Eur. Phys. J. B (2015) 88: 158
DOI: 10.1140/epjb/e2015-60180-8

THE EUROPEAN PHYSICAL JOURNAL B

Colloquium

Colloquium: Large scale simulations on GPU clusters

Massimo Bernaschi[1,a], Mauro Bisson[2], and Massimiliano Fatica[2]

[1] Istituto per le Applicazioni del Calcolo, National Research Council of Italy, Via dei Taurini 19, 00185 Roma, Italy
[2] NVIDIA Corporation, 2701 San Tomas Expressway, Santa Clara, CA 95050, USA

Received 6 March 2015 / Received in final form 11 May 2015
Published online 17 June 2015 –

Abstract. Graphics processing units (GPU) are currently used as a cost-effective platform for computer simulations and big-data processing. Large scale applications require that multiple GPUs work together but the efficiency obtained with cluster of GPUs is, at times, sub-optimal because the GPU features are not exploited at their best. We describe how it is possible to achieve an excellent efficiency for applications in statistical mechanics, particle dynamics and networks analysis by using suitable memory access patterns and mechanisms like CUDA *streams*, profiling tools, etc. Similar concepts and techniques may be applied also to other problems like the solution of Partial Differential Equations.

1 Introduction

In the past years, graphics processing units (GPU) have established themselves as a viable alternative to traditional platforms for high performance computing. A number of scientific codes have a GPU version, most of the times implemented by using NVIDIA compute unified device architecture (CUDA) C and Fortran *bindings*. However many interesting problems simply do not fit in the memory available on a single GPU so large scale simulations or simulations whose execution time must be reduced as much as possible (i.e., pseudo real-time simulations or high-frequency data processing) require more than one GPU working concurrently. Multi-GPU programming can be carried out in different ways but the most common approach is to combine CUDA for programming the single GPU and MPI to exchange data among the GPUs involved in the computation. In other words, a hybrid parallel programming model whose efficiency depends on the ability of the developer to exploit at its best the huge computing power of the single GPU and to overlap computation and communication with a proper combination of problem decomposition, data structures, algorithms, and technology exploitation. In the present paper we describe, through some representative examples, how it is possibile to achieve excellent scalability using multiple GPUs. We assume a working knowledge of CUDA and MPI.

The paper is organized as follows: Section 2 provides a general introduction to multi-GPU programming and shows how to reduce the communication overhead; Section 3 presents few selected applications and techniques that can be of interest for researchers who need to run large scale simulations on clusters of GPUs. Section 4 concludes the paper with a look to future perspectives of high performance GPU-based parallel processing.

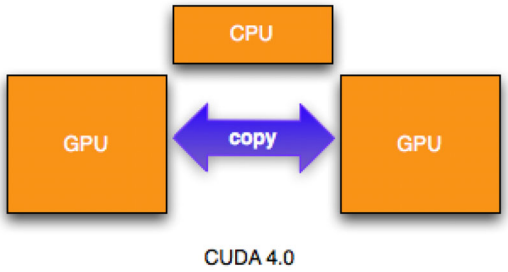

Fig. 1. Direct memory copy between GPUs enabled since CUDA 4.0.

2 Effective multi-GPU programming

Starting with CUDA version 4.0, NVIDIA GPUs can, under some specific conditions, move data to/from the memory of another GPU (see Fig. 1). Basically the mechanism, named in CUDA as *peer-to-peer*, requires that:

- source and target GPUs are connected to the same PCI-e root complex;
- both source and target GPU are, at least, of *Fermi* generation.

With the *peer-to-peer* support, the copy operation does not need to be staged through the CPU. However, in general, GPUs cannot exchange data directly without the support of the hosting CPU. If the GPUs that need to communicate are in two distinct systems, then the most

[a] e-mail: massimo.bernaschi@gmail.com

used procedure is:

1. the sender GPU uploads data to its controlling CPU (a *device to host* memory copy operation in CUDA jargon);
2. sender CPU sends data to the CPU controlling the target GPU with MPI. MPI guarantees portability and efficiency of the inter-systems communication;
3. target CPU downloads data to the target GPU (*host to device* memory copy).

It is apparent that the memory copy operations introduce an overhead and that if the GPUs remain idle during both the memory copy operations and the inter-CPU data exchange, the efficiency of a multi-GPU configuration may be seriously impaired. In general, it is desirable to carry out concurrently computation and communication (regardless of the availability of GPUs) but, most of the times, even if MPI supports a *non-blocking* communication mode that can be interleaved with other useful work, the actual overlap is very limited (if any).

However, CUDA provides a very good support for concurrency within an application through its execution *streams*. In CUDA, a stream is a sequence of operations that execute on the GPU in the order in which they are issued by the CPU. While operations within a stream are guaranteed to execute in the prescribed order, operations in different streams can be interleaved and, when possible, they can even run concurrently. The amount of execution overlap between two streams depends on the order in which the requests are issued to each stream and on the availability of resources for the execution (further information can be found in Ref. [1]). It is possible to overlap:

- computations on GPU (depending on the availability of resources);
- memory copy operations (from/to CPU) and computations on GPU;
- CPU operations (including communication) and GPU computations.

For the sake of simplicity, let us assume that the GPU computations can be divided somehow in two parts: the first one deals with data that need to exchanged with other GPUs (let us call them *boundaries*) whereas the second one deals with data that are completely local to the GPU (let us call them *bulk*). This sub-division is pretty natural using domain decomposition techniques but it can also be applied to other situations (e.g., networks or graph partitioning).

Two streams are required to achieve overlap between communication and computation: one stream is assigned to the bulk and one to the boundaries. Then the following scheme can be applied:

1. update the boundaries by using the first stream;
2. first stream:
   - copy data in the boundaries from the GPU to the CPU;
   - exchange data between nodes by using MPI;
   - copy data in the boundaries from the CPU to the GPU;
3. second stream:
   - update the bulk;
4. start a new iteration.

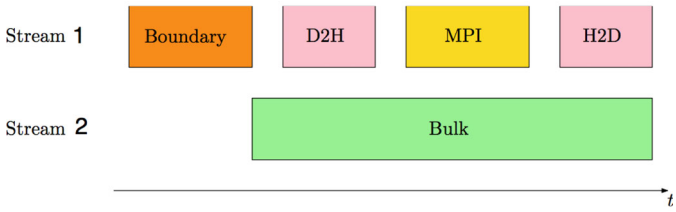

Fig. 2. Overlap between computation and communication using two streams. D2H is the *device to host* memory copy. H2D is the *host to device* memory copy.

The overlap, shown in Figure 2, is between the exchange of data within the *boundaries* (carried out by the first stream and the CPU) and the update of the bulk (carried out by the second stream). The CPU acts as a data-exchange-coprocessor of the GPU. The communication overhead can be completely *hidden* under the condition that the computations in the bulk require more time than the data exchange.

2.1 GPU-aware MPI implementations

CUDA 4.0 introduced a new feature called Unified Virtual Addressing, or UVA, which maps all CPU and GPU memories in the system into a single virtual address space. This allows CUDA applications and libraries to determine the location of a variable based on the value of its virtual address. This feature made possible the development of GPU-aware MPI implementations in which MPI functions are able to also accept pointers to GPU memory. There are several GPU-aware MPI implementations available: MVAPICH2[1], OpenMPI[2], CRAY MPI and IBM Platform MPI.

The ability to pass GPU pointers directly to MPI functions may simplify the programming effort since, with GPU-aware MPI, all the complexity of the CPU-GPU data exchange is hidden inside the library and only MPI calls are required. It also allows applications to automatically benefit from optimizations possible from GPU Direct[3].

GPU Direct is a name used to refer to several specific technologies (from peer to peer to GPU RDMA). In the context of MPI the GPU Direct technologies cover all kinds of inter-rank communication: intra-node, inter-node, and RDMA inter-node communication.

Every GPU-aware MPI implementation has different capabilities, and being a rapidly evolving technology, it is difficult to describe what it is available in a particular

[1] http://mvapich.cse.ohio-state.edu/overview/mvapich2/
[2] http://www.open-mpi.org/faq/?category=building#build-cuda
[3] http://devblogs.nvidia.com/parallelforall/introduction-cuda-aware-mpi

distribution. There are also dependencies on the particular hardware configuration (for example, peer to peer only works if the two GPUs are under the same PCI-e root complex, GPU Direct RDMA may require particular Infiniband drivers and cards).

It is also difficult to estimate the performance gains obtainable from using GPU-aware MPI instead of a traditional approach. Glaser et al. [2] compared the performance of a traditional approach versus a GPU-aware MPI (with GPU Direct both enabled and disabled) and showed that for single precision the traditional approach is always faster then GPU aware MPI with GPU Direct disabled. Once GPU Direct is enabled the gap is significantly reduced, bringing the two approaches on par. For double precision, the GPU-aware MPI with GPU Direct is always faster with gains up to 20%.

2.2 Advanced profiling

When porting an application to a cluster with GPUs, it is important to be able to understand where the time is spent on both the CPU and GPU sides. The standard profiling tools in CUDA, nvprof and nvvp, are able to show the GPU timeline but do not present CPU activity. The NVIDIA Tools Extension (NVTX) is a C-based API to annotate the profiler time line with events and ranges and to customize their appearance and assign names to resources such as CPU threads and devices. The use is very simple, once the NVTX header file is included, the developer needs to mark the region of interest with nvtxRangePush and nvtxRangePop calls.

```
#include "nvToolsExt.h"
...
void init_host_data( int n, double * x )
{
  nvtxRangePushA("init_host_data");
  //initialize x on host
  ...
  nvtxRangePop();
}
...
```

To eliminate profiling overhead during production runs and to remove NVTX from release builds, it is possible to use a set of macros that also allow full color and text customization[4].

During the runs, one or more MPI processes generate the traces that are later imported and visualized with nvvp, the NVIDIA Visual Profiler. For example, in order to generate the profiler traces for a run on 4 GPUs, where each trace will have the host name (%.h) and the process number (%p) appended to the name of the output file, it is possible to use the following command:

```
mpirun  -n 4  nvprof -o output_hitMPI.%h.%p
./hitMPI
```

[4] http://devblogs.nvidia.com/parallelforall/cuda-pro-tip-generate-custom-application-profile-timelines-nvtx/

3 Examples of multi-GPU applications

In this section we present some examples of applications that can take advantage of clusters of GPUs and provide hints on how to best exploit the capabilities offered by this computing platform. We start from simple, regular computations and move to more complex applications.

3.1 Stencil computations and statistical mechanics models

As a first example, we consider the simulation of the three-dimensional Ising spin glass, a statistical mechanics model defined on a cubic lattice by the Hamiltonian

$$H = -\sum_{\langle ik \rangle} J_{ik}\, \sigma_i \sigma_k, \tag{1}$$

where the $\sigma_i \in \{-1, +1\}$ are the so-called spin variables, the $J_{ik} \in \{-1, +1\}$ are coupling constants (representing *quenched* disorder) randomly drawn according to a given probability distribution $P(J_{ik})$ and the sum $\sum_{\langle ik \rangle}$ is restricted to nearest neighbours. With a bimodal disorder the probability distribution reads

$$P(J_{ik}) = \frac{1}{2}\left[\delta_K(J_{ik} - 1) + \delta_K(J_{ik} + 1)\right], \tag{2}$$

where $\delta_K(a - b) = \delta_{ab}$ stands for the Kroneker delta. Such a model describes, in three dimensions, a disordered and frustrated magnetic system showing a glassy dynamics below a finite critical temperature $T_c = 1.1019(29)$. Spin systems have been studied by using GPUs by several authors [3–6]. Most Monte Carlo simulations of statistical mechanics models, like the Ising spin glass, require a *stencil computation*. Stencil computations are used to solve a large number of important scientific computing problems not only in Statistical Mechanics but also in partial differential equations (PDE) and Lattice Boltzmann based modeling. The computing kernels for stencil computations use an outer-most time loop to make a large number of sweeps over a multi-dimensional mesh. The value of the model variables living on each node of the mesh is modified at each iteration based on the values of the neighboring nodes. In general stencil computations can be parallelized (both on GPUs and standard CPUs) either by adopting a checkboard decomposition of the mesh or by using a double buffer to store old and new values of the variables. Besides that, a domain decomposition allows for a further level of parallelization by using more than one GPU. Stencil computations usually involve only few neighboring nodes (e.g., in 3D, 6 for the simulation of the Ising spin glass or 26 for the solution of most PDE). As a consequence the *boundaries* that need to be exchanged are small in size compared to the *bulk* and we can expect that the computation in the bulk can hide completely the data exchange even if the computation is pretty simple as it happens to be in the case of the Ising spin glass. Apparently, the lower bound on the number of MPI communications it is achieved by adopting a mesh decomposition

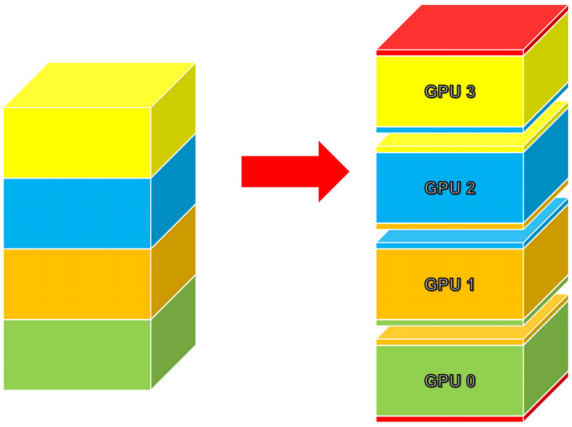

Fig. 3. Domain decomposition along 1 direction.

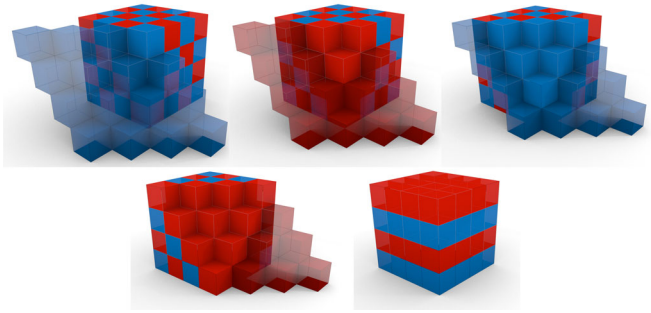

Fig. 4. A depiction of the slicing procedure. Lighter cells are the periodic ones.

along a single direction (e.g., along the z direction) so that each GPU has to exchange boundary data only with two other GPUs (see Fig. 3). Actually, it is possible to reduce the number of communications by using a different memory layout for the cubic stencil access pattern in the checkerboard scheme. The *sliced* layout, recently proposed in reference [7] works only with periodic boundary conditions but it allows to achieve a faster memory access and a higher cache efficiency. It is defined by taking sites belonging to planes orthogonal to one of the diagonals $\boldsymbol{d} = (\pm 1, \pm 1, \pm 1)$. Then, each slice only contains sites of one of the colours of the checkboard decomposition. In the case of nearest neighbours interactions this means that in each slice all sites are decoupled. At the end of the procedure one ends up with a new cubic lattice, with periodic boundary conditions, for which it is always possible to find two opposite one-coloured surfaces. The *slicing* procedure is depicted in Figure 4.

The method, which holds in an arbitrary number of dimensions, has been shown to be rather robust with respect to the usual checkerboard memory layout under variations of the memory load. Further details can be found in reference [7].

The *sliced* memory layout makes possible, as shown in reference [7], a reduction of about 20% of the time required for the update of a single spin with respect to the classic checkboard partitioning scheme and performs better also with respect to a very optimized scheme (based on bitwise operations) that works only when the linear size of the lattice is a power of two, whereas the *sliced* scheme works for any linear size (provided that the lattice has periodic boundary conditions). However, here the intriguing aspect is how the multi-GPU approach is modified when the *sliced* memory layout is adopted. In the classic checkerboard memory layout each boundary is two-coloured so that each process must communicate with two other different processes. In the sliced scheme what happens is that each boundary is only one-coloured, by construction, so that each process only needs to communicate with another one. Of course, the total amount of transferred data is conserved but the number of transaction requests is halved. In Figure 5 we report a depiction of the communication pattern with the *sliced* scheme. For problems that require a domain decomposition among GPUs, this memory layout represents a potential improvement to the overall performance since the total latency of the communication halves (the number of communications is divided by two).

The combination of a suitable memory access pattern and the *streams* support to communication/computation overlap allows to achieve, as shown in Figure 6, a pretty good multi-GPU efficiency (defined as $\eta_{SC} = \frac{T_1}{N \times T_N}$ where T_1 is the execution time on a single GPU and T_N is the execution time by using N GPUs) even for simulations with a low computing intensity like those for the Ising spin glass.

3.2 Particle dynamics in irregular domain decompositions

We now move to describe the issues that arise in multi-GPU programming when the simulation deals with an irregular domain.

Many interesting physical systems are modeled by means of microscopic particles interacting via a predefined set of rules. In some cases the geometry in which the particles move is not relevant and thus the simulations could be performed in box-shaped domains with periodic boundary conditions. In other cases, however, it is interesting to study the dynamics of the particles when their movement is confined in domains with specific geometries. In hemodynamics simulations, for example, particles resembling red blood cells are simulated in spatial domains whose geometries closely resemble those of blood vessels.

Large scale Particle Dynamics (PD) simulations can easily involve the concurrent tracking of hundreds of millions of particles [8]. The memory and computing resources required by those simulations are so high that the only viable approach is to distribute the computations among multiple GPUs. However, there are, at least, two non-trivial problems related to the distribution: interdomain interactions and particles migration.

Hereafter we present a general method to perform efficiently interdomain interactions and particles migration when parallelizing PD with irregular domain decompositions. This approach has been proved to be efficient on

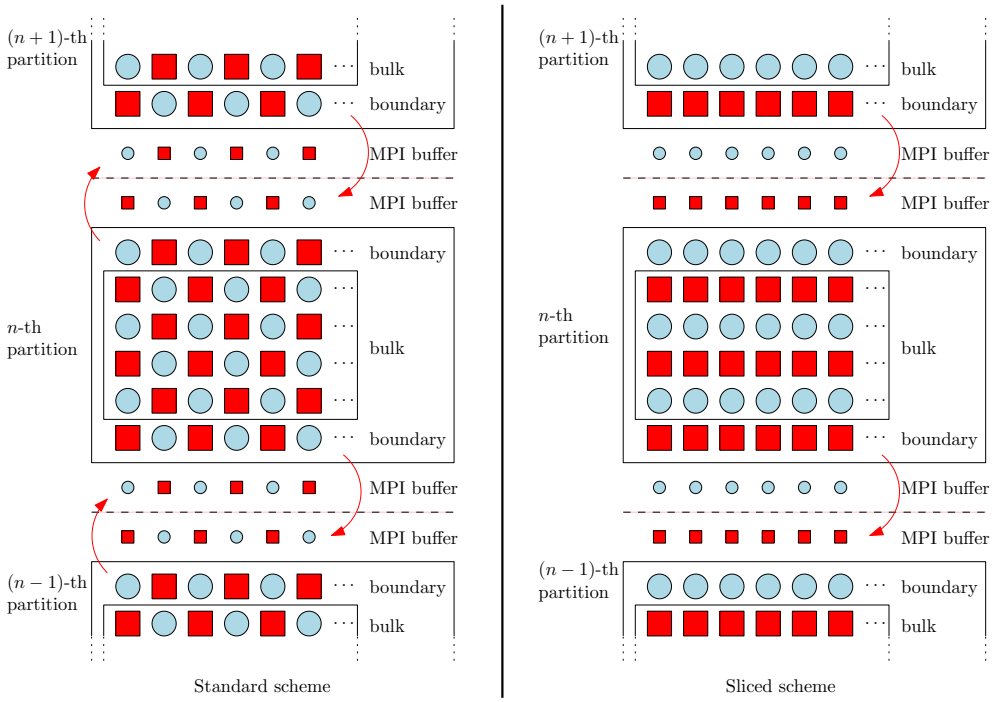

Fig. 5. Partitioning the system along the z axis. Left panel: standard checkerboard. Each boundary contains sites of both colours; a two-ways communication is required during the update. Right panel: sliced scheme. Boundaries are one-coloured, so that the communication is one-way for the same amount of data, thus halving the number of transaction requests.

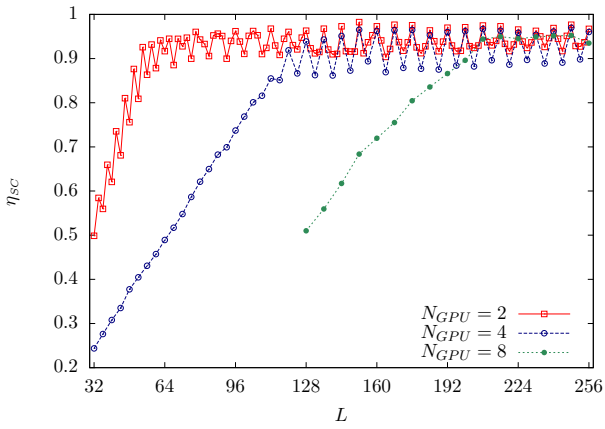

Fig. 6. Strong-scaling efficiency η_{SC} of Ising spin glass simulations for different numbers of GPUs and different system size (L).

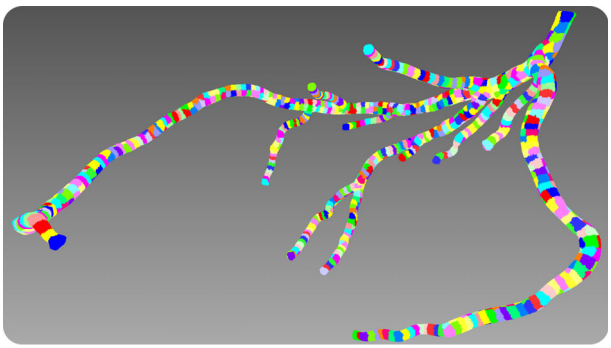

Fig. 7. Optimal partitioning for 1024 processors of an irregular domain representing a full coronary tree. Different colors represent different subdomains.

GPUs [9] even if it has not been designed exclusively for those architectures.

For the sake of generality, we consider a generic PD model where the shape of particles is not relevant and interactions are limited by a cutoff distance that is much smaller than the linear sizes of the bounding box of the subdomains assigned to each processor. Moreover, we assume that the particles are evenly distributed inside the domain (this latter assumption can be easily relaxed by dividing the domain).

3.2.1 Interdomain interactions

The general problem is to run, on a cluster of GPUs, a large scale simulation in which the particles move in a domain with an arbitrarily complex geometry. In order to guarantee a good load balancing among the GPUs, the domain must be partitioned in subdomains with volumes as similar as possible. Moreover, in order to limit the communication loads it is necessary to minimize the contact areas between subdomains. High quality decompositions with such features can be obtained by using graph partitioning algorithms [10]. These solutions, however, produce subdomains with irregular shapes and non-flat contact surfaces that result in a non-trivial communication pattern. As an example, Figure 7 shows a domain representing a coronary

artery tree partitioned in 1024 subdomains with the PT-SCOTCH software [11].

Irregular domain decompositions are not exclusively employed for simulations in complex geometries. There are a number of efforts to optimize load-balancing of PD simulations in regular, box-shaped domains where particles are not uniformly distributed. For example in reference [12], subdomains represented with six tetrahedrons are allowed to deform during the simulation to balance the number of particles handled by the processors. Such approaches typically tile the simulation box with a choice of geometric primitives that represent a trade-off between the level of balancing and the complexity of the resulting communication pattern. In the scenario presently considered however, domain decompositions at the bounding box level may not be a viable option. It is not clear whether these approaches can provide high quality results when partitioning highly sparse, complex geometries for a high number of domains (tens of thousands and more), especially for what concerns both the size and the contact surfaces of the subdomains.

The computation of interdomain interactions requires the processors to exchange all the particles having a distance less than the cutoff from neighboring subdomains. The ideal solution would be to send to neighboring GPUs only the particles that are close enough to the boundaries of their subdomain. However, given the irregularities of contact surfaces, an exact identification of such particles can be impractical. On the other hand, a rough overestimation would result in an unacceptable communication overhead due to the exchange of non-interacting particles. The technique hereafter presented allows to efficiently identify a limited superset of the particles involved in interdomain interactions and to perform selective transfers of the particles based on the distance with target subdomains.

The solution [9] relies on an additional decomposition step performed on the subdomains. Each GPU tiles the bounding box of its subdomain with cubic cells of side equal to the interaction cutoff. The cells having empty intersection with the subdomain are discarded with the exception of those that are neighbors of cells intersecting the contact surfaces with other subdomains. The remaining cells are then grouped into three sets (*internal*, *frontier* and *external* cells) that verify the following properties.

1. Every point of the subdomain is within either an internal or a frontier cell.
2. Internal cells contain only points of the subdomain at distance greater than the cutoff distance from the subdomain boundary.
3. Frontier cells contain all the points of the subdomain at distance less than or equal to the cutoff distance from the subdomain boundary.
4. External cells contain only points outside the subdomain and, together with frontier cells, cover all the external region at distance less than or equal to the cutoff distance from the subdomain boundary.

Figure 8 illustrates an example of such decomposition applied to a simplified two-dimensional geometry. To each

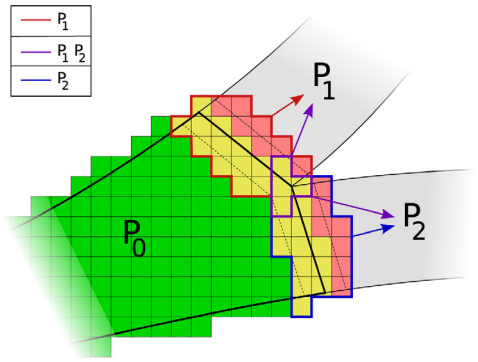

Fig. 8. Tiling of a 2D domain in *external* cells (red), *frontier* cells (yellow) and *internal* cells (green). The dashed line represents the region within a cutoff distance from the domain (solid line). The colored contours around *frontier* and *external* cells represent the neighboring processors lists. The red line encloses the cells at interacting distance only with processor 1, the purple line those at interacting distance with both processors 1 and 2 and the blue one encloses the cells at interacting distance only with processor 2.

cell are then assigned two indices:

- a global index obtained by linearizing the three-dimensional coordinates of the tiling $c_k \times N_j \times N_i + c_j \times N_i + c_i$ ($N_{i,j,k}$ and $c_{i,j,k}$ are, respectively, the number of cells and the cell indices along the x, y and z axes);
- a local index encoding the cell group it belongs to and its index inside the group.

As an example of local indexing, all the cells could be numbered consecutively running on internal, frontier and external cells and saving the first index for each set.

This decomposition requires to maintain the following data structures:

- a *cell array*, containing all the index couples and sorted by global index;
- a *cell matrix*, with a row for each cell containing pointers to the particles in that cell (the particles can be stored in dedicated data structures in no particular order);
- a *connection matrix*, with a row for each frontier and external cell containing the list of neighboring processors that handle domains at interacting distance with the cell.

The connection matrix is typically built using the information about the subdomain interconnection returned by the initial partitioning step.

The cell array is used to bin the particles in the corresponding cells in $O(\log 2(\#cells))$ time. Given a particle at position (p_x, p_y, p_z) the global id (gid) of the cell that contains it can be determined with:

$$gid = \left\lfloor \frac{p_z}{coff} \right\rfloor \times N_j \times N_i + \left\lfloor \frac{p_y}{coff} \right\rfloor \times N_i + \left\lfloor \frac{p_x}{coff} \right\rfloor$$

the local id of the cell can be found by performing a binary search of gid in the cell array. If the search fails, then the particle is not positioned inside the subdomain.

Using this decomposition, interdomain interactions are processed in the following way. Each processor first identifies and exchanges with neighboring processors all the particles that could interact with the outside. Property 3 guarantees that all particles involved in interdomain interactions are contained inside the set of frontier cells. Thus the particles listed in the rows of the cell matrix corresponding to frontier cells are sent to the neighboring processors that are specified in the corresponding rows of the connection matrix. In this way only a limited superset of the particles that could interact with the outer region is transferred. Moreover, data are transferred only to GPUs in close proximity with the particles. This mechanism reduces dramatically the number of duplications within the set of transmitted particles with respect to a multicast approach where all particles in the boundaries are sent to all neighbors. Replication takes place only when more than two subdomains are in touch. Particles located in those regions are sent to more than one processor. On the receiving side, only the particles that could interact with the inner region are considered. Given property 4, the received particles to be retained are filtered depending on the cells they are positioned into. All particles located inside either external or frontier cells are kept and the others are discarded. Finally, particles that are retained can be used to compute interdomain interactions.

3.2.2 Particles migration

Particles migration can be handled following an approach similar to the one used for interdomain interactions. After particles positions are updated, all particles are binned inside the cells they moved into. Particles that moved outside of the GPU subdomain must be exchanged with neighboring processors. Departing particles are found by selecting those that moved into external cells (property 4) and by selecting a subset of those that moved into frontier cells. The identification of particles sitting in frontier cells and moving to neighboring subdomains is carried out by means of a membership test that allows to decide exactly whether a particle lies inside or outside the subdomain. While the implementation of this test strongly depends on the particular representation of the subdomain (i.e. Cartesian mesh, finite elements, etc.) it is reasonable to assume that it exists as a fundamental building block of any PD simulation. Once the list of departing particles is built, a selective transfer to neighboring domains is done by using the connection matrix. In this way, processors send exactly the particles that leave their subdomains, only to the neighbors in proximity of their updated positions. On the receiving side, incoming particles are filtered by using again the membership test and adding newcomers to the local list of particles.

It is apparent how the membership of each particle to one of the three subsets not only reduces the amount of exchanged data but also supports the division of the local work done by each GPU in a boundary (the work done on particles living in external and frontier cells) and in a bulk (the work done on particles living in internal cells) part. This division, in turn, makes possible to overlap communication and computation as described in Section 2.

3.3 Parallel algorithms for large scale graphs

In this section we describe how good performances can be obtained for problems for which a cluster of GPUs does not appear as a very suitable platform. To that purpose, we selected the study of large scale graphs as a representative example of a class of problems characterized by low arithmetic intensity and irregular memory access patterns. Graphs having hundreds of millions of nodes and billions of edges can be found studying the web graph, social networks, protein-protein interaction networks, and bibliographical networks just to mention some application fields.

One of the fundamental building blocks for the development of graph algorithms is the Breadth First Search (BFS). Starting from a *root* node, the connected component of a graph is explored iteratively by traversing, at each step, the unvisited neighbors of the nodes previously discovered. The set of nodes whose neighbors are visited at each step is called *frontier* and the traversal of their edges is called *expansion*. BFS implementations typically result in memory-bandwidth-bound codes whose efficiency depend on the memory access pattern employed during the search. Parallel architectures with very high memory bandwidth, such as GPUs, are often used to accelerate BFS operations by exploiting the fact that the frontier expansion phase can be performed in no particular order with respect to the nodes in the frontier and thus multiple edges can be followed simultaneously. Outstanding results on a single GPU have been reported by some authors [13,14] following different approaches.

However, the need to analyze graphs of steadily-growing size has lead to the development of distributed-memory implementations that combine shared-memory computing nodes (CPUs and/or GPUs) to process graphs whose size is in the Terabytes range. In such parallel and distributed BFS codes, the graph is partitioned among the available GPUs. The partitioning represents a problem in itself and a number of solutions, mostly based on heuristics that depend on the features of the graphs, have been proposed [11,15]. Regardless of the adopted partitioning scheme, the frontier is typically expanded in two phases. First, a local expansion is performed on all the GPUs containing a part of the frontier nodes. Their neighbors are identified and processed in the second phase. Local vertices form the so-called *next level frontier set* (NLFS) and are kept local whereas remote vertices are sent to the processors owning them. On the receiving side, each GPU completes its NLFS by adding the received vertices and a new BFS iteration begins. Although the exact amount of exchanged data depends on (i) the graph being traversed; (ii) the graph partitioning scheme and (iii) the BFS level, communications represent, most of the times, the major bottleneck of distributed-memory BFS codes. For that reason it is always desirable to limit the amount of data exchanged during the BFS iterations.

In the rest of this section, we describe a technique to drastically limit the size of the messages required to exchange information about vertices to be visited during the most communication intensive levels of a BFS and how it can be efficiently implemented for GPUs. For simplicity, we consider a directed graph $G = (V, E)$ (where V indicates the set of vertices and E the set of edges) partitioned among k GPUs. The k GPU are arranged as a logical grid with R rows and C columns and mapped onto the adjacency matrix $A_{N\times N}$. The partitioning is such that:

- the edge lists of the vertices handled by each GPU are partitioned among the processors in the same grid column;
- for each edge, the GPU in charge of the destination vertex is in the same grid row of the edge owner.

The main advantage of the 2D partitioning is a reduction of the number of communications. If P is the number of GPUs, a naive 1D partitioning requires $O(P)$ data transfers at each step whereas the 2D partitioning only requires $2 \times O(\sqrt{P})$ communications (see [16] for further details). With such decomposition, each step of the BFS requires two communication phases, called *expand* and *fold*. The first one involves the processors in the same grid column whereas the second those in the same grid row. With this partitioning to each processor are assigned the edges in a $\frac{N}{R} \times \frac{N}{C}$ sub-matrix of A and $N/(RC)$ vertices. In the most communication intensive steps, each transfer can involve up to the total number of $N/(RC)$ vertices handled by the target processor. In those steps, limiting the amount of data being transferred is crucial to achieve high performance.

The size of the messages can be reduced by following an approach proposed in references [17,18] and extended in reference [16] consisting in using bitmaps for data transfers. The idea is that when the size of the lists of vertices to be sent exceeds, in bits, the number of indices local to the receiving process, then it is more convenient to send a bitmap with the bits corresponding to the outgoing vertices set equal to one and the others equal to zero. This technique reduces the communication times by limiting the data transmitted to a fixed amount whenever the number of vertices to be transferred grows over a given threshold. Obviously, that happens in the most expensive steps of the visit. With the partitioning scheme we adopted the size of bitmap would be $N/(8RC)$ bytes. Assuming that the indices to be transferred are 32-bit words, it is more convenient to transfer the bitmap whenever more than:

$$\frac{N}{32RC}$$

vertices need to be sent. This *trick* makes possible a reduction up to a factor 32 (for 32-bit words) in the size of the messages during the most expensive communication phases. Obviously, the approach is advantageous as long as the overhead imposed by the packing of vertices lists into a bitmap (on the sending side) and the unpacking of the bitmap (on the receiving side) into a vertices list is smaller than the time saved by reducing the size of the messages in the communications.

Since the adjacency matrix and other commonly used data structures, such as level and predecessor arrays, are addressed by using vertices as indices, it is more convenient to process them via their integer representation. For that reason, before and after data transfers, vertices need to be packed and unpacked into/from bit-masks, respectively.

Algorithm 1 GPU unpack of bitmap into vertex list.

Require: bitmap array **bmap** of size **n**
Require: exclusive scan array **cbuf** of size **n**

```
 1: tid = blockIdx.x*blockDim.x+threadIdx.x
 2: wid = tid/warpSize
 3: lid = tid%warpSize
 4: mask = (1≪lid)

 5: if (tid < n) then
 6:    word = bmap[tid]
 7:    offs = cbuf[tid]
 8: end if

 9: for i = 0 to warpSize-1 do
10:    if (wid*32 + i) >= n) then
11:       break
12:    end if
13:    w = __shfl(word, i)
14:    o = __shfl(offs, i)
15:    loc = __popc(w & (mask-1))
16:    if (w & mask) then
17:       out[o + loc] = wid*warpSize*32 + lid + i*32
18:    end if
19: end for
```

The *pack* operation can be easily implemented in CUDA by using *atomicOR* operations. Given a list of vertices, it is sufficient to launch a kernel with a thread per element where thread i reads index v_i and atomically sets the bit at position (v_i mod 32) of the bitmap word ($v_i/32$).

The *unpack* operation, on the other hand, is not so simple because for each bit set equal to one in the bitmap, the position at which the corresponding vertex must be inserted depends on the number of the preceding 1s in the bitmap. For that reason, before the unpack takes place, an exclusive scan on the bitmap array is performed considering, for each word, the number of bits set equal to one in place of the integer value. The resulting array contains, for each word, the starting offset in the output list at which the vertices corresponding to the bits set equal to one should be inserted. This information makes possible to process each word of the bitmap independently from the others having to compute only intra-word vertex offsets. In order to utilize the GPU memory bandwidth efficiently, it is advisable to use a warp-centric approach for the unpack kernel taking advantage of the *shuffle* instructions, available starting on the Kepler architecture (in a few words, threads of a warp can read each others' registers). The idea is to read one bitmap word per thread and to process consecutive blocks of 32 words within the warp.

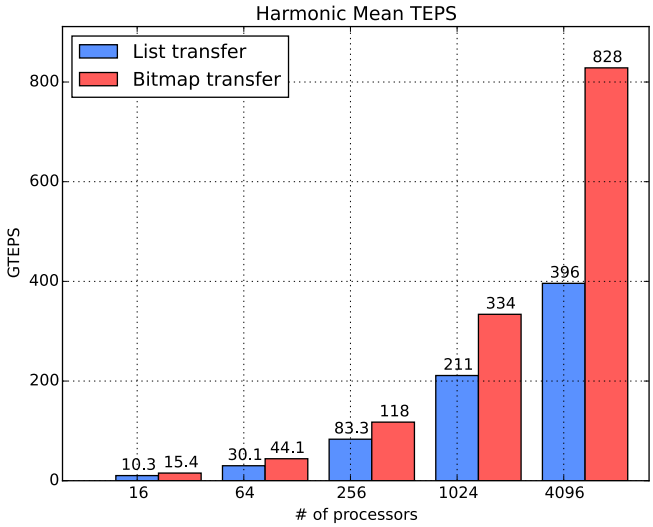

Fig. 9. Comparison of (Giga) Traversed Edges Per Seconds (TEPS) obtained by using a number of K20X GPUs ranging from 1 to 4096 keeping the graph scale per processor fixed. Blue bars (left) report the TEPS obtained with *plain* transfers (exchange of vertices lists). Red bars (right) report the TEPS with *bitmap* transfers.

For each word, only the threads whose position in the warp match the bits set equal to one write the output vertices at consecutive positions. Shuffle operations are used to broadcast the 32 words to the warp threads avoiding additional shared/global memory reads/writes and block synchronizations. Algorithm 1 shows a pseudo-code for the unpack kernel. At the beginning, each thread computes its warp id (line 2), its index inside the warp (the so-called lane id, line 3) and its mask (line 4). In lines 5–8 the bitmap and the offset array are read, one element per-thread. Words are processed consecutively starting from lane 0 to lane 31. Only the words effectively read are processed (the last warp may read less than 32 elements if the bitmap size is not a multiple of 32, lines 10–12). With the shuffle instructions at lines 13–14 the current thread broadcasts its word and offset to the other threads in the warp. Now each thread computes a thread offset by counting the number of bits set equal to one at positions less than its lane id. This is done with the *popc* intrinsic that returns the number of bits that are set in its argument (line 15). Finally, each thread with lane id corresponding to a bit set equal to one (line 16) writes in the output array the global bitmap index of the corresponding bit (i.e. the vertex id, line 17). Consecutive threads write at consecutive positions obtained by adding the thread and the word offsets.

The advantage provided by the reduction of the message size can be dramatic for data intensive applications like graph traversal. As shown in Figure 9, the number of Traversed Edges Per Seconds (TEPS), a performance metric introduced by the graph500 benchmark, doubles by using bitmap transfers and reaches the outstanding value of more than 800 billions by using 4096 K20X GPUs.

4 Conclusions and future perspectives

We have presented results obtained with multi-GPU codes in several disciplines. We showed how the *stream* concept offered by CUDA allows to hide the communication overhead provided that the computation executed concurrently with the communication is long enough. A hybrid multi-GPU code based on a suitable combination of these two mechanisms may reach a very good parallel efficiency. Clusters of GPUs are very promising platforms for large scale simulations of physical systems.

The architecture of high performance clusters with GPUs has not changed very much in the past years. A typical node in a high performance cluster has one or more accelerators connected to a CPU using PCI Express. Even at the fastest PCIe 3.0 speed (8 Giga-transfers per second per lane) and with the widest links (16 lanes), the bandwidth provided over this link is just a fraction of the bandwidth available between the CPU and its system memory. In a multi-GPU node, there may also be PCI-e switches that further reduce the available bandwidth. The systems could still be effectively utilized in most situations, but the developers must take care of overlapping data transfers with computation to hide the transfer time and orchestrate GPU access over PCI-e to maximize performance, using for example some of the techniques discussed in this paper.

The need for this level of tuning is going to disappear in the near future. Upcoming NVIDIA GPUs will offer NVLink, a new high speed connection to the CPU and system memory. Recently, the U.S. Department of Energy announced its plans to build two of the world's fastest supercomputers – the *Summit* system at Oak Ridge National Laboratory and the *Sierra* system at Lawrence Livermore National Laboratory – using this new interconnect. NVLink is a high-bandwidth, energy-efficient path between the GPU and the CPU capable of achieving peak data rates of 80 Gigabytes per second, almost 5 times faster than PCI-e 3.0 and comparable in speed to the current CPU memory systems. NVLink will be available with the next generation Pascal GPU in 2016. IBM has already announced that the POWER CPU will have this new interconnect on die. In addition to increasing the speed of the communication between CPU and GPU, NVLink could also be utilized for GPU to GPU (peer to peer) communications enabling multiple GPUs to share data with very high bandwidth.

Another important hardware feature in upcoming accelerators is stacked memory. Most computational workloads are bandwidth limited. While there have been progress in the memory subsystem (the current NVIDIA GPUs have a memory bandwidth close to 300 GB/s), it is difficult to imagine wider memory interfaces and higher clocks to move in the TB/s range. All the major vendors have announced upcoming products (NVIDIA Pascal GPU, Intel Xeon Phi Knight Landing) featuring stacked memory. Stacked memory is a technology which enables multiple layers of DRAM components to be integrated vertically on the package along with the GPU. It provides several times greater bandwidth, more than twice

the capacity, and quadrupled energy efficiency, compared to current off-package GDDR5. By combining large high-bandwidth memory in the same package with the GPU, the voltage regulators could be placed close to the chip for efficient power delivery. This will also result in a new Pascal module that is one-third the size of current PCIe boards, allowing system vendors to build denser solutions. The large increase in GPU memory size and bandwidth provided by stacked memory will enable GPU applications to access a much larger working set of data at higher bandwidth, improving efficiency and computational throughput, and reducing the frequency of off-GPU transfers.

Starting with CUDA 6, Unified Memory simplifies memory management by giving a single pointer to data, and automatically migrating pages on access to the processor that needs them. On Pascal GPUs, Unified Memory and NVLink will provide the ultimate combination of simplicity and performance. The full-bandwidth access to the CPUs memory system enabled by NVLink means that NVIDIAs GPU can access data in the CPUs memory at the same rate as the CPU can.

We would like to thank Giancarlo Carbone, Matteo Lulli, Enrico Mastrostefano, Giorgio Parisi, Davide Rossetti and Flavio Vella for many useful discussions. We also thank the Swiss National Supercomputing Centre for the access to the "Piz Daint" supercomputer.
All authors contributed equally to the present work.

References

1. NVIDIA CUDA Compute Unified Device Architecture Programming Guide, http://www.nvidia.com/cuda
2. J. Glaser, T.D. Nguyen, J.A. Anderson, P. Lui, F. Spiga, J.A. Millan, D.C. Morse, S.C. Glotzer, Comput. Phys. Commun. **192**, 97 (2015)
3. M. Bernaschi, G. Parisi, L. Parisi, Comput. Phys. Commun. **182**, 6 (2011)
4. T. Preis, P. Virnau, W. Paul, J. Schneider, J. Comput. Phys. **228**, 4468 (2009)
5. M. Weigel, Comput. Phys. Commun. **182**, 1833 (2011)
6. M. Weigel, J. Comput. Phys. **231**, 3064 (2012)
7. M. Lulli, M. Bernaschi, G. Parisi, accepted in Comput. Phys. Commun.
8. M. Bernaschi, G. Amati, M. Bisson, S. Melchionna, S. Succi, Comput. Phys. Commun. **184**, 2 (2012)
9. M. Bisson, M. Bernaschi, S. Melchionna, Commun. Comput. Phys. **10**, 1077 (2011)
10. G. Karypis, V. Kumar, SIAM J. Sci. Comput. **20**, 359 (1999)
11. C. Chevalier, F. Pellegrini, Parallel Comput. **34**, 318 (2008)
12. C. Begau, G. Sutmann, Comput. Phys. Commun. **190**, 51 (2015)
13. D. Merrill, M. Garland, A. Grimshaw, Scalable gpu graph traversal, in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '12* (ACM, New York, 2012), pp. 117–128
14. T. Hiragushi, D. Takahashi, in *Algorithms and Architectures for Parallel Processing*, Lect. Notes Computer Science (Springer, 2013), Vol. 8286, pp. 40–50
15. G. Karypis, V. Kumar, SIAM J. Sci. Comput. **20**, 359 (1999)
16. M. Bernaschi, M. Bisson, E. Mastrostefano, submitted to *IEEE Transactions on Distributed and Parallel Systems*, arXiv:1408.1605 (2014)
17. N. Satish, C. Kim, J. Chhugani, P. Dubey, Large-scale Energy-efficient Graph Traversal: A Path to Efficient Data-intensive Supercomputing, in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC), 2012*
18. F. Checconi, F. Petrini, J. Willcock, A. Lumsdaine, A.R. Choudhury, Y. Sabharwal, Breaking the speed and scalability barriers for graph exploration on distributed-memory machines, in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC), 2012*