Noname manuscript No.
(will be inserted by the editor)

An Study of the Effect of Process Malleability in the Energy Efficiency on GPU-based Clusters

Sergio Iserte · Krzysztof Rojek

Received: date / Accepted: date

Abstract The adoption of Graphic Processors Units (GPU) in high-performance computing (HPC) infrastructures is determining, in many cases, the energy consumption of those facilities. For this reason, an efficient management and administration of the GPU-enabled clusters is crucial for the optimum operation of the cluster. The main aim of this work is to study and design efficient mechanisms of job scheduling across GPU-enabled clusters by leveraging process malleability techniques, able to reconfigure running jobs, depending on the cluster status. This paper presents a model that improves the energy efficiency when processing a batch of jobs in an HPC cluster. The model is validated through the MPDATA algorithm, as a representative example of stencil computation used in numerical weather prediction. The proposed solution applies the efficiency metrics obtained in a new reconfiguration policy aimed at job arrays. This solution allows the reduction of the processing time of workloads up to 4.8 times and reduction of the energy consumption up to 2.4 times the cluster compared to the traditional job management, where jobs are not reconfigured during their execution.

The researcher from Universitat Jaume I (UJI) was supported by the project TIN2017-82972-R from MINECO and FEDER. The National Polish Science Centre supported the researcher from Czestochowa University of Technology under grant no. UMO-2015/17/D/ST6/04059 and under grant no. UMO-2017/26/D/ST6/00687. This work was partially performed during a short term scientific missing (STSM) from Krzysztof Rojek to UJI supported by the EU COST IC1305. Authors are also grateful to the BSC for letting them use the HPC facilities.

S. Iserte
Universitat Jaume I, Castelló de la Plana, Spain
Universitat de València, València, Spain E-mail: siserte@uji.es

K. Rojek
Czestochowa University of Technology, Czestochowa, Poland
byteLAKE S.C., Wroclaw, Poland

Keywords Dynamic Resource Management · Job Reconfiguration · MPI Malleability · Job Array-aware Scheduling · MPDATA Algorithm · Heterogeneous Programming Model

1 Introduction

The current trend in high-performance computing (HPC) facilities is towards the centralization of large amounts of resources in a data center, which, in turn, are shared by thousands of users. Users submit their applications to the system as jobs that request computational resources. In production systems, it is easy to find a wide variety of jobs with different resource requirements: from single-node jobs that may even share the same host, to massive jobs running exclusively over significant portions of the data center. Jobs request resources to the system in order to be initiated. Those requests are defined by users who know the characteristics of their jobs. Furthermore, jobs are not homogeneously submitted across time [15]. Some users can sporadically submit jobs to the system, while others periodically submit bursts of jobs. This latter case, for example, can be easily caused by a job which, at each iteration, spawns a new set of independent jobs with different input data but with the same resources requirements. This specific kind of bursts of jobs are viewed as *job arrays* by many resource manager systems (RMS) such as Slurm[1], SGE[2], MOAB[3], etc.

This research aims to realize a highly efficient management of the cluster resources in order to improve the global productivity, in terms of completed jobs per unit of time, when a collection of similar jobs is submitted to the system in burst mode. For this purpose, malleability methods are leveraged in order to reconfigure jobs during their execution. In other words, these techniques dynamically reallocate the assigned resources as well as change the number of processes in charge of executing the job. Although process malleability has been integrated into different types of applications such as: master-worker [4], non-iterative [10] or benchmarks [17], the main target of malleability are iterative applications [7] since they present clear processes synchronization points where job reconfigurations can be easily triggered [5,11,16,26]. Malleability has been also implemented with different approaches and frameworks, such as: non-standard MPI with ULFM [14], Checkpoint-restart [6], CHARM++ [8], Java virtual machine [25], etc. Concretely, this research relies on the dynamic management of resources (DMR) process malleability framework [12], a standard MPI-based solution which provides a modular design that allows its integration with other programming models, such as CUDA. Indeed, this is the crucial difference that distinguishes DMR from other malleability solutions that present a very high level of coupling among their components or are bound to an ad-hoc programming model.

[1] https://slurm.schedmd.com/job_array.html
[2] http://wiki.gridengine.info/wiki/index.php/Simple-Job-Array-Howto
[3] http://docs.adaptivecomputing.com/mwm/7-0/Content/topics/jobAdministration/jobarrays.html

This paper presents a malleable version of the multidimensional positive definite advection transport algorithm (MPDATA), which is one of the main parts of the dynamic core of the Eulerian/semi-Lagrangian fluid solver (EULAG). Following the architecture of systolic procurement [13], EULAG submits MPDATA job arrays during its execution, making this an ideal candidate to study the efficiency of the resource manager in the event of a burst of jobs [19,27]. As DMR is based on the Slurm workload manager [28], it has been possible to design a novel job reconfiguration policy that extends Slurm in order to improve the productivity and energy efficiency for job arrays.

In summary, this paper reports MPDATA as the first GPU-capable malleable application developed and evaluated [22]. The proposed solution is a systematic procedure to balance the jobs across all the nodes. Moreover, it also demonstrates that the DMR malleability framework can be efficiently adapted to heterogeneous programming models, such as CUDA;

The rest of the paper is structured as follows: Section 2 briefly describes the MPDATA application and the DMR framework. Section 3 develops a theoretical study of an energy-efficient scheduling policy for executing job arrays on a GPU-based cluster and details on its practical implementation. Section 4 explains the design, deploy, and evaluation of the first GPU-capable malleable application, MPDATA. Section 5 presents strong experimental evidence that the proposed policy outperforms the productivity results obtained by a traditional rigid workload that has been evaluated in two GPU-enabled clusters. Finally, the conclusions of this work can be found in Section 6.

2 Background

2.1 3D MPDATA Overview

The 3D MPDATA [18,24] application is an iterative algorithm that solves the continuity equation describing the advection of a nondiffusive quantity Ψ. A single iteration of MPDATA is called a time step and returns a single array. The set of states of the Ψ array is obtained after each time step creating the simulation of a studied phenomenon.

The algorithm is positive defined, and by appropriate flux correction can also be monotonic, a desirable feature for advection of positive definite variables such as specific humidity, cloud water, cloud ice, rain, snow, aerosol particles, and gaseous substances. The spatial discretization of MPDATA is based on finite difference approximations. The algorithm is iterative and fast convergent. In the first sub-step, advection of the Ψ field is computed with the standard donor-cell approximation, which ensures the first-order of accuracy only. In the subsequent time step, corrections are applied to make the scheme more accurate (i.e., second-order in space and time). In the corrective sub-step, the donor-cell approximation is used again but with new anti-diffusive velocities computed based on the advected fields. The procedure can be repeated

many times; however, typically, after more than two corrections, no significant improvements are observed.

In fact, to implement this algorithm, 11 arrays are allocated. The first six, $v1$, $v2$, $v3$, $v1P$, $v2P$ and $v3P$, represent velocities in each direction, where $v1P, v2P$, and $v3P$ are required to store intermediate results. The next two correspond to scalar quantities, one for odd time steps and another for even time steps. The array h represents the vector of density, while cp, cn store intermediate results.

As most stencil-based applications, MPDATA belongs to a group of memory-bound algorithms. To provide high efficient memory access [20], each row of arrays is aligned to a size of 128B. For this purpose, it is required to add some extra data at the beginning of each array to align the first element of the data and an appropriate number of extra columns to provide data alignment (padding) [21] for each row. This data reorganization allows the attainment of GPU coalesced memory access [1]. The structure of the array is shown in Figure 1.

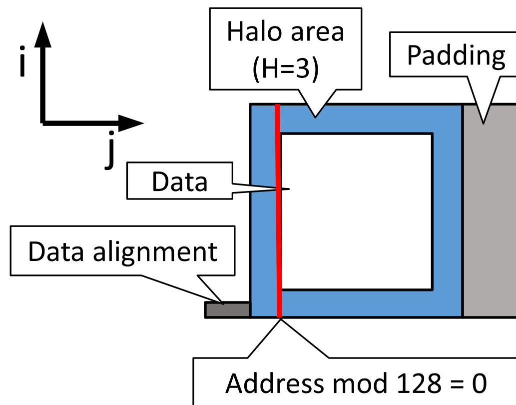

Fig. 1 Data structure of each array of MPDATA.

The current GPU implementation consists of four GPU kernels that perform a sequence of stencil computations. In order to compute a single element of the output array, it is required to perform around 343 flops (floating-point operations) per each time step (with some differences at the borders of the computational domain). After each time step, halo areas need to be exchanged between adjacent subarrays of the output array. The communication requires sending and receiving halo areas of size 3 on each side of a subarray.

2.2 Dynamic Management of Resources

This work is based on the DMR malleability framework. This solution provides a series of tools to introduce process malleability into a message passing interface (MPI) application easily. DMR includes two major components: On the one hand, a resource management system (RMS) responsible for resource allocation. Concretely, DMR relies on the Slurm Workload Manager and an extension with a module for scheduling job processes reconfigurations. Furthermore, thanks to its application programming interface (API), other components can interact with it exploiting all its potential. On the other hand, a parallel runtime that spawns and terminates MPI processes, handles the data redistributions, and resumes the execution at the exact point where malleability was triggered. Specifically, DMR is based on a version of the Nanos++ runtime [23] with support for detached offloading [11].

Figure 2 depicts how the DMR components communicate and how DMR is linked with an application. The communication layer between the RMS and the runtime, and an API for the users, enables job malleability. A typical scenario for DMR malleability is the following:

- A job, periodically, sends a reconfiguration request to the RMS. The most common case is in iterative applications, where each timestep represents an ideal point for resizing.
- The RMS receives the request and checks the system. Although this process depends on the selected reconfiguration policy, usually, the RMS takes into account the resources utilization and the queue of jobs.
- If the system can be optimized (the reconfiguration policy determines this optimization), the RMS will notify the runtime about the imminent reconfiguration.
- The runtime retrieves the process communication pattern and the resuming point. With this information, the runtime will perform the data redistribution, according to the communication pattern. Once the data has been redistributed, the runtime will resume the execution in the timestep where it was left.
- Finally, the job continues with the remaining iterations but with a new process layout.

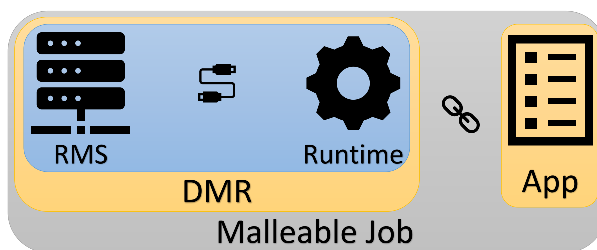

Fig. 2 DMR architecture and integration in an application.

DMR allows applications to change the number of resources during the execution of a job [11] by: i) expanding jobs, assigning more nodes to a job; or ii) shrinking jobs, releasing resources from a job, setting them available to be reassigned. Those decisions are made by Slurm, taking into account the following information:

- The workload size. In other words, the number of jobs in the queue, paying special attention to the number of pending jobs and their resource requirements.
- The scalability of each job and their lower and upper malleability limits.
- The number of available nodes within the cluster.

Summarizing, DMR expands and shrinks jobs on-the-fly by reassigning the underlying resources, spawning new MPI processes, redistributing the data among processes, and resuming the execution where it was left for reconfiguring.

The integration of the DMR library in the user's application allows the attainment of consistency of the internal data processed by the application. Reconfiguring a job with a new layout of nodes usually involves a reshape of the computational domain of the application, where the domain needs to be redistributed to the new set of nodes. It may affect the size of the domain, topology of nodes, and communication among nodes.

3 Preliminary study on the energy efficiency

With an efficient management of resources, the goal is to increase the global efficiency when executing job arrays, henceforth referred to as workload. The efficiency of the workload is defined as the relation of speedup of an parallel algorithm to the number of nodes used by the algorithm. Speedup is a multiplier indicating how many times faster the parallel algorithm is than its sequential version. The efficiency is expressed as $Eff_n = S_n/n$, where $S_n = t_1/t_n$ is the speedup of the job executed using n nodes in time t_n over a job executed sequentially in time t_1. The range of the efficiency is $Eff \in [0..1]$. In the MPDATA application, the execution time using n nodes is expressed as the communication time (t_c) of a single node (all nodes communicate in parallel) and a serial time of computation (t_1) over number of nodes $t_n = t_c + t_1/n$. The data transfer occurs between neighboring nodes and has a fixed size (the halo area is of size 3), independent on the number of nodes. For this reason, the communication time is assumed to be constant in the above equation.

By increasing the number of nodes, the size of the sub-domain is decreased, and the communication between neighboring nodes is required. In an ideal scenario, the communication would be at no temporal cost. In a real scenario, communication results in overhead, so it is assumed that the efficiency of MPDATA decreases when using more nodes. The communication overhead is related to a smaller size of computation for a higher number of nodes. As a result, the computation time decreases while the communication time is constant. For this reason, the ratio of computation to communication decreases.

To conclude the above discussion it is expected to achieve the highest efficiency by:

- executing in parallel as many small jobs as possible, since lowering the number of nodes per a single job increases the efficiency: $Max(Eff_i) = Eff_i(min(i))$, where $i \in [1..n]$;
- allocating in parallel all the available nodes by maximizing the number of independent jobs (the most efficient scenario is to execute n jobs per n nodes allocating one job per node).

The main idea behind the proposed solution is to minimize the resources for a single job while keeping the utilization of the entire cluster. There are to main scenarios to consider:

- the traditional scenario, based on allocating the maximum number of nodes per a single job for which the algorithm is scalable;
- the proposed scenario, based on executing the maximum number of jobs in parallel by reducing the number of nodes per a single job.

Let us consider the proposed scenario from the perspective of energy consumption. Generally, the power consumption of current CPUs and GPUs can be described as the sum of static power and dynamic power. Concretely, focusing on GPUs and considering n nodes, the power consumption generated by all the GPUs used by an algorithm is given by:

$$P(n) = (P_S + P_D^n) \cdot n, \tag{1}$$

where P_S is the static power per node (dependent on-chip layout and circuit technology and independent of the workload execution), P_D^n is the dynamic power per node (dependent on transistors switching overhead). Since the static power is independent on the resource utilization by an algorithm, the dynamic power changes with the active state of GPUs. The active state is related to the efficiency (Eff), which is relatively determined by the number of nodes (n). High scalable algorithms achieve the maximum efficiency when $Eff(n) = 1$ (the exception is a super scalability of algorithms that is not considered in this paper). It is assumed that it is worth to analyze the algorithm when its efficiency is higher than $1/n$. The efficiency below $1/n$ suggests that it is more efficient to reduce the number of nodes. So the lower limit of the efficiency that is considered in this work is $Eff(n) < 1/n$. It is assumed that there is no performance gain with a higher number of nodes. In summary, there are taken into account the following assumptions:

$$P_D^n \propto Eff(n); \quad Eff \in [\frac{1}{n}...1]; \quad E \approx P \cdot t. \tag{2}$$

In accordance with the above assumptions, the execution time of an algorithm executed using $n + 1$ nodes is within the following range:

$$t_{n+1} \in [t_n \cdot \frac{n}{n+1} ... t_n]. \tag{3}$$

Assuming a high limit of efficiency ($\mathit{Eff} = 1$), the active state of GPUs is constant and $P_D^{n+1} = P_D^n$, as well as $t_{n+1} = t_n \cdot n/(n+1)$. In this scenario, the energy reduction from using less number of nodes is lost by a higher execution time of a job. Then the energy reduction from utilizing the proposed scenario of allocating the lower number of nodes per job is insignificant:

$$E(n+1) - E(n) \approx n \cdot t_n \cdot (P_D^{n+1} - P_D^n) \approx 0 \tag{4}$$

Considering a low limit of efficiency ($\mathit{Eff}(n) = 1/n$), the dynamic power per node is decreasing with a higher number of nodes per job ($P_D^{n+1} < P_D^n$) and the execution time is $t_{n+1} = t_n$. In this scenario, there is a possibility to observe the energy reduction whether the dynamic power per node meets the following condition:

$$E(n+1) - E(n) > 0 \implies P_D^{n+1} > P_D^n \cdot \frac{n}{n+1} - \frac{P_S}{n+1} \tag{5}$$

In the MPDATA algorithm, in accordance to the assumptions (2), the formula (5) is equivalent to:

$$E(n+1) - E(n) > 0 \implies \frac{1}{n+1} \cdot P_D^n > \frac{1}{n+1} \cdot P_D^n - \frac{P_S}{n+1}, \tag{6}$$

which is true when $P_S > 0$. Based on the above considerations, it is expected that the proposed scenario provides maximization of the efficiency and improves the performance. It is also expected that the energy consumption will be reduced, or in the worst scenario it will not be lost.

In order to discuss the proposed approach to process job arrays, we briefly review our method by an example. Let us consider an example when a 3-job workload is distributed across a 6-node cluster. Assume that each job scales up to 4 nodes. The execution time of each job consists of the computation and communication time among nodes, where the number of nodes is greater than one (a single node execution does not require communication). In this regard, the computation time can be reduced by half when the number of nodes is doubled. With a single dimension mesh decomposition, the size of data transfers between nodes is the same, so the communication time is expected to be constant. This assumption is confirmed [20] for a group of iterative algorithms, such as MPDATA, where the halo area size is unchanged with the number of nodes greater than 1 and the size of sub-domain per node greater than the size of halo area. Figure 3 depicts this example. The blue arrows indict neighboring nodes that communicate with each other. White boxes indict the communication part, while grey boxes indict the computation part of jobs.

Traditionally, jobs are submitted to the system with the resource configuration that maximizes performance to complete the execution earlier (the traditional scenario). However, this is not what usually happens when many jobs compete for the same resources [9]. This argument is illustrated via an example that describes this scenario. Concretely, in this example, each job is executed in 4 nodes. Since there are only 6 available nodes, the Slurm manager

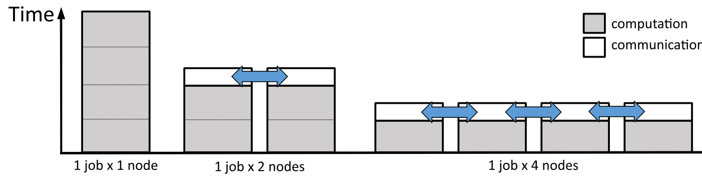

Fig. 3 Reduction of execution time depending on number of nodes.

enqueues the jobs and executes them one by one. Although this approach provides the highest performance, it also has side effects such as: (i) some nodes will remain idle; (ii) the efficiency of the allocated nodes is not maximized; and (iii) the global productivity is completely ignored. This working mode is illustrated in Figure 4, where jobs are executed one by one.

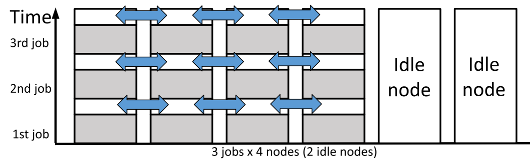

Fig. 4 Traditional distribution of nodes among jobs executed at their maximum performance.

Adopting more flexible strategies, which take into account the system status regarding jobs and resources, improves the efficiency. In particular, the process malleability via DMR has been leveraged, which will resize jobs on–the–fly depending on the cluster status. DMR originally is equipped with a job reconfiguration policy that allows users to define the malleability boundaries for each job. This policy accommodates heterogeneous workloads where jobs instantiate distinct applications with different scalability configurations [12]. In an effort to improve the efficiency of job arrays submitted to a queue, a new malleability scheduling strategy has been designed. For this purpose, the proposed strategy analyzes the entire workload and dynamically adapts the jobs to the platform's available resources. As a result, the policy maximizes the efficiency of each node. In this approach, all the jobs are resized in order to (i) minimize the communication between nodes; (ii) allocate all the available resources and reduce the idle time of the nodes; (iii) execute as many jobs in parallel as possible; and (iv) increase the global throughput by completing more jobs in less time. The proposed scenario is represented in Figure 5.

The potential gain in performance of the proposed scenario over the traditional ones is depicted in Figure 6. Considering a job array, it is expected that the execution of 3 jobs using 4 nodes per job (Figure 4) offers a lower performance than executing 3 jobs with 2 nodes per job (Figure 5) for a clus-

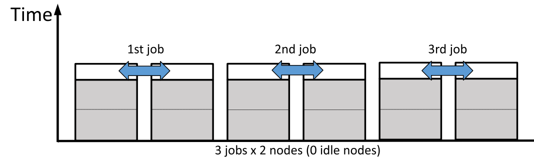

Fig. 5 Distribution of nodes among jobs by analyzing the current workload and the available resources.

ter equipped with 6 nodes. The proposed scenario is more efficient due to the following reasons:

- the traditional scenario includes 3 jobs, each of them uses 4 nodes that require to execute nine bidirectional data transfers (Figure 4);
- the proposed scenario uses 6 nodes simultaneously, but only 2 nodes need to be synchronized per job that gives three bidirectional data transfers (Figure 5);
- the efficiency decreases with the higher number of nodes per a single job;
- the proposed scenario has lower granularity, so it is more adaptable to a cluster with 6 available nodes (6 is divisible by 2) than the traditional scenario, where there are 2 idle nodes (the number of available nodes need to be divisible by 4).

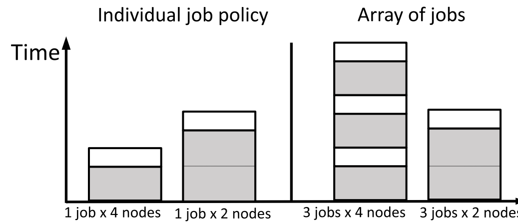

Fig. 6 Comparison of the traditional (when jobs are evaluated as individual) and the job arrays aware policies.

Furthermore, different working modes are provided for efficient job reconfiguration that, in some cases, can be less restrictive and, in consequence, can provide higher performance of some applications. Those modes are dedicated to the applications, where the parallelization model design is too complex to provide the most flexible management of application resources. Here, four reconfiguration working modes can be selected by a user:

- fixed - the job is launched with a determined number of processes, and they stay invariable during the whole execution. This mode is the traditional working mode in production clusters.

– moldable - the job can be initiated with different number of processes, but once it is initiated, the number of processes will remain invariable during the whole execution;
– malleable - the job is launched with a determined number of processes, but during the execution, the job can be resized;
– flexible - the job can be initiated with a different number of processes, and during the execution, the job can be resized.

The idea of each policy is depicted in Figure 7.

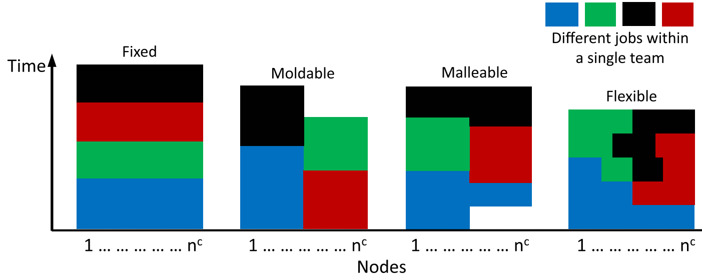

Fig. 7 Different policies of the proposed scheduler.

4 Dynamic Reconfiguration of the Computational Domain of MPDATA

. The data structure of MPDATA is decomposed in order to provide GPU coalesced memory access. In the proposed solution, the data allocation is evaluated by a data manager. This evaluation determines a set of input parameters with the most appropriate data organization. Finally, the data manager returns the array that is allocated both on the host and device. This array stores one chunk of the computational domain of MPDATA that is distributed across a set of nodes. The list of input parameters of the proposed data manager is: 3D computational domain ($N \times M \times L$) that represents the size of MPDATA task, number of nodes (n), topology (T_N) of nodes defined as $[n/T_N \times T_N]$, data alignment (see Figure 1) expressed in bytes (A), number of GPUs per node (G), topology (T_G) of GPUs per node defined as $[G/T_G \times T_G]$, sizes of CUDA blocks [1] for each of four GPU kernels ($K_i^x \times K_i^y$, where $i \in [1, 2, 3, 4]$), and halo area size (H) for data transfers (the MPDATA algorithm requires to keep halo area of size $H = 3$, expressed in a number of rows/columns). The topology parameters characterize the number of rows and columns within a mesh of GPUs per node (G) or nodes (n). For example, the algorithm configured with $n = 16$ number of nodes and $T_N = 2$ topology of nodes creates a 2D mesh of size 8×2.

The data manager uses a 2D decomposition over a 3D computational domain. The decomposition is performed across the N and M dimensions. The CUDA blocks are organized into 2D blocks across the N and M dimensions. Each GPU kernel then computes a single loop iterating across the third dimension. The memory organization is designed in the following order: the memory is continuous across the M dimension, then across the L dimension, and finally across the N dimension. To prepare the data structures, the proposed solution to calculates the number of rows (T^y) that represents the topology of the domain decomposition and the size of each sub-domain ($D_N \times D_M$):

$$T^y = G \cdot T_N; \quad D_N = \lceil N/(n/T_N) \rceil; \quad D_M = \lceil M/T_N \rceil. \tag{7}$$

Then, the maximum size of a CUDA block for all the kernels are calculated:

$$mx^x = \max(K_i^x); \quad mx^y = \max(K_i^y); \quad i \in [1, 2, 3, 4]. \tag{8}$$

The size S_n of a sub-array is obtained based on the following equation:

$$S_n = 2 \cdot H + rndT(rndT(\lceil D_N/T^y \rceil, mx^y), T^y), \tag{9}$$

while the size S_m is calculated as:

$$S_m = rndT(rndT(rndT(\lceil D_M/T_G \rceil, mx^x), T_G) + 2 \cdot H, A), \tag{10}$$

where the function $rndT(x, y)$ returns the x value rounded up to y: $rndT(x, y) = \lceil x/y \rceil \cdot y$. The number of threads ($Th_n \times Th_m$) that should be created following the CUDA programming model, organized into two-dimensional blocks is computed based on the following equations:

$$Th_n = rndT(\lceil D_N/T^y \rceil, mx^y), \text{ and } Th_m = rndT(\lceil D_M/T_G \rceil, mx^x). \tag{11}$$

Finally, the data manager returns the array size that is organized as it is shown in Figure 1:

$$size = S_n \cdot S_m \cdot (L + 2 \cdot H) + A. \tag{12}$$

This corresponds to the number of asynchronous streams that needs to be created is equal to the number G - GPUs per node. The size of CUDA grid expressed in number of CUDA blocks per GPU kernel is equal to:

$$S_B = \lceil T_m/K_i^x \rceil \times \lceil T_n/K_i^y \rceil, \quad i \in [1, 2, 3, 4]. \tag{13}$$

Developing a malleable version of MPDATA requires the preparation of two procedures within the MPDATA application leveraged by the scheduler. The first one is executed when the number of nodes needs to be increased and, therefore, can be regarded as an expanding procedure. The second one is responsible for releasing the cluster resources and is referred to as a shrinking procedure. The MPDATA application design is based on the PCAM model [2], where the parallelization of the code is performed in four stages, namely Partition, Communication, Agglomeration, and Map (Figure 8), discussed later in this section.

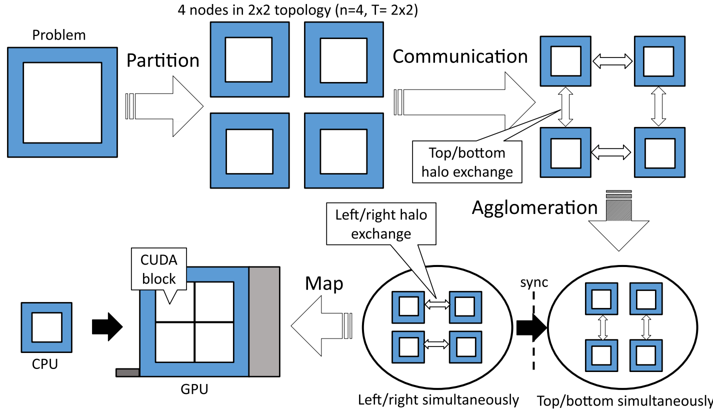

Fig. 8 Design methodology of MPDATA based on PCAM model.

Both the procedures (expanding and shrinking) are responsible for rebuilding the computational domain of the application and modifying the number of computing nodes. The procedures start with the partition stage. Here, the scheduler modifies the following parameters: number of nodes n and topology T_N. By default, each time the application is requested to be expanded or shrunk, the number of nodes n is doubled or reduced by a half. The new topology of nodes (T_N) is recalculated in accordance with the following formula: $T_N = \lfloor sqrt(n) \rfloor$. This formula provides the same number of rows as columns if possible; otherwise, the number of rows is greater than number columns. Then the data manager returns new sub-arrays for each node based on Equations $1 - 8$.

The second stage, called communication, is responsible for estimating the halo areas between neighbor nodes within the new computational domain. The MPI rank of the right/left neighboring node is calculated by adding/subtracting the value n/T_n to/from the current MPI rank. The top/bottom neighboring node is obtained by adding/subtracting 1 value to/from the current MPI rank. In a third step, called agglomeration, the communication is performed. The data exchange is organized in the following sequence of data transfers in order to improve the performance of communications between neighbor nodes: (i) transfer data between left and right neighbors simultaneously - here the columns of the sub-matrices are copied without its top and bottom halo areas; (ii) communicate the top and bottom neighbors simultaneously with their halo areas. With this mechanism, it is not necessary performing diagonal updates of data located at the corners of the computational domain. Furthermore, it reduces the maximum number of data transfers to four (left/right/top/bottom) instead of nine (all-around nodes). The last stage, map, is responsible for sending data from the host memory to the GPU global memory, where it is further processed by the GPU kernels in a CUDA blocks fashion [1]. All the data

transfers between host and device are performed using asynchronous CUDA streams.

5 Experimental Evaluation

5.1 Testing Environment

The effectiveness of the proposed scheduler is validated using the GPU-based cluster Minotauro at the Barcelona Supercomputing Center (BSC). This cluster has two different configurations for which the following scenarios have been deployed:

- 41 nodes equipped with an NVIDIA Tesla M2090 GPUs (512 CUDA cores and 6 GB of GDDR5 memory). Each node comprises 2 Intel E5649 sockets (6 cores at 2.53 GHz each) for a total of 12 cores with 24 GB of main memory. The nodes are interconnected through a 40 Gbit/s Infiniband QDR network.
- 21 nodes equipped with an NVIDIA Tesla K80 GPUs (4992 CUDA cores and 2 x 12 GB of GDDR5 memory). Besides, nodes contain 2 Intel Xeon E5-2630 v3 sockets (8 cores at 2.4 GHz each) for a total of 16 cores with 128 GB of main memory. The nodes are interconnected through a 56 Gbit/s Infiniband FDR network.

The software stack was composed of CUDA 8.0, MPICH 3.2, OmpSs 15.06, and Slurm 15.08. Slurm was configured with the following plug-ins:

- Job scheduling: sched/backfill with 10-second interval time among scheduling attempts.
- Job priority: priority/multifactor without wall time duration of jobs.
- Resource selection: select/linear.

One node of each scenario hosted the Slurm management daemon, while the remaining ones were used as compute nodes.

5.2 Experimental Results

This work relies on power profiles [3], to provide an accurate energy measurement. All the energy estimations are based on the power dissipation of GPU devices. The power consumption of communication/networking is not measured. In more detail, `nvidia-smi` tool is leveraged to query GPU sensors and capture the current power draw of the GPU and time stamps. Based on them, the energy consumption of MPDATA is obtained by integrating power curves over the corresponding executions. All the experiments include workloads of 9 MPDATA jobs with a grid of size $2048 \times 1024 \times 64$ and 10.000 time steps. The power profiles of MPDATA obtained in the first cluster, equipped with NVIDIA Tesla M2090 GPUs, are shown in Figure 9. The power profiles are estimated for the algorithm executed using from 2 to 32 nodes. Experiments

were repeated ten times to validate their correctness and diminish the effect of noise. The result reported next corresponds to the median. Repeatability is examined through the calculation of the maximum root mean squared error (RMSE) that is below 2.8 for all the achieved results. These results allow the

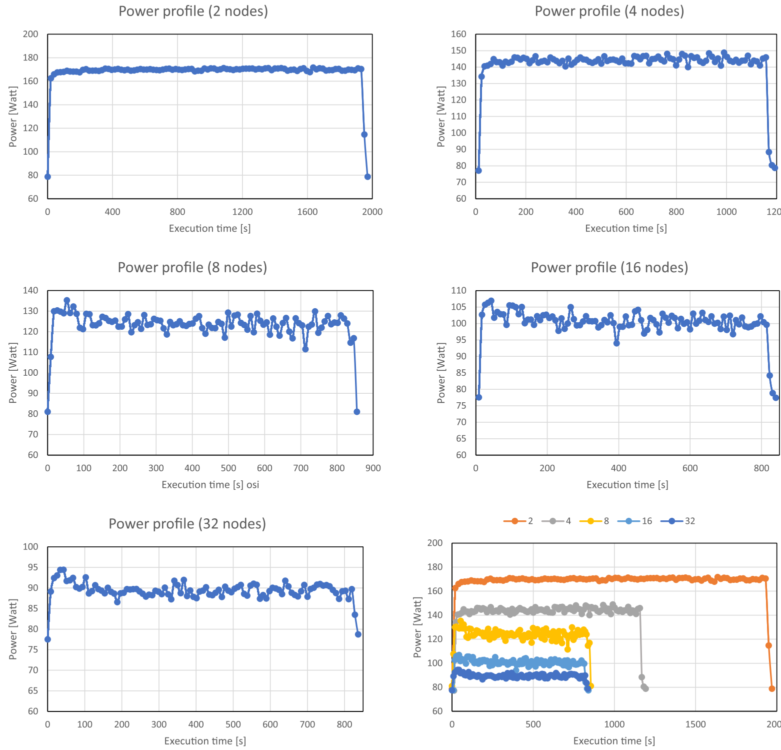

Fig. 9 Power profiles of MPDATA achieved on the cluster equipped with NVIDIA Tesla M2090 GPUs.

formulation of the following conclusions: (i) The average power dissipation decreases with the number of nodes. This is due to the increase in the number of nodes reduces the efficiency. (ii) Using more nodes generates a less smooth power curve than the one visualized when fewer nodes are involved. This results from the fact of a higher impact of the communications in the algorithm execution that is not included in the power measurements. A high number of nodes reduces the number of computations per node within a single time step of MPDATA, increasing the communication overhead. (iii) The average power dissipation is lower when more nodes are used. This supports the hypothesis that the efficiency of the algorithm is higher when the number of nodes is reduced.

Figure 10 shows the power profiles of MPDATA on the K80-GPU based cluster. The method for evaluating power curves is the same as in the first configuration. The power profiles are estimated based on MPDATA executions in 2 to 16 nodes. The maximum RMSE of results is 1.8. The achieved results confirm the assumptions that the highest power dissipation is obtained when using the lowest number of nodes. The increased communication overhead is observed with idle states of GPUs, where a higher number of nodes is used.

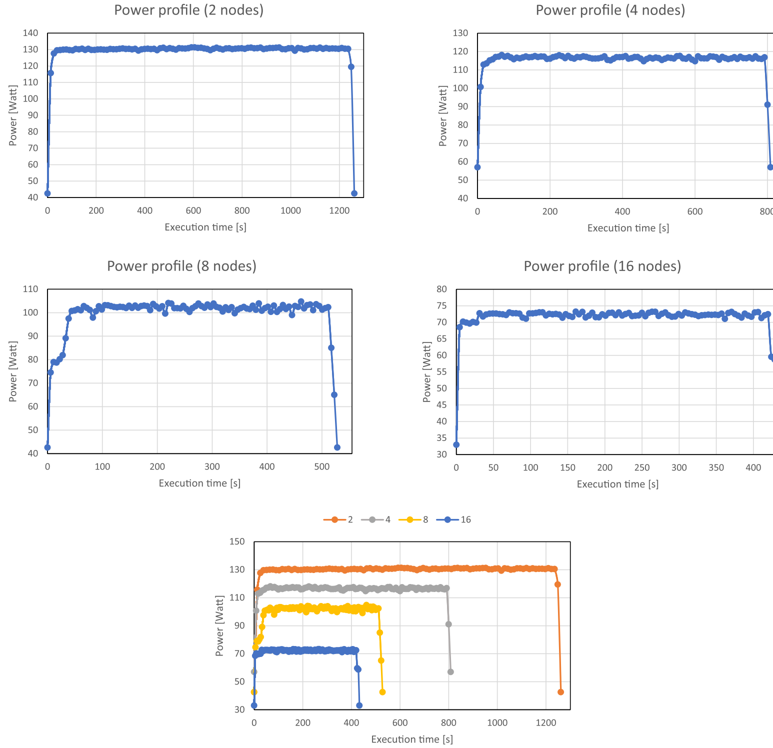

Fig. 10 Power profiles of MPDATA achieved on the cluster equipped with NVIDIA Tesla K80 GPUs.

The scalability of the algorithm executed on the first cluster is shown in Table 1. The second column contains the execution time for each node configuration. The third column shows the average power dissipation of a single GPU, while the fourth column shows the energy consumed by a single GPU, estimated based on the power profiles. The total energy column shows the energy consumed by all GPUs used in the experiment. The last column reports the ratio between the execution time using a single node and the execution time achieved for each experiment. The highest performance is observed using 32 nodes, while the lowest energy consumption corresponds to the execution

with a single node. The energy consumption is more than 5× higher compared with that on a single node. However, the execution time is reduced by a factor of 3. This confirms the assumption that the highest efficiency is achieved using a single node.

Table 1 Execution times and energy results of MPDATA achieved on the cluster equipped with NVIDIA Tesla M2090 GPUs.

#n	time [s]	avg. P [W]	E/n [kJ]	Tot. E[kJ]	t_1/t_n
1	2355	198.5	467.47	467.47	1.00
2	1970	168.3	331.60	663.19	1.20
4	1192	141.6	168.84	675.35	1.98
8	855	123.6	105.69	845.54	2.75
16	831	100.4	83.41	1334.57	2.83
32	796	94.0	74.81	2393.92	2.96

Table 2 shows the execution times and energy results of MPDATA on the cluster equipped with NVIDIA Tesla K80 GPUs. The execution time compared to the M2090-based cluster is reduced varying from a factor of 1.3 in a single node to 1.9 on 16 nodes. This is a result of using more advanced architecture but also more efficient network in the K80-based cluster. A faster intercommunication network allows achieving higher speedup and reducing the communication overhead compared with the previous cluster. In accordance with the initial assumptions, the power dissipation per GPU is inversely proportional to the number of nodes. For this reason, the highest efficiency is achieved using a single node.

Table 2 Execution times and energy results of MPDATA achieved on the cluster equipped with NVIDIA Tesla K80 GPUs.

#n	time [s]	avg. P [W]	E/n [kJ]	Tot. E[kJ]	t_1/t_n
1	1705	135.2	230.55	230.55	1.00
2	1248	131.3	163.82	327.65	1.37
4	808	116.4	94.07	376.29	2.11
8	528	101.0	53.32	426.56	3.23
16	433	73.0	31.58	505.32	3.94

The next experiments compare the different working modes: *fixed*, *moldable*, *malleable*, and *flexible*. In those experiments the following measurements are taken: (i) the execution time, (ii) energy consumption of busy nodes (nodes where the jobs are running), (iii) energy consumption of idle nodes (during the execution of the experiment), (iv) energy consumed by the cluster expressed as a sum of the energy consumed both by the busy and idle nodes, and (v) resource allocation expressed as a percentage of busy nodes with respect to all the nodes in the cluster. All the experiments measure the malleability overhead due to rebuilding the computational domain for each job. The overhead

varies in the range from 1.1% to 3.5% of the application execution time. This overhead does not apply to the *fixed* working mode neither the *moldable* one, as in those two cases, the computation domain is not modified during runtime.

The results of the first experiment performed on the cluster equipped with NVIDIA Tesla M2090 GPUs are shown in Table 3. The *flexible* mode allows the reduction of the workload execution time by a factor of 4.8 compared with that of the *fixed* mode. The energy consumption is reduced by a factor of 2.4. Here, using the *fixed* mode, all the jobs are executed sequentially, and each of them allocates 32 nodes because jobs are launched at their maximum performance. The *flexible* mode shrinks all the jobs, and in consequence, it executes nine jobs in parallel, where each of them allocates 4 nodes. There is no significant difference between the *moldable* and *malleable* working modes in terms of performance and energy consumption. Both of them reduce the performance and energy consumption compared with the *fixed* one, but they are at about 3× slower and about 1.6× more energy consuming than the *flexible* configuration.

Table 3 Comparison of performance results of different malleability working modes for a workload of 9 jobs on the cluster equipped with NVIDIA Tesla M2090 GPUs.

Mode	Time [s]	Busy E [kJ]	Idle E [kJ]	Cluster E [kJ]	Res. Alloc. [%]
Fixed	7112	21389	853	22242	88.89
Moldable	4608	14178	731	14909	86.46
Malleable	4440	13681	1475	15156	69.25
Flexible	1472	8887	248	9135	84.43

Table 4 compares the working modes for a cluster equipped with NVIDIA Tesla K80 GPUs. The highest performance is achieved using the *flexible* mode, which resulted in the performance increase of a workload by a factor of 2.5 over the traditional working mode. The final jobs distribution within the *flexible* mode is to use 2 nodes per job that allocates 18 nodes per 9 jobs simultaneously. The proposed reconfiguration policy allows the reduction of the energy consumption by a factor of 1.4 compared with the results obtained using the *fixed* mode. Similarly to the results achieved on the Fermi-based cluster, there are no significant differences between the *moldable* and *malleable* modes. Both of them yield a reduction of 1.44× in execution time and 1.14× in energy reduction compared with the results achieved using the *flexible*.

6 Conclusion

The presented work has studied the effect of malleability when GPU-capable jobs are submitted in burst mode. For this purpose, the 3D MPDATA application has been redesigned to support MPI process malleability. For this purpose, the project has leveraged the DMR malleability framework, which

Table 4 Comparison of performance results of different malleability working modes for a workload of 9 jobs on the cluster equipped with NVIDIA Tesla K80 GPUs.

Mode	Time [s]	Busy E [kJ]	Idle E [kJ]	Cluster E [kJ]	Res. Alloc. [%]
Fixed	3826	4466	459	4925	80.00
Moldable	2652	3856	446	4302	98.99
Malleable	2653	3858	446	4303	93.50
Flexible	1536	3274	173	3448	81.21

eased the adoption of malleability during the coding stage. At the same time, this study has demonstrated the versatility of DMR and its readiness to be used together with other programming models like CUDA.

The theoretical study presented in this work shows that the best malleability strategy for this kind of application is to fairly share the resources among all the jobs in the queue to increase job concurrency, at the expense of reducing the number of allocated nodes per job. For this reason, a new malleability plug-in for the DMR was designed to implement the policy that followed the deliberate strategy.

Furthermore, this paper unveils the first CUDA application with support for MPI malleability through the integration of DMR in MPDATA. The proposed solution presents a very low overhead which varies from 1.1% to 3.5% in the execution time, since the implementation takes into account a set of factors that keep the high performance of the application including the data alignment and padding, while preserving the CUDA block sizes of the computational domain.

The experiments performed on two GPU-enabled HPC clusters, expose impressive results not only in throughput (executed jobs per second) but also in energy consumption when comparing the *flexible* working mode with the traditional job management mode (*fixed*). To sum up, the proposed solution allows the significant reduction of the energy consumption of the application and increase the system productivity. The experiments confirm the assumptions of the scheduling strategy for job arrays, which benefit from higher concurrency and smaller resource allocation.

References

1. Nvidia web page. http://www.nvidia.com (2018). . Accessed: 2018-12-17
2. Barlas, G.: Multicore and GPU Programming: An Integrated Approach. Elsevier Science (2014)
3. Burtscher, M., Zecena, I., Zong, Z.: Measuring GPU Power with the K20 Built-in Sensor. pp. 28:28–28:36. ACM (2014)
4. Comprés, I., Mo-Hellenbrand, A., Gerndt, M., Bungartz, H.J.: Infrastructure and API Extensions for Elastic Execution of MPI Applications. In: Proceedings of the 23rd European MPI Users' Group Meeting on - EuroMPI 2016, pp. 82–97. ACM Press (2016)
5. El Maghraoui, K., Desell, T.J., Szymanski, B.K., Varela, C.A.: Malleable Iterative MPI Applications. Concurrency and Computation: Practice and Experience **21**(3), 393–413 (2009)

6. El Maghraoui, K., Szymanski, B.K., Varela, C.: An Architecture for Reconfigurable Iterative MPI Applications in Dynamic Environments. In: International Conference on Parallel Processing and Applied Mathematics, pp. 258–27 (2006)
7. Feitelson, D.G.: Packing Schemes for Gang Scheduling. In: Lecture Notes in Computer Science book series (LNCS, volume 1162), pp. 89–110. Springer, Berlin, Heidelberg (1996)
8. Gupta, A., Acun, B., Sarood, O., Kalé, L.V.: Towards Realizing the Potential of Malleable Jobs. In: 21st International Conference on High Performance Computing (HiPC) (2014)
9. Iserte, S.: High-throughput Computation through Efficient Resource Management. Ph.D. thesis, Universitat Jaume I, Castelló de la Plana (2018)
10. Iserte, S., Martínez, H., Barrachina, S., Castillo, M., Mayo, R., Peña, A.J.: Dynamic Reconfiguration of Noniterative Scientific Applications. The International Journal of High Performance Computing Applications p. 109434201880234 (2018)
11. Iserte, S., Mayo, R., Quintana-Ortí, E.S., Beltran, V., Peña, A.J.: Efficient Scalable Computing through Flexible Applications and Adaptive Workloads. In: 10th International Workshop on Parallel Programming Models and Systems Software for High-End Computing (P2S2). Bristol (2017)
12. Iserte, S., Mayo, R., Quintana-Ortí, E.S., Beltran, V., Peña, A.J.: DMR API: Improving cluster productivity by turning applications into malleable. Parallel Computing **78**, 54–66 (2018)
13. Leiserson, H.T.K.C.E.: Algorithms for vlsi processor arrays (1979)
14. Lemarinier, P., Hasanov, K., Venugopal, S., Katrinis, K.: Architecting Malleable MPI Applications for Priority-driven Adaptive Scheduling. In: Proceedings of the 23rd European MPI Users' Group Meeting (EuroMPI), pp. 74–81 (2016)
15. Lublin, U., Feitelson, D.G.: The workload on parallel supercomputers: Modeling the characteristics of rigid jobs. Journal of Parallel and Distributed Computing **63**(11), 1105–1122 (2003)
16. Martín, G., Singh, D.E., Marinescu, M.C., Carretero, J.: Enhancing the Performance of Malleable MPI Applications by Using Performance-aware Dynamic Reconfiguration. Parallel Computing **46**, 60–77 (2015)
17. Prabhakaran, S., Neumann, M., Rinke, S., Wolf, F., Gupta, A., Kale, L.V.: A batch system with efficient adaptive scheduling for malleable and evolving applications. In: 2015 IEEE International Parallel and Distributed Processing Symposium, pp. 429–438 (2015)
18. Prusa, J., Smolarkiewicz, P., Wyszogrodzki, A.: Eulag, a computational model for multiscale flows. Computers & Fluids **37**, 1193–1207 (2008)
19. Rojek, K.: Machine Learning Method for Energy Reduction by Utilizing Dynamic Mixed Precision on GPU-based Supercomputers. Concurrency and Computation: Practice and Experience **e4644** (2018). DOI https://doi.org/10.1002/cpe.4644
20. Rojek, K., Quintana-Ortí, E.S., Wyrzykowski, R.: Modeling power consumption of 3d mpdata and the cg method on arm and intel multicore architectures. The Journal of Supercomputing **73**(10), 4373–4389 (2017)
21. Rojek, K., Wyrzykowski, R.: Performance modeling of 3d mpdata simulations on gpu cluster. The Journal of Supercomputing **73**(2), 664–675 (2017)
22. Rojek, K., Wyrzykowski, R., Kuczynski, L.: Systematic Adaptation of Stencil-based 3D MPDATA to GPU Architectures. Concurrency and Computation: Practice and Experience **29**(9) (2017)
23. Sainz, F., Bellon, J., Beltran, V., Labarta, J.: Collective Offload for Heterogeneous Clusters. In: 22nd International Conference on High Performance Computing (HiPC) (2015)
24. Smolarkiewicz, P.: Multidimensional Positive Definite Advection Transport Algorithm: An Overview. Int. J. Numer. Meth. Fluids **50**, 1123–1144 (2006)
25. Spenke, F., Balzer, K., Frick, S., Hartke, B., Dieterich, J.M.: Malleable parallelism with minimal effort for maximal throughput and maximal hardware load. Computational and Theoretical Chemistry **1151**, 72 – 77 (2019)
26. Sudarsan, R., Ribbens, C.: Scheduling Resizable Parallel Applications. International Symposium on Parallel and Distributed Processing (2009)

27. Szustak, L.: Strategy for data-flow synchronizations in stencil parallel computations on multi-/manycore systems. The Journal of Supercomputing **74**(4), 1534–1546 (2018)
28. Yoo, A.B., Jette, M.A., Grondona, M.: SLURM: Simple Linux Utility for Resource Management. In: 9th International Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP), pp. 44–60 (2003)