INTERNATIONAL JOURNAL OF COMPUTERS COMMUNICATIONS & CONTROL
ISSN 1841-9836, 13(4), 503-520, August 2018.

A Simulation based Analysis of an Multi Objective Diffusive Load Balancing Algorithm

I.D. Mironescu, L. Vinţan

Ion Dan Mironescu*
Lucian Blaga University of Sibiu, Romania
*Corresponding author: ion.mironescu@ulbsibiu.ro

Lucian Vinţan
Lucian Blaga University of Sibiu, Romania
lucian.vintan@ulbsibiu.ro

Abstract: In this paper, we presented a further development of our research on developing an optimal software-hardware mapping framework. We used the Petri Net model of the complete hardware and software High Performance Computing (HPC) system running a Computational Fluid Dynamics (CFD) application, to simulate the behaviour of the proposed diffusive two level multi-objective load-balancing algorithm. We developed an meta-heuristic algorithm for generating an approximation of the Pareto-optimal set to be used as reference. The simulations showed the advantages of this algorithm over other diffusive algorithms: reduced computational and communication overhead and robustness due to low dependence on uncertain data. The algorithm also had the capacity to handle unpredictable events as a load increase due to domain refinement or loss of a computation resource due to malfunction.
Keywords: Petri Net simulation, High Performance Computing, load balancing, diffusive algorithm, multi-objective optimisation

1 Introduction

1.1 The problem

The main problem approached through our research is how we can optimally distribute the computational effort on an application among the multiple processing units (PUs) of a High Performance Computing (HPC) machine. Computational Fluid Dynamics (CFD) applications [8], which are the focus of our research, perform alternatively intensive computation and communication (data exchange) steps for a high number of points of a discretized computational domain. The hardware on which these applications run is a collection of computing nodes connected through a hierarchical communication network. Each node has a heterogeneous collection of processing units with different processing speed communicating through a hierarchy of global and local memories. Shorter computation time, lower energy consumption and an equilibrated load are all important for the application user and for the hardware operator. Therefore, this represents a multi-objective optimization problem.

1.2 State of the art

Taking into account that the computation platform and the application have many parameters, each one with many distinct discrete values, the load-balancing problem is a NP-hard one [7]. This type of application needs a large-scale distributed computing system. As the underlying numerical method is an approximation method, the computational load can be known completely only at computation time. To handle this problem's particularities, modern solutions use metaheuristic [3], distributed [16] and dynamic load balancer [17]. The scale of the system

and the heuristic behaviour of the used algorithm make modelling and simulation indispensable in the development and assessment of a load balancer algorithm. The modern tools for modelling and simulation of computing systems are using the discrete event formalism [5] [9] [22]. From the current existing methods, Petri Nets presents the advantage of having graphical representation and sound formal definition [2]. The graphical representation facilitates rapid development of models and expressive simulation. The formal definition allows formal verification of the resulted models.

2 The proposed high performance computing system

In this research, we continue and complement our work presented in [20]. The hardware we consider in our approach is a cluster of heterogeneous nodes, implementing a High Performance Computer System (HPC). For the simulations, we modelled the nodes of this HPC after those of an existing system, the COKA cluster [25]. A node from COKA has two CPU sockets and eight PCIe slots. The CPU sockets are occupied by Intel Xeon E5 2630v3 CPUs with 8 cores. Nvidia K80 GPU boards, each having two GPU processors, occupy the PCIe slots. The CPUs and GPUs are communicating through the PCIe bus. Four GPU cards are sharing the same bus to a CPU socket. Each node has two network interfaces – one on each socket; they connect each node to an Infiniband (IB) interconnection network with a fat tree topology. We selected this model because performance and energy consumption models and measurements were presented in [4] for the same class of application we use.
The software runtime allowing the transparent and distributed use of the hardware resources has a two level architecture. The internode level is implemented in the CHARM++ framework [14]. The CHARM++ runtime supports the transparent, asynchronous, message based communication among chares distributed on the system nodes. Chares are the basic entities at this level and follow the Actor paradigm. The heterogeneous node level is managed with the support of the StarPU framework [1]. In this framework, the application is represented as a Directed Acyclic Graph of tasks. The tasks are dispatched to the processing units by a scheduler component that uses a scheduling/load balancing algorithm. Required memory transfers and dependencies between tasks are managed transparently.
The application we designed implements the Lattice Boltzmann Method (LB) CFD numerical method [8]. The computation in LB is an iterative process. In each iteration, for each point of the discrete computational space, two consecutive computational steps are performed: collision (only local data to the point is used) and *propagation* (data from neighbours is needed). The distributed character of the computation is implemented through domain decomposition. At the upper level (CHARM++) the workload is distributed / redistributed among the nodes. The global domain is divided in subdomains. Each subdomain is distributed to a chare, which will manage it until the end of computations. The chare divides the subdomain in blocks, creates for each the corresponding tasks and feeds them to the lower level (StarPU); this dispatches the task on PU and executes them.

3 Proposed load balancer

The proposed load balancing algorithm described in [20] and schematically presented in figure 1 has two levels, corresponding to the software architecture levels. At the CHARM++ level, the load distribution is performed by the chares. In the first iteration, one of the chares receives the domain and starts, as initiator, a negotiation process with other nodes; the negotiations are following the Contract Net Protocol [24]. In each negotiation round the initiator selects its

neighbours from the nodes bidding to its announcement; the initiator then sends to its neighbours bigger subdomains to be further redistributed. The process continues recursively with each responder becoming initiator until no node responds to the call. The resulting neighbourhood relation will be kept until the computation ends and each node will communicate only with its neighbours implementing thus the diffusive character of the algorithm [12]. After receiving its subdomain, each node's chare decomposes it in blocks and creates tasks for the next level. At the StarPU level, the local scheduler dispatches the tasks received from the node's chare to the queues of the node's PUs.

We developed a scheduler based on the dequeue model data aware sorted decision (dmdasd) scheduler [15] with some adaptations. The scheduler is using a cost function representing a weighted sum of the three optimization objectives. The function has the following form:

$$f_{obj} = \alpha \cdot t_{computation} + \beta \cdot E_{computation} + \gamma \cdot Load \tag{1}$$

where f_{obj} is the scheduling function, $t_{computation}$ is the estimated computation time , $E_{computation}$ is the estimated energy consumption, $Load$ is the estimated queue load. The weighting coefficients must satisfy the normalisation (equilibrium) relation $\alpha+\beta+\gamma = 1$. The scheduler estimates the values of the objective function for each PU; it sends the task for computing the block to the PU with the lowest value of the objective function. Our variant includes a supplementary border queue, priority based dispatching and work stealing. Exterior blocks are placed in the border queue based on assigned priority, but they can be transferred to another node if this becomes idle and requests ghost border zones [18]. When all PUs become idle, StarPU signals to the node's chare the end of computation for the current iteration. Then, the chare demands the ghost border zones from its neighbours and its frequency is lowered. If the neighbour responds with a ghost border zone, the computation continues normally; if it responds with a bigger memory zone, the load is redistributed and the neighbour frequency is raised. The receiving chare integrates the new memory zone in its subdomain. If the neighbour is not responding in a pre-set time, it is considered malfunctioning; its subdomain is recovered from backup by its neighbours and recomputed.

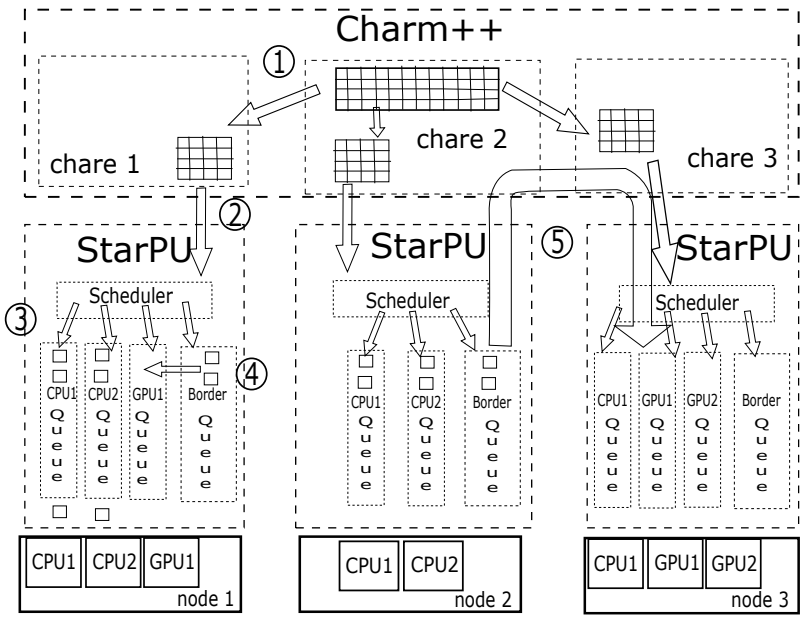


Figure 1: Scheme of the load balancing two level architecture: 1) distribution; 2) task creation; 3) scheduling; 4) work stealing; 5) redistribution

4 Modelling of the system

4.1 Underlying mathematical model

The execution of the HPC application on the given hardware architecture was modelled as a combination of atomic computation and communication steps. These steps can be performed concurrently on the available units. We adopted from literature the models for the time and the energy needed to perform these steps on a given PU [4] [21]. The computation is divided in two phases: *collision* and *propagation*. On the CPU, the unit optimally processed by each core is the 1024x1024 sites block. The time of performing the computation for this number of sites in the *collision* phase (t_{col_CPU}, in s) depends on the clock's frequency (f, in MHz) through the formula:

$$t_{col\_CPU}(f) = \frac{1380}{f}, \tag{2}$$

The energy used in this phase (E_{col_CPU}, in J) is related to the clock's frequency (f, in MHz) through the following relations:

$$E_{col\_CPU}(f) = P_{col\_CPU}(f) \cdot t_{col\_CPU}(f) \tag{3}$$

$$P_{col\_CPU}(f) = 30 + 0.0062 \cdot f^{1+0.2}, [W] \tag{4}$$

The time needed to perform the *propagation* phase for the same block (t_{prop_CPU}) is independent of the frequency:

$$t_{prop\_CPU}(f) = 0.15s \tag{5}$$

The energy used by the CPU in the *propagation* phase E_{prop_CPU} is related to the clock's frequency (f, in MHz) through the following relations:

$$E_{prop\_CPU}(f) = P_{prop\_CPU}(f) \cdot t_{prop\_CPU} \tag{6}$$

$$P_{prop\_CPU}(f) = 30 + 0.055 \cdot f^{1+0.2}, [W] \tag{7}$$

The GPU processes in one step a block of 8192x1024 sites. The time needed by the GPU in the *collision* phase (t_{col_GPU}, in s) depends on the clock's frequency (f, 'in MHz) by the expression:

$$t_{col\_GPU}(f) = \begin{cases} \frac{71.4}{f} & f < 650MHz \\ 0.033 & f \geq 650MHz \end{cases} \tag{8}$$

The energy used in this phase (E_{col_GPU}, in J) is given by the following formulas:

$$E_{col\_GPU}(f) = P_{col\_GPU}(f) \cdot t_{col\_GPU}(f) \tag{9}$$

$$P_{col\_GPU}(f) = \begin{cases} 42.5 + 0.109f & f < 650MHz \\ 42.5 + 0.109f + +0.005e^{0.0099f} & f \geq 650MHz \end{cases} \tag{10}$$

where (P_{col_GPU}) is the power (in W) required for the *collision* step on the GPU. For the *propagation* phase on the GPU, the time needed (t_{prop_GPU}, in s) is:

$$t_{prop\_GPU}(f) = \begin{cases} \frac{27.2}{f} & f < 800MHz \\ 0.085 & f \geq 800MHz \end{cases} \tag{11}$$

The energy used for the *propagation* phase (E_{prop_GPU}, in J) is calculated on the basis of the power required (P_{prop_GPU}, in W) with the relation:

$$E_{prop\_GPU}(f) = P_{prop\_GPU}(f) \cdot t_{prop\_GPU}(f) \tag{12}$$

$$P_{prop\_GPU}(f) = 42.94 + 0.096f \tag{13}$$

The blocks to be processed are in the main memory. Time and energy for accesses to the local memory of the PUs are included in the models presented above. In order to be processed by the GPUs, blocks must be transferred from the main memory (host memory) to the GPU's local memory (device memory). The results should be transferred back from device to host memory. In order to model these transfers over PCIe, we used the model described in [23]; this model adapts the latency (L), overhead (o), short message gap (g), long message gap (G) model to the host-device communication over the PCIe bus. The expression of the transfer time from host to device (t_{hd}) as a function of the number of transferred bytes (B) is given by the following formula:

$$t_{hd}(B) = L_{hd} + o_{hd} + G_{hd} \cdot B + g_{hd} \tag{14}$$

With the values from [23], the expression (14) is:

$$t_{hd}(B) = 0.009420 + 8.318392E - 008B + 0.002503, [ms] \tag{15}$$

The energy used for the host-device transfer ($E_{hd}(B)$, in J) is:

$$E_{hd}(B) = P_{PCI} \cdot t_{hd}(B) \tag{16}$$

P_{PCI} is the power consumed on the PCIe Bus; $P_{PCI} = 25W$ [26].
The transfer time from device to host (t_{dh}) is given by the formula:

$$t_{dh}(B) = L_{dh} + o_{dh} + G_{dh} \cdot B + g_{dh} \tag{17}$$

With the values from [23] the expression is:

$$t_{dh}(B) = 0.009023 + 7.924734E - 008B + 0.002674, [ms] \tag{18}$$

The energy used for the host-device transfer ($E_{dh}(B)$) is:

$$E_{dh}(B) = P_{PCI} \cdot t_{dh}(B) \tag{19}$$

For the network, the LoOgGP model proposed in [19] was used. This model introduces a second overhead (O) supplementary to the latency (L), overhead (o), short message gap (g), long message gap (G) parameters; O is linearly dependent on data size. The transfer time between two nodes over IB (t_{IB}) for B bytes is then:

$$t_{IB}(B) = L_{IB} + o_{IB} + O_{IB} \cdot B + G_{IB} \cdot B + g_{IB} \tag{20}$$

With the values from [19], $t_{IB}(B)$ is (in ms):

$$t_{IB}(B) = 181.5 + 34.7 + 1.88B + 37.9B + 1.88 \tag{21}$$

4.2 Reference model

As reference we developed a model estimating the time and the energy needed to perform one iteration for a given system configuration. A system configuration specifies: the number N_N of nodes and for each node i with $0 \leq i < NN$: the number of PUs of each type ($n_{CPU}(i)$ and $n_{GPU}(i)$); the clock frequencies of each PU($f_{CPU}(j,i)$,$0 \leq j < n_{CPU}(i)$ and $f_{GPU}(k,i)$,$0 \leq k < n_{GPU}(i)$); the number of blocks allocated to each node NB(i). The time of computing a block on the CPU j of the node i ($t_{CPU}(j,i)$) is:

$$t_{CPU}(j,i) = t_{col\_CPU}(f_{CPU}(j,i)) + t_{prop\_CPU}(f_{CPU}(j,i)) \tag{22}$$

where t_{col_CPU} and t_{prop_CPU} are calculated with the formulas (2), respectively (5). The energy is given by the following formula:

$$E_{CPU}(j,i) = E_{col\_CPU}(f_{CPU}(j,i)) + E_{prop\_CPU}(f_{CPU}(j,i)) \tag{23}$$

where E_{col_CPU} and E_{prop_CPU} are calculated with the formulas (3) , respectively (6). If the GPU is alone on the link to the host memory, the total time of computing eight blocks ($Bl_{GPU} = 8 \cdot Bl_{CPU}$ bytes) on the GPU k of the node i ($t_{GPU}(k,i)$) is:

$$t_{GPU}(k,i) = t_{hd}(Bl_{GPU}) + t_{col\_GPU}(f_{GPU}(k,i)) + t_{prop\_GPU}(f_{GPU}(k,i)) + t_{dh}(Bl_{GPU}) \tag{24}$$

where t_{hd} and t_{dh} are the transfer times from host to device and from device to host, respectively, given by equations (14) and (17), t_{col_GPU} and t_{prop_GPU} are the times for computing in *collision* and in *propagation* phase respectively, calculated with the formulas (8) and (11). The energy used to compute eight blocks on the GPU k of the node i ($E_{GPU}(k,i)$) is given by the following equation:

$$E_{GPU}(k,i) = E_{hd} + E_{col\_GPU}(f_{GPU}(k,i)) + E_{prop\_GPU}(f_{GPU}(k,i)) + E_{dh} \tag{25}$$

where the computation energies E_{col_GPU} and E_{prop_GPU} are calculated with the formulas (9) and (12) and the transport energies E_{hd} and E_{hd} are calculated with the formulas (16) and (19). Therefore, for the maximal time needed for the computation on the GPU (t_{max_GPU}), we derived the formula for the case that m GPUs are sharing the same link to memory:

$$\begin{aligned} t_{max\_GPU} = (m \mod 2) \cdot t_{hd}(Bl_{GPU}) + floor(\frac{m}{2}) \cdot t_{hd}(2 \cdot Bl_{GPU}) + t_{col\_GPU}(f_{GPU}(k,i)) \\ + t_{prop\_GPU}(f_{GPU}(k,i)) + ((m-1) \mod 2) \cdot t_{dh}(2 \cdot Bl_{GPU}) + (m \mod 2) \cdot t_{dh}(Bl_{GPU}) \end{aligned} \tag{26}$$

With these considerations, if we denote with $N_{BO}(i)$ the number of blocks which completely occupy the units of the node i, with q the quotient of $N_B/N_{BO}(i)$ and with r the reminder of $N_B/N_{BO}(i)$, the time to compute N_B blocks on node i is given by the following formula:

$$t(N_B,i) = (q+1) * max(t_{CPU}(j,i), t_{max\_GPU}), j = 1, n_{CPU}(i) \tag{27}$$

The energy is given by the equation:

$$\begin{aligned} E(N_B,i) = q * (\sum_{j=1}^{n_{CPU}(i)} E_{CPU}(j,i) + \sum_{k=1}^{n_{GPU}(i)} E_{GPU}(k,i)) + \sum_{k=1}^{\frac{r}{8}} E_{GPU}(k,i) \\ + \sum_{j=1}^{r \mod 8} E_{CPU}(j,i) + \sum_{j=(r \mod 8)+1}^{n_{CPU}(i)} E_{CPUidle}(j,i) + \sum_{k=\frac{r}{8}+1}^{n_{GPU}(i)} E_{GPUidle}(k,i) \end{aligned} \tag{28}$$

where $E_{CPUidle}(j,i)$ is the energy consumed by the CPU j of the node i in idle mode and $E_{GPU}(k,i)$ is the energy consumed by the GPU k of the node i in idle mode.
The CPU and GPU idle energies are calculated with the equations:

$$E_{CPUidle}(j,i) = P_{CPUidle} \cdot t_{CPUidle}(j,i) \tag{29}$$

$$E_{GPUidle}(k,i) = P_{GPUidle} \cdot t_{GPUidle}(k,i) \tag{30}$$

The CPU and GPU idle times are given by the formulas:

$$t_{CPUidle}(j,i) = \begin{cases} t(N_B,i) - (q+1)\cdot t_{CPU}(j,i) & if j \leq r \mod 8 \\ t(N_B,i) - q\cdot t_{CPU}(j,i) & if j > r \mod 8 \end{cases} \quad (31)$$

$$t_{GPUidle}(k,i) = \begin{cases} t(N_B,i) - (q+1)\cdot t_{GPU}(k,i) & if k \leq \frac{r}{8} \\ t(N_B,i) - q\cdot t_{GPU}(k,i) & if k > \frac{r}{8} \end{cases} \quad (32)$$

We considered $P_{CPUidle} = 30$ W and $P_{GPUidle} = 42$ W, the free coefficients from equations (4) and (10).
After all PUs have terminated their local computation, the exchange phase takes place. In this phase, each node transfers the border zones to its neighbours. The averaged dimension (d_{bord}) of the border zone for a subdomain with N_B blocks is:

$$d_{bord}(N_B) = 4\cdot\sqrt{N_B}\cdot 1024\cdot w_{bord}\cdot d_{site}, \text{ bytes} \quad (33)$$

where w_{bord} is the border width (stencil dimension for the computation of one site) and d_{site} is the dimension in bytes of the data for one site. The average is computed considering the square with the equivalent number of blocks. Considering a duplex channel of communication, the time for the node i to be ready for the next iteration ($t_{iter}(i)$) is the time needed to compute the N_B blocks and the time to transfer the border zones:

$$t_{iter}(N_B,i) = t(N_B,i) + t_{IB}(d_{bord}(N_B)) \quad (34)$$

The energy consumed for this is:

$$E_{iter}(N_B,i) = E(N_B,i) + E_{IB}(d_{bord}(N_B)) \quad (35)$$

An iteration is finished when all nodes have completed their transfers. The performance objectives of the whole system for a given distribution of blocks $N_B(i)$, $1 < i \leq N_N$ are: a) the time of computation for one iteration (t_{iter}) (Eq. 36); b) the energy of computation for one iteration(E_{iter}) (Eq. 37), with the idle energy E_{idle} given in (Eq. 38); c) the load imbalance $Li = 1 - Lb$ where the load balancing Lb is given by (Eq. 39) as defined in [27] . We used Li in order to perform the multi-criterial minimisation for all the objectives, as Lb is an objective to be maximized.

$$t_{iter} = \max_{i\in[0...N_N-1]} t_{iter}(N_B(i),i) \quad (36)$$

$$E_{iter} = \sum_{i=0}^{N_N-1} E_{iter}(N_B(i),i) + E_{idle} \quad (37)$$

$$E_{idle} = \sum_{i=0}^{N_N-1} (t_{iter} - t_{iter}(N_B,i))\cdot(\sum_{j=1}^{n_{CPU}(i)} P_{CPUidle}) + \sum_{k=1}^{n_{GPU}(i)} P_{GPUidle}) \quad (38)$$

$$Lb = \frac{\underset{i\in[0..N_N-1]}{AVG} t_{iter}(N_B(i),i)}{t_{iter}} \quad (39)$$

4.3 Petri Net model

The system was modelled using the Petri net formalism. Specifically, Extended Petri Nets were used [2]. The modelling and simulation were performed using the Snoopy tool [9]. Hardware and software components were modelled as described in [20]. Each PU is represented by a token; this token is initially located in the idle place and travels to the other places representing points of the computation reached by the PU. The process of computation is represented by two transitions corresponding to the two phases: propagation and collision. Each of the computation phases introduces a time delay and an energy consumption; to model them, we implemented in the net the relations described in Section 4.1. The frequency of each PU is modelled by tokens residing in the Frequency place; this place is connected to the transitions representing each of the two computing phases, collision and propagation, through a read arc. In this way, transition's delay and number of generated tokens are expressed as function of the number of frequency tokens. The process of modifying the frequency of PU for Dynamic Voltage and Frequency Scaling (DVFS) is simulated by increasing or decreasing the number of tokens in the frequency place. In the CPU model, each transition corresponds to the computation of an elementary block. The delay of the transitions corresponds to the time given by the formulas (2) for *collision* and (5) for *propagation*. For the GPU model, each transition corresponds to the computation of eight blocks representing a GPU block. The delay corresponds to the time obtained with the formulas (8) for *collision* and (11) for *propagation*. The transition for each computing phase produces tokens that accumulate in an *Energy* place. The number of produced energy tokens depends on the number of frequency token through the formulas (3) for collision on CPU, ((6) for propagation on CPU, (9) for collision on GPU and respectively (12) for propagation on GPU. By counting the tokens in the *Energy* place, the energy consumed by each computational process can be evaluated.
The PUs, global memories and network interfaces are connected through a simplified model of the PCIe bus. The tokens flowing through this part of the model represent data packets. Each component connected to the bus can gain the exclusive access to the bus by putting a token in a place which inhibits the transition for other units connected to the bus. The values of the coefficients from the formulas (15) and (18) were used as timing values for the transitions of the CPN model. Overhead (o) is used for the transition representing the preparing of the message. This transition needs a CPU core token in the idle place. Latency (L) is used as a delay in the next transition modelling the transport on the PCI. The bandwidth for a large message (G) is used to calculate the passing rate through the transition. The bandwidth for short messages (g) is used for the transition enabling sending of a new data stream. The transition delay simulates the bus contention. The specific mainboard architecture is also considered; it has a separate link to each processor. In this way, only half of the GPU boards from a node shares the same host-device link.
For the network simulation, the values of the parameter from the formula (21) were used for the transitions delays. The two overheads (o, O) were used in the transmission preparing transition, which also consumes a CPU core token from the idle state. The two gaps (g, G) were used in the transmission transition simulating the network bandwidth and contention.
The application is composed of the scheduler net, which connects to the StarPU run time net, and the load balancer, which connects to the Charm++ runtime. We re-used the Charm++ structure presented in [20]. Details of the scheduler for one PU are presented in Figure 2. The scheduling is done by computing an objective function for each PU. The next arriving task is dispatched to the PU with the minimal value for the objective function. The computation of the objective function is implemented through a net simulating the addition of values through the accumulation of corresponding number of tokens. For each PU, one such net was build. This net has an accumulator place for each optimisation criteria. The place contains the current value

of the criteria for the PU. For each task in the waiting queue and for each PU, a number of tokens are generated for energy consumption, completion time and load. The number of tokens corresponds to the estimated values of the criteria for the task execution on the PU. These tokens are transferred to an estimation place. An amount equal to the tokens for the current values is also transferred in the same place. This gives the estimation of the cumulative values for energy, completion time and load after the task execution on the given PU. All estimated values are connected to a cumulative transition. This transition functions like a minimum operator. It transfers tokens until the place with the smallest number of tokens becomes empty. The empty place activates the scheduling transition. The transition transfers the task token from the waiting queue to the corresponding PU queue. The accumulator places of the selected PU are then updated with the number of tokens corresponding to the scheduled task. If a frequency change takes place, the computational speed and the energy used by the PU are changing, meaning that the estimation for the completion time and energy for the tasks still in queue are not accurate any more. The number of tokens in energy and completion time accumulators places are increased correspondingly. In addition, if a task is transferred from queue to execution, a token is removed from the load accumulator place. By explicitly modelling the scheduling process, it is possible to model the incurring overhead. Each operation corresponding to a computational process is represented through a transition. The transition is enabled only if a token for a CPU core is present in the idle core place; it consumes this token and returns it to the idle place only after the processing time passed.

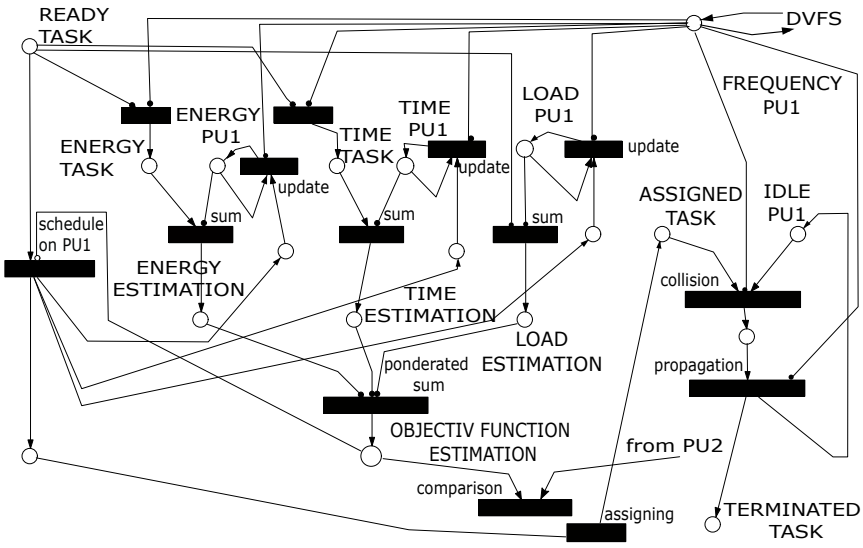


Figure 2: Schematics of the scheduling function for a PU

5 Simulation

5.1 Computing system modelled

We modelled a system with 16 nodes. To give relevance to the computational case, each node was configured differently from the other in respect to the available CPUs and GPUs. This configuration becomes relevant in the context of the partition of computing resources among different applications [10] or in the context of temporary or permanent resource unavailability. In both cases, even if the nodes are identical at start, the resources available to an application differ from a node to the other. The configuration for each node is given in Table 1, where

Table 1: Nodes configuration

Node i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$n_{CPU}(i)$	8	8	8	8	8	8	8	8	16	16	16	16	16	16	16	16
$n_{GPU}(i)$	2	4	6	8	10	12	14	16	2	4	6	8	10	12	14	16
$NBO(i)$	23	38	53	68	83	98	113	128	31	46	61	76	91	106	121	136

nCPU(i) and nGPU(i) are the number of PUs of each type and NBO(i) the number of blocks completely occupying the units of the node i.

Considering the optimal blocks load of each PU given in [4] and that a CPU core is needed to manage the operation of a GPU board (=two processors), $N_{BO}(i)$ is given by the formula:

$$N_{BO}(i) = n_{CPU}(i) - \frac{n_{GPU(i)}}{2} + 8 \cdot n_{GPU}(i) \tag{40}$$

5.2 Design space

The input variables for the three objective functions of the multi-criterial optimisation problem are the partition of the LB grid in 16 subdomains and the running frequency for each PU of the system. The dimension of the design space (i.e. the number of possible hardware-software combinations) is equal to the number of possible partitions x number of possible frequency configurations. The number of possible frequency configurations is equal to ((number of CPU frequency levels) x (number of GPU frequency levels))^{16}. Even by considering only the frequencies effectives for DVFS (as shown in [4]), we have four levels for CPU and 12 levels for GPU giving a set of 48^{16} possible configurations. The number of possible partitions depends on the dimension of the partition unit(the smallest part to be transferred between nodes). If a node is idle, the minimal optimal load to be transferred to it will be of 16 blocks(the load of a GPU card with two GPU processor). The node 15 can host maximal 256 blocks over its optimal load of 136; this gives 256/16 = 16 units of partition. For the 16 nodes, we determined experimentally 300540195 possibilities of distributing these 16 units. This gives a total of $48^16 \cdot 300540195 \approx 10^{35}$ possible configurations. This design space obviously cannot be explored exhaustively in order to obtain the Pareto set.

5.3 Reference surface

To have a measure of the algorithm efficiency, we built an approximation of the Pareto surface in the space of the three objectives: energy consumption, completion time and load balancing. We designated this surface as reference. Given the design space dimension, we used a metaheuristic algorithm to generate a reference surface approximating the Pareto set. We preferred this approach to other metaheuristics (e.g. NSGA2 multi-objective genetic algorithm) because it can find points in the Pareto set like the configurations using minimal energy and the configurations with minimal computing time. The pseudocode is presented in Algorithm 1

To reduce calculation, we used a dictionary type structure *perf_ data* to store the values for the time needed and energy consumed by each individual node for the computing of each of the possible subdomain's dimension (in blocks), with each of the possible frequency configurations. The values were calculated with formulas (27) and (28) from Section 4.2. The dictionary allows the retrieving for each node i and for each subdomain dimension B of the minimal computing time (t_{min}) and corresponding energy ($E_{t_{min}}$), of the minimal energy (E_{min}) and corresponding time ($t_{E_{min}}$) and of the corresponding frequencies configurations, $freq_{t_{min}}$ and $freq_{E_{min}}$. We obtained a set of starting points by generating a manageable set of partitions. We considered

Algorithm 1 Metaheuristic search

generate perf_data dictionary
generate schedules
for all schedules, S **do** ▷ global candidate selection
  $(freq_{t_{min}}, t_{min}, E_{t_{min}}, Li_{t_{min}}) \leftarrow$ generate tmin configuration$(S, perf_data)$
  $(freq_{E_{min}}, t_{E_{min}}, E_{min}, Li_{E_{min}}) \leftarrow$ generate Emin configuration$(S, perf_data)$
  if $insert(ref\_set, (S, freq_{t_{min}}, t_{min}, E_{t_{min}}, Li_{t_{min}}))$ **then**
    $new\_freq \leftarrow freq_{t_{min}}$
    repeat ▷ local search for candidates that minimise Li
      $new\_freq \leftarrow decrease_frequency(new\_freq)$
      $(t_{new}, E_{new}, Li_{new}) \leftarrow generatenewLiconfig(S, new\_freq, perf\_data)$
    until $insert(ref\_set, (s, new\_freq, t_{new}, E_{new}, Li_{new}))$
  end if
  if $insert(ref\_set, (S, freq_{E_{min}}, t_{E_{min}}, E_{min}, Li_{E_{min}})$ **then**
    $new\_freq \leftarrow freq_{E_{min}}$
    repeat ▷ local search for candidates that minimise Li
      $new\_freq \leftarrow increase_frequency(new\_freq)$
      $(t_{new}, E_{new}, Li_{new}) \leftarrow generatenewLiconfig(S, new\_freq, perf\_data$
    until $insert(ref\_set, (s, new_c\_Emin, t_new, E_new, li_new)$
  end if
end for

packets of 16 blocks as the partition units and we generated all possible distributions of these 16 units on the 16 nodes (300540195 schedules). The reference set *ref_ set* is the structure that will contain the approximation of the Pareto set at the algorithm completion. For the *ref_ set* we used a dictionary of dictionaries structure. Each of the three optimization objectives is key at one level. The keys are sorted, allowing efficient comparison of a given element with the set members. This is used by the insert operation testing the argument element before insertion. If the element is not dominated, it is inserted in the set and the operation returns true. If the element is dominated, it is discarded and the operation returns false. If the element dominates a set member, this member is deleted from *ref_ set*. Using the same algorithm, we built reference sets with less blocks by decreasing the dimension of the partition unit. These configurations were used as starting point for the investigation of load increase case (see Section 5.5). With a similar process, reference sets were calculated for partitions on 10 to 15 nodes to be used for the node failure case (see Section 5.6).

5.4 Initialization case

The simulation aims to test if the developed algorithm can achieve a mapping (domain decomposition plus scheduling) near to the reference set and how fast it can reach this mapping. An initialization message is sent to one of the nodes with the location and dimension of the complete domain. The node starts the Contract Net Protocol initiation. After receiving biddings, the starting node divides the domain, considering its processing power (type and number of PUs) and the processing power of the bidding nodes. The distribution phase continues recursively until no nodes are bidding. Each node starts computation in parallel with redistribution. The computations start with the blocks situated at the domain core, allowing the outer blocks to be redistributed. The normal redistribution takes place automatically at the end of each iteration. A simulation was run for each node as first node.

5.5 Load increase case

The simulation goal was to show how the algorithm reacts in case of a dynamic increase in load; this situation arises when the calculation imposes the mesh refining in order to increase computation's accuracy. Refining is done by halving the distance between the grid sites (one step of refining). The number of sites in each direction is doubled. The total number of block sites is multiplied by four. We started with a configuration that is on the reduced reference set and increased four times the load of some selected nodes. In the CPN model, the load increase was simulated by marking the places corresponding to the selected nodes with four times more tokens than in the reference mapping. The effect of a single affected node was investigated at first. Each node was loaded with four times its complete load indicated in Table 1. The cases of multiple nodes were then investigated. As reference we used the reference set computed for the maximal number of sites. As starting configuration we used a configuration in the reference set for a lower number of sites.

5.6 Node failure case

In this case, we investigated how the load balancing will react to a node failure. We used the reference set computed with the same number of sites for a system with fewer nodes. In the simulation case, a scheduled transition injects at a random time a token, blocking the functioning of one or more nodes. The first goal is to find if the system could recover after losing one or more nodes; the second one is to see in how much time the mapping reaches the reference surface for the corresponding number of nodes.

6 Results and discussion

6.1 Initialisation case

In the first step, the chare on the init node distributed the data to eight other nodes with the shortest path to it. Each of these nodes already have some fixed neighbours (the node from which it received the task and some of the nodes that accepted the task.) The recursive process ends when no other nodes respond or a node has a subdomain that it can certain compute.
Figure 3 presents one example of the distribution process for a 4x4 grid. In each step, the announcing nodes are in blue, the nodes that have finished distribution and started computing are in green and the nodes that cannot distribute further are in red. The arrows are presenting established contracts.

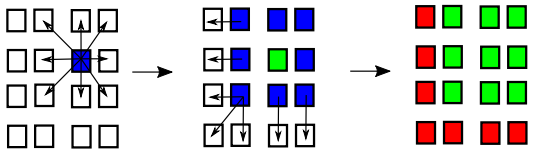


Figure 3: Initial distribution of load on available nodes

After all nodes start calculation, blocks are also exchanged in the communication step to balance the load. This happens until the computation time is approximately equal on all nodes. After the computations were started, the simulations were run until a stable mapping (meaning no full block exchange) took place in the communication step. The simulations were run for each node as starting node. A mean of 40 iterations are needed for reaching a grid distribution that is near to the reference set.

Figure 4 presents one of the reached mapping as a red point, compared with the reference set in blue. We considered the time and the used energy as more important than the load; so, we selected the values $\alpha = 0.40$; $\beta = 0.40$; $\gamma = 0.2$ for the weighting coefficients of the objective function given by formula (1). Therefore we expected that the resulted mapping will be nearer to the points representing also the weighted combination of the three objectives on the reference set. The cause of not reaching the configuration on the reference set is that the redistribution stopped before reaching the optimal schedule. The faster node has finished earlier and requested the border block. The request arrived only after the slower node started to process the last blocks so they were not transferred anymore.

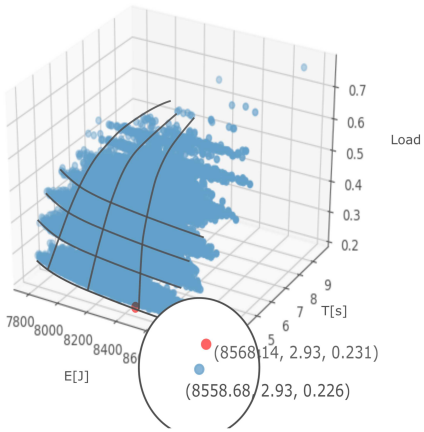
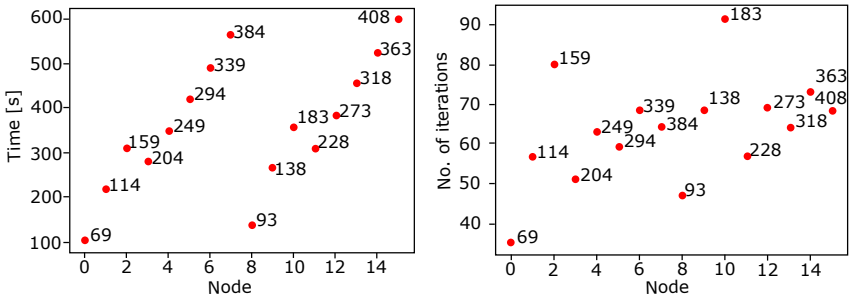


Figure 4: Reference set (blue) with stable configuration (red) reached after redistribution

6.2 Load increase case

Figure 5 presents time (a) and iteration number (b) needed to reach the proximity of the Pareto front for a single affected node.



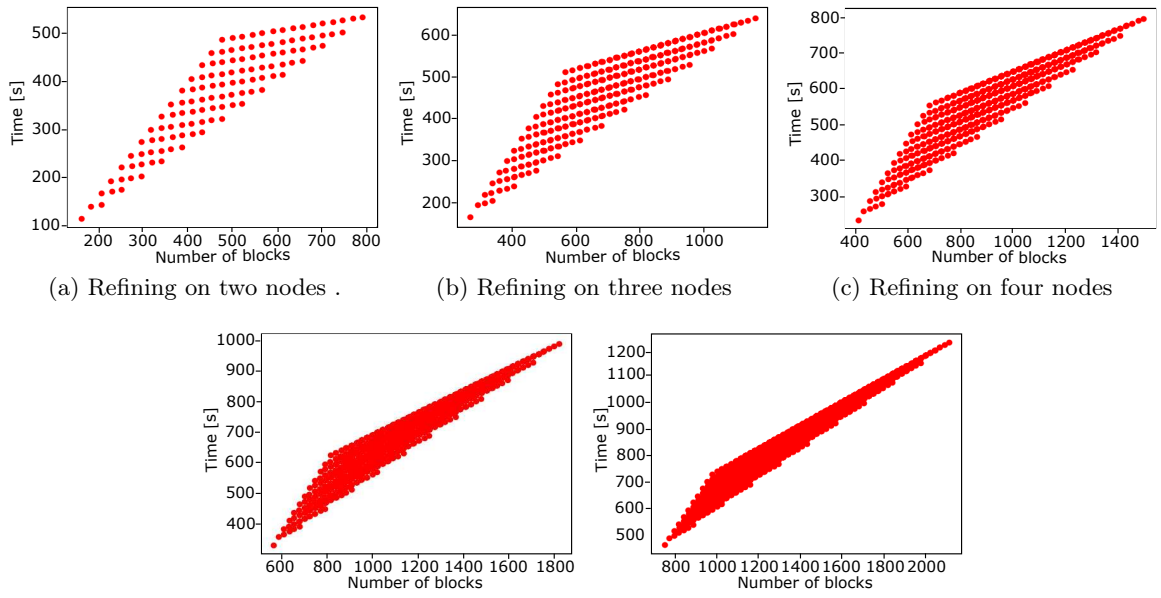
(a) Time needed to reach the near optimal solution.

(b) Number of iterations needed to reach optimal solution.

Figure 5: Handling of the load increase on one node. Point label indicates number of blocks to be transferred

The numbering of the nodes are in the order from the slowest node (node 0 only two GPUs available) to the fastest node (node 15 - all possible GPUs=16 available). In each case, the affected node was considered to have the minimal optimal load ($N_{BO}(i)$ from Table 1) before the refinement was started. In Figure 5, each point is labelled with the number of blocks to be

transferred which is approximated to 3*optimal load. Point annotation is the number of supplementary blocks to be redistributed (3*optimal load, see Table 1). The break in the ascending trend at the nodes 3 and 11 can be explained by some architectural details. Starting with the two nodes, there is a better distribution of GPUs on the CPU-GPU interconnect network. In this way, the transfer time between the host and devices is smaller for the same amount of data. The time needed to reach a stable near optimal mapping, in the proximity of the reference set, for the refinement started in two(a) to six(f) nodes is presented in Figure 6. Number of blocks on the x-axis is the total number of blocks to be redistributed from the affected node. All possible distinct combinations of two, three, four, five and respectively six nodes that can be formed from the 16 nodes, were simulated. As the configurations are limited, there are more combination



(a) Refining on two nodes . (b) Refining on three nodes (c) Refining on four nodes

(d) Refining on five nodes (e) Refining on six nodes

Figure 6: Handling of the load increase on 2 to 6 nodes

of nodes with the same total number of blocks. Depending of the node combination, we have different timings for the same total point number (grid sites). This is the result of their hardware configuration and of the number of neighbours from each of them. The time is increasing with the number of nodes as they share the same communication bandwidth.

6.3 Node failure case

Figure 7 shows the time until the stable, near optimal mapping is reached depending on the number of points to be redistributed. The points label indicates the number of defective nodes. Three classes of nodes failure behaviour are observed:

- slow recovering nodes: 0, 1, 3, 13 and 15;
- intermediate recovering nodes: 8, 9, 2, 10, 4, 12, 6, 14 and 7;
- fast recovering nodes: 11 and 5.

The three classes are given by the variation in the number and the hardware configuration of the node neighbours. Failure in nodes with many neighbours with better architectural characteristics

(such as nodes 11 and 5) is recovered faster than failure in nodes that have less and/or suboptimal neighbours (such as nodes 0 and 5).

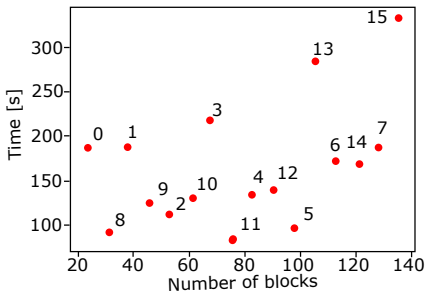


Figure 7: Time for the redistribution of the load of a failed node. The label of each point indicates the node that has failed.

7 Conclusion

We have investigated through simulation the behaviour of the proposed algorithm for the multi-objective optimization of the workload distribution on a heterogeneous computing system. The simulations have shown that the algorithm can perform as expected. The algorithm succeeds:

- To distribute the workload by passing it from neighbour to neighbour. The maximal number of iterations needed is equal to the maximal distance between the starting node and the other nodes in the tree of neighbourhood that is forming;
- To reach a stable state in the proximity of the reference set of a tri-objective optimisation problem. The factors influencing the iteration number and the time needed for reaching a stable configuration were also revealed by this investigation. They are the hardware configuration and the position in the node neighbourhood hierarchy/topology. From the hardware perspective, the number and distribution of GPUs on the available slots is determined. The optimum is for a moderate number of GPUs uniformly distributed on the slots, to fully exploitation of the communication hierarchy. From the topological perspective, nodes with fewer neighbours recover slower from the increase of load as they cannot efficiently evacuate the load;
- To handle well sudden load increases caused by mesh refining or node failure.

The behaviour observed through the simulation reveals that our algorithm has some advantages over other algorithms:

- It does not need an explicit, distinct imbalance check. This is implicitly detected by the slower node that receives a border zone request from is faster neighbours. The load balancing is also resolved with the transfer that is already taking place only with an increased payload. Therefore, our developed algorithm has a speed and resource advantage over all algorithms that make an explicit balance check and have an explicit load balancing phase;
- It can detect and solve imbalances resulted from special situations, like mesh refining or node failure without explicit checks. The detection is implicitly codified in the exchanges protocol between neighbours. After the supplementary load is dispatched to some queues, the load balancing is taking place like in a normal situation. The consideration of special cases- refining and failure - is an original aspect. This is important because these cases are unexpected, but not rare for the application investigated;

- It uses less uncertain data. It starts the transfer only when the imbalance is certain and directs the data only to the nodes that can handle the supplementary load. So, our proposed algorithm is more robust than algorithms basing their decision exclusive on estimations. The second level scheduler uses estimates in its decision, but at this level predictability is higher. At first level, the decision of transferring the load is not based on estimations but triggered by a real imbalance;

- It follows three objectives for optimisation, instead of classical solution that considers only time and energy [6] [13]. Analysis of computing load is also important in the context of system cooling and of a datacenter reliability.

The heuristic algorithm used to approximate the Pareto surface is also original and has showed potential at least for producing reference sets and for off-line scheduling. It must be also further analysed. More investigation has to be done on the model of a system with a much higher number of nodes in the magnitude order of a real HPC configuration. This will be done using a simulator that can handle this number of nodes and a metaheuristic automatic design space tool (FADSE) [11] to build the reference set.

Conflict of interest

The authors declare no conflict of interest.

Bibliography

[1] Augonnet, C.; Samuel, T.; Namyst, R.; Wacrenier, P.-A. (2011); StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures, *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, 23, 187-198, 2011.

[2] Blätke, M.A.; Heiner, M.; Marwan, W. (2015); Engineering with Petri Nets, In *R. Robeva (Ed.), Algebraic and Discrete Mathematical Methods for Modern Biology*, Elsevier Inc., 141–193, 2015.

[3] Brahambhatt, M.; Panchal, D. (2015); Comparative Analysis on Heuristic Based Load Balancing Algorithms in Grid Environment, *International Journal of Engineering Research & Technology (IJERT)*, 4(4), 802–806, 2015.

[4] Calore, E.; Gabbana, A.; Schifano, F.S.; Tripiccione, R. (2017); Evaluation of DVFS techniques on modern HPC processors and accelerators for energy-aware applications, *Concurrency and Computation: Practice and Experience*, DOI: https://doi.org/10.1002/cpe.4143, 29(12), 1–19, 2017.

[5] Casanova, H.; Giersch, A.; Legrand, A.; Quinson, M.; Suter, F. (2014); Versatile, Scalable and Accurate Simulation of Distributed Applications and Platforms, *Journal of Parallel and Distributed Computing*, DOI: https://doi.org/10.1016/j.jpdc.2014.06.008, 74(10), 2899–2917, 2014.

[6] Chatterjee, N.; Paul, S.; Mukherjee, P.; Chattopadhyay, S.(2017); Deadline and energy aware dynamic task mapping and scheduling for Network-on-Chip based multi-core platform, *Journal of Systems Architecture*, DOI: https://doi.org/10.1016/j.sysarc.2017.01.008, 74, 61–77, 2017.

[7] Chen, B.; Potts, C.N.; Woeginger, G.J. (1998); A Review of Machine Scheduling: Complexity, Algorithms and Approximability, In *D.Z. Du, P.M. Pardalos (Eds.), Handbook of Combinatorial Optimization*, Springer, 21–129, 1998.

[8] Guo, Z.; Shu, C.(2013); *Lattice Boltzmann Method and Its Applications in Engineering Advances in computational fluid dynamics Volume:3*, World Scientific, 2013.

[9] Heiner, M.; Herajy, M.; Liu, F.; Rohr, C.; Schwarick, M.(2012); Snoopy - a unifying Petri net tool, In *S. Haddad, L. Pomello, (Eds.) Application and Theory of Petri Nets*, Springer, 7347, 398–407, 2012.

[10] Hugo, A. E.; Guermouche, A.; Wacrenier, P.A.; Namyst, R. (2013); Composing Multiple StarPU Applications over Heterogeneous Machines: A Supervised Approach, In *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*, 1050–1059, 2013.

[11] Jahr, R.; Calborean, H.; Vintan, L.; Ungerer, T. (2012); Finding Near-Perfect Parameters for Hardware and Code Optimizations with Automatic Multi-Objective Design Space Explorations, In *Concurrency and Computation: Practice and Experience*, DOI: 10.1002/cpe.2975, 27(9), 2196–2214, 2012.

[12] Jeannot, E.; Vernier, F., (2006); *A Practical Approach of Diffusion Load Balancing Algorithms*, INRIA, RR5875, 2006.

[13] Juarez, F.; Ejarque, J.; Badia, R.M. (2018); Dynamic energy-aware scheduling for parallel task-based application in cloud computing, *Future Generation Computer Systems*, 78, 257–271, 2018.

[14] Kale, L.V.; Bhatele, A.; (2013); *Parallel Science and Engineering Applications: The Charm++ Approach (1st ed.)*, CRC Press, 2013

[15] Kasmi, N.; Zbakh, M.; Samadi, Y.; Cherkaoui, R.; Haouari, A. (2017) Performance evaluation of StarPU schedulers with preconditioned conjugate gradient solver on heterogeneous (multi-CPUs/multi-GPUs) architecture, In *3rd International Conference of Cloud Computing Technologies and Applications (CloudTech)*, 1–6, 2017.

[16] Kaur, N.; Chhabra, A. (2017); Comparative Analysis of Job Scheduling Algorithms in Parallel and Distributed Computing Environments, *International Journal of Advanced Research in Computer Science*, 8(3), 948–956, 2017.

[17] Khan, S.; Nazir, B.; Khan, I. A.; Shamshirband, S.; Chronopoulos, A. T. (2017); Load balancing in grid computing: Taxonomy, trends and opportunities, *Journal of Network and Computer Applications*, 88, 99–111, 2017.

[18] Kjolstad, F.B.; Snir, M.(2010); Ghost Cell Pattern, In *Proceedings of the 2010 Workshop on Parallel Programming Patterns (ParaPLoP'10)*, DOI=http://dx.doi.org/10.1145/1953611.1953615, 4, 2010.

[19] Martinez, D. R.; Cabaleiro, J.C.;Pena, T.F.; Rivera, F.F.; Blanco,V. (2009); Accurate analytical performance model of communications in MPI applications, In *2009 IEEE International Symposium on Parallel & Distributed Processing*, DOI: https://doi.org/10.1109/IPDPS.2009.5161175, 1–8. 2009.

[20] Mironescu, I.D.; Vintan, L. (2017); A task scheduling algorithm for HPC applications using colored stochastic Petri Net models, In *Proceedings of 13th International Conference on Intelligent Computer Communication and Processing*, 479–486, 2017.

[21] Rauber, T.; Rünger, G.; Schwind, M.; Xu, H.; Melzner, S. (2014); Energy measurement, modeling, and prediction for processors with frequency scaling, *The Journal of Supercomputing*, 70, 1451–1476, 2014.

[22] Ubal, R.; Byunghyun, J., Mistry, P.; Schaa, D.; Kaeli, D., (2012); Multi2Sim: a simulation framework for CPU-GPU computing, In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques (PACT '12)*, ACM, 335–344, 2012.

[23] van Werkhoven, B.V.; Maassen, J.; Seinstra, F.J.; Bal, H.E. (2014); Performance Models for CPU-GPU Data Transfers, In *14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 11–20, 2014.

[24] Wu,J.; Contract Net Protocol for Coordination in Multi-Agent System, In *2008 Second International Symposium on Intelligent Information Technology Application*, doi: 10.1109/I-ITA.2008.273, 1052-1058, 2008.

[25] [Online]. Available: http://www.fe.infn.it/coka/doku.php?id=start, Accesed on 26 february 2018

[26] [Online]. Available: https://pcisig.com/specifications/pciexpress/base2/, Accesed on 26 february 2018

[27] [Online]. Available: https://pop-coe.eu/node/69, Accesed on 26 february 2018