Received October 12, 2020, accepted October 26, 2020, date of publication November 16, 2020, date of current version December 8, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3037731

Deployment and Analysis of a Hybrid Shared/Distributed-Memory Parallel Visualization Tool for 3-D Oil Reservoir Grid on OpenStack Cloud Computing

ALI A. EL-MOURSY[1], (Senior Member, IEEE), FADI N. SIBAI[2], HANAN KHALED[3], SALWA M. NASSAR[3], (Member, IEEE), AND MOHAMED TAHER[4]

[1]Computer Engineering Department, University of Sharjah, Sharjah, United Arab Emirates
[2]College of Computer Engineering and Science, Prince Mohammad Bin Fahd University, Al-Khobar 31952, Saudi Arabia
[3]Electronics Research Institute, Cairo 12622, Egypt
[4]Department of Computer and Systems Engineering, Ain Shams University, Cairo 11566, Egypt

Corresponding author: Ali A. El-Moursy (aelmoursy@sharjah.ac.ae)

This work was supported in part by the Cloud Computing Center of Excellence under Grant 5220, in part by the Science and Technology Development Fund (STDF), Egypt, and in part by the Distributed and Networked Systems Research Group, University of Sharjah, United Arab Emirates, under Grant 150410. Through those grants the Cloud system is established to perform the experiments and simulations.

ABSTRACT The main goal of oil reservoir management is to provide more efficient, price-effective and environmentally more secure oil production. Oil production management includes an accurate characterization of the reservoir and strategies that involve interactions between reservoir data and human assessment. Hence, it is important to graphically visualize and handle massive data sets of oil and gas pressure / saturation levels to help decision makers in statistical analysis, history matching and recovery of hydrocarbons of the reservoir. In this article, we experimentally study the parallelization of intensive computation for a 3-D (three dimensional) oil reservoir data visualization tool. For this tool, we develop and implement a transformation and lighting model to visualize and react with the grid. Herein, we propose a hybrid (shared memory and distributed memory) parallelization technique to adapt with the data processing scalability. We tested these implementations on OpenStack Cloud Virtual Cluster. Our results indicate that although the virtual platform adds overhead for running parallel implementations, utilizing knowledge of the VM location on the compute host and network traffic among VMs to deploy the virtual environment can achieve significant performance enhancements. Hybrid parallel implementation using large data size can achieve 70× speedup over serial execution without owning a costly HPC infrastructure as the conventional parallel processing deployment model.

INDEX TERMS Cloud computing, data visualization tool, HPC, hybrid (distributed/shared)-memoryparallel programming, MPI, multi-threading, OpenStack.

I. INTRODUCTION

Cloud computing has been rapidly developing its level of maturity and popularity, and has provided to researchers a large pool of computation resources for High Performance Computing (HPC) applications at possibly low cost [1]. Cloud computing encompasses computer networks and virtualization technologies which can facilitate both remote computing resources interaction and workload management. Aside from the technical concepts, Cloud computing offers a commercial enterprise model as customers pay for the demanded resources in contrast to conventional HPC platforms. In cloud computing, customers can swiftly modify their pool of resources, via elasticity mechanism [2] (one of the cloud computing characteristics), based on the infrastructure size controlled by the Cloud Service Provider (CSP) [1]. Authors in [3] show three usage scenarios for HPC with cloud computing; i. HPC over Cloud platform; focused on

The associate editor coordinating the review of this manuscript and approving it for publication was Jing Bi.

moving HPC programs totally to a private/public cloud computing environment which is called Infrastructure as a Service (IaaS). ii. HPC with Cloud platform; focused on using the Cloud to complement HPC resources to support application workloads such as heterogeneous requirements and unexpected demands. iii. HPCaaS (HPC as a service); emphasised on offering HPC via Cloud services using demand elasticity of the Cloud platform by merging the resilient access of the cloud computing environment with the HPC performance environment [4].

As the computation power of hardware infrastructure increases, researchers are running more complicated applications that may utilize a large number of computing resources and generate huge amounts of data. It is hard to automate parallelization for scientific applications due to the customized performance analysis needed to identify execution time hotspots and the performance tuning needed to utilize the underlying computation capability [5], [6]. Recent research in parallel processing indicates that deep algorithm analysis and investigation are inevitable to tune the performance of the parallel implementation. The performance of the parallel implementation for different scientific applications depends on the application design and the used platform. For shared memory applications, compiler capability such as OpenMP is not efficient to parallelize complex applications automatically due to dependencies either in data or in tasks. For distributed memory applications, there are no tools yet developed for auto-parallelization [7]. Furthermore, combining both parallel processing paradigms (shared-memory and distributed-memory) is extremely challenging for auto parallel tools. One of the applications which needs parallelization in the field of reservoir engineering is a 3-D oil reservoir data visualization tool which can render and visualize the output data of the reservoir simulator such as pressures and saturations in a 3-D environment to assist the decision maker in statistical analysis, historical matching and recovery of hydrocarbons of the oil reservoir.

The focus of this article is the hybrid shared/distributed parallel implementations of an oil reservoir data visualization tool, deployment on Virtual Cluster (VC), and conducting a comparative analysis of the performance to enable users of the tool to use a virtual platform instead of bare-metal one.

The contributions of this article are:

- Developing hybrid shared/distributed parallel implementations of an oil reservoir data visualization tool.
- Deploying Virtual Cluster (VC) over OpenStack Cloud and studying the performance of the 3-D oil reservoir data visualization tool on this virtual cluster.
- Providing comparative and quantitative analysis for various virtual hardware resources (VCPU or Virtual network) using the shared and distributed parallel oil reservoir data visualization tool for different cases of VCPU deployment.
- Analyzing and comparing the performance between HPC as a dedicated hardware platform [8] and VC as a virtual platform for the parallel oil reservoir data visualization tool.

The rest of the paper is structured as follows. Section II presents the related work. Section III describes the serial oil reservoir data visualization tool. The parallel implementation is depicted in section IV. Section VI discusses some concepts of OpenStack Cloud. Section VII describes the experimental platforms. Section VIII discusses results and analysis. Finally, the article is concluded in section V.

II. RELATED WORK

HPC experts have leveraged the benefits of new technology trends including, parallel processing and cloud computing technologies. Researches and studies have begun showing the feasibility of running HPC applications on the remote resources of cloud computing. Authors in [9] used two benchmarks of the SPECMPI suite [10] in addition to two Parallel benchmarks of the NASA benchmarks [11] as the workload to analyze their performance model. They predict the performance of MPI applications on Cloud bare-metal multi-core machines, achieving, on average, 86% accuracy for the used benchmarks.

Software Defined Networks (SDN) [12]–[14] is a new technology to manage the Virtual Networks through Software controller as a replacement for the physical routers controls. SDN separates the data plane from the controller plane of the network to allow much flexible and dynamic network management in the virtual Networks. Authors in [15] used two Hypercube and Mesh multiplication algorithms, to evaluate the performance of distributed HPC model over OpenStack cloud under SDN infrastructure. A simple Matrix by Matrix multiplication is used as the benchmark for the study with no real application is evaluated. While authors [16] in focus in multi-application scenarios trying to enhance revenues. No parallel applications are considered in their study. The authors in [17] and [18] study the performance impact of *OpenFlow* (a well-known SDN implementation) on the Message Passing Interface (MPI) collective communication. SDN is proposed to support the dynamic nature of the virtual servers and networks. Although SDN is utilized in enhancing the network performance of HPC on Cloud, we do not consider it in our research in this article for many reasons. The research in the SDN is kind of orthogonal to our work since we explore visualization performance components and focus on the computation load distribution rather than the network routing. Our assumption is a dedicated Cloud platform to run HPC which much less sensitive to the Network routing compared to a typical Cloud system utilized by different flavors of workload mix running HPC and non-HPC concurrently. Our system is solely utilized by our parallel application. The impact of SDN is more elaborated in large virtual Cluster Network where the efficient routing is a crucial component in the system utilization.

Another approach is presented in [19] for estimating the cloud preparedness for parallel applications. The authors reported that native cloud and parallel applications design have inconsistent goals since native cloud applications are throughput-based. In contrast, parallel application design aims at headstrong performance enhancement in terms of parallel efficiency and speedup. The authors' approach advocated the migration of parallel application to the cloud depending on the parallel design. Authors in [20] used two HPC benchmarks, the HPC Challenge (HPCC) and the High Performance Conjugate Gradient (HPCG), to estimate the performance of Azure and AWS cloud platforms. They concluded that it is better to test the application on the desired Cloud platform to determine the suitable platform. OpenStack over KVM (Kernel-based Virtual Machine) [21] hypervisor provides the best virtualized CPU according to [22] which analyzed the virtual CPU performance based on KVM hypervisor for three open source Cloud middleware: Apache CloudStack [23], Eucalyptus [24] and OpenStack [25]. The configuration of KVM on each Cloud platform provides several abstractions of the underlying CPU hardware that has its impact on the performance of the running programs. All the above-mentioned researches depend on benchmarks to test the Cloud environment for running parallel processing, but this article focuses on a real parallel application with GUI overhead to test the performance of the Cloud. In addition, most of the above-mentioned studies estimate HPC applications on public Clouds with lack of knowledge of the underlying hardware infrastructure to conduct fair analysis. Hence, this study targets the deployment of a parallel 3-D oil reservoir data visualization tool over an OpenStack Cloud utilizing the KVM hypervisor.

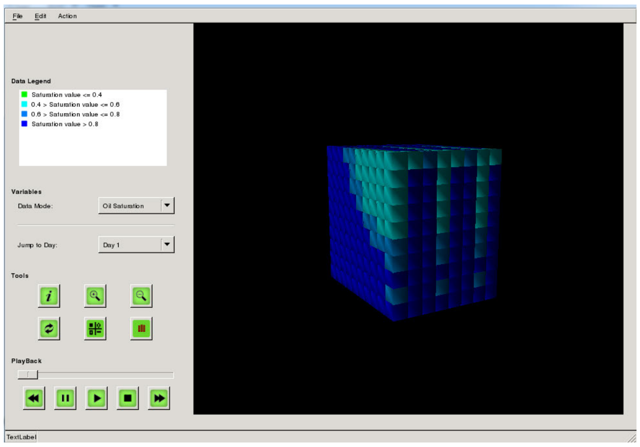
III. SERIAL OIL RESERVOIR DATA VISUALIZATION TOOL

The data grid of the oil reservoir visualization tool consists of individual cube-shaped cells. Pressure, oil and water saturation values are generated from a serial oil reservoir simulator [26] and then mapped against a color set to visually represent a single cell (grid point). The tool also has buttons for forward, playback and rewind to allow the user and to visualize the grid at a specific time for checking the produced data. Each interaction renders a day of simulation hence any pressure or saturation value changes are emphasised with a color changing on the grid. The user can choose any cell on the grid and check on a specific day the values of pressure and oil or water saturation. The final output of grid rendering for oil saturation data of grid size 10 × 10 × 8 is shown in Fig. 1.

The Transformation and Lighting Models are at the heart of the oil reservoir data visualization processing. Hence, we will discuss them in more details in the following subsections.

A. TRANSFORMATION

In the visualization tool, transformation of an object in 3-D environment consumes significant computation power for matrix manipulations in translation, rotation, and camera matrices. Translation [27] is a kind of displacement of an object in the 3-D space to a new position, thus moving the



Data Legend
🟩 Saturation value <= 0.4
🟦 0.4 > Saturation value <= 0.6
🟦 0.6 > Saturation value <= 0.8
🟦 Saturation value > 0.8

Variables
Data Mode: Oil Saturation
Jump to Day: Day 1

FIGURE 1. Data grid rendering.

object depending on a vector of translation; Rotation [27] is intelligent combinations of sine and cosine with an angle in radians or degrees for each unit axis (X, Y or Z) in 3-D space; and camera [28] for freely moving in a 3-D scene. We consider the following vectors to implement camera features: Camera position, Camera direction, Camera up vector (vector of the y-axis) and Camera right vector (vector of the x-axis). The camera matrix is given by:

$$\begin{pmatrix} rightVectorx & rightVectory & rightVectorz & 0 \\ upVectorx & upVectory & upVectorz & 0 \\ zVectorx & zVectory & zVectorz & 0 \\ Originalx & Originaly & Originalz & 1 \end{pmatrix}$$

The camera materix depends on vectors in 3-D dimension (x, y and z) where *Original* is the vector of the camera position in the world space. *zVector* is the vector of camera direction looking at. *upVector* is the vector pointing upwards from the camera. *rightVector* is the vector pointing to the right direction of the camera.

B. LIGHTING MODEL

Lighting is highly complex in the natural environment and depends on too many aspects. Therefore, lighting in the simulation setting is based on reality approximations using simpler models which are much easier to handle and look closer to reality [29]. Our model uses the Basic Per-Vertex Lighting Model [30]. The light per vertex is calculated, and the shading model smoothens out the region color. The color of each vertex relies on ambient, diffuse and specular lighting terms. Each term depends on the combination of the surface material properties, the properties of light source (such as color and position of light) and location of the viewer. The general Eq. 1 describes the Basic Per-Vertex Lighting Model:

$$Color\_Vertex = Diffuse + Specular + Ambient \qquad (1)$$

Next, we discuss the components of Equation 1.

`Ambient Lighting`: The ambient component represents light that has rebounded around the scene that returns from the surrounded area [31]. Since there is no direct source of light, the ambient intensity component is simplified by the ambient light color scaled by the property of the ambient material. The formula for ambient component is given by Eq. 2:

$$Ambient = ambient_{surface} * ambient_{light} \tag{2}$$

`Diffuse Lighting`: The diffuse component represents direct light reflected in all directions equally from a surface [31]. The intensity of diffuse component relies on the incident light angle and can be represented as a dot product between surface normal vector and the surface towards light vector. The formula for diffuse component is given by Eq. 3:

$$Diffuse = diffuse_{surface} * diffuse_{light} * max\,(n.l, 0) \tag{3}$$

where l is the normalized vector toward the light source and n is the normalized surface normal.

`Specular Lighting`: The specular component represents light scattered all around the mirror direction from a surface [31]. The specular intensity term relies on the viewer's position relative to the surface. The viewer will not see a specular highlight on the surface if the viewer is not at a place that receives the reflected rays. The formula for specular component is given by Eq. 4:

$$Specular = specular_{surface} * specular_{light} * (max\,(n.h, 0))^{brightness} \tag{4}$$

where n is the normalized surface normal, h is the normalized vector which is halfway between the normalized viewer vector and the normalized light source vector.

We note that all lighting model equations need the normalized surface normal. In the proposed lighting model, we need to calculate the normalized surface normal for each and every vertex/cell of the grid as follows: i. surface normals for every triangle which can make up the cell are calculated; ii. the surface normals of shared triangles for every vertex are averaged out; iii. the final average normal is normalized for each vertex. Eqs. 2, 3 and 4 are combined to give a single vertex final color value, provided by Eq. 1.

To implement `transformation` (such as grid Zooming in/out or Rotation): i. Model_View matrix is generated by multiplication of the following three matrices: a. camera matrices which consist of the previously mentioned vectors to support three functions: Moving forward, Moving backward by increasing and decreasing forward vector (Z-axis) values and Rotation camera around Y-axis by changing right vector and forward vector values; b. translational matrix c. rotational matrix; ii. Each vertex is multiplied by the created Model_View matrix to get the new vertex position where transformation calculation is shown in Figure 2. To implement `Lighting Model`: Surface normal should be calculated and is initially computed when the data grid is loaded to avoid repetition of computation of the surface normal during each transition through the following steps: i. compute surface normal for all triangles of each cell, where the surface normal for a triangle can be calculated by creating two

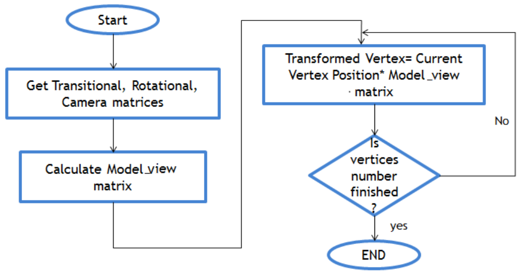

Start
Get Transitional, Rotational, Camera matrices
Calculate Model_view matrix
Transformed Vertex= Current Vertex Position* Model_view matrix
Is vertices number finished ?
No
yes
END

FIGURE 2. Transformation steps.

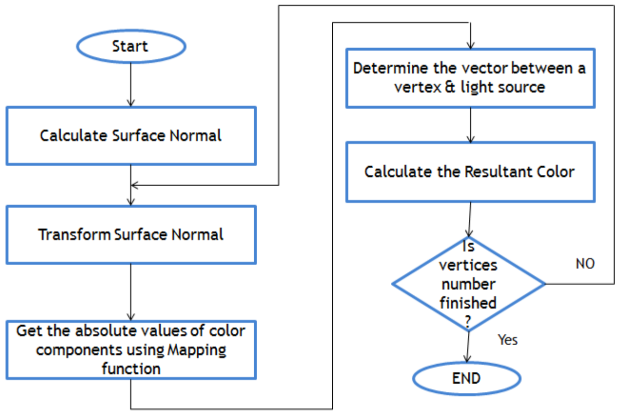

Start
Calculate Surface Normal
Transform Surface Normal
Get the absolute values of color components using Mapping function
Determine the vector between a vertex & light source
Calculate the Resultant Color
Is vertices number finished ?
NO
Yes
END

FIGURE 3. Ligting model steps.

triangle vectors using each of the three vertices and obtaining a two-vector vector product; ii. Average out surface normal of triangles which share the same vertex. The above two steps produces the surface normal for each vertex then the vertex is lit using the Basic Per-Vertex Lighting Model. For each transition, the surface normal for the transformed vertex depends on the original surface normal and transformed matrix according to the following procedure to obtain the parameters for each color component:

1) The surface normal for each vertex is transformed and is normalized.
2) Using mapping function, specular, diffuse and ambient color values are calculated for every vertex.
3) The vector between any vertex and source of light is determined as well as the vector between the vector to the viewer and the vector to the source of light.

Then, by using Eqs. 1, 2, 3 and 4, the resultant color is derived for each vertex as shown in Figure 3. Finally, the new vertex positions for the whole grid are drawn and the resultant color values for each vertex are passed to OpenGL [32], [33].

IV. PARALLEL ALGORITHM AND IMPLEMENTATION

The first goal of our study is evolving parallel methods for visualization of serial oil reservoir [34] to allow a real-time processing of the data, smooth and fast data processing by the analyst, and to promptly adapt to any data changes in the oil reservoir simulator (pressure, oil and water saturation). The oil reservoir visualizer allows the user to load the data of the 3-D grid and sketch each cell, and enter the data for each block of the oil reservoir. Adjacent cells along the X axis direction forms a full row. The group of adjacent rows along the Y axis direction forms one full plane. Moreover, the adjacent planes along the Z axis direction form the whole 3-D Grid.

Execution time analysis is performed to check the serial implementation hotspots of the Visualization application using Gprof (GNU profiler) tool which is used for performance analysis of Linux/Unix applications [35]. This analysis provides the relative execution time of each task, which is an essential indicator of the application hotspots to explore the parallelization opportunities. The tool computation load is splitted into two main tasks: visualization (necessary functions for grid computations and rendering) and, Main_Window (necessary functions for GUI events). Main_window tasks cannot be parallelized due to GUI events and tasks are sequential by nature since multiple GUI events cannot be executed at the same time. Thus, we focus on visualization tasks as shown in Figure 4 especially m3dNormalizeVector and CalculateLight (transformation and Lighting essential functions) which have prominent percentages with respect to the whole execution time of program. The remaining routines consume smaller quantities of execution time (from 5% to 10%). The remaining functions do not only have little relative execution time but additionally have a high degree of data dependency causing very frequent data exchanges among the processing elements. Accordingly, any aggregate performance gain from the parallelization of the remainder tasks could be a source of considerable communication overhead.

The `Master_Slave` (MS) model is among the most popular paradigms for parallel applications. In this model, one processing element known as the master is responsible for executing the optimized functions and distributing the heavy work among slave processing elements. The main advantage of the MS parallel model is the ability to balance the load among the slaves [36]. Due to the nature of the proposed data visualization application, the MS model fits well the oil visualizer simulation implementation since the tool is divided into two basic tasks, GUI and processing of the grid data. The master processing element will be responsible for executing GUI routines and its events (such as uploading data or navigation across days) and the data is distributed among the slave elements to calculate heavy computation (transformation and lighting model), then the overall results are gathered by the master. Lighting and transformation kernels are iterative routines as shown in Figures 2 and 3. In each iteration, a set of calculations is executed on each vertex for each cell to get the final transformation and color during user interaction.

A. DATA DECOMPOSITION APPROACHES

The 3-D grid data is either pressure, or saturation levels of oil or water for blocks of the reservoir. Each block is represented by a cell in the proposed data visualization tool,

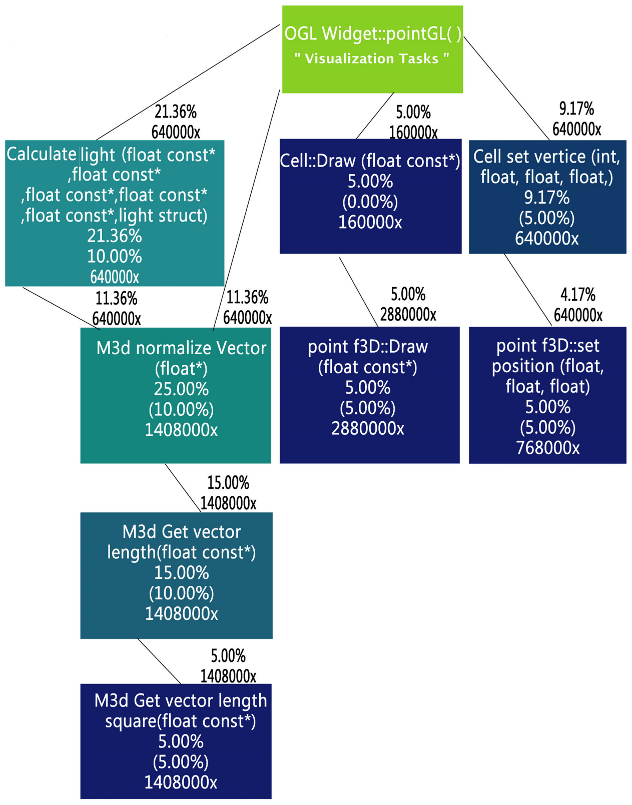

OGL Widget::pointGL( )
" Visualization Tasks "
21.36%
640000x
Calculate light (float const* ,float const* ,float const*,float const* ,float const*,light struct)
21.36%
10.00%
640000x
5.00%
160000x
Cell::Draw (float const*)
5.00%
(0.00%)
160000x
9.17%
640000x
Cell set vertice (int, float, float, float,)
9.17%
(5.00%)
640000x
11.36%
640000x
11.36%
640000x
M3d normalize Vector (float*)
25.00%
(10.00%)
1408000x
5.00%
2880000x
point f3D::Draw (float const*)
5.00%
(5.00%)
2880000x
4.17%
640000x
point f3D::set position (float, float, float)
5.00%
(5.00%)
768000x
15.00%
1408000x
M3d Get vector length(float const*)
15.00%
(10.00%)
1408000x
5.00%
1408000x
M3d Get vector length square(float const*)
5.00%
(5.00%)
1408000x

FIGURE 4. **Call diagrams and relative execution times for visualization task.**

and each cell consists of eight vertices. For each vertex, lighting model and transformation routines are calculated to color and transform its cell. Hence, the same routines are executed MAXX*MAXY*MAXZ* 8 times on different data with each grid transaction where MAXX, MAXY and MAXZ represent the grid size in x, y and z directions, respectively. We found that the best fitting parallel technique is Single Program Multiple Data (SPMD) since the same functions are executed for each vertex.

Our parallel implementation method consists of a sequence of three phases: loading, calculation and gathering. In the loading phase, the data is divided among the processing elements (threads or nodes), then every processing element executes needed calculations on its data. Finally in the gathering phase, the final output values are collected in master/main processing element to draw the final results.

Our parallel technique for 3-D oil reservoir data visualization tool has the following advantages: i. Scalability, our parallel implementations scale properly with increasing data size and increasing number of processing elements; ii. Decreasing communication overhead, by gathering all data when all processing element complete their calculations.

According to the grid creation nature in the tool, Two parallel data decompositions can be implemented: *coarse grain* data decomposition and *fine grain* data decomposition. In the *coarse grain* data decomposition, each processing element computes Lighting and transformation routines on one plane

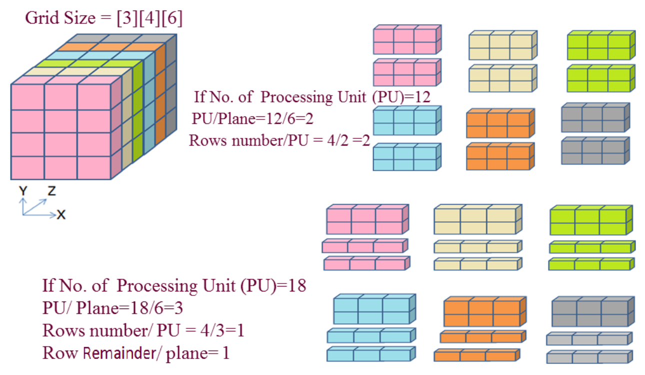


FIGURE 5. **Fine grain (2D) data decomposition.**

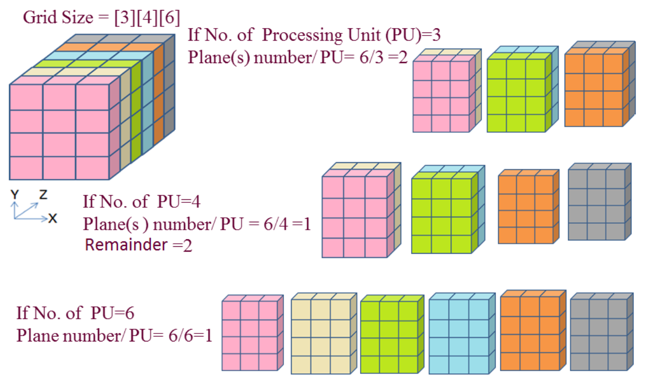
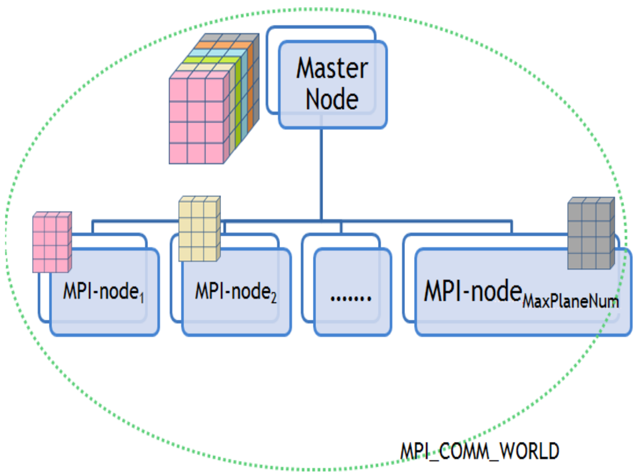


FIGURE 6. **Coarse grain (1D) data decomposition.**

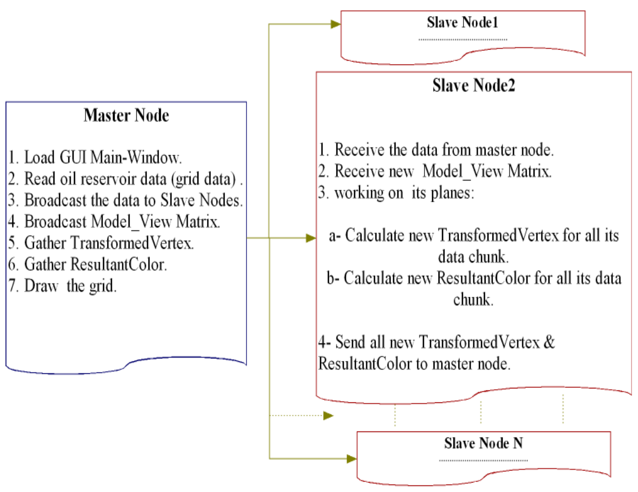
of the 3-D grid at least. In the *fine grain* data decomposition, parallelization is executed on the level of plane since every processing element can manage one or more cells. Assigning cell operations to the processing elements (nodes or threads) in adjacent chunks of cells (row wise, 2D or *fine grain* data decomposition) as shown in Figure 5, or adjacent rows (plane wise, 1D or coarse grain data decomposition) [37] as shown in Figure 6, is the most efficient assignment to avoid sparse data processing and achieve a more optimized communication handling.

B. PARALLEL IMPLEMENTATIONS

The parallel implementation depends on the underlying infrastructure of the hardware system. Typically two models are utilized, shared memory and distributed memory. Each of these models has its advantages and drawbacks. In shared memory architecture, shared address space among processing elements provides the ability to develop programs effectively since the software engineer does not need to explicitly exchange messages among processing elements (send and receive) for data sharing. Furthermore, shared address space is characterized by the uniform data among threads due to relative short distance between CPUs and memory. However the system scalability is limited. The shared memory model drawback is the lack of CPUs scalability as growing numbers of CPUs can increase the traffic on the shared



(a) Logical communication


Slave Node1
Slave Node2
1. Receive the data from master node.
2. Receive new Model_View Matrix.
3. working on its planes:
a- Calculate new TransformedVertex for all its data chunk.
b- Calculate new ResultantColor for all its data chunk.
4- Send all new TransformedVertex & ResultantColor to master node.
Master Node
1. Load GUI Main-Window.
2. Read oil reservoir data (grid data) .
3. Broadcast the data to Slave Nodes.
4. Broadcast Model_View Matrix.
5. Gather TransformedVertex.
6. Gather ResultantColor.
7. Draw the grid.
Slave Node N

(b) Implementation details

FIGURE 7. **Coarse grain data decomposition on distributed memory model.**

CPU-memory bus and congest data transmission. On the other hand, distributed memory machines can be scaled in terms of processor numbers. A network that connects the far distant processing elements allows much more scalability compared to the shared memory. However, in the distributed memory model, data communication among processors is much slower, highly dependent on the network speed, and is carried out explicitly by the software engineer [38]. Detailed parallel implementations of 3-D oil reservoir data visualization based on different memory models, are proposed in the following sections.

1) *Distributed-Memory Implementation*

First, the Model_View matrix and oil reservoir data are broadcasted by the master node since this information is accessible from GUI Main_Window after user transaction (such as uploading data, rotation or zooming). Then the three phases; loading, calculating and gathering are executed. Implementation of "coarse grain

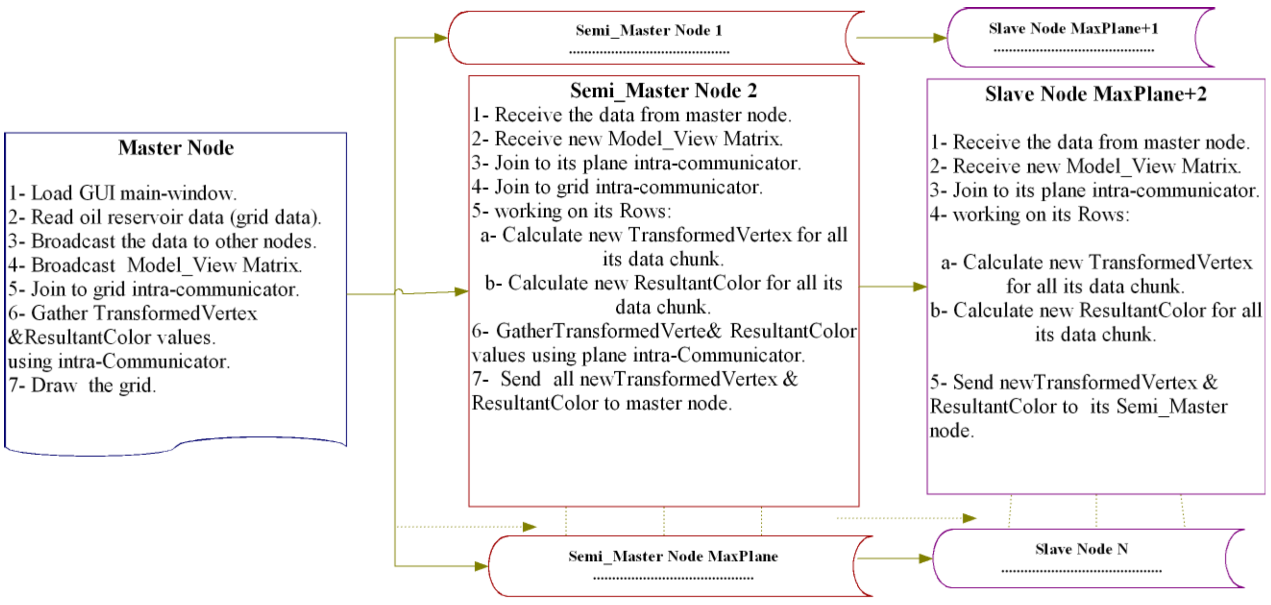

Semi_Master Node 1
Slave Node MaxPlane+1
Master Node
1- Load GUI main-window.
2- Read oil reservoir data (grid data).
3- Broadcast the data to other nodes.
4- Broadcast Model_View Matrix.
5- Join to grid intra-communicator.
6- Gather TransformedVertex &ResultantColor values.
using intra-Communicator.
7- Draw the grid.
Semi_Master Node 2
1- Receive the data from master node.
2- Receive new Model_View Matrix.
3- Join to its plane intra-communicator.
4- Join to grid intra-communicator.
5- working on its Rows:
a- Calculate new TransformedVertex for all its data chunk.
b- Calculate new ResultantColor for all its data chunk.
6- GatherTransformedVerte& ResultantColor values using plane intra-Communicator.
7- Send all newTransformedVertex & ResultantColor to master node.
Slave Node MaxPlane+2
1- Receive the data from master node.
2- Receive new Model_View Matrix.
3- Join to its plane intra-communicator.
4- working on its Rows:
a- Calculate new TransformedVertex for all its data chunk.
b- Calculate new ResultantColor for all its data chunk.
5- Send newTransformedVertex & ResultantColor to its Semi_Master node.
Semi_Master Node MaxPlane
Slave Node N

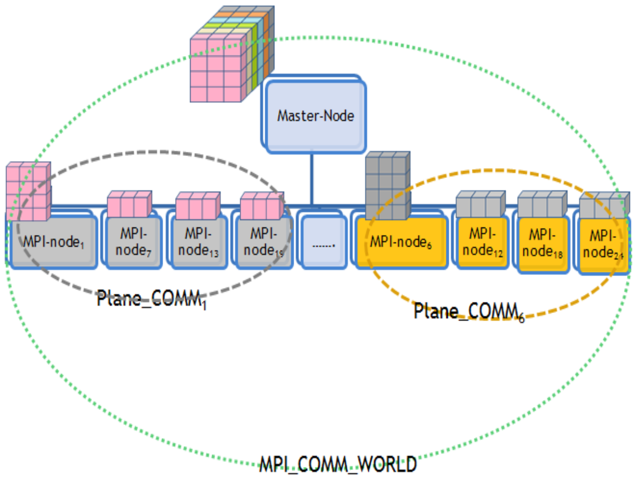
FIGURE 8. **Fine grain data decomposition approach on distributed memory model.**

data decomposition approach" is shown in Figure 7 as in the "loading phase", planes are circulated among the slave nodes. In the "calculating phase", every slave node executes lighting and transformation kernels and stores its final outcomes into local buffers. Finally, the master node collects the final outcomes of lighting and transformation models in the "gathering phase" to draw the grid using rendering since the data transfer between compute processors requires explicit communication in the distributed memory techniques.

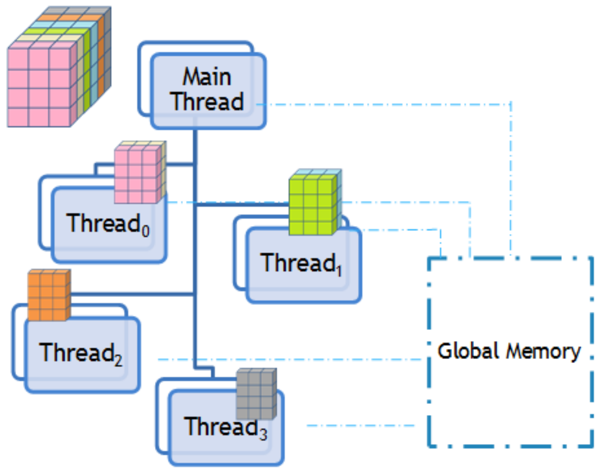
Implementation of "fine grain data decomposition approach" is shown in Figure 8 as the rows are divided among the slave nodes in the "loading phase" since more than one node work in the same plane but on distinct rows. A recursive (hierarchical) decomposition of the planes into rows is the most effective way to split the rows between the nodes. This will allow few node subsets to interact and partly collect information to one node (semi_master) then this node will send the collected data to the master node. The MPI library supports this hierarchical strategy through the capability of sub_communicators. This is accomplished by using the `MPI_Comm_Split` function to create `(MAXZ+1)` intra-sub-communicators from the (`MPI_COMM_WORLD`) (main communicator of MPI). The GRID-COMM is utilized for accomplishing the communication between the master node and semi_master nodes. An example is shown in Figure 9, when operating on 25 processors, MPI COMM WORLD divides into six sub_communicators for data grid size $3 \times 4 \times 6$. Each slave node executes Lighting and Transformation kernels in the "calculating phase" and stores its final outcomes in local buffers. In the "gathering phase" and using PLANE-COMM-i, every semi_master node collects the outcomes from its slave nodes for an entire plane, then the master node collects the final outcomes of semi-master nodes of lighting and transformation models to render the entire grid.

2) *Shared-Memory Implementation*
In this implementation, reservoir data and Model_View matrix are shared among all threads. Implementation of "coarse grain data decomposition approach" is shown in Figure 10. Synchronization among threads is required and data grid processing is limited to a single compute node RAM capacity. In the "calculating phase", every slave thread executes lighting and transformation kernels. Afterward, each slave thread copies the final outcome of lighting and transformation models to global buffers in the "gathering phase" so the master thread can render the grid as soon as the slave threads are synchronized.
Implementation of "fine grain data decomposition approach" is similar to the coarse grain one in all phases, however every slave thread operates at the row level instead of the plane level as shown in Figure 11, thus allowing more threads to work together on the required calculations.

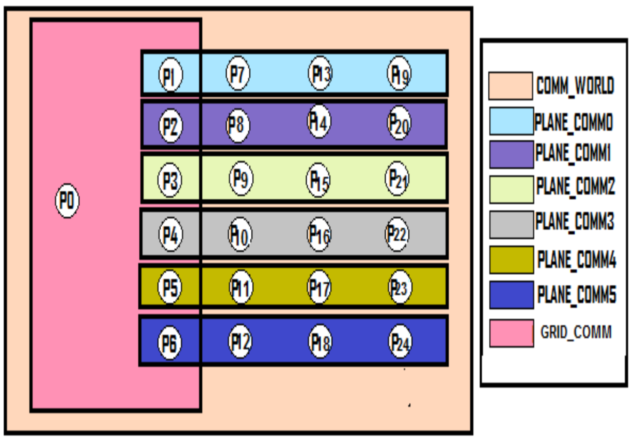
3) *Hybrid Shared/Distributed Memory Implementation*
Each strategy of data decomposition (Coarse grain or Fine grain) has its advantages. Coarse grain data decomposition strategy operates on adjacent big pieces of data (planes). Hence the data is divided among a small number of slaves a small number of processing elements are utilized. However, the fine grain data decomposition strategy operates on adjacent small pieces of data (Rows). Fine-grain is more scalable with



(a) Logical communication

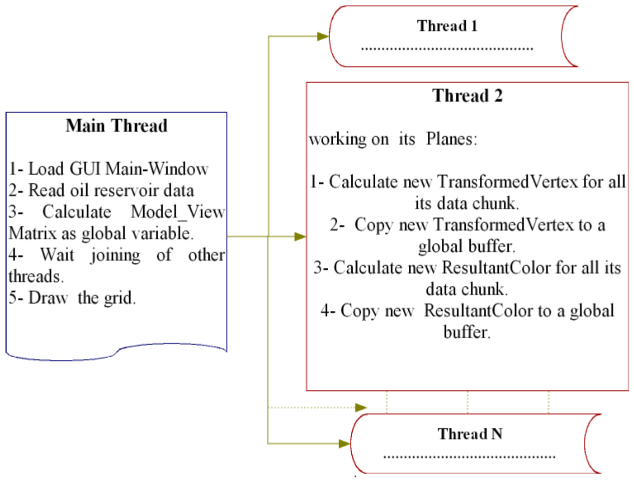


(a) Logical synchronization



(b) sub_communicators detail

FIGURE 9. **Six sub_communicators from MPI_COMM_WORLD.**



Main Thread

1- Load GUI Main-Window
2- Read oil reservoir data
3- Calculate Model_View Matrix as global variable.
4- Wait joining of other threads.
5- Draw the grid.

Thread 2

working on its Planes:

1- Calculate new TransformedVertex for all its data chunk.
2- Copy new TransformedVertex to a global buffer.
3- Calculate new ResultantColor for all its data chunk.
4- Copy new ResultantColor to a global buffer.

(b) Implementation details

FIGURE 10. **Coarse grain data decomposition on shared memory model.**

running more processing elements. However, it may involve an elevated level of communication with the master processing element. In terms of processing elements, the distributed memory model is scalable. On the other hand, implementation of shared memory has the benefits of less overhead communication.

The combination of both strategies in one hybrid approach can add further enhancement in calculation ability [39]–[41]. Figure 12 shows the detailed implementation since in the "loading phase", grid planes are distributed using MPI among the semi-masters, then every node forks multiple slave threads to distribute the rows among them. Every thread executes the routines of the lighting and transformation model for its rows in the "calculating phase". Each semi-master sends its final outcomes to the master node in the "gathering phase". Then the master node collects the data to render the entire grid. Hence, the combination of both implementations into one hybrid shared/ distributed parallel technique makes it possible to capitalize on the benefits of both techniques to adapt effectively with the recent computing systems and to use suitable data decomposition granularity hierarchically.

V. COMPLEXITY ANALYSIS

In this section, a quantitative complexity analysis of our parallel implementations is presented and compared to the serial implementation of the parallel visualization tool. The computational complexity depends on the number of mathematical operations performed in each implementation. The computational complexity of the serial implementation is *O(dimension * vertex_per_cube * MAXX * MAXY * MAXZ)* since the dimension is provided by the called functions inside the loops which are shown in Figure 2 and 3 flowcharts. In addition, each cube in the grid has eight vertices as well as the dimension is small so these parameters can be neglected from the complexity with the large grid

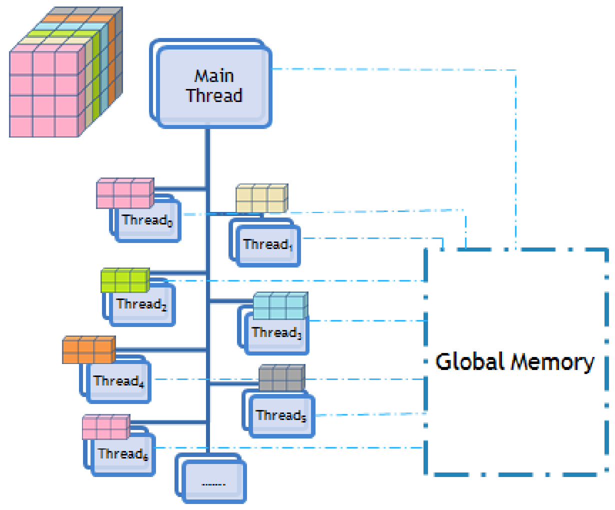
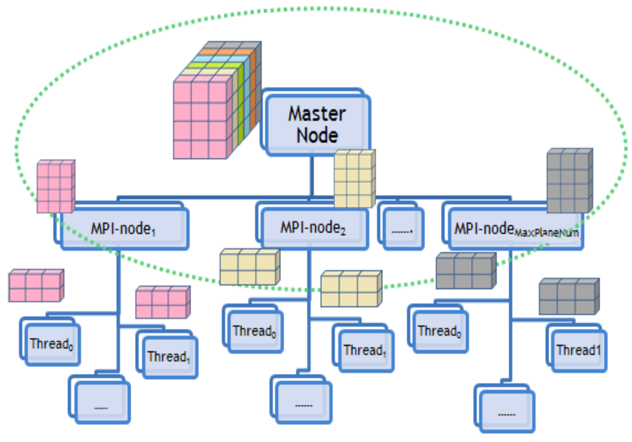
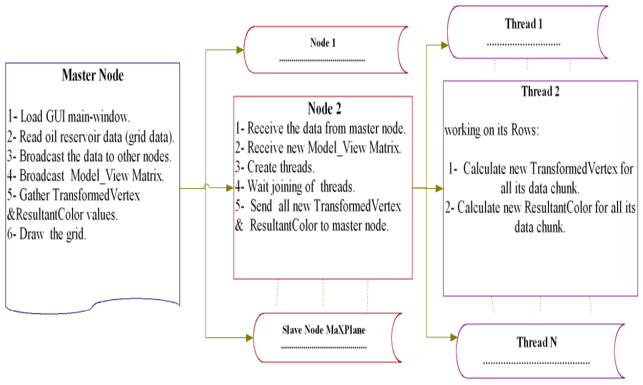


FIGURE 11. Fine grain data decomposition on shared memory model.



(a) Logical synchronization and communication


Master Node
1- Load GUI main-window.
2- Read oil reservoir data (grid data).
3- Broadcast the data to other nodes.
4- Broadcast Model_View Matrix.
5- Gather TransformedVertex &ResultantColor values.
6- Draw the grid.
Node 1
Node 2
1- Receive the data from master node.
2- Receive new Model_View Matrix.
3- Create threads.
4- Wait joining of threads.
5- Send all new TransformedVertex & ResultantColor to master node.
Slave Node MaXPlane
Thread 1
Thread 2
working on its Rows:
1- Calculate new TransformedVertex for all its data chunk.
2- Calculate new ResultantColor for all its data chunk.
Thread N

(b) Implementation details

FIGURE 12. Coarse/ fine grain data decomposition on hybrid memory model.

sizes, therefore the serial implementation has a cubic complexity *O(dimension * vertex_per_cube * MAXX * MAXY * MAXZ)*. On the other hand, the computational complexity for the parallel distributed memory implementations is performed on multiple stages depends on the computational operations and communication operations and the final complexity is the largest one. For the coarse grain implementation with MPI, the computational complexity can be illustrated through two stages as it is shown in Figure 7:

- Iteration operations: each node calls the function inside the loops with *(planes_per_node * MAXY * MAXX * vertex_per_cube * dimension)* times so its computational complexity *O(dimension * vertex_per_cube * MAXX * MAXY * MAXZ / num_nodes)* since *planes_per_node = (MAXZ / num_nodes)*.
- Gathering operations: the master node gathers a number of elements from the slave nodes so the computational complexity depends on the number of elements that is received from each slave node as well as the number of nodes. The master node collects *vertex_per_cube * dimension * MAXX * MAXY * MAXZ / num_nodes* elements from each slave so the computational complexity is *O(dimension * vertex_per_cube* MAXX * MAXY * MAXZ * log (num_nodes) / num_nodes)*. According to the aforementioned analysis, the overall computational complexity for the coarse grain implementation is therefore *O(dimension * vertex_per_cube* MAXX * MAXY * MAXZ * log (num_nodes) / num_nodes)*.

For the fine grain implementation with MPI, the computational complexity can be explained within three stages as shown in Figure 8 for this implementation:

- Iteration operations: each node calls the function inside the loops with *(planes_per_node * rows_per_node * MAXX * vertex_per_cube * dimension)* times, as more than one node are working on the same plane, and *planes_per_node = 1, rows_per_node = (MAXY * MAXZ / num_nodes)*. Therefore the computational complexity is *O(dimension * vertex_per_cube * MAXX * MAXY * MAXZ / num_nodes)* which is similar to the coarse grain implementation.
- Semi_Gathering operations: each semi-master node collects the data from its slaves since the number of collected data elements from each slave is *(rows_per_node * MAXX * vertex_per_cube * dimension)*, and the number of nodes for each plane *(i.e, nodes_per_plane) is num_nodes/MAXZ*. Therefore the computational complexity is *O(dimension * vertex_per_cube * MAXX* MAXY* MAXZ *log (num_nodes/MAXZ) / num_nodes)*.
- Final_Gathering operations: the master node collects the overall data from the semi-master nodes since the number of collected data elements from each semi-master node is a complete plane *(MAXX* MAXY)* and the number of nodes equals *MAXZ*. Therefore the computational complexity is *O(dimension * MAXX * MAXY * log*

TABLE 1. Complexity analysis results.

Implementation	Complexity
Serial	*S*
Coarse Grain	*S * log(num_nodes) / num_nodes*
Fine Grain	*S * log(num_nodes / MAXZ) / num_node*

(MAXZ)). According to the aforementioned analysis, the overall computational complexity for the fine grain implementation is *O(dimension * vertex_per_cube * MAXX * MAXY * MAXZ *log (num_nodes / MAXZ) / num_nodes)*.

Comparing the complexities of the three implementations, and setting S to *dimension * vertex_per_cube * MAXX * MAXY * MAXZ*, reveals the results shown in Table 1. The fine grain implementation has the best complexity when *1 <= num_nodes / MAXZ < 2, i.e., num_nodes is near MAXZ*, but not larger than *2 * MAXZ*. When *num_nodes > 2 * MAXZ*, the coarse grain implementation is better.

VI. OpenStack CLOUD COMPUTING

In 2010, OpenStack was presented and the first contributors for its development were Rackspace and NASA. It is a rapidly growing free open source middleware for Cloud [42]. OpenStack is a Cloud middleware which manages huge pools of storage, networking and computing resources of the data center. The whole resources are overseen via a dashboard that provides administrators with a high level of supervision (e.g., Quota of each project, status for all services, creating new flavors for machine creation) while enabling their clients to provision resources by a web page portal. OpenStack is the most common open source solutions of Cloud which offers Infrastructure as a Service (IaaS) architecture. There are three main advantages that make OpenStack an attractive candidate among other private Cloud solutions [43], [44]:

- Publicity: OpenStack is already deployed in different large distributed environments (deployed by 4500 individual members and 850 organizations [42]).
- Flexibility: OpenStack supports most of the existing hypervisors (such as Xen, KVM, Hyperv, XenServer, VMware) in order to support virtualization for the environment.
- Open source: OpenStack code can be modified according to the requirements and OpenStack maturation for new releases and new features can be developed easily since the code is openly distributed.

A. OPERATION OF OpenStack NETWORK

This section provides an overview of virtual network components of the OpenStack Neutron service which handle network traffic among virtual machines (VMs). These components collaborate to provide simple and friendly steps for VMs network communication. Multiple VMs can be connected with each other via a private network to establish Virtual Cluster(VC) which can be used as a distributed memory

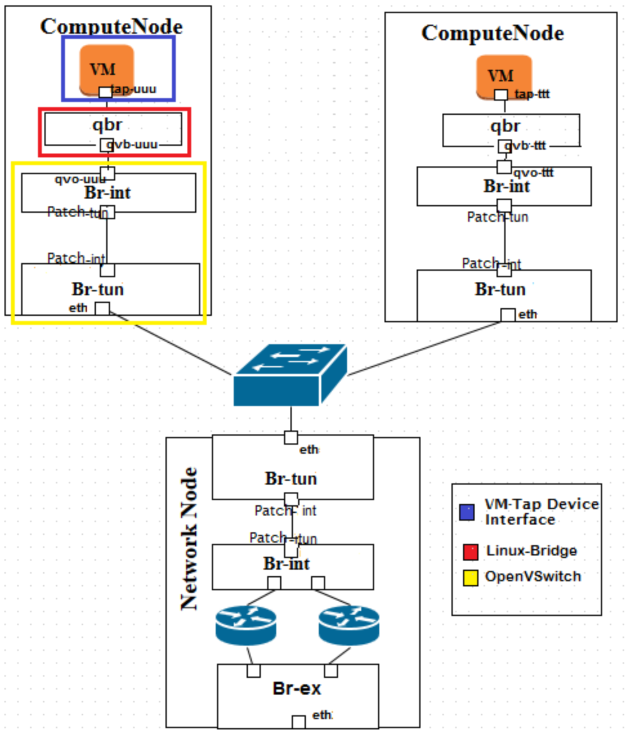


FIGURE 13. Logical architecture of compute and network nodes' bridges and interfaces [46].

platform. When the VM is created, the Neutron service can create a private network and associate this private network with a subnet. During VM booting, it connects to the private network [25]. Two IP addresses, private IP and floating IP are assigned to each VM. A DHCP server offers a private IP address from the private network of OpenStack to the VM interface, and the VM guest operating system knows this IP. Thus, VMs interact with each other using these private IPs through virtual switches and bridges configured on each compute node. Floating IP addresses are features assigned to VMs by the Neutron service to access VMs from the Openstack external network. Virtual Linux networking components are shown in Table 2 since VM traffic has to go through them before reaching its VM destination. For more understanding of VM network communication on OpenStack, it is required to investigate the internal logical architecture of compute and network nodes with their bridges and interfaces as shown in Figure 13. The network components inside compute node can be divided into three parts as follows [45]:

- Tap Device: The Virtual Network Interface Card (VNIC) of a VM that accepts Ethernet frames. Tap interfaces are special software entities which instructs the Linux bridge to forward Ethernet frames. In other words, the virtual machines connected to tap interfaces will be able to receive raw Ethernet frames. IP-table rules are used on this tap device to implement the security groups (such as, SSH and Pinging protocols) associated with the VM.

TABLE 2. Definition of Linux network components.

Linux network Component	Definition
TAP device	A tap device is a software-only interface that a userspace program can attach to itself and send/receive packets to it. TAP devices are the way that KVM/QEMU implement Virtual Network Interface Card (VNIC) attached to the VMs.
veth pair	A veth pair is a pair of virtual NIC cards connected via a virtual cable. If a packet is sent on one of them, it will come out of the other one and vice versa. Veth are usually used to connect two entities.
Linux bridge	Linux bridge is a virtual switch implemented in Linux.
Openvswitch	Openvswitch is a more complicated virtual switch implemented in Linux. It allows openflow rules to be applied to traffic at layer 2 such that decisions are made on MAC addresses, VLAN ID of the traffic flow. Openvswitch provides native support for VXLAN tunnels.
Patch interfaces in openvswitch	A special kind of interface that is used to connect two openvswitch switches.

- Linux-Bridge: OpenVSwitch (OVS) cannot directly attach a tap device where IP-table rules are applied; the bridge device offers a route to the kernel for filtering.
- OpenVSwitch (OVS): OVS is an interaction component to connect virtual ports with other network components such as Linux bridges and underlying interfaces. OVS consists of two bridges, integration bridge (Br-int) and tunnel bridge (Br-tun). The integration bridge enables communications between VMs on the same compute host, whereas the tunnel bridge connect VMs to an external network.

When VMs are deployed on the same compute host, the network traffic is handled using Br-int since each network has a VLAN-ID which enables traffic isolation. On the other hand, deploying VMs on different compute hosts needs another technique and network traffic flows between compute hosts via an overlay network by converting the network VLAN-ID to tunnel using VXLAN or GRE key depending on the configuration choice.

Building Virtual Cluster (VC) on OpenStack which consists of two VMs at least, needs awareness of network traffic flow for all cases of VMs deployment since performance of VMs communication depends on the hosting of VMs on compute hosts (the same or different compute host(s)) as well as IP of VMs (the same or different network(s)). Thus, deploying two VMs on Opensatck can fall into one of four cases depending on VM hosting and its IP [46].

i. Case1, the same compute host and the same network.

ii. Case2, the same compute host and different networks as shown in Fig. 14.

iii. Case3, different compute hosts and the same network.

iv. Case4, different compute hosts and different networks as shown in Fig. 15.

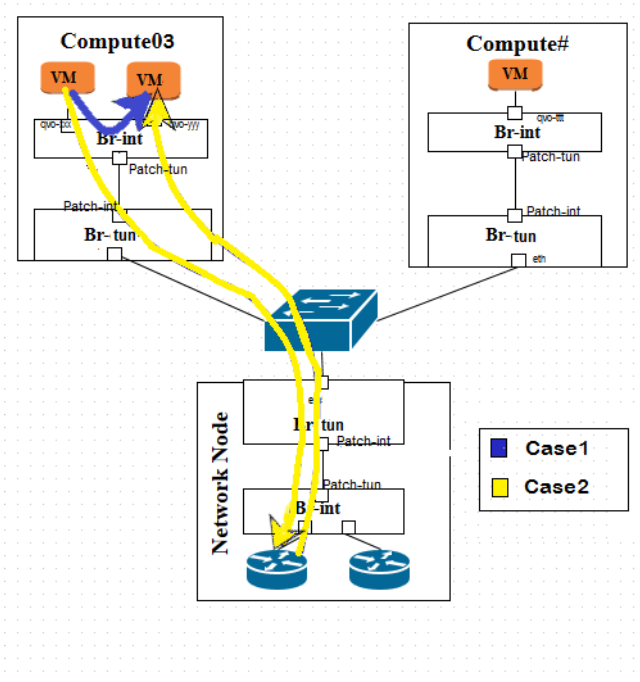


FIGURE 14. Traffic flow for Case1 and Case2 [46].

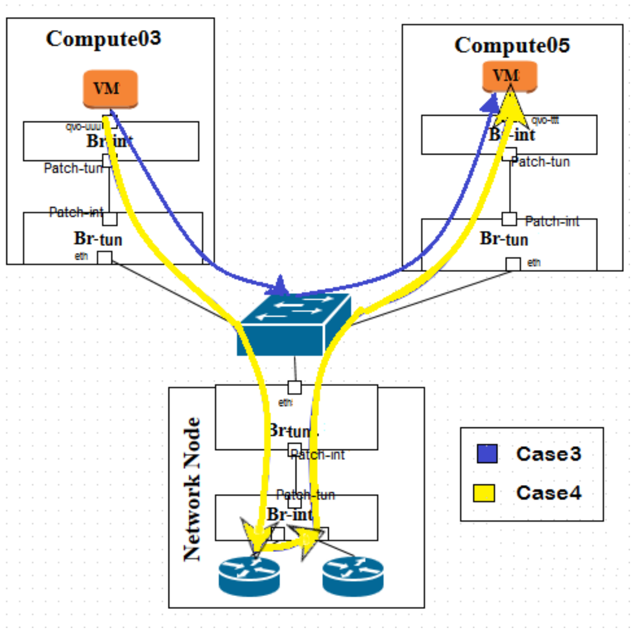


FIGURE 15. Traffic flow for Case3 and Case4 [46].

VII. EXPERIMENTAL SETUP AND PLATFORMS

Since our implementations are based on distributed and shared memory platforms, the following subsections present the details of the two experimental platforms.

A. DISTRIBUTED MEMORY PLATFORM

A private OpenStack Cloud is used as an IaaS platform. Our OpenStack environment consists of one controller node and five OpenStack compute-nodes with hardware specifications

TABLE 3. Hardware specifications for private OpenStack Cloud.

	1X Controller Node	5X Compute Nodes
System Model	Dell PowerEdge R720	Dell PowerEdge M620 Blade Server
Processors/node	2X Intel(R) Xeon(R) CPU E5-2640 (2.4 GHz,25 MB, 10 cores, 20 threads)	2X Intel (R) Xeon (R) CPU E5-2670 (2.6 GHz, 20MB cache, 8 cores, 16 threads)
Memory/node	128GB DDR3 with NUMA architecture	128GB DDR3 with NUMA architecture
NICs /node	2x 10 Gbps	1x 10 Gbps

TABLE 4. Deployment parameters for ERI-OpenStack Cloud.

Distribution	OpenStack Liberty 1.7.3
Operating System	Centos7.0
Hypervisor	KVM
Storage Backend	ISCSI (500TB)
Network (Nova FlatDHCP)	Network #1: Public, Storage, VM Network #2: Admin, Management

as shown in Table 3. The used release of OpenStack is Liberty version [47] and the hypervisor layer is KVM. Table 4 shows the other deployment parameters of the Electronics Research Institute (ERI) HPC Center of Excellence environment.

Building efficient virtual cluster platform needs understanding of the traffic flow of virtual network so our study is carried out in two phases. The first phase of the experiments is to investigate traffic flow patterns among the VMs and the second is to build Virtual Cluster (VC) to use all capabilities of our OpenStack infrastructure. The investigation of network traffic flow is conducted by deploying virtual machines (VMs) on the same and/or different compute nodes. All VMs use the Centos7.0 operating system as their base OS. For the first phase experiments, we test the four cases of VMs deployment to demonstrate the behavior of our parallel implementation on the OpenStack virtual platform.

Table 5 recaps the locus of VMs on compute host and their IP addresses for the different cases as mentioned in VI-A. VM-1 and VM-2 are deployed on the same compute host and are assigned to the same network and this is considered as Case1. VM-3 and VM-4 are assigned to different networks but they occupy the same compute host which we refer to as Case2. VM-5 and VM-6 are located on different compute hosts but have the same network, which is Case3. Finally VM-7 and VM-8 reside on different compute hosts and different networks and this is Case4.

For experiments of the second phase, we create a cluster with five VMs on the same tenant. Each VM has 32 VCPUs, 8GB RAM and 500GB disk with Centos7.0 operating system. The topology of VC network is shown in Figure 16. The five VMs are connected through the same virtual network which is connected to the public internet via virtual tenant-router.

TABLE 5. Experimental setup for four deployment scenarios.

Name of scenario	Name of VM	Private IP address	Compute node name
Case1	VM1 and VM2	192.168.1.2 and 192.168.1.4	Compute03
Case2	VM3 and VM4	10.0.3.8 and 172.16.0.2	Compute03
Case3	VM5 and VM6	10.0.3.8 and 10.0.3.9	Compute03 and Compute05
Case4	VM7 and VM8	10.0.3.2 and 172.16.0.2	Compute03 and Compute05

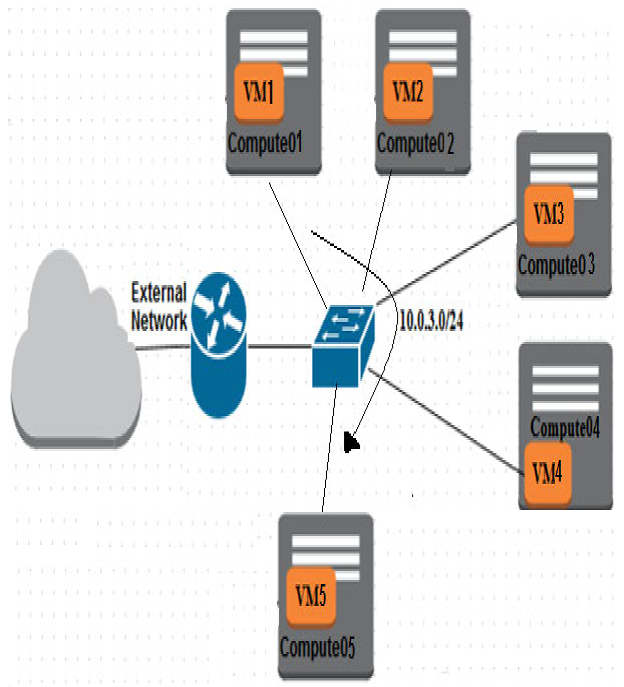


FIGURE 16. Virtual cluster network topology.

B. SHARED MEMORY PLATFORM

We test our multi-threaded implementations on one of the VM with as many as 32 threads. For the two platforms (Distributed/ shared memory), we use GCC compiler version 5.2 [48] and MPICC compiler version mpich-3.2.0 which is freely available on all Linux platforms [49].

The number of processing elements is not the only parameter to test our implementations but we used three different grid block sizes which are comparable in size to those in several researches [50]–[53], to test the performance of our implementations on Intel Xeon CPU:

- Small Grid Size;for grid sizes $10 \times 10 \times 8$ and $20 \times 20 \times 20$.
- Medium Grid Size; for grid sizes $40 \times 40 \times 40$ and $200 \times 200 \times 6$.

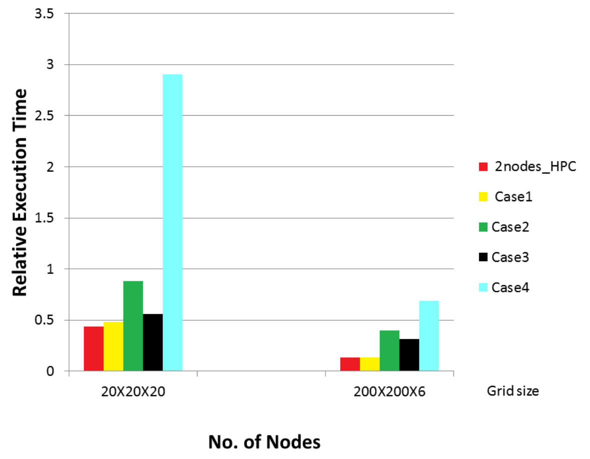


FIGURE 17. HPC [8] versus virtual cluster scenarios relative execution time.

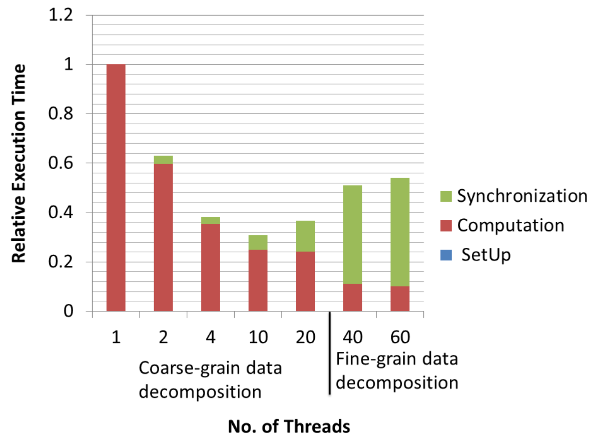
- Large Grid Size; for grid sizes $10 \times 80 \times 8$, $20 \times 400 \times 20$ and $40 \times 1600 \times 40$.

VIII. RESULTS AND ANALYSIS

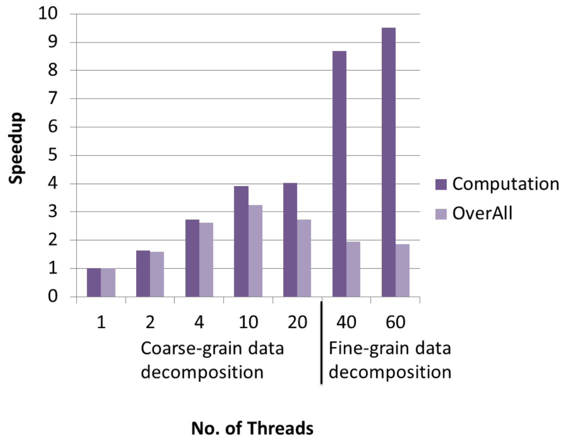
Parallel implementations of the oil reservoir data visualization tool are analyzed by evaluating the execution times and speedups with respect to the serial implementation. The setup time is negligible for all parallel implementations as the setup time is for just a few straightforward equations to determine the data border for each processing element (node/ thread). The computation time is the time for calculating the lighting model and transformation of grid vertices. The communication time for MPI coarse grain data decomposition is the time for Model_View matrix broadcasting and time for master node data gathering. In contrast, the MPI fine grain data decomposition communication time is the sum of the time spent broadcasting the Model_View matrix, creating sub-communicators, collecting the data of the same plane and collecting the data of the whole grid at the master node. Analysis of small grid data size is performed by studying the behavior of $20 \times 20 \times 20$ grid size, while the $200 \times 200 \times 6$ grid size is used for the analysis of medium grid data sizes.

A. DIFFERENT MPI DEPLOYMENT SCENARIOS ON CLOUD

Deep investigation is required to demonstrate the major factors which can affect the Virtual Cluster throughput. Figure 17 shows the relative execution time of coarse grain MPI implementation using four cases of VM deployment and two nodes of HPC with respect to serial running on virtual machine. The relative execution time shrinks by 4% only when running on two nodes of HPC compared to running on Case1 of VMs deployment since the two VMs are on the same compute host and on the same network. The traffic does not flow through network node or physical switch between the compute nodes. For the case2, 44% is the increase in execution time compared to HPC since



(a) Relative execution time for 20x20x20 grid size

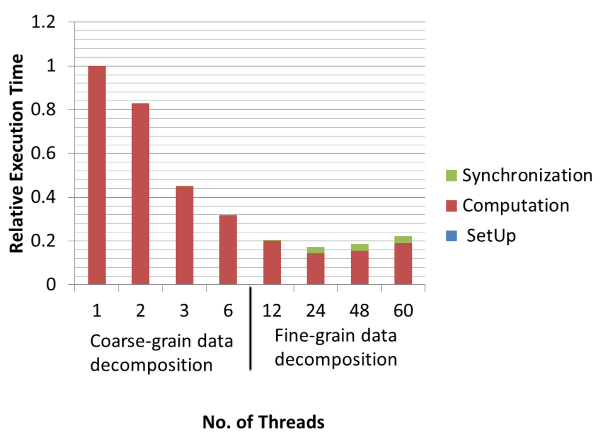


(b) Speedup for 20x20x20 grid size

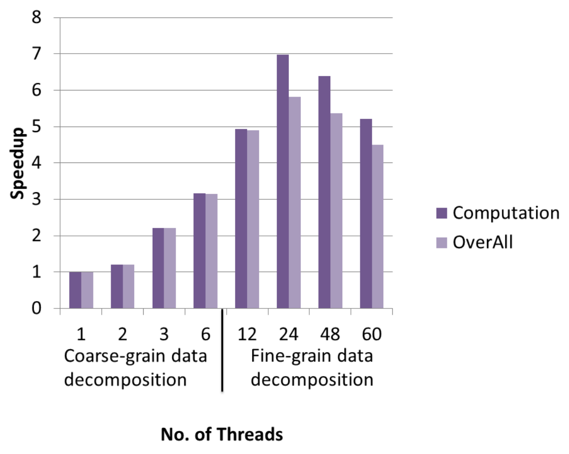
FIGURE 18. Small grid size performance on shared memory model.

the two VMs are on the same compute node but on different networks. The traffic has to pass through network node bridges through the physical switch. The increase in execution time shrinks to 12% for case3 when tasks are running on different compute nodes but on the same network. Hence, the traffic flows through the virtual switches of the same host node. On the other hand, case4 shoots the execution time as high as 25X compared to HPC since the two VMs are on different compute nodes and different networks.

According to the aforementioned investigations for internal traffic flow between VMs through compute and network nodes, it can be confirmed that when Virtual machines (VMs) with private IPs are located on the same network, then they only use switches in order to communicate with each other independently of their location on the compute nodes. Thus Case1 and Case3 consume smaller times than Case2 and Case4 as shown in Figure 17. The results show that the locus of machines in terms of compute node and network address matters for the performance. Thus, when VMs are on the same compute node and the same network, they perform

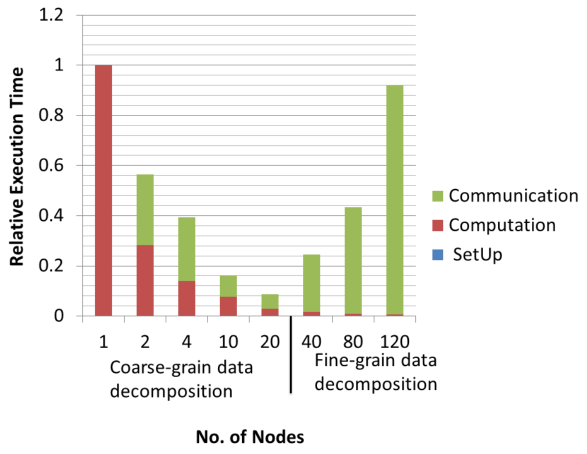


(a) Relative execution time for 200x200x6 grid size

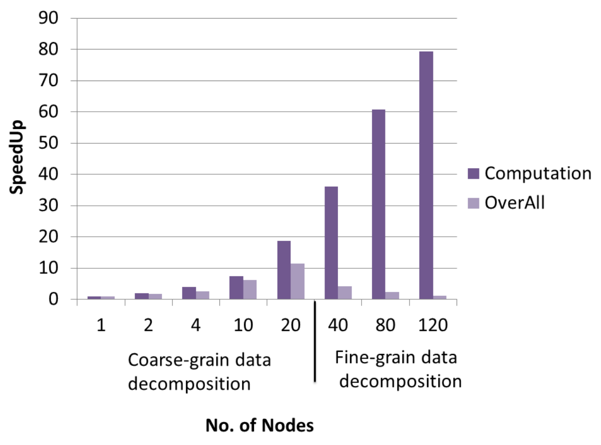


(b) Speedup for 200x200x6 grid size

FIGURE 19. **Medium grid size performance on shared memory model.**



(a) Relative execution time for 20x20x20 grid size



(b) Speedup for 20x20x20 grid size

FIGURE 20. **Small grid size performance on distributed memory model.**

better than other scenarios. This is because the transmission path is shorter (in term of delay) than the other scenarios. In the next set of experiments we will adopt the best performing configuration for the Cloud out of the four scenarios tested.

B. SHARED MEMORY (MULTI-THREADING) EXPERIMENTS

Our results for shared memory experiments are performed on one VM hosted on one compute node. For medium and small grid data sizes, computation speedup rises almost sub-linearly with more handling threads for the data grid due to resource sharing (particularly caches and cores) among operating threads. as shown in Figs 18(a) and 19(a).

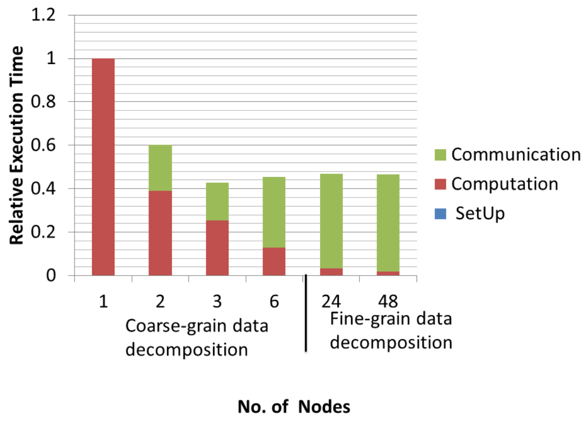
For *small grid data sizes*, the synchronization time is nearly fixed and represents about 0.2% of the entire time provided that the operating threads number is equal to or less than four threads running. When the number of threads is greater than four, the synchronization time is gradually increased since the work partitioning among so many slave threads provides little work for each thread. In this case the start and finish thread overhead exceeds the speedup advantages as shown in Figure 18(b). Due to the overhead synchronization of threads, the general speedup is 90% smaller than the computation speedup.

For *medium grid data sizes*, the synchronization time is almost steady and about 0.1% of the overall time provided that the operating threads number is equal to or less than twelve running threads. For thread numbers higher than twelve threads, the synchronization time is slightly increased as each slave thread has enough work to compensate for the start and finish threads overhead as shown in Figure 19(b).

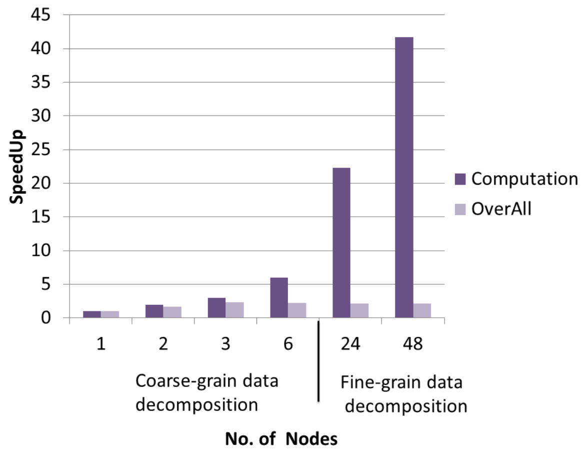
Even though the high efficiency scaling of the performance for shared memory implementation, the restricted integration of more cores per node and the restricted size of used data set are the primary drawback of this parallel paradigm.

C. DISTRIBUTED MEMORY (MPI) EXPERIMENTS

For *small grid data sizes* as shown in Fig. 20(a), the communication time of coarse grain data decomposition reduces as the number of operating nodes increases due to the small data bulk gathered from each node as well as the MPI-task numbers. Despite the fact that the communication time for



(a) Relative execution time for 200x200x6 grid size



(b) Speedup for 200x200x6 grid size

FIGURE 21. **Medium grid size performance on distributed memory model.**

fine grain data decomposition rises as the number of operating tasks increases (because more communication is required, especially the time to create sub_communicators, which accounts for about 70% of the total communication time). The decrease in computation time, however, falls short of the heavy communication overhead between slave nodes and master node.

For *medium grid data sizes* as shown in Figure 21(a), coarse grain data decomposition communication time is almost fixed across a variety of running nodes as the data volume gathered from each node is significant. From the results, we can observe that the implementation of distributed memory can only enhance the execution time with large data footstep for the coarse grain data decomposition.

For small and medium grid data sizes, the speedup of computation grows linearly as the number of MPI tasks increases to reach the number of planes of the data grid (Maximum coarse grain data decomposition) as shown in Figures 20(b) and 21(b) respectively. For fine grain (2D data decomposition), the speedup of computation rises sub-linearly. Increasing the number of MPI-tasks on the same

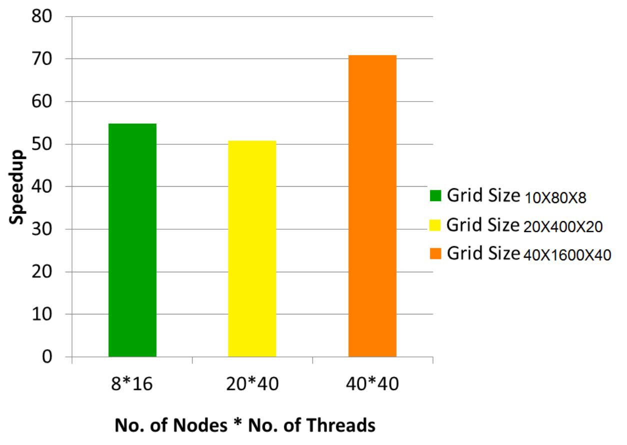


FIGURE 22. **Speedup on hybrid memory model.**

physical node while each node works on a subset of data causes the memory-overhead to outfit more tasks than distributing the computational job among them. Even though the computation time scales gradually with incrementing the MPI-tasks, the overhead noticeably affects the performance scaling. Furthermore, the overhead communication for coarse grain decomposition is restricted due to infrequent communication of nodes. Accordingly, the aggregate speedup is satisfactory when using the coarse grain (plane-level/ 1D) decomposition. For twenty MPI-tasks, where one complete plane is processed by each node, more than 10X speedup is accomplished as shown in Figure 20(b).

D. HYBRID MEMORY EXPERIMENTS

Hybrid development is the best option for large data sets. Although, the primary advantage of using MPI is the scalability of running node numbers, only coarse grain parallel implementation can be used effectively for this parallel paradigm. Whereas, MPI fine grain data decomposition technique is not efficient because the communication overhead is very high with respect to computation speedup benefit. On the other hand, the fine grain data decomposition technique can be used effectively in the shared memory paradigm.

In this study, the combination of both paradigms in one hybrid approach is a significant contribution that customizes the parallelization to the need of the data visualization tool and effectively exploits both paradigm advantages concurrently. On average, the execution time of hybrid parallel paradigms for big grid data sizes can accelerate 59X over serial execution using single Power Processor Unit (PPU) of IBM Cell BE [54] as shown in Figure 22.

E. COMPARISON OF REAL AND VIRTUAL PLATFORMS

In this section, we evaluate the performance of the virtual platform on OpenStack private Cloud, as the most popular open source Cloud platform, by providing a comparison with respect to traditional real platform which is proposed in [8]. The authors in [8] provide a good comparison with CUDA

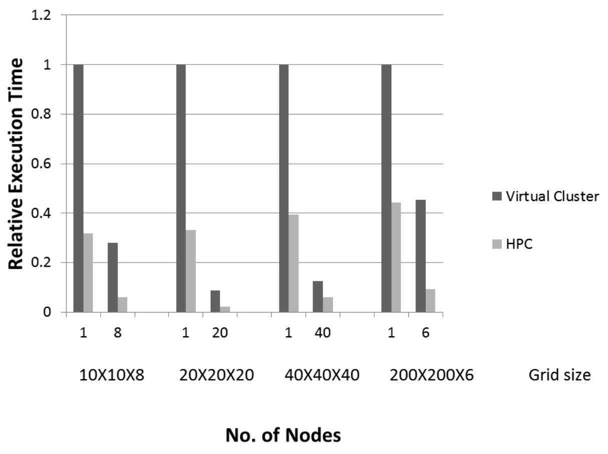


FIGURE 23. HPC [8] versus VC relative execution time using distributed memory implementations.

implementation in [34]. CUDA falls short in scalability for large data-sets hence, we limit our comparison to the performance of a virtual cluster of multiple VMs running MPI implementation with respect to the HPC system described in [8] to better understand the effect of network bandwidth and latency.

By using MPI implementation with coarse grain data decomposition since it gives the best throughput according to the previous analysis, the execution time shrinks by 70% for HPC compared to Virtual Cluster for all grid sizes. This overhead of Virtual Cluster is due to mainly two reasons, first multiple compute node deployment for VMs since each VM is located on different physical compute node. The second reason is due to network virtualization as shown in Figure 23.

IX. CONCLUSION AND FUTURE WORK

In this article, we propose new parallel techniques for implementing the 3-D oil reservoir data visualization tool using Multi-threading, MPI, and hybrid (MPI/ Multi-threading) implementations. The Distributed Memory (MPI) approach is more suitable for coarse grain data decomposition to avoid frequent and small data transform among the tasks. However, it has the ability to scale well with the grid scaling. On the other hand, the Shared Memory approach achieves high performance for fine grain data decomposition while its hardware systems scalability is limited due to its resource sharing nature.

Hybrid shared/distributed memory parallelism that utilizes coarse grain and fine grain data decomposition simultaneously is the best customed approach that better fits the 3-D grid data set nature while utilizing the heterogeneous systems composed of a mix of parallel processing units. Deploying Virtual Cluster for running real parallel applications needs awareness with respect to the real infrastructure of the cloud to reduce network communication among VMs and overcome the causes of this overhead. Running our proposed hybrid parallelization technique on Virtual Cluster can achieve speedup reaching to 70X over the single processing implementation.

As future work for this study, we plan to pursue a MPI-CUDA Implementation to enhance the performance of the Oil Reservoir Data Visualization tool using a multi-GPU Cluster.

REFERENCES

[1] R. Jothikumar, K. Subramaniam, S. G. Shanmugam, and S. Susi, "Electronic voting system with cloud based high performance computing," *J. Comput. Theor. Nanosci.*, vol. 16, no. 2, pp. 768–772, Feb. 2019.

[2] P. M. Mell and T. Grance, "The NIST definition of cloud computing," Nat. Inst. Standards Technol., Gaithersburg, MD, USA, Tech. Rep. Special Publication 800-145, 2011.

[3] M. Parashar, M. AbdelBaky, I. Rodero, and A. Devarakonda, "Cloud paradigms and practices for computational and data-enabled science and engineering," *Comput. Sci. Eng.*, vol. 15, no. 4, pp. 10–18, Jul. 2013.

[4] M. R. Abid, "HPC (High-performance the Computing) for big data on cloud: Opportunities and challenges," *Int. J. Comput. Theory Eng.*, vol. 8, no. 5, pp. 423–428, Oct. 2016.

[5] A. A. El-Moursy, W. S. Afifi, F. N. Sibai, and S. M. Nassar, "Parallel PPI prediction performance study on HPC platforms," *J. Circuits, Syst. Comput.*, vol. 24, no. 05, Jun. 2015, Art. no. 1550074.

[6] R. Nane, V.-M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels, "A survey and evaluation of FPGA high-level synthesis tools," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 35, no. 10, pp. 1591–1604, Oct. 2016.

[7] S. Prema, R. Jehadeesan, and B. K. Panigrahi, "Identifying pitfalls in automatic parallelization of NAS parallel benchmarks," in *Proc. Nat. Conf. Parallel Comput. Technol.*, Feb. 2017, pp. 1–6.

[8] H. Khaled, A. A. El-Moursy, S. M. Nassar, M. Taher, and F. N. Sibai, "Parallel study of 3-D oil reservoir data visualization tool using hybrid distributed/shared-memory models," in *Proc. IEEE 16th Intl Conf Dependable, Autonomic Secure Comput.*, Aug. 2018, pp. 1016–1021.

[9] A. Saad, A. El-Mahdy, and H. El-Shishiny, "Performance modeling of MPI-based applications on cloud multicore servers," in *Proc. Rapid Simulation Perform. Eval.*, 2019, p. 5.

[10] (SPEC). (2010). *Spec MPI 2007 Benchmark Suite Documentation*. [Online]. Available: https://www.spec.org/auto/mpi2007/Docs/

[11] (2019). *Nasa Advanced Supercomputing Division*. [Online]. Available: https://www.nas.nasa.gov/publications/npb.html/

[12] S. Jamaliannasrabadi, "High performance computing as a service in the cloud using software-defined networking," Ph.D. dissertation, Dept. Comput. Sci., Bowling Green State Univ., Green, OH, USA, 2015.

[13] S. Jamalian and H. Rajaei, "ASETS: A SDN empowered task scheduling system for HPCaaS on the cloud," in *Proc. IEEE Int. Conf. Cloud Eng.*, Mar. 2015, pp. 329–334.

[14] S. Jamalian and H. Rajaei, "Data-intensive HPC tasks scheduling with SDN to enable HPC-as-a-Service," in *Proc. IEEE 8th Int. Conf. Cloud Comput.*, Jun. 2015, pp. 596–603.

[15] S. R. Basnet, R. S. Chaulagain, S. Pandey, and S. Shakya, "Distributed high performance computing in OpenStack cloud over SDN infrastructure," in *Proc. IEEE Int. Conf. Smart Cloud (SmartCloud)*, Nov. 2017, pp. 144–148.

[16] H. Yuan, J. Bi, M. Zhou, and K. Sedraoui, "WARM: Workload-aware multi-application task scheduling for revenue maximization in SDN-based cloud data center," *IEEE Access*, vol. 6, pp. 645–657, 2018.

[17] S. Date, "SDN-accelerated hpc infrastructure for scientific research," *Int. J. Inf. Technol.*, vol. 22, no. 1, 2016.

[18] S. Date, H. Abe, D. Khureltulga, K. Takahashi, Y. Kido, Y. Watashiba, P. U-Chupala, K. Ichikawa, H. Yamanaka, E. Kawai, and S. Shimojo, "An empirical study of SDN-accelerated HPC infrastructure for scientific research," in *Proc. Int. Conf. Cloud Comput. Res. Innov. (ICCCRI)*, Oct. 2015, pp. 89–96.

[19] S. Kehrer and W. Blochinger, "Migrating parallel applications to the cloud: Assessing cloud readiness based on parallel design decisions," *SICS Softw.-Intensive Cyber-Phys. Syst.*, vol. 34, nos. 2–3, pp. 73–84, Feb. 2019.

[20] C. Kotas, T. Naughton, and N. Imam, "A comparison of amazon Web services and microsoft azure cloud platforms for high performance computing," in *Proc. IEEE Int. Conf. Consum. Electron. (ICCE)*, Jan. 2018, pp. 1–4.

[21] S. Ali, "Virtualization with KVM," in *Practical Linux Infrastructure*. Berkeley, CA, USA: Apress, 2015, pp. 53–80.

[22] F. Gomez-Folgar, A. J. Garcia-Loureiro, T. F. Pena, J. I. Zablah, and N. Seoane, "Study of the KVM CPU performance of open-source cloud management platforms," in *Proc. 15th IEEE/ACM Int. Symp. Cluster, Cloud Grid Comput.*, May 2015, pp. 1225–1228.

[23] T. A. S. Foundation. *Apache Cloudstack*. Accessed: Nov. 2020. [Online]. Available: https://cloudstack.apache.org/

[24] A. Systems. *Eucalyptus*. Accessed: Nov. 2020. [Online]. Available: https://www.eucalyptus.cloud/

[25] *Openstack*. Accessed: Nov. 2020. [Online]. Available: https://www.openstack.org/.

[26] T. Ertekin, J. H. Abou-Kassen, and G. R. King, *Basic Pactical Reservoir Simulations*. Richardson, TX, USA: SPE Textbook Series, 2001.

[27] J. de Vries. (2018). *Learn Opengl Transformation*. [Online]. Available: https://learnopengl.com/Getting-started/Transformations

[28] J. de Vries. (2018). *Learn Opengl Camera*. [Online]. Available: https://learnopengl.com/Getting-started/Camera

[29] J. de Vries. (2018). *Learn Opengl Basic-Lighting*. [Online]. Available: https://learnopengl.com/Lighting/Basic-Lighting

[30] F. Dunn and I. Parberry, *3D Math Primer for Graphics and Game Development*. Boca Raton, FL, USA: CRC Press, 2011.

[31] O. Guide. (2016). *Opengl Programming Guide Chapter 5 Lighting*. [Online]. Available: http://www.glprogramming.com/red/chapter05.html

[32] T. Mcreynolds and D. Blythe, "Using OpenGL extensions," in *Proc. Adv. Graph. Program. Using*, 2005, pp. 593–604.

[33] N. Baek, "Design of OpenGL SC 2.0 shader language features," in *Proc. Mobile Wireless Technol.*, 2017, pp. 248–252.

[34] F. N. Siba, S. Mohammad, H. K. Kidwai, B. Qamar, and F. Awwad, "Parallel implementation and performance analysis of a 3D oil reservoir data visualization tool on the cell broadband engine and CUDA GPU," in *Proc. IEEE 14th Int. Conf. High Perform. Comput. Commun. IEEE 9th Int. Conf. Embedded Softw. Syst.*, Jun. 2012, pp. 970–975.

[35] S. L. Graham, P. B. Kessler, and M. K. McKusick, "Gprof: A call graph execution profiler," *ACM SIGPLAN Notices*, vol. 39, no. 4, p. 49, Apr. 2004.

[36] K. E. Parsopoulos, "Parallel cooperative micro-particle swarm optimization: A master–slave model," *Appl. Soft Comput.*, vol. 12, no. 11, pp. 3552–3579, Nov. 2012.

[37] H. Eldeeb, H. Elsadek, M. Dessokey, H. Abdallah, and N. Bagherzadeh, "High performance parallel computing for FDTD numerical technique in electromagnetic calculations for SAR distribution inside human head," in *Proc. 14th WSEAS Int. Conf. Comput.*, 2010, pp. 23–25.

[38] J. K. Bennett, J. B. Carter, and W. Zwaenepoel, "Munin: Distributed shared memory based on type-specific memory coherence," in *Proc. ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, vol. 25, 1990, pp. 168–176.

[39] Y.-Y. Jiao, Q. Zhao, L. Wang, G.-H. Huang, and F. Tan, "A hybrid MPI/OpenMP parallel computing model for spherical discontinuous deformation analysis," *Comput. Geotechn.*, vol. 106, pp. 217–227, Feb. 2019.

[40] P. Ouro, B. Fraga, U. Lopez-Novoa, and T. Stoesser, "Scalability of an eulerian-lagrangian large-eddy simulation solver with hybrid MPI/OpenMP parallelisation," *Comput. Fluids*, vol. 179, pp. 123–136, Jan. 2019.

[41] B. Yan and R. A. Regueiro, "Comparison between pure MPI and hybrid MPI-OpenMP parallelism for discrete element method (DEM) of ellipsoidal and poly-ellipsoidal particles," *Comput. Part. Mech.*, vol. 6, no. 2, pp. 271–295, Apr. 2019.

[42] S. Yadav, "Comparative study on open source software for cloud computing platform: Eucalyptus, openstack and opennebula," *Int. J. Eng. Sci.*, vol. 3, no. 10, pp. 51–54, 2013.

[43] R. Nasim and A. J. Kassler, "Deploying OpenStack: Virtual infrastructure or dedicated hardware," in *Proc. IEEE 38th Int. Comput. Softw. Appl. Conf. Workshops*, Jul. 2014, pp. 84–89.

[44] S. Lima and A. Rocha, "A view of OpenStack: Toward an open-source solution for cloud," in *Proc. Adv. Intell. Syst. Comput.*, 2017, pp. 481–491.

[45] *Scenario: Provider Networks With Open Vswitch*. Accessed: Nov. 2020. [Online]. Available: https://docs.openstack.org/kilo/networking-guide/scenario_provider_ovs.html

[46] M. B. Gebreyohannes, "Network performance study on openstack cloud computing," M.S. thesis, Dept. Inform., Univ. Oslo, Oslo, Norway, 2014.

[47] *Liberty the Twelfth Release of Openstack*. Accessed: Nov. 2020. [Online]. Available: https://www.openstack.org/software/liberty/

[48] GCC. (2018). *Gnu Compiler Collection*. [Online]. Available: https://gcc.gnu.org/

[49] MPICH. (1992). *High-Performance Portable MPI*. [Online]. Available: http://www.mpich.org/

[50] C. Wang, P. Winterfeld, B. Johnston, and Y.-S. Wu, "An embedded 3D fracture modeling approach for simulating fracture-dominated fluid flow and heat transfer in geothermal reservoirs," *Geothermics*, vol. 86, Jul. 2020, Art. no. 101831.

[51] Z. L. Jin, Y. Liu, and L. J. Durlofsky, "Deep-learning-based surrogate model for reservoir simulation with time-varying well controls," *J. Petroleum Sci. Eng.*, vol. 192, Sep. 2020, Art. no. 107273.

[52] L. Yuyang, L. Shiqi, and P. Mao, "Finite element simulation of oil and gas reservoir *in situ* stress based on a 3D corner-point grid model," *Math. Problems Eng.*, vol. 2020, pp. 1–14, Feb. 2020.

[53] R. P. Batycky, M. R. Thiele, K. Coats, A. Grindheim, and D. Ponting, "Reservoir simulation," *Trans., AIME*, vol. 213, p. 28, Dec. 1958.

[54] S. Koranne, *Practical Computing on the Cell Broadband Engine*. New York, NY, USA: Springer, 2009.

ALI A. EL-MOURSY (Senior Member, IEEE) received the Ph.D. degree in the area of high-performance computer architecture from the University of Rochester, Rochester, NY, USA, in 2005. He has worked with the Software Solution Group, Intel Corporation, CA, USA, till early 2007. In 2007, he joined the Electronics Research Institute, Giza, Egypt. He has also participated as a Visitor Research Scientist with the IBM Cairo Technology Development Center, Egypt, from February 2007 to January 2010. In September 2010, he joined the ECE Department, University of Sharjah, Sharjah, United Arab Emirates, as an Assistant Professor. In January 2017, he has been promoted to the Associate Professor Rank. His research interests include high-performance computer architecture, multi-core multi-threaded mirco-architecture, power-aware micro-architecture, simulation and modeling of architecture performance and power, workload profiling and characterization, cell programming, high-performance computing, parallel computing, and cloud computing.

FADI N. SIBAI received the B.S. degree from The University of Texas at Austin and the M.S. and Ph.D. degrees from Texas A&M University, all in electrical (computer) engineering. He joined Prince Mohammad Bin Fahd University, in 2019, as the Dean of the College of Computer Engineering and Science. He taught at several Universities in USA and Middle East. From 2006 to 2011, he directed the Computer Systems Design Program and the IBM Competence Center, United Arab Emirates University, where he received the IBM's Highest Research Award. From 1990 to 1996, he was an Assistant Professor of electrical engineering with the University of Akron. He also has extensive industrial experience with Saudi Aramco and Intel Corporation. He has authored or coauthored over 200 publications and technical reports and served on the Organizing or Program Committees of over 20 International Conferences. He is a member of PMI, (ISC)^2, and Eta Kappa Nu. He holds PMP, CISSP, CCNA, and CQRM certifications.

HANAN KHALED received the B.Sc. degree (Hons.) in electronics and communication engineering from Al Azhar University, Cairo, Egypt, in 2012, the Diploma degree in cloud application from the Information Technology Institute (ITI), in 2013, and the M.Sc. degree in computer and systems engineering from Ain Shams University, in 2019. From 2014 to 2019, she was a Research Assistant with the Cloud Computing Laboratory, Electronics Research Institute (ERI), Giza, Egypt, where she is currently an Associate Researcher. Her research interests include performance optimization, high-performance computing, and cloud computing.

SALWA M. NASSAR (Member, IEEE) received the Ph.D. degree in the field of parallelism in programming languages from the Electronics and Communication Department, Faculty of Engineering, Cairo University, in 1984. Her professional experience started, in 1974, by being a Research Assistant with the Computer and System Department, Electronics Research Institute. She became an Instructor and an Assistant Professor with the Computer and System Department, Electronics Research Institute. She taught in the American University in Cairo (AUC), from 1987 to 1997. In 1991, she became an Associate Professor in 1991 and a Full Professor in 1996. She was the Head of EU information Point InP. She is an Ex-ERI President and the PI of the Cloud Center of Excellence and the Head of the HPCloud Group, ERI. She has 50 publications. Her research interests include parallel processing, parallel logic languages, modeling and simulation of parallel programs, distributed systems, computer networks, parallel applications, parallel virtual machine, grid computing, cluster computing, and cloud computing. She has led a number of European and USA funded projects. She is a member of the IEEE Computer Society and the Information Technology Academia Collaboration (ITAC) Steering Committee, and a Juror in the Egypt. She received the WSIS-Award, organized by the MCIT, Egypt, in 2006.

MOHAMED TAHER received the B.Sc. (Hons.) and M.Sc. degrees in computer engineering from Ain Shams University, Cairo, Egypt, in 1996 and 2001, respectively, and the Ph.D. degree in electrical and computer engineering from George Washington University, Washington, DC, USA, in 2006. He is currently a Professor with the Department of Computer and Systems Engineering, Ain Shams University. His research interests include high-performance computing, reconfigurable computing, embedded systems, and computer architecture.