© International Journal of Combinatorial Optimization Problems and Informatics, Vol. 9, No. 1, Jan-April 2018, pp. 12-22. ISSN: 2007-1558.

A multi-agent architecture for scheduling of high performance services in a GPU cluster

Joel Antonio Trejo-Sánchez[1], José Luis López-Martínez[2], J. Octavio Gutierrez-Garcia[3], Julio César Ramírez-Pacheco[4], Daniel Fajardo-Delgado[5]
CONACYT-Centro de Investigación en Matemáticas[1]; Universidad Autónoma de Yucatán-Facultad de Matemáticas[2]; Department of Computer Science[3], ITAM, Rio Hondo 1, Ciudad de México 01080, México; Universidad del Caribe- Departamento en Ciencias Básicas e Ingenierías[4]; Instituto Tecnológico de Cd Guzmán[5]
CICESE Research Center, Ensenada, Baja California, México[2]
joel.trejo@cimat.mx,jose.lopez@correo.uady.mx,octavio.gutierrez@itam.mx, jramirez@ucaribe.edu.mx, dfajardo@itcg.edu.mx

Abstract. Nowadays, clusters containing multiple GPU nodes are widely used to execute high-performance computing applications. Diverse disciplines use these clusters to improve the performance of several services that consume high computational resources. The challenge of executing high-performance computing applications becomes harder when the applications are executed concurrently and each one of them may demand multiple GPU nodes for different periods of time. To tackle this challenge, we propose a multi-agent architecture for scheduling multiple services in a heterogeneous GPU cluster. We provide simulation results of our agent-based system utilizing three commonly used scheduling heuristics for several configuration settings.

Keywords: High performance computing, GPU cluster, scheduling.

1 Introduction

GPU clusters are attracting attention from the parallel and the scientific computing community due to their computing power that enables the efficient execution of high-performance computing applications. One of the main challenges for these applications is to design efficient implementations capable of fully taking advantage of GPU clusters. In order to tackle this challenge, the designed implementations require an efficient scheduling protocol that equitably distributes the workload among all GPU nodes in a cluster.

There exists several scheduling protocols for distributing the workload in GPU clusters [23, 21]. In grid environments, multi-agent scheduling protocols using distributed architectures have been proposed [4, 3] to efficiently utilize computing resources. However, to our knowledge, there has not been agent-based scheduling protocol managing clusters containing multiple GPU nodes.

In this paper, we propose an agent-based scheduling protocol for scheduling services in a GPU cluster. By using the agent paradigm, agents (as autonomous entities) can interact with each other in order to decide the most feasible allocation according to the profile of the services and the nodes in the GPU cluster. In addition, based on [11, 9], we implement three commonly used scheduling heuristics whose performance evaluation was conducted using simulation.

The structure of the paper is as follows. In Section 2, we describe related work on tasks scheduling in GPU clusters; in Section 3, we present the agent-based scheduling protocol for scheduling services in a GPU cluster and describes some scheduling heuristics; in Section 4, we present the discussion of the simulation results; and finally in Section 5, we give some concluding remarks.

Received Dec 2, 2017 / Accepted Jan 22, 2018

2 Task scheduling in a GPU cluster

Nowadays, clusters containing multiple GPU nodes are widely used to execute high-performance computing applications. The computational power of GPU nodes exceeds the computational power of CPU nodes and this power increases every year. Hence the use of GPU clusters is now popular in the scientific com- munity.

Diverse disciplines use these clusters to improve the performance of several services that consume high computational resources. For instance, Fan, Qiu, Kaufman, and Yoakum-Stover [5] have implemented a parallel architecture based on the lattice Boltzmann model to simulate the dispersion of airborne contaminants. Thibault and Senocak [24] have implemented a multi-GPU 3D Naiver- Stokes solver using the CUDA programming model. Micikevicius [16] has pro- posed a multi-GPU implementation of the discretization of the wave equation. Schive et al. [19] described a GPU cluster with 16 nodes to conduct N-body simulations. More recently, Obrecht, Kuznik, Tourancheau, and Roux [17] presented an implementation of the lattice Boltzmann solver for GPU clusters.

One of the main challenges in mananing a GPU cluster is the distribution of tasks among the different nodes of the cluster. The problem of finding an optimal feasible solution to allocate T tasks into P processors is NP-hard [2].

Nevertheless, there exists some research efforts that consider the problem of distributing tasks among all the nodes in a cluster. Song and Dongarra [23] proposed to use a multi-level distribution method to allocate data to different heterogeneous nodes (either CPU or GPU nodes). The method receives as input a matrix, which is divided into tiles of hybrid size. Then, they use a direct acyclic graph to represent the tasks and their relationships. Their distribution protocol distinguishes between small and large tasks. Small tasks are assigned to CPU nodes and large tasks are assigned to GPU nodes. Their multi-level distribution method is supported by the execution of a runtime system in each node. The runtime system solves data dependencies dynamically. Experimentally, Song and Dongarra explored the performance of their distribution protocol using Cholesky and QR factorizations.

Shirahata, Sato, and Matsuoka [21] designed a hybrid scheduling mechanism to distribute tasks onto CPU and GPU cores. Particularly, they have focused on executing MapReduce tasks in GPU clusters. MapReduce is a programming model for big data processing in clusters. Their system receives a MapReduce task that is assigned to a job tracker. In addition, the system contains a set of task trackers, which monitor the behaviour of each task and create a profile for each of the received task. Using these profiles, the job tracker decides which map task to run on which node (either CPU or GPU). They demonstrated experimentally, that their mechanism outperforms the Hadoop scheduling algorithm when running in a 64-node GPU cluster.

Lang and Rünger [14] propose to distribute the workload (resulting from executing a parallel conjugate gradient method) between both CPU and GPU. Ravi et al. [18] considered two strategies for the distribution of work among the nodes of a GPU cluster. The first one handles tasks that can be executed in a single node (either a CPU or GPU node). The second one handles tasks that require any number of GPU and/or CPU nodes to be executed. In the first scheme, the challenge of the scheduler is to decide on which resource the task should be assigned. In order to assign the task to a resource, the scheduler takes into account the relative multi-core and GPU speedups as well as the expected execution times. In the second scheme, the scheduler groups the submitted tasks based on the resources requested (e.g., number of GPUs or CPUs), then it sorts the requested tasks in each group and assigns the resources according to the order of the submitted tasks. Recently, Wen, Wang, and O'Boyle [25] developed a scheduling mechanism that, based on the structure of the OpenCL code, determines at runtime the expected execution time of tasks either on CPU or GPU nodes. They consider the speedup prediction and the size of the input data to define the scheduling of tasks.

With respect to multi-agent based scheduling mechanisms. Shen, Wang, and Hao [20] have defined the state of the art of agent-based planning and scheduling mechanisms in manufacturing systems. They have described the advantages of using agent-based approaches for tackling the scheduling problem in the manufacturing domain. One of these advantages is that agents' negotiation capabilities help to maximize the efficiency of the scheduling. Take into account this advantage, we propose an agent-based scheduling protocol for scheduling high- performance computing applications in GPU clusters.

In this regard, there is a myriad of agent-based scheduling approaches in the context of distributed and parallel processing. For instance, Chavez, Moukas, and Maes [4] propose a multi-agent scheduling mechanism for CPU allocation. A central agent broadcasts requests to distributed agents, and based on the bids of these agents, the central agent allocates the task to the best option. Cao et al. [3] implemented a three-layered agent-based resource management in grid computing. In [6], Ghosh, Basu, and Das propose a multi-agent system for scheduling tasks supported by game theory. The agent architecture consists of a job allocator and node agents. The job allocator and the node agents play a non-cooperative game with incomplete information. In the context of Cloud computing, Jung and Sim [12] present an agent-based resource allocation protocol. To allocate resources, they take into account both the geographical location of the data center and the current workload.

In this work, we are particularly interested in tasks that can be resolved using a single GPU node in a short period of time. This case is interesting when (i) multiple tasks require the use of the cluster at the same time, and (ii) the time in solving the problem is crucial. We propose an agent-based architecture to schedule these concurrent jobs across the nodes of a GPU cluster. Now, we describe the multi-agent architecture for scheduling high-performance computing applications in GPU clusters.

3 Problem formulation

In this work, we consider tasks that can be executed in a single node; specifically to tasks that can be solved quickly in a GPU cluster, but its execution time takes several minutes in a single CPU. We refer to such tasks as small tasks. Given a small task, it can be classified as low demand, medium demand or high demand. Later in the description of the multi-agent architecture we describe in detail this classification. This approach is interesting when we receive several requests for small tasks in a short period. The time response for this requests is crucial, and it is not quick enough when we only consider a CPU. For instance, algorithms that use numerical computations for solving linear systems to restore images [15] in real time. Our multi-agent architecture considers the problem of scheduling small tasks among the nodes in the GPU cluster. We consider three services that can be requested by a user. We assume that we have a sequential and a parallel implementation of each one of the three services implemented in each of the workstations of the cluster.

- **Matrix-vector Multiplication** Let A be a matrix of size $n \times n$ and *a* vector b of size $n \times 1$. A sequential implementation of this problem requires approximately $O(n^2)$ flops (floating point operations) for large n [7, 13]. We consider a parallel implementation using n threads, each one assigned to a GPU core, that performs $O(n)$ flops in each thread. Since the threads are concurrent, we outperform the computing of this problem to a computational time proportional $O(n)$.
- **Histogram of a digital image**. Computing the histogram of a digital image is useful to enhance such image [8]. Computing the histogram of an 8-bit digital gray-scale image I of size $n \times n$ using the vector of counters h of size 1×256 that represents the discrete function of the histogram, requires $O(n^2)$ flops. We implement in parallel using n^2 concurrent threads of execution, reducing the execution time to $O(1)$, unfortunately, sharing the vector h requires a process of synchronization between the threads that can consume significant time.
- **Matrix-Matrix Multiplication** Let D and B matrices of size n x n. The computing of $DB = A$ requires approximately $2n^3$ flops for large n and its sequential complexity is $O(n^3)$. With parallelization with n (n^2) threads to perform the matrix-matrix computation reduce the complexity to $O(n^2)$ ($O(n)$). We implement both cases in our multi-agent architecture.

The GPU cluster contains workstations with heterogeneous architecture. In this paper, we consider three kinds of workstations.
- A CPU-workstation contains a CPU with eight cores. This workstation is best suitable for low demand tasks.
- A Quadro-GPU workstation contains a GPU with 256 cores. This workstation is best suitable for medium demand tasks.
- A Tesla-GPU workstation contains a GPU with 2096 cores. This workstation is best suitable for high demand tasks.

Let $W_1, W_2, \ldots, W_m$ be a set of n heterogeneous connected workstations, each one with different CPU or GPU capabilities, and let $T_1, T_2, \ldots, T_n$ be a set of n small tasks, that can be allocated to one of the workstations W_i. Each task T_i has an starting time $start_{T_i}$, an estimated sequential execution time $execution_{T_i}$, and a deadline to execution deadline T_i. If a task T_i is not allocated before the $deadline_{T_i}$ units of time after $start_{T_i}$, the task T_i is aborted. Note that a workstation W_i is unavailable if it is serving a task T_j; otherwise, the workstation is idle. The problem consists in assign the maximum number of requests to the workstations, minimizing the waiting time of each one of such requests.

3.1. Multi-agent architecture

To allocate efficiently the set of small tasks in the GPU cluster, we propose a multi-agent architecture that provides simple scheduling strategies. In our architecture, we focus on small tasks. Figure 1 illustrates our agent architecture Multiagent-GPU.

The multi-agent architecture Multiagent-GPU consists of three type of agents: Job agents, Node agents, and Server agents.

Job agent is assigned to a specific small task represented by a job T_i in the system. The job agent communicates with the server agent requesting the allocation of task T_i. The job agent defines a profile of the task T_i. The profile considers the sequential execution time, the deadline and the time of arrival of the T_i. The job agent classifies the task into one of three possible categories. The small task T_i is *low demand* if a CPU-workstation can satisfy its deadline. The small task T_i is *medium demand* if requires at least a Quadro-GPU workstation to satisfy its deadline. The small task T_i is *high demand* if it requires at least a Tesla-GPU workstation to satisfy its deadline. Note that it is possible that

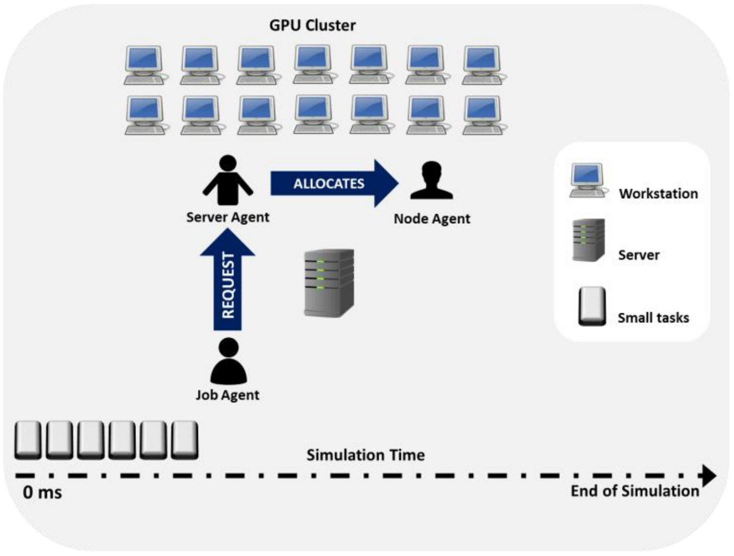

Fig. 1. Multi-agent architecture.

the same task takes different classifications according to the parameters of the small task. For instance, a matrix-matrix multiplication can be classified as low demand if the dimension of matrices is 10×10, as medium demand, if the dimension of matrices is 100×100, and as high demand if the dimension of matrices is 1000×1000.

Server agent represents to the server in the architecture Multiagent-GPU. The server agent receives the request of a service T_i through a job agent. Then, the server agent coordinates with the node agents to determine the best suitable allocation of T_i according to a particular scheduling policy to distribute the request of the job agent representing the task T_i to an available workstation. Finally, the server agent coordinates with the job agent to supervise the job T_i during its timeline in the system. The server agent contains a set of scheduling policies to determine the allocation. Later in this section we describe in detail these scheduling policies.

Node agent represents a workstation node W_i and computes the availability of W_j to the system requirements. The node agent informs to the server agent about its availability. Without loss of generality, we assume that each j serves at most to one service at the same time. Note that the cluster is heterogeneous, i.e., there exists different architectures, some of them more powerful than others. The node agent gathers information about the features of its associated work- station. The node agent estimates the time of execution of the requested task T_i according to the profile of such task. When the server agent requests for a service, the node agent returns the status of the represented workstation (available or unavailable). In case that the workstation W_i is unavailable, the node agent returns the estimated waiting time to the job agent.

Now, we describe in detail the protocol of interaction of the agents for the architecture Multiagent-GPU.

3.2. Interaction of agents

The Figure 2 shows the interaction protocol for the architecture Multi-GPU- Agent

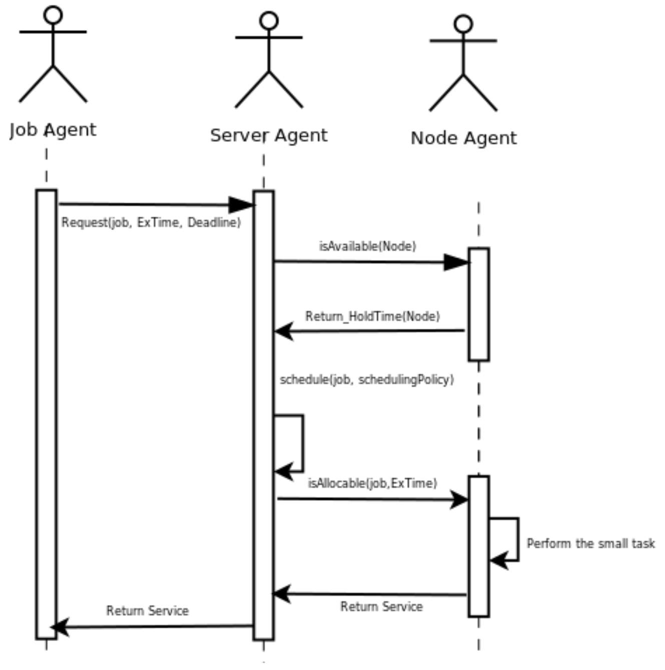

Fig. 2. Multi-agent communication.

When a user requests for a small task T_i service, the user indicates a deadline to get that service. A job agent is assigned to represent T_i. Since there is a set of valid requests, the job agent determines if T_i is a valid small task. If so, the job agent delivers its request to the server agent.

The server agent receives the request from the job agent. Then the server agent broadcast the request to the set of node agents to determine which the most suitable workstation to accomplish the request is. The server agent considers a set of online scheduling heuristics to deliver the request to a subset of node agents. After receiving the response of the subset of node agents, the server agents allocates the task T_i to the best suitable workstation according to their responses. In case there no workstation can serve the request, the task T_i is aborted.

When the node agent, representing the workstation W_j receives a request from the server agent, it first responses with the availability of W_j. If W_j is busy with a previous service, the node agent responds to the server agent with waiting time for the requesting job. In case W_j is available, the waiting time is zero. Afterward, the server agent allocates the small task T_i to the selected workstation W_j.

As mentioned earlier, the server agent performs a set of scheduling policies to the decision making of the small task allocation among the workstations. Now we describe such scheduling policies of our Multiagent-GPU architecture.

3.3. Scheduling policies

We implement a total of three scheduling policies to determine the destination of the small tasks requested by users. We consider the following heuristics based in [11, 9].

In the *round-robin heuristic*. This heuristic consists in allocating the small tasks cyclically as they are arriving at the system. In this heuristic the workstation W_1 receives the small task, workstation W_2 receives small task T_2,…, workstation W_m receives task T_m, workstation W_{m+1} receives task T_{m+1}, and so forth. In a perfect round robin, if there exist *n* small task and *m* workstations, each workstation perform *n/m* tasks.

The Pseudocode 1 presents the round robin heuristic for the Multi-GPU-Agent. In Line 10 of Pseudocode 1 the function *Execution_Time*(T_i) is computed by the node agent assigned in the workstation W_j, and computes the time that W_j requires to perform the small task T_i considering the information of T_i and the resources available in W_j.
In the *best-fit heuristic* when a small task T_i arrives, it is assigned to such workstation W_j that offers the best option when receiving the requesting task. To determine the workstation W_j we use the contract net [22] protocol to the decision making. The server agent broadcast the request to node agents. The node agents respond with their bids, and with the responses, the server agent decides to allocate the task to the workstation with the best fit.

Pseudocode 1: Pseudocode for the round robin heuristic

```
Input  : A set of $n$ small tasks and a set of $m$ workstations
Output: For each task $\mathcal{T}_i$ an allocation of such task in the workstation $\mathcal{W}_j$. For
        each allocated small task $\mathcal{T}_i$ the round robin heuristic returns the
        waiting time for every task
1  begin
2      Let $Q_{\mathcal{W}}$ be a queue of workstations ;
3      Upon server agent receives a task $\mathcal{T}_i$;
4      Let $\mathcal{A}_{w_j}$ be the node agent representing the workstation in the top of $Q_{\mathcal{W}}$ ;
5      Get $\mathcal{W}_j$ from the top of $Q_{\mathcal{W}}$;
6      if $\mathcal{W}_j$ is available then
7          $\mathcal{T}_i.waiting\_time \leftarrow 0$;
8      else
9          Compute the release time ($R_j$) for workstation $\mathcal{W}_j$;
10         $\mathcal{T}_i.waiting\_time \leftarrow R_j + Execution\_Time(\mathcal{T}_i)$;
11         if $\mathcal{T}_i.waiting\_time \leq \mathcal{T}_i.Deadline$ then
12             $\mathcal{T}_i.waiting\_time \leftarrow$ allocate($\mathcal{W}_j, \mathcal{T}_i$);
13         else
14             $\mathcal{T}_i.waiting\_time \leftarrow \infty$;
15         end
16     end
17     Enqueue $\mathcal{W}_j$ to $Q_{\mathcal{W}}$;
18 end
```

Pseudocode 2: Pseudocode for the best-fit heuristic for the server agent $\mathcal{S}_A$

```
Input  : A set of $n$ small tasks and a set of $m$ workstations
Output: For each task $\mathcal{T}_i$ an allocation of such task in the workstation $\mathcal{W}_j$. For
        each allocated small task $\mathcal{T}_i$ the best fit heuristic returns the waiting
        time for every task
begin
    Upon server agent receives a task $\mathcal{T}_i$;
    The server agent $\mathcal{S}_A$ broadcast the profile of $\mathcal{T}_i$ to every node agent in
     MULTI-GPU-AGENT.;
    When receiving the response of all the agents ;
    if If there exists a feasible allocation then
        Select the best-fit workstation $\mathcal{W}_j$ over all the responses ;
        $\mathcal{T}_i.waiting\_time \leftarrow$ allocate($\mathcal{W}_j, \mathcal{T}_i$);
    else
        $\mathcal{T}_i.waiting\_time \leftarrow \infty$;
    end
end
```

Pseudocode 3: Request for allocation of task $\mathcal{T}_i$ to the the node agent $\mathcal{N}_A$

```
Input  : A profile of the small task $\mathcal{T}_i$
Output: Execution time of workstation $\mathcal{W}_j$ for task $\mathcal{T}_i$
begin
    Upon receiving the profile of task $\mathcal{T}_i$;
    Compute the release time ($R_j$) for workstation $\mathcal{W}_j$;
    $\mathcal{T}_i.waiting\_time \leftarrow R_j + Execution\_Time(\mathcal{T}_i)$;
    if If it is a feasible time according to profile of $\mathcal{W}_j$ then
        return $Execution\_Time(\mathcal{T}_i)$;
    else
        return $\infty$;
    end
end
```

Now in Pseudocode 2 and Pseudocode 3 we presents the best-fit heuristic for the server agent and the node agents respectively.

In the First come first serve heuristic the workstations are ordered according to its availability in a list. When a new task T_i arrives, that task is allocated to the workstation W_j in the front of the list. Once the task T_i is allocated to the workstation W_j, such workstation is removed from the list until finishing the processing of T_i. Afterward, W_j is added at the end of the list. This heuristic allows that the faster workstations be more available than the slower ones.

Pseudocode 4: Pseudocode for the first come first serve heuristic in the server side

```
Input  : A set of $n$ small tasks and a set of $m$ workstations
Output: For each task $\mathcal{T}_i$ an allocation of such task in the workstation $\mathcal{W}_j$. For
        each allocated small task $\mathcal{T}_i$, the first come first serve heuristic returns
        the waiting time for every task
1  begin
2      Let $Q_{\mathcal{W}}$ be a queue of workstations sorting starting from the most powerful ;
3      Upon server agent receives a task $\mathcal{T}_i$;
4      Let $\mathcal{A}_{w_j}$ be the node agent representing the workstation in the top of $Q_{\mathcal{W}}$ ;
5      Get $\mathcal{W}_j$ from the top of $Q_{\mathcal{W}}$;
6      Request to the node agent for the estimated time $\phi$ of the task $\mathcal{T}_i$ to the $\mathcal{W}_j$;
7      if $\phi \leq \mathcal{T}_i.deadline$ then
8          Allocate($\mathcal{T}_i, \mathcal{W}_j$);
9      else
10         No allocate;
11     end
12     Assign the task $\mathcal{T}_i$ to the $\mathcal{W}_j$ ;
13 end
```

Pseudocode 5: Allocate the task $\mathcal{T}_i$ to the workstation $\mathcal{W}_j$

```
Input  : A set of $n$ small tasks and a set of $m$ workstations
Output: For each task $\mathcal{T}_i$ an allocation of such task in the workstation $\mathcal{W}_j$. For
        each allocated small task $\mathcal{T}_i$, the first come first serve heuristic returns
        the waiting time for every task
1  begin
2      Upon receiving he profile of task $\mathcal{T}_i$;
3      Perform the task $\mathcal{T}_i$;
4      Enqueue $\mathcal{W}_j$ to $Q_{\mathcal{W}}$;
5  end
```

Now, in Pseudocode 4 we present the pseudocode of the first come first serve heuristic. Note that in contrast with the round robin heuristic, on the first come first serve heuristic, the workstations enqueue to the queue of workstations according to their performance, and not as a cyclic loop. In the next section, we perform simulations to determine performance analysis of the scheduling policies.

4 Discussion and results

Assume that there exist 20 workstations with heterogeneous capabilities. Six CPU-workstations, eight Quadro-workstations and six TESLA-workstations. We perform the simulation of different service requests with different parameters: Matrix-vector multiplication, the histogram of a digital image, and Matrix- Matrix multiplication.

We assume that the speed-up of the Matrix-vector multiplication is approximate S/p, where $S \in O(n^2)$ is the sequential execution time, n is the size of the vector, and p represents the number of cores used during the computing of such service. In the case of the histogram of a digital image the speed up is $S/p + s$ where $S \in O(n^2)$ is the sequential execution time, the image I is of size n x n, and p is the number of cores used, and s is the synchronization factor to implement the histogram of a digital image. The speed-up of matrix-matrix multiplication of $n\times n$ is s/p, where $S \in O(N^3)$ is the sequential time and p is the number of processors.

We compare the throughput of the three policies by performing ten simulations. Each simulation considers two hours of service requests. We measure the average waiting time per service (Figure 3), the average execution time per service (Figure 4), and the number of accepted tasks for each one of the ten simulations (Figure 5). The blue plot represents the round robin heuristic, the red plot represents the best-fit heuristic, and the yellow plot represents the first come first serve heuristic.

As illustrated with Figure 3, Figure 4 and Figure 5, the best performing is obtained with the best fit heuristic, whereas both, the round robin heuristic and the first come first serve heuristic perform similarly. Unfortunately, we noted that the number of tasks assigned is unbalanced with the best fit heuristic, since the scheduler assigns the most of the tasks to the most powerful workstations, whereas the remaining workstations are idle the most of the simulation time.

5 Concluding remarks and future work

We present a multi-agent architecture for task scheduling in a GPU cluster. Three kinds of agents are involved in the scheduling policy. The server agent receives the request from the tasks; these tasks are represented by a job agent, and the workstation are represented by node agents. We perform three scheduling

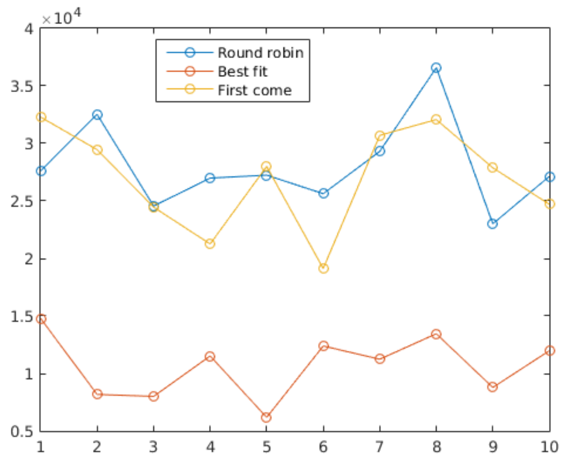

Fig. 3. Average waiting time during ten simulations of two hours each one.

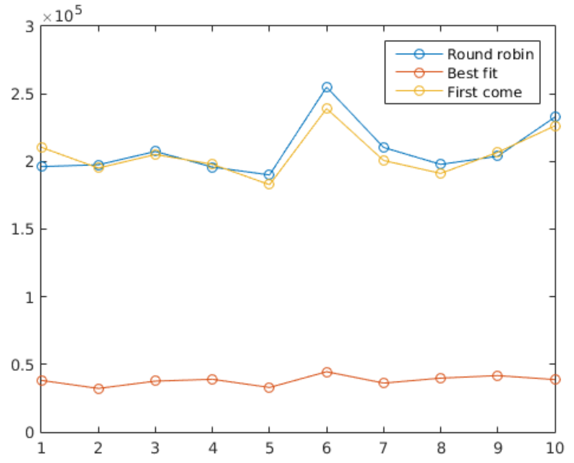

Fig. 4. Average execution time during ten simulations of two hours each one.

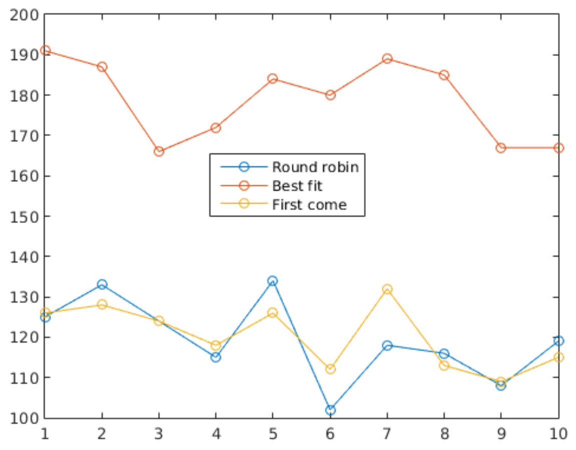

Fig. 5. Number of accepted tasks during ten simulations of two hours each one.

policies. The best performance is obtained with the best fit heuristic, but unfortunately, the amount of work assigned to each workstation is unbalanced. To our knowledge, our work is one of the first agent-based approaches for scheduling in a GPU cluster.

As future work, we will implement more sophisticated scheduling strategies that involve game theory in the negotiation between agents. Additionally, we will also consider large task as input request services. Such strategies will dis- tribute the tasks in such a way that the allocation is balanced among all the workstations.

References

1. Alessandro Agnetis, Jean-Charles Billaut, Stanislaw Gawiejnowicz, Dario Pacciarelli, and A Souhal. Multi-agent scheduling. *Berlin Heidelberg: Springer Berlin Heidelberg*. *doi*, 10(1007):978–3, 2014.
2. Alan Burns. Scheduling hard real-time systems: a review. *Software Engineering Journal*, 6(3):116–128, 1991.
3. Junwei Cao, Stephen A Jarvis, Subhash Saini, Darren J Kerbyson, and Graham R Nudd. Arms: An agent-based resource management system for grid computing. *Scientific Programming*, 10(2):135–148, 2002.
4. Anthony Chavez, Alexandros Moukas, and Pattie Maes. Challenger: A multi-agent system for distributed resource allocation. In *Proceedings of the first international conference on Autonomous agents*, pages 323–331. ACM, 1997.
5. Zhe Fan, Feng Qiu, Arie Kaufman, and Suzanne Yoakum-Stover. Gpu cluster for high performance computing. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 47. IEEE Computer Society, 2004.
6. Preetam Ghosh, Kalyan Basu, and Sajal K Das. A game theory-based pricing strategy to support single/multiclass job allocation schemes for bandwidth-constrained distributed computing systems. IEEE *Transactions on Parallel and Distributed Systems*, 18(3):289–306, 2007.
7. Gene H Golub and Charles F Van Loan. Matrix computations. 1996. *Johns Hopkins University, Press, Baltimore*, MD, USA, pages 374–426, 1996.
8. RC Gonzalez and RE Woods. Digital image processing: Pearson prentice hall. *Upper Saddle River*, NJ, 2008.
9. J Octavio Gutierrez-Garcia and Adrian Ramirez-Nafarrate. Agent-based load balancing in cloud data centers. *Cluster Computing*, 18(3):1041–1062, 2015.
10. J Octavio Gutierrez-Garcia and Adrian Ramirez-Nafarrate. Agent-based load balancing in cloud data centers. Cluster Computing, 18(3):1041–1062, 2015.
11. Heath A James, Ken A Hawick, Paul D Coddington, et al. Scheduling independent tasks on metacomputing systems. In *Proceedings of Parallel and Distributed Computing Systems* (PDCS99), *Fort Lauderdale, Florida*, 1999.
12. Gihun Jung and Kwang Mong Sim. Agentbased adaptive resource allocation on the cloud computing environment. In 2011 *40th International Conference on Parallel Processing Workshops*, pages 345–351. IEEE, 2011.
13. Janusz Kowalik and Tadeusz Pu´zniakowski. *Using OpenCL: Programming Massively Parallel Computers*, volume 21. IOS Press, 2012.
14. Jens Lang and Gudula Rünger. Dynamic distribution of workload between cpu and gpu for a parallel conjugate gradient method in an adaptive fem. *Procedia Computer Science*, 18:299–308, 2013.
15. José L López-Martínez and Vitaly Kober. Blind adaptive method for image restoration using microscanning. IEICE TRANSACTIONS *on Information and Systems*, 95(1):280–284, 2012.
16. Paulius Micikevicius. 3d finite difference computation on gpus using cuda. In *Proceedings of 2nd workshop on general purpose processing on graphics processing units*, pages 79–84. ACM, 2009.

17. Christian Obrecht, Frédéric Kuznik, Bernard Tourancheau, and Jean-Jacques Roux. Scalable lattice boltzmann solvers for cuda gpu clusters. *Parallel Computing*, 39(6):259–270, 2013.
18. Vignesh T Ravi, Michela Becchi, Wei Jiang, Gagan Agrawal, and Srimat Chakradhar. Scheduling concurrent applications on a cluster of cpu–gpu nodes. *Future Generation Computer Systems*, 29(8):2262–2271, 2013.
19. Hsi-Yu Schive, Chia-Hung Chien, Shing-Kwong Wong, Yu-Chih Tsai, and Tzihong Chiueh. Graphic-card cluster for astrophysics (gracca)–performance tests. *New Astronomy*, 13(6):418–435, 2008.
20. Weiming Shen, Lihui Wang, and Qi Hao. Agent-based distributed manufacturing process planning and scheduling: a state-of-the-art survey. IEEE *Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews*), 36(4):563– 577, 2006.
21. Koichi Shirahata, Hitoshi Sato, and Satoshi Matsuoka. Hybrid map task scheduling for gpu-based heterogeneous clusters. *In Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 733– 740. IEEE, 2010.
22. R Smith. The contract net protocol: Highlevel communication and control in a distributed problem solver, 1980. IEEE *Trans. on Computers*, C, 29:12.
23. Fengguang Song and Jack Dongarra. A scalable framework for heterogeneous gpu- based clusters. In *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures*, pages 91–100. ACM, 2012.
24. Julien C Thibault and Inanc Senocak. Cuda implementation of a navierstokes solver on multi-gpu desktop platforms for incompressible flows. *In Proceedings of the 47th AIAA aerospace sciences meeting*, pages 2009–758, 2009.
25. Yuan Wen, Zheng Wang, and Michael FP O'Boyle. Smart multi-task scheduling for opencl programs on cpu/gpu heterogeneous platforms. In *2014 21st International Conference on High Performance Computing (HiPC)*, pages 1–10. IEEE, 2014.