Contemporary Engineering Sciences, Vol. 8, 2015, no. 15, 677 - 698
HIKARI Ltd, www.m-hikari.com
http://dx.doi.org/10.12988/ces.2015.55153

A Parallel Computational Model Based on Mobile Agents for High Performance Computing

Mohamed Youssfi, Omar Bouattane

Lab. SSDIA, ENSET, University Hassan II of Casablanca, Morocco

Mohammed Ouadi Bensalah

Faculty of Science, University Mohammed V of Rabat, Morocco

Abstract

The goal of this paper is to present a new massively parallel virtual machine model, designed for parallel and high performance computing on distributed systems. The proposed model allows us to build a polymorphic grid computing assigned to solve fine grained parallel problems over different well known, SIMD (Single Instruction Multiple Data), SPMD (Single Program Multiple Data), MIMD (Multiple Instruction Multiple Data)) and MPMD (Multiple Program Multiple Data) machine structures. This model is built using dynamic distributed Virtual Processors Units (VPU). Each VPU corresponds to a mobile agent deployed in a physical processing unit. Each physical processing unit is assigned to a node of the considered distributed system. These units may be heterogeneous machines, they are characterized by their physical performance indices. The proposed model is able to include different computing unit technologies, such as supercomputers, simple desktop or smart phones etc. The VPUs are designed to communicate with each other asynchronously by exchanging, in local or remote manner, the ACL messages (Agent Communication Language) containing data, instructions or any task to be performed. In this model a special agent is designed to represent the host of the parallel virtual machine and to manage the following activities: the life cycle of the VPUs, the load balancing control and the parallel programs to run. In this model VPUs can also use a virtual shared memory represented by hierarchical mobile agents. All the properties offered to the proposed model, are easily designed thanks to the flexibility and the mobility of the multi agent systems. This leads to a strong computation model for high perform-

ance computing for physical applications. Some illustrative parallel program samples are presented and implemented to show the effectiveness of the proposed machine.

Keywords: Parallel Processing, distributed computing, parallel virtual machine, middleware, multi-agent system, Load balancing

1. Introduction

Pervasive computing applications need more processing power and availability of storage resources and computing. In the recent decade, the analysis tools of the computation methods and their technological computational models have known a very high level of progress. This progress has oriented the scientists toward new computation strategies based on parallel approaches. Due to the huge volume of data to be processed and to the large amount of computations needed to solve a given problem, the basic idea of the parallel computing is to split tasks and data so that one can easily perform their corresponding algorithms concurrently on different physical computational units. Naturally, the use of the parallel approaches implies important data exchange between computational units. Subsequently, this generates new challenges on data exchanging and communications. To manage these communications, it is important to exam how the data in query are organized. This examination leads to several parallel algorithms and several corresponding computational models. Actually, we distinguish several computing models starting from a single processor until the massively fine grained parallel machines having a large amount of processing elements interconnected according to several topological networks. Indeed, the analysis of the performance enhancement in terms of processing ability and execution speed must take into account the data types and their pointer management problem.

The need of the new architectures and the processor efficiency improvement has been excited and encouraged by the VLSI development. As a result, we have seen the new processor technologies (e.g. Reduced Instruction Set Computer “RISC”, Transputer, Digital Signal Processor “DSP”, Cellular automata etc.) and the parallel interconnection of fine grained networks (e.g. Linear, 2-D grid of processors, pyramidal architectures, cubic and hyper cubic connection machines, etc.). Several parallel architectures were created to solve parallel problems. In the past decade some simple grids of cellular automata have been created. Due to some technological enhancements, the cellular automaton became a fine grained processing element and the resulted grid became the well known mesh connected computer MCC [1]. Using some additional communication Buses, the MCC became the mesh with multiple broadcast [2] and polymorphic torus [3]. Finally, the reconfigurable mesh computer (RMC) integrates a reconfiguration network at each processing element [4, 5]. Recently, the graphical processing unit "GPU" architectures [6,7,8] that are largely used for graphical parallel computing, are

introduced as the most feasible structure to represent some parallel architectures for more complicated problems.

The theorists innovate more and more by imagining new parallel machines having well adapted topologies to specific problems. These innovations lead to new algorithms for high performance calculation. Unfortunately, the technology is not able to follow the scientist's imaginations. Due to the unavailability or to the high cost of this kind of real parallel machines, we conclude that creating emulators for these architectures is the first good way to test and validate the parallel algorithms on serial machines. In this context, the realization of an emulator for SIMD parallel machines has been the first exploited model in our research works [9], [10], [11] and [12]. This emulator has been of great use to elaborate, validate and test new parallel algorithms without need the real parallel machines. However, it is clear that emulating a parallel machine on a single processor machine do not introduce any enhancement in terms of performance at runtime. It is therefore necessary to look for other ways to achieve these emulated parallel applications in a real environment that offers a high performance gain.

The parallel and distributed systems have become a natural solution for different types of environments [13]. In such systems many middleware have been designed and used to build grid computing. Subsequently, new challenges appear, such as load balancing and fault tolerances. The quality and performance of a grid computing depend on the performance of the machines used in its nodes, the interconnection networks and mainly the quality of the associated middleware. Researchers intensify their efforts to find new solutions and new high-performance middleware having new innovative features to improve the performance of parallel virtual machines based on grid computing.

Another aspect in term of high performance computing is related to the load balancing problem. Indeed, several load balancing algorithms have been designed for distributed systems [14] [15] [16] [17]. In the distributed systems, tasks progress is viewed as a dynamic process that can be sometimes unpredictable. This means that the state of load at each node cannot be static. So, a dynamic load distribution process is necessary. In this way, as an example, the diffusion algorithm [18] is based on collecting information about nodes of the system. The communication policy, used in this collection procedure, can have a negative impact on the convergence speed of the load balancing algorithm. In the diffusion algorithms [19, 20, 21], authors used a method that involves a domain based collecting information from each group of nodes, where each group corresponds to a domain. This method tries to balance loads in each domain. It acts naturally by the fact that excess loads must be transferred to under loaded nodes in their respective domains, in order to reduce the communication costs. To move the loads in a distributed system the authors have used in [22] a mobile agent, which migrates loads from overloaded nodes to under loaded ones, but they consider that all the grid nodes are homogeneous.

In this paper, we present a new model of a virtual massively parallel machine, assigned for parallel and distributed high performance computing systems. This

parallel virtual machine allows us to build an extensible polymorphic computing grid used to solve fine grained parallel problems. This model is built using distributed Virtual Processors Units (VPU). Each VPU is a mobile agent deployed in a physical processing unit. The physical processing units of the distributed system nodes can be heterogeneous machines; they may be supercomputers, simple desktops, smart phones, etc.
Some parallel algorithms are implemented on the realized machine to assess the model and show how it may be feasible and effective in the high performance computing domain.

This paper is organized as follows: In the second section we will describe the architecture and the functional aspect of the designed parallel virtual machine. The third section is devoted to the main software components of the proposed model using some illustrative figures and their UML class diagrams. These components correspond to the Virtual Processor Unit (VPU) and the virtual shared memory components that are associated to synthesis the global parallel virtual model. The fourth section describes shortly the load balancing process designed for this model. Some obtained results by performing an application example of parallel algorithms on the proposed model are presented in section V. The finale section gives some concluding remarks and perspectives of this work.

2. Parallel Virtual Machine Architecture

Before giving details of the different parts of the designed polymorphic model, an overview of its functional and technical architecture is necessary. Thus, the proposed massively parallel virtual machine model is viewed as a distributed computing system. It is mainly composed of a set of n virtual processor units (VPU_i (i=1...n)). Each VPU_i is a mobile agent of performance index Pi(t) at a given iteration time t when it is deployed in a physical processing unit N_i. At each node, the processing units may be heterogeneous machines. Each VPU_i is designed to perform basic tasks and other processing procedures on data items. Each task should be assigned to a VPU. The VPUs can communicate with each other asynchronously by exchanging ACL messages (Agent Communication Language). In the proposed machine, the life cycle of VPUs, the load balancing task and the parallel programs, are managed by a special agent named Virtual Host Agent (VHA). It represents the Host of the parallel virtual machine. It is an agent that is responsible to launch the execution of the parallel programs by distributing data and tasks to the deployed VPUs in the nodes of the proposed system. At the same time, the load balancing task is generated and launched to distribute, efficiently, loads to different nodes of the system. The corresponding agent to this task, must communicate with other agents named Performance Node Collector Agents (PNCA). Each PNCA is deployed in a node to collect, at runtime, the performance node. At runtime, the VPUs can migrate from one node to another in order to move loads from overloaded nodes to under loaded ones.

Figure 1 shows the physical structure of a parallel virtual machine based on the four nodes N_0, N_1, N_2 and N_3. The VHA agent is arbitrarily deployed in the node N_0. In this example, the deployed application has created a parallel virtual machine of 24 VPUs, distributed over the four nodes according to a given topology. In this figure we show that in each node a special agent for the Performance Node Collection (PNCA) is deployed to provide, to the VHA, the information about loads of these nodes. In this parallel virtual machine model, the VPUs can be interconnected through their ports in order to exchange data inside their neighborhoods.

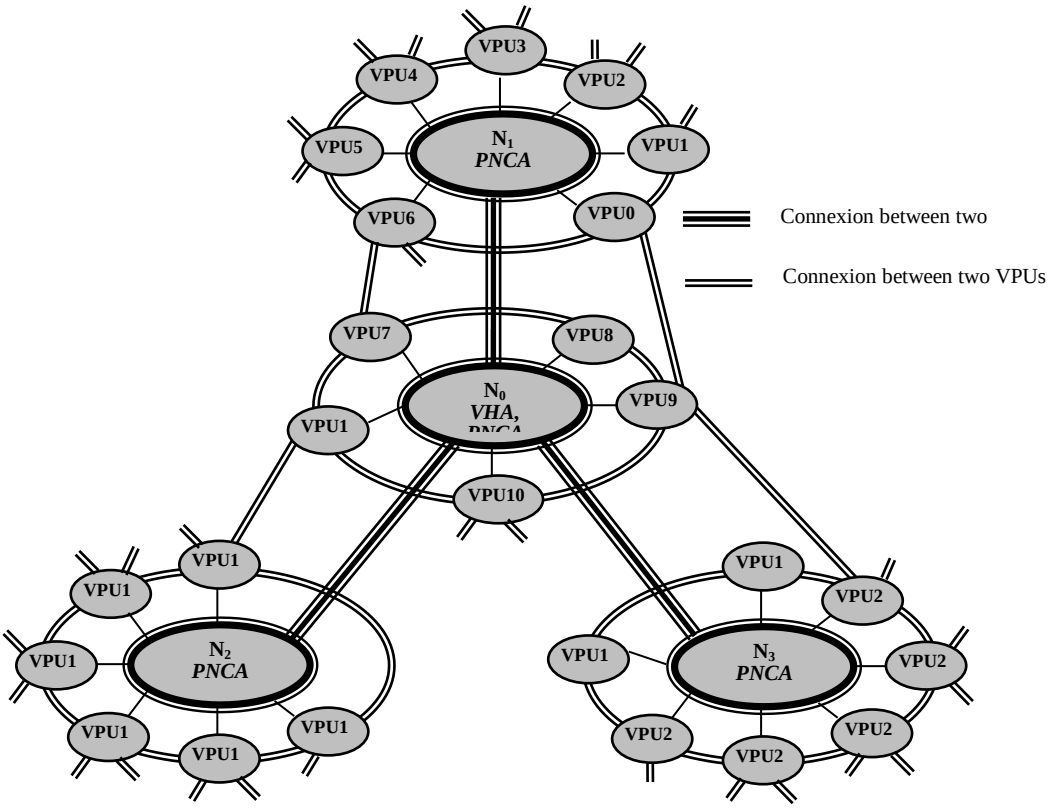


Figure 1. Distributed System with 4 nodes and 24 distributed VPUs

Figure 2 represents an example of parallel virtual machine whose VPUs are associated according to 2D Mesh topology of size (6 x 6). This machine has 36 virtual processing units VPU(i, j) *(i=0...5 and j=0...5)* which are deployed in 4 physical processor machines of the realized virtual system. In this configuration, we assume that these 4 physical processors have the same performance index P. Each VPU is connected to its four VPUs neighbors, if they exist, by its North, South, East, and West communication ports. All the VPUs can use their own internal memory components and can share a component memory represented by a distributed memory agent.

Each VPU of the virtual machine has its virtual arithmetic and logic unit, its virtual internal registers and its virtual internal memory. All the VPUs can share a distributed memory that is represented by the special hierarchical memory agents.

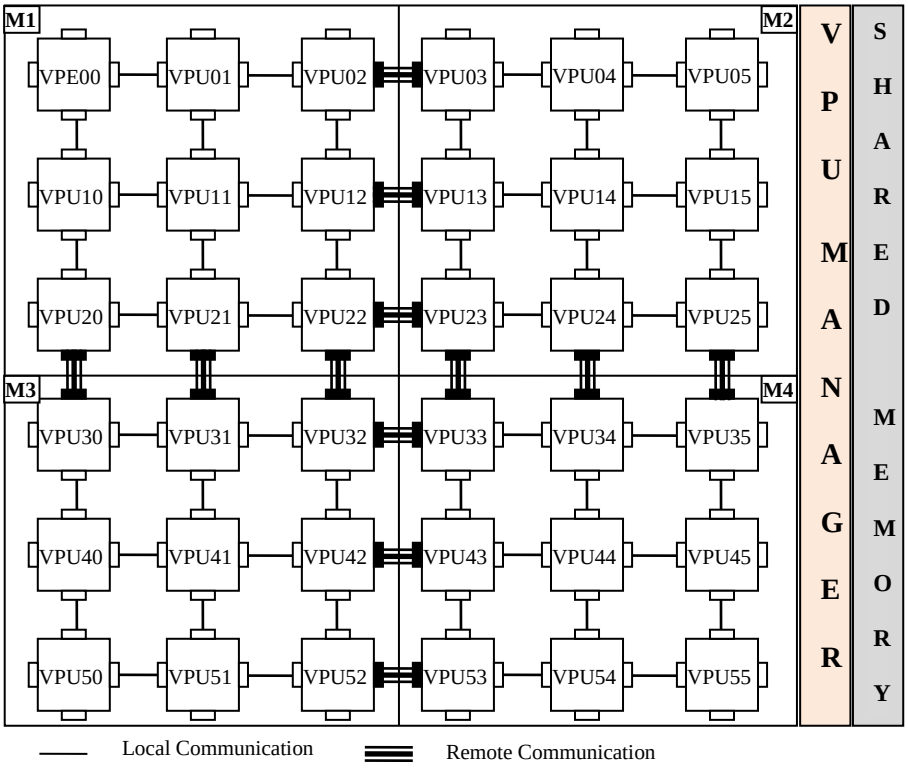


Figure 2. An example of Parallel Virtual Machine using 2D Mesh topology.

The proposed Massively Parallel Virtual Machine is viewed as a grid computing which is assigned to perform parallel programs according to the different parallel structures (SIMD, SPMD, MIMD, or SPMD). To belong to the corresponding grid, any physical computer machine must deploy an agent container, named *Node Manager Agent Container* (NMAC), which is based on a specific middleware of multi-agent systems. In this paper, we present the results of an implementation of this parallel virtual machine model, based on the JADE [26] middleware. To stand up this machine, all the NMAC agents must be connected to a special node named *Main Node Manager Agent Container* (MNMAC) to be part of the multi-agent platform. Figure 3 shows the used standard model of an agent platform as defined by the Foundation for Intelligent Physical Agents (FIPA) [27]. In this platform, the Agent Management System (AMS) is the one that supervises the access and the use of the Agent Platform. Only one AMS will exist in a given platform where it provides a white-page and a management life-cycle services. Each agent must be registered inside an AMS in order to get a valid AID. The Directory Facilitator (DF) is the agent that provides the default yellow page service in the platform. The message transport system, called Agent Communi-

cation Channel (ACC), is the software component controlling all the exchanges of messages within the platform, including remote messages to/from platforms.
In the proposed virtual machine model, each deployment is associated to a specific Virtual Host Agent (VHA). This latter is responsible to build the parallel virtual machine. To do so, the VHA starts by initializing the load balancing system to determine the performance index of each physical node. Then, it proceeds by deploying, over the platform nodes, the required VPUs for a given parallel application. This VHA is also assumed to distribute elementary data and parallel tasks to the VPUs. During the program execution of the parallel application, the load balancing system is reconfigured by setting up its parameters. Therefore, in order to keep a balanced load in the system, the VHA agent must ask some VPUs to migrate from the overloaded nodes to the under loaded ones. Also, it must deal with conventional operations of parallel and distributed systems such as the synchronization of parallel tasks, managing life cycles of other components, nodes monitoring, logging and load balancing in different grid nodes, etc.

3. Parallel Virtual machine components

3.1. Virtual Processing Unit (VPU).

In Figure 4, we show two physical machine nodes N_1 and N_2. Each machine has its own standard hardware components (CPU, RAM, ROM, HDD etc.), classical software (BIOS, OS, JVM) and our software solution for parallel virtual machine. Both machines are assumed to be connected by a conventional network. In the first physical machine at node N_1, two VPUs are deployed, while the third one is deployed in the second machine at node N_2. The three VPUs are connected together through their ports. In this figure, we show that the communication between VPU01 and VPU02 is established using their local ports. While the communication between VPU02 and VPU03 is established using their remote ports. Each VPU is a virtual software entity that represents one of the main components of our parallel virtual machine. In addition to its standard components as listed above, we propose to offer to a VPU a set of useful internal registers from which we distinguish:

- *Identifier Registers*: Used to store the location coordinates of VPU according to the chosen topology. In the 2D Mesh case, we define: iReg and jReg registers to save row and column index of the VPU in a given matrix (n x m).
- *Data Registers* : Like any CPU, the VPU must use its data registers to perform arithmetical and logical operations. To do so, we defined in the VPU model a collection of internal registers named: data Registers.
- *Flag Register*: As any standard processor, we introduce in the VPU model a special flag register, where each bit indicates the VPU state after any instruction. In our model, this register is arbitrarily defined by an array of 16 bits, but it can be extended to largest size according to the usefulness of the additional instructions.

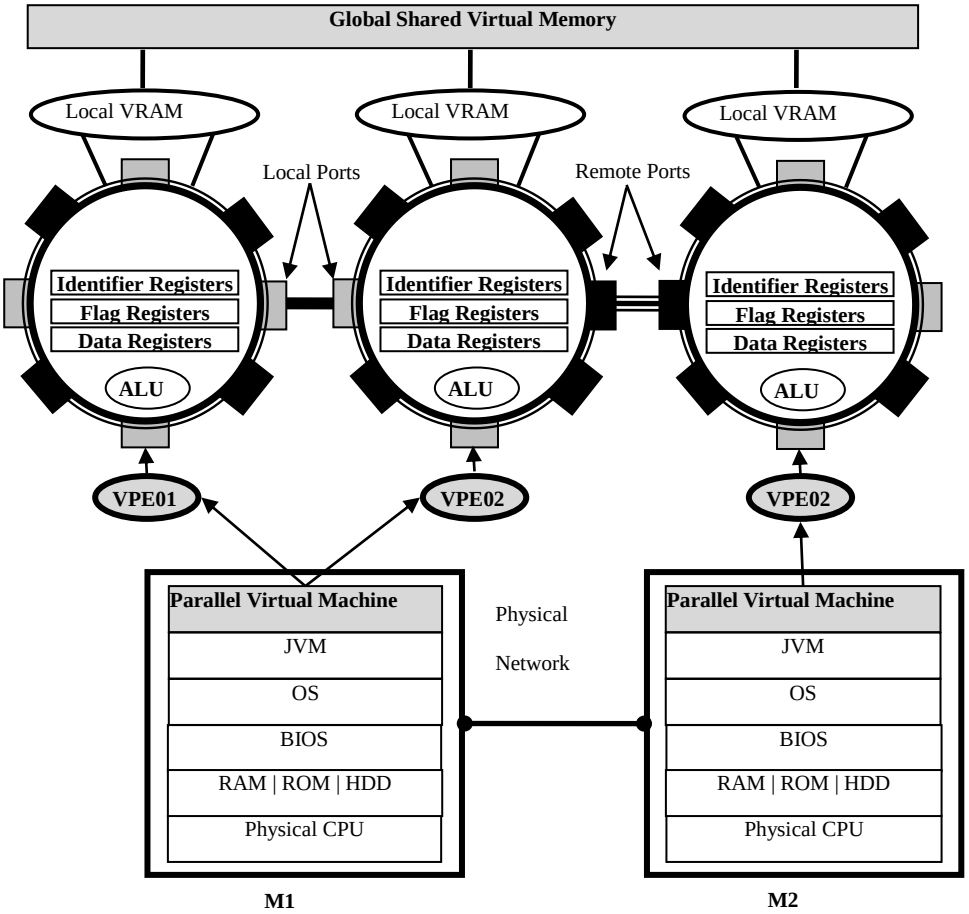


Figure 4. The main components of the VPU.

In the case of a SIMD structure, for any fine grained parallel algorithm, the data of the problem must be stored in internal registers of the VPUs. While in the SPMD or MPMD structure, the VPUs need to store data in a virtual local memory VRAM. When the amount of data is very large and exceeds the physical storage capacity of the local real machine. Our model provides the ability to use a global virtual memory shared by all the VPUs of the virtual parallel machine. The design and organization of this virtual shared Memory is described in section 3.B.
Any VPU can communicate with other local or remote VPU of the grid using its logical ports. The communication cost between VPUs, depends on the chosen virtual machine topology. It is therefore crucial to choose, for each kind of parallel program, the adequate topology that optimizes the algorithm complexity.

In this work, all the machine components models are based on mobile agents. As shown in Figure 5, the VPU model is structured to give opportunity to other developers, using this framework, to customize their own implementations; In fact, we have defined an abstract implementation for the VPU. It is represented by *AbstractVPU* class. Thus, we associate to the VPU, a list of data registers, a list of

identifier registers and a list of flag registers. Each register is defined by its name and its content. To manage communications between agents of the platform, we assign to each VPU a cyclic behavior named *CyclicBehaviour*. In order to offer to a VPU the ability to identify its neighbors in the chosen topology, we associated it to a list of agent identifiers *AID*. To select one ACL message that the VPU must carry out, a message template *MessageTemplate* is defined using a specific ontology for VPU agents. At run time, each active VPU is asked to load its data items and parallel tasks to execute. To do so, the VPU is associated to a list of parallel tasks *ParallelTask*, where each task is executed as a new thread.
The elementary data of the VPU are represented by extended class objects *AbstractData*. Therefore, the programmers can implement their own data structures according to the problem to solve. In this kernel, we have also included an abstract implementation of VPU which is adapted to 2D Mesh architecture, in which we need to save the rows and columns indices of the 2D grid. In order to allow the programmer to extend the operations set of the VPU, it may define a new extended class of the defined abstract implementations in the core of this Framework. This is the case of *MyVPUImpl* class, where we have implemented two new operations representing the parallel tasks to be executed by all the VPUs of our parallel virtual machine. Also, we associated to the VPU a graphical interface where we can display its current activities states.
The listing 1 represents a simple example where we show an implementation of a VPU which extends PEmesh2D and in which an image processing procedure sample is implemented. Il is the parallel task named *paralleleSobel* which consists in applying SOBEL operator on an elementary image for contour detection. This problem consists of splitting a large image into tiny image fragments. Then distribute the obtained list of image items, to all the VPUs of our parallel virtual machine. Thereafter, all VPUs execute, at the same time, the Sobel operator using their elementary images. Then each VPU returns the elementary result image to the host. This latter will perform the assembly operation of the global image.

Listing 1. Example of a java implementation of the VPU.

```
/* New VPU Implementation extending the PUMesh2D /*
public class MyVPUImpl extends PUMesh2D {
/* Implemtation of sobel operator /*
public ElementResult parallelSobel (ParallelTask t) {
 /* Getting de input data /*
 DataMatrixDouble dm=(DataMatrixDouble)
 getData(t.getInputDataName());
 double[][]m=dm.getData();
 double[][] res=new double[m.length][m[0].length];
 /* sobel algorithm implementation /*
 for(int i=1;i<m.length-1;i++)
  for(int j=1;j<m[0].length-1;j++){  // Sobel operator … }
   /*Return the elementary result to the host /*
 return new ElementResult(new DataMatrixDouble(res));
}}
```

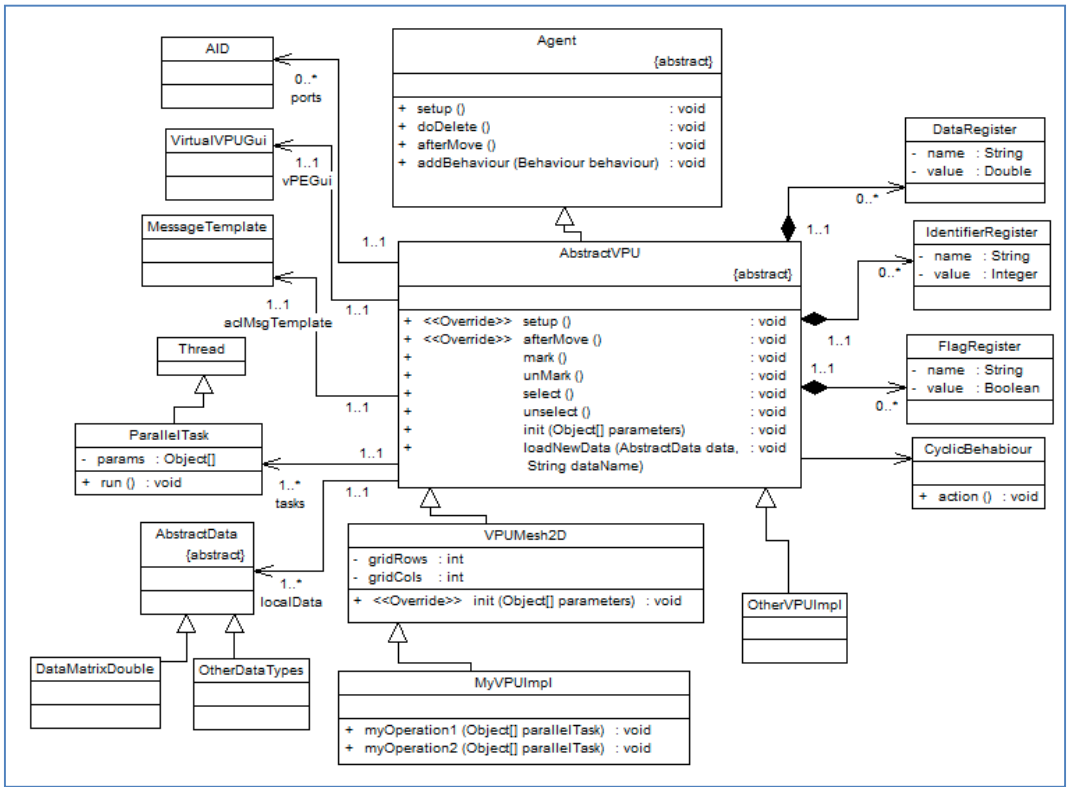


Figure 5. UML Class Diagram representing the logical structure of a VPU

3.2. Parallel Virtual Machine Model

The UML class diagram of Figure 7, shows the main components of the proposed parallel virtual machine. We define an abstraction layer that represents the core of the virtual machine, and then we let the user define its own implementation. In order to represent the host of the parallel virtual machine, we introduce the component *VPUManager* as an extended one of the abstract class *AbstractParallelVirtualMachineAgent*. This abstract class translates the polymorphism behavior of our model. It allows several implementations according to the topologies of the virtual machine (Mesh2D, mesh3D, Pyramid, Hypercube etc.).

In this paper, we focus on the simplest 2D Mesh topology implementation. It is represented by a class named: *Mesh2DVirtualMachine*. The VPU Manager must identify all the distributed VPUs agents and the Virtual Shared Memory through their local stubs *VPUStub* and *VirtualMemoryStub* respectively. If the shared memory mechanism is not enabled, then the data of the global application are stored in a data list of type *AbstractData*. To establish a parallel program in this platform, we must create an extended class of *AbstractParallelProgram*, then

override *deploy()* and *run()* methods that are executed by the parallel virtual machine when we deploy and execute the parallel program respectively.
To stand up our virtual machine, two principal phases are necessary:

A. Deployment and building phase

The *deploy()* method of any implemented parallel program is the first executed by the parallel virtual machine. In this method, the programmer must decide and select the topology to be associated to the parallel program. Also it must specify the implementation of VPU to be used. These actions can be performed using the *createVirtualGrid()* statement that triggers a set of operations to build the parallel virtual machine, these operations are:

1. Create and start a new agent controller,
2. Deploy the virtual host manager agent, representing the selected parallel virtual machine topology. In the proposed example, we use *Mesh2DVirtualMachine*. This virtual host manager agent handles the control of the life cycle of the parallel Virtual machine.
3. Get the node list of the system using the behavior *GetAvailableLocationsBehaviour*. This latter uses the yellow pages service of the multi-agent system.
4. Retrieve the identifier AID from the naming service, for the *VirtualMemoryAgent (VMA)* if this option is activated in the platform.
5. Launch a cyclic behavior where the communication strategy with all other agents (VPUs and VMA), is implemented. The execution of the ACL messages, by the virtual host, requires a Message Template (MT) where a specific ontology of the framework is defined.
6. Perform a diagnostic test to evaluate the performance index of each node. This step is necessary to get the initial parameters for the load balancing procedure.
7. Deploy and distribute the *VPUs* in all nodes taking into account the result of the load balancing program. At this stage, the parallel virtual machine is built and ready to launch the program execution.
8. Display the results on the Graphical User Interface of the built machine.

In the *deploy()* method, it is possible to use the *addInitParameter* statement to specify the parameters and configure the execution of the deployed parallel application.

B. Execution phase

In the implementation *MyParallelProgramImpl*, described in the UML class diagram of figure 7, the override method *run()* contains the parallel program code to be executed in the parallel virtual machine.
The code in Listing 2, shows an example of a parallel program that creates, at the deployment phase, a 2D Mesh grid using an interactive interface in the GUI to define the size of the 2D mesh *(row and column numbers)*. The *run()* method is

established to contain the program code example to realize the following tasks:

1. Split a given input image into a list of elementary images using the *ImageSpliStream* statement that was defined in *Mesh2DVirtualMachine* implementation.

2. Define and select the parallel task. In this case the *parallelSobel* task is assigned to be performed by all the created VPUs.

3. Distribute the list of image items to all the deployed VPUs. This operation uses *broadCastStreamData* statement.

4. Launch all the VPUs to execute the *parallelSobel* task. This task is carried out by the operation *executeParallelTask* which is implemented in the *AbstractParallelVirtualMachine* class. Once all the VPUs have returned their elementary results, the *executeParallelTask* operation returns a result as a list of data items.

5. The *viewImage* statement is used to show the global resulting image in the GUI.

The *end()* statement is used to ask the host manager to destroy all the components of the created parallel virtual machine.

Listing 2. Example of a parallel program implementation

```
public class MyParallelProgram extends AbstractParallelProgram{
@Override
public void deploy(Object[] params) {
   createGrid(MESH_2D,"test.MyParallelTask",params);
   addNewInputParam("image", "DS6/2.jpg");
}
@Override
public void run(){
      String image=getInputParamValue("image");
  1.  ImageSplitStream is=new ImageSplitStream(image, 1,
      gridRows,gridCols);
  2.  parallelMachine.broadCastStreamData("m1", is);
  3.  ParallelTask t1=new ParallelTask("sobel","m1","res1");
      t1.setOutputDataName("res1");
  4.  List<AbstractData> liste1=parallelMachine.executeParallelTask(t1);
  5.  parallelMachine.viewImage("res1");
  6.  end();
  }
}
```

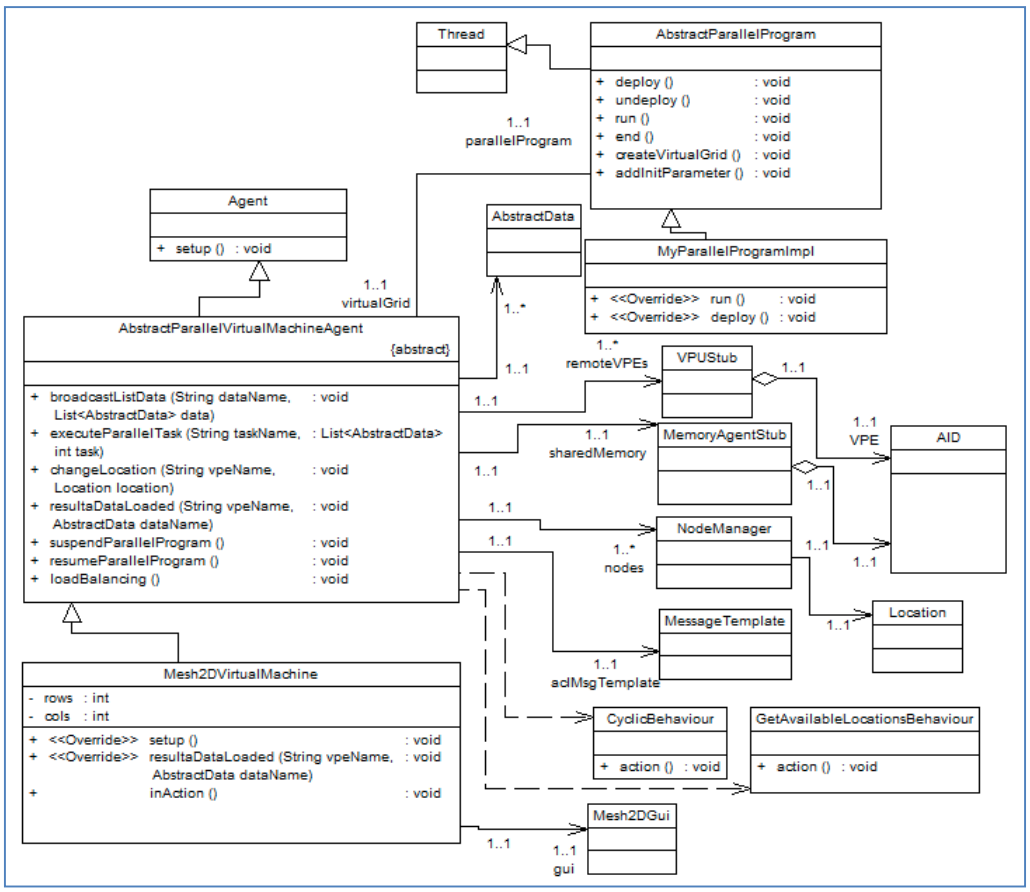


Figure 7. The class diagram of the parallel virtual machine

4. Load Balancing System

In this section, we deal with some essential properties of the proposed load balancing procedure. This latter is performed by the VHA which is deployed in the host node N_0. This agent should deploy, in the n nodes of the grid, a set of m virtual processing units (VPUs). Each *VPU* will be charged to execute a task T_k on a data structure D_k. Before performing this task, the VHA agent should determine the performance index of each node of the system. To do so, this agent starts by identifying the list of the n agents *NPCA***,** from the naming service of the system, to launch the diagnostic performance test.

Each agent *NPCA* is asked to perform an arbitrary reference task T_0, of a known complexity $C_0(x)$**,** on a known reference data structure D_0 of size x_0 (for example, a matrix of 30 x 30 numbers of type double). In this test, it is important to choose the kind of the task in which communication latency will be very small and negligible compared to the computation duration. Also, the computation time of this test program should be very short to achieve a quick load balancing convergence of the system. The node performance index determination is realized by the following steps:

- **Step 1: Diagnostic test of the local node N_0:**

The reference task T_0, is firstly executed in the local node N_0, where the agent VHA is deployed alone, without using a distributed agents. This step allows us to know the processing duration $\theta_{REF} = \theta P_0(T_0)$ of the reference task T_0 in the local node N_0. Since the same task must be executed by all the distributed agents *NPCA* deployed in local or remote nodes, we can measure the communication cost between the distributed agents including those deployed in the local node N_0.

- **Step 2: Diagnostic test of the node N_i:**

After the first step, the reference data D_0 is sent at time t_0 to all the distributed agents *NPCA* over the nodes N_i. Upon receipt of these data, each agent *NPCA* executes the reference task T_0 and returns, to the *VHA*, its resulted data R_0 in the node N_i. The data structure R_0, of size y_0, includes the processing duration θP_i elapsed during the execution of this task in N_i.
Upon receipt of the result returned by each node N_i, at time $t_1(i)$, the *VHA* agent can determine the total duration ($\theta_i = t_1(i) - t_0$), necessary to perform the task T_0 in each node N_i. Notice that the duration θ_i includes the processing duration and the communication latency between N_i and N_0.
The communication Latency of each node N_i, can be easily reported as:

$$\theta L_i = \theta_i - \theta P_i$$

- **Step 3: Determination of the performance index of the node N_i**

The performance index of each node N_i can be calculated as:

$$P_i = \frac{\theta_{REF}}{\theta_i} \quad (1)$$

The performance index P_i can also be measured using the minimum of the execution durations in all nodes as:
$MIN(\theta_j)_{(j=0 \text{ to } n-1)}$, instead of θ_{REF}, So, P_i becomes Q_i where;

$$Q_i = \frac{\underset{j=0}{\overset{n-1}{MIN}}(\theta_j)}{\theta_i} \quad (2)$$

Notice that: The first coefficient P_i measures the ratio between the local node N_0 , without using any distributed agent, and node N_i through a distributed agent. While Q_i measures the performance ratio, taking into account the communication latency between N_i and the fastest node.

- **Step 4: Load determination**

The load L_i of data D_k to be assigned to each node N_i can be calculated by dividing the performance factor Q_i, by the average of performance factors in all

nodes and multiplying by the unweighted load L_{REF}. This later is obtained by dividing the number m of data, to be processed, by the number n of nodes. So,

$$L_{REF}(i) = m.\frac{Q_i}{\sum_{i=0}^{n-1} Q_i} \quad (3)$$

- **Step 5: Rebalancing procedure**

At the first iteration, each node $\boldsymbol{N_i}$ must start its task by determining the amount of data $L_{REF}(i)$. This reference load distribution is determined, at the deployment time of the parallel program, in order to initialize the load balancing system. The goal is to achieve a perfect load balancing system, where the duration, required to perform the list of tasks assigned to each node, θD_i should be independent on the node $\boldsymbol{N_i}$. This means that, all the nodes should finish the execution of their jobs at the same time.

As mentioned above, the load balancing procedure is managed by the VHA which represents the Host of the parallel virtual machine. The VHA performs a load balancing algorithm to assign, efficiently, loads to different nodes of the distributed system. Subsequently, at runtime, the VPUs can migrate from one node to another in order to move loads from overloaded to under loaded nodes. The undertaken procedure for this task is based on deploying distributed agents in all the nodes. Hence, at each iteration, new parameters must be calculated in order to keep the system in a balanced state. Thus, we have proposed an algorithm which determines the required migrations to carry out loads from overloaded to under-loaded nodes. To do so, we engaged mobile agents that represents virtual processing unit of the parallel machine. Also, this model allows us to estimate, for any given task, the processing costs in the sequential and distributed modes. This feature can be used to adapt easily the most appropriate architecture to any parallel task

In an iterative distributed computing program, we should check, at each iteration *t*, whether unbalanced loads appear in the distributed system. If yes, the *VHA*, must activate the deployed *VPUs* agents, in the overloaded nodes, to migrate to under loaded ones.

To clarify this procedure, we consider the following situation at iteration *t* :

- *m*: Represents the number of *VPUs* of the parallel virtual machine.
- $L_i(t)$: Represents the load assigned to the node N_i during iteration *t*. Also, it corresponds to the number of VPUs deployed in this node N_i.
- $\Theta D_i(t)$: Represents the processing duration of the load $L_i(t)$, by the node N_i, during iteration *t*.

To rebalance loads, we will proceed by the following steps:

a) Calculate the performance index Q_i of each node N_i during iteration *t-1*.

$$Q_i(t-1) = \frac{\underset{j=0}{\overset{n-1}{MIN}}(\theta D_j(t-1))}{\theta D_i(t-1)} \quad (4)$$

b) Calculate the new loads, for iteration *t*, as a function of the performance index Q_i *(t-1)* at *t-1* of the node *i*, the average of the performance factor of all nodes, the loads number *m* and the nodes number *n*.

$$L_i(t) = \frac{Q_i(t-1)}{\langle Q_i(t-1)\rangle}.\frac{m}{n} \quad (5)$$

c) Determine load error between iterations *t* and *t-1*, to distinguish the overloaded and under loaded nodes. We denote this error term ΔLi.

$$\Delta L_i(t) = L_i(t) - L_i(t-1) \quad (6)$$

d) Use the following marks to distinguish: Overloaded ***(ON)***, Under-loaded ***(UN)*** and Balanced ***(BN)*** node sets. So;

$$\begin{aligned} ON &= \{N_i / \Delta L_i < 0\}_{(i=0..n-1)} \\ UN &= \{N_i / \Delta L_i > 0\}_{(i=0..n-1)} \\ BN &= \{N_i / \Delta L_i = 0\}_{(i=0..n-1)} \end{aligned} \quad (7)$$

e) Determine the required migration of the VPUs from the overloaded to the under-loaded nodes in order to keep the parallel virtual machine in a balanced state.

5. Application: Parallel C-MEAN Classification Algorithm

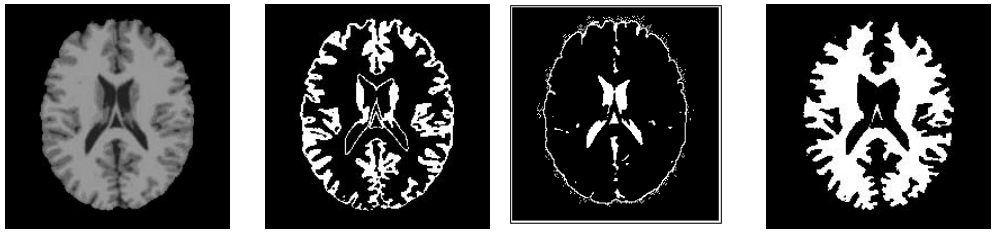
To assess the proposed virtual machine, we define the problem as follows:
Given a stream of *m=500* images, each of size (row, column) = (640, 480) pixels. We perform an application algorithm that consists of a parallel program for medical image segmentation.The results presented in this section are obtained using a distributed system with 10 physical nodes (*n= 10*). The node N_0 is the host from which the application is launched. In this experience, the list of m images is distributed, using the load balancing algorithm, to m agents of the parallel virtual machine. Each agent is asked to perform the classification algorithm using its assigned MRI image. In this application, we show how to create a parallel application as an SPMD (Single Program Multiple Data) program.
Image segmentation [23,24,25] is a splitting process of images into a set of regions, classes or homogeneous sub-sets according to some criteria. Usually, grey levels, texture or shapes constitute the well-used segmenting criteria. Their choice is frequently based on the kind of images and the goals to reach.
This program consists of the following steps:

1. The host initializes the load balancing system
2. The host loads a set of **m** images from the streaming data.
3. The host distributes the list of **m** images to the VPUs of the parallel virtual machine
4. The host broadcasts the current class centers over this parallel virtual machine.

5. All the VPUs, run C-MEAN classification algorithm and return results
6. The host binds a list of the received results.
7. The host repeats the steps (1 to 7) if a new set of **m** images appears in the stream.

5.1. Results

After performing the presented parallel program, we obtain the following results: The image of figure 8a) corresponds to an example of human brain cut of the list of images to be performed, it is the original input image of the program. Figures 8b), 8c) and 8d) represent the three brain matters. They are named the grey matter, cerebrospinal fluid and white matter respectively.



a) original image b) grey matter c)cerebrospinal fluid d) white matter

Figure 8. Segmentation results of the elaborated parallel program

The reference diagnostic test, for evaluating the performance of the nodes of the distributed system, has been carried out by performing in a node N_0 a reference task T_0 having a known complexity C_0 (x), and a known reference data D_0. In the table I, we show the results obtained for at the first iteration of the classification. This table shows, for each node, the performance index the performance index $Q_{REF}(i)$ obtained by equation (2), the load distribution obtained by equation (3) and the duration, ϴD(i), required to perform the set of tasks assigned to each node.

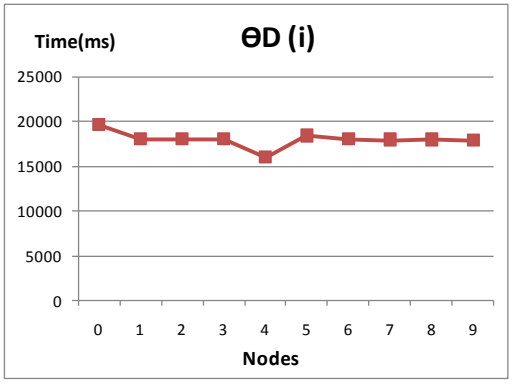


Figure 9. Classification durations in each node

TABLE I
RESULTS OF DISTRIBUTED CLASSIFICATION.

Node i	$Q_{REF}(i)$	$L_{REF}(i)$	EXP_$ϴD_i$ (ms)
0	1,00	61	19806
1	0,70	43	18028
2	0,78	48	18032
3	0,83	51	17969
4	0,99	61	15821
5	0,91	56	18389
6	0,65	40	18048
7	0,93	57	17923
8	0,72	44	17951
9	0,64	39	17838

From these results, we can notify that the required time to achieve the segmentation of 500 images in this parallel virtual machine is the maximum of the durations *ΘD(i)* obtained in all nodes. In figure 9, we have: *ΘDMAX=19806 ms*. This value means that the node 1 is the most overloaded node. The fastest node, in this case, is the node 4 with a value of *ΘDMIN= 15821 ms*. The unbalanced load magnitude is computed as: *ΘDMAX- ΘDMIN= 3985 ms = 3.98 s*.

We notice that, at the first iteration, the first distribution of loads using L_{REF} leads our parallel virtual machine to a satisfactory load balancing state. This state is more improved by performing the second iteration of the load balancing algorithm as shown in figure 10.

To compare with the sequential mode, the same algorithm has been executed in a single processor machine to perform the same stream of images. The obtained duration is *ΘS= 144500 ms*. The performance ratio, between sequential mode and distributed one is evaluated as: $\lambda = \Theta S / \Theta D_{MAX} = 7{,}37$.

Then, the performance ratio of this virtual machine of n=10 nodes is :

$\Omega = \lambda / n = 73.7$ %.

This result proves that our virtual machine provides very good performance ratio, at the first iteration. This ratio will be more improved in the next iterations thanks to the load balancing algorithm. We should apply the refinement load balancing algorithm which determines the required migrations of the VPUs, from the overloaded nodes to under-loaded ones, to keep the system in perfect load balancing state. The obtained results by this refinement procedure are presented in the following two tables II and III.

Table II shows, for each node, the values of $L_i(t-1)$, $\theta D_i(t-1)$, $Q_i(t)$, $L_i(t)$, $\Delta L_i(t)$, STATE obtained respectively by the equations 4, 5, 6 and 7.

The values : $AN\_\theta D_i(t)$ and $EXP\_\theta D_i(t)$ represent the analytical and the experimental θD_i respectively, obtained for the iteration t=1.

In table III, we notice that to improve the load balancing system, 5 VPUs migrate from the node 0 to the node 4 and 1 VPU migrates from the note 5 towards the node 4. After this migration process, we obtained the following results:

- $\Theta D_{MAX}(t) = 17412$ ms . The saving time according the previous iteration is $\Delta\Theta D = \Theta D_{MAX}(t-1) - \Theta D_{MAX}(t) = 2394$ ms . So, the relative improvement ratio is about 12%.
- $\lambda(t) = \Theta S / \Theta D_{MAX}(t) = 8.29$. So, the improvement is about 11%.
- $\Omega(t) = \lambda(t) / n = 82.9$ % . So, the improvement is about 9.2%.

TABLE II
REFINEMENT LOAD BALANCING ALGORITHM RESULTS

i	$L_i(t-1)$	$\theta D_i(t-1)$	$Q_i(t)$	$L_i(t)$	$\Delta L_i(t)$	STATE	$AN\_\theta D_i(t)$	$EXP\_\theta D_i(t)$
0	61	19806	0,82	56	-5	ON	17928	17928
1	43	18028	0,63	43	0	BN	17928	18428
2	48	18032	0,70	48	0	BN	17928	17651
3	51	17969	0,75	51	0	BN	17928	17528
4	61	15821	1,00	68	7	UN	17928	17913
5	56	18389	0,80	54	-1	ON	17928	17846
6	40	18048	0,58	40	0	BN	17928	17412
7	57	17923	0,84	57	0	BN	17928	17834
8	44	17951	0,65	44	0	BN	17928	19032
9	39	17838	0,58	39	0	BN	17928	17649

In figure 10, we notice that, thanks to the VPUs mobility, the parallel virtual machine is in a perfect load balancing state at the iteration t comparing to the iteration t-1. This distribution is illustrated in figure =11.

TABLE III
MIGRATIONS, BETWEEN NODES, TO REBALANCE LOADS.

Migrating Source Node	VPUs Count	Percentage of carried loads	Migate Destination Node
0	5	8,14%	4
5	1	1,79%	4

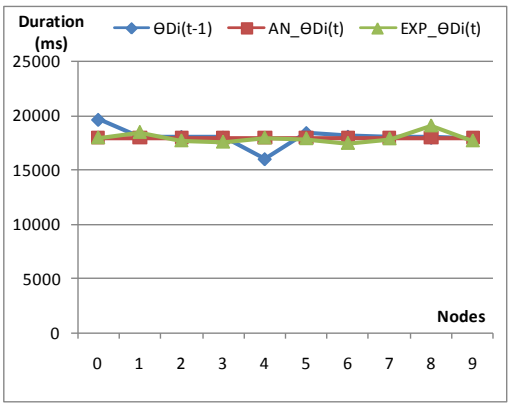


Figure10. Comparison between analytical durations, The experimental durations at the first and its next iteration

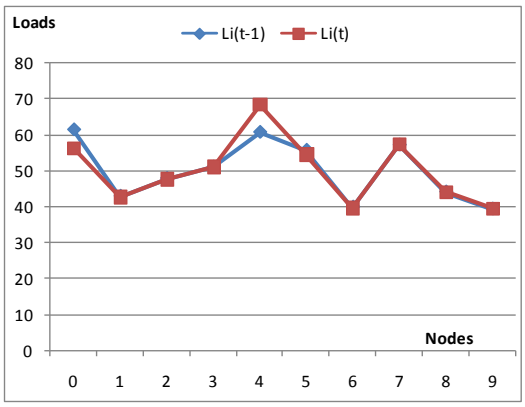


Figure11. Load distribution between two successive iterations

Conclusion

In this paper, we have presented a massively parallel virtual machine model, which is designed for parallel and distributed high performance computing. This model is built using mobile agents to model the virtual processing units that correspond to the main components of any parallel machine. This computational model is characterized by a special agent designed to represent the host of the parallel virtual machine. This agent manages the life cycle and other features of the virtual processors. The efficiency of the proposed model is improved by an associated fast load balancing algorithm. In this model the virtual processors can also use a virtual shared memory represented by the hierarchical mobile agents. However, if the cost of communication latency, caused by the network communication between nodes, is unknown, it remains to see whether the impact of asynchronous communication with agents, using ACL messages, do not alter this latency. An example of parallel algorithm for medical imaging that was performed on the proposed machine is also presented. The efficiency of the proposed model was discussed taking into account the size of the model in terms of node numbers. We show also the impact of the proposed load balancing procedure to enhance the acceleration ratio between serial and parallel models.

References

[1] R. Miller, Q. F. Stout, Geometric algorithms for digitized pictures on a mesh connected computer, IEEE Transactions on PAMI, Vol. **7,** No. 2, pp. 216–228, 1985. http://dx.doi.org/10.1109/tpami.1985.4767645

[2] V. K. Prasanna and D. I. Reisis, Image computation on meshes with multiple broadcast, IEEE Transactions on PAMI, Vol. 11**,** No. 11, pp. 1194–1201, 1989. http://dx.doi.org/10.1109/34.42857

[3] H. LI, M. Maresca, Polymorphic torus network, IEEE Transaction on Computer, Vol. C-38**,** No. 9, pp. 1345– 1351, 1989. http://dx.doi.org/10.1109/12.29479

[4] T. Hayachi, K. Nakano, and S. Olariu, An O ((log log n)²) time algorithm to compute the convex hull of sorted points on re-configurable meshes, IEEE Transactions on Parallel and Distributed Systems, Vol. 9, No. 12, 1167–1179, 1998. http://dx.doi.org/10.1109/71.737694

[5] R. Miller, V. K. Prasanna-Kummar, D. I. Reisis, and Q. F. Stout, Parallel computation on re-configurable meshes, IEEE Transactions on Computer, Vol. 42, No. 6, pp. 678–692, 1993. http://dx.doi.org/10.1109/12.277290

[6] Yue Zhao, Francis C. M. Lau, Implementation of Decoders for LDPC Block Codes and LDPC Convolutional Codes Based on GPUs, IEEE Transactions on Parallel and Distributed Systems, vol. 25, no. 3, pp. 663-672, March 2014. http://dx.doi.org/10.1109/tpds.2013.52

[7] Jing Wu, Joseph JaJa, Elias Balaras, An Optimized FFT-Based Direct Poisson Solver on CUDA GPUs, IEEE Transactions on Parallel and Distributed Systems, vol. 25, no. 3, pp. 550-559, March 2014. http://dx.doi.org/10.1109/TPDS.2013.53

[8] Abid Rafique, George A. Constantinides, Nachiket Kapre, Communication Optimization of Iterative Sparse Matrix-Vector Multiply on GPUs and FPGAs, IEEE Transactions on Parallel and Distributed Systems, 28 Feb. 2014. IEEE Computer Society Digital Library. IEEE Computer Society, vol. 26, no. 1, pp. 24-34. http://dx.doi.org/10.1109/tpds.2014.6

[9] M. Youssfi, O. Bouattane and M. O. Bensalah, A Massively Parallel Re-Configurable Mesh Computer Emulator: Design, Modeling and Realization, J. Software Engineering & Applications, 2010, 3: 11-26 Published Online January 2010. http://dx.doi.org/10.4236/jsea.2010.31002

[10] Bouattane, B. Cherradi, M. Youssfi and M.O. Bensalah, Parallel c-means algorithm for image segmentation on a reconfigurable mesh computer, Parallel Computing, 37 (2011) pp 230-243. http://dx.doi.org/10.1016/j.parco.2011.03.001

[11] M. Youssfi, O. Bouattane, and M.O. Bensalah, On the Object Modelling of the Massively Parallel Architecture Computers, Proceedings of the IASTED Inter. Conf. Software Engineering, February 16 - 18, 2010, Innsbruck, Austria, pp 71-78.

[12] M. Youssfi, O. Bouattane, and M.O. Bensalah, Parallelization of the local image processing operators, Application on the emulating framework of the reconfigurable mesh connected computer, Proceeding of scientists meeting in Information Technology and Communication JOSTIC, Rabat, November 3-4, 2008, pp 81-83.

[13] C. Kesselman, S. Tuecke, The Anatomy of the Grid: Enabling Scalable Virtual Organizations, International Journal of High Performance Computing Applications, vol. 15, issue3, pp 200–222, 2001. http://dx.doi.org/10.1177/109434200101500302

[14] D.R. Karger and M. Ruhl, Simple Efficient Load-Balancing Algorithms for Peer-to-Peer Systems, Theory of Computing Systems, vol. 39, pp. 787-804, Nov. 2006. http://dx.doi.org/10.1007/s00224-006-1246-6

[15] Ioannis Konstantinou, Dimitrios Tsoumakos, Nectarios Koziris, Fast and Cost-Effective Online Load-Balancing in Distributed Range-Queriable Systems, IEEE Transactions on Parallel and Distributed Systems, vol. 22, no. 8, pp. 1350-1364, Aug. 2011. http://dx.doi.org/10.1109/tpds.2010.200

[16] Hung-Chang Hsiao, Hsueh-Yi Chung, Haiying Shen, Yu-Chang Chao, Load Rebalancing for Distributed File Systems in Clouds, IEEE Transactions on Parallel and Distributed Systems, vol. 24, no. 5, pp. 951-962, May 2013. http://dx.doi.org/10.1109/tpds.2012.196

[17] H.-C. Hsiao and C.-W. Chang, A Symmetric Load Balancing Algorithm with Performance Guarantees for Distributed Hash Tables, IEEE Trans. Computers, 2012, http://doi.ieeecomputersociety.org/10.1109/TC.2012.13.

[18] G. Cybenko, Load balancing for distributed memory multiprocessors, Journal of Parallel and Distributed Computing, 7:279-301, 1989.

[19] Rupali Bhardwaj, V.S.Dixit, Anil Kr.Upadhyay, A Propound Method for Agent Based Dynamic Load Balancing Algorithm For Heterogeneous P2P

Systems, in International Conference on Intelligent Agent and MultiAgent Systems, 2009. http://dx.doi.org/10.1109/iama.2009.5228060

[20] F.M. auf der Heide, B. Oesterdiekhoff, and R. Wanka, Strongly adaptive token distribution, Algorithmica 15 (1996), pp. 413–427.

[21] Berenbrink, P. and Friedetzky, T. and Martin, R., Dynamic diffusion load balancing, in Automata, languages and programming: 32nd International Colloquium, ICALP 2005, 11-15 July 2005, Lisbon, Portugal; proceedings. Berlin: Springer, pp. 1386-1398. http://dx.doi.org/10.1007/11523468_112

[22] Liu, J., Jin, X. and Wang, Y., Agent-Based Load Balancing on Homogeneous Mini-grids: Macroscopic Modeling and Characterization, IEEE Transactions on Parallel and Distributed Systems, 586-594. 2005. http://dx.doi.org/10.1109/tpds.2005.76

[23] B. Cherradi, O. Bouattane, M. Youssfi, A. Raihani, M. Tarbouchi, A Fast and Robust Segmentation Algorithm for Cerebral T1-Weighted MR Images, Journal of Computing and Information Technology 01/2011, 3:137-153.

[24] J.H. Chang, K.C. Fan, Y.L. Chang, Multi-modal gray-level histogram modeling and decomposition, Image and Vision Computing, 20 (2002) 203–216. http://dx.doi.org/10.1016/s0262-8856(01)00095-6

[25] Bellifemine F., Poggi A., Rimassa G., JADE - A FIPA-compliant agent framework, CSELT internal technical report, Part of this report has been also published in Proceedings of PAAM'99, London, pp. 97-108, April 1999.

[26] C. Castelfranchi, Y. Lespérance, Developing Multi-agent Systems with JADE Intelligent Agents, In Intelligent Agents VII Agent Theories Architectures and Languages, Vol. 1986 (2001), pp. 42-47.

Received: May 19, 2015; Published: July 14, 2015