Journal of Parallel and Distributed Computing 158 (2021) 138–150

Contents lists available at ScienceDirect

Journal of Parallel and Distributed Computing

www.elsevier.com/locate/jpdc

Multi-GPU systems and Unified Virtual Memory for scientific applications: The case of the NAS multi-zone parallel benchmarks

Marc González *, Enric Morancho

Department of Computer Architecture, Universitat Politècnica de Catalunya - BarcelonaTECH, Spain

A R T I C L E I N F O

Article history:
Received 15 January 2021
Received in revised form 15 June 2021
Accepted 5 August 2021
Available online 13 August 2021

Keywords:
Multi-GPU
Unified Virtual Memory
Single address space
NAS parallel benchmarks

A B S T R A C T

GPU-based computing systems have become a widely accepted solution for the high-performance-computing (HPC) domain. GPUs have shown highly competitive performance-per-watt ratios and can exploit an astonishing level of parallelism. However, exploiting the peak performance of such devices is a challenge, mainly due to the combination of two essential aspects of multi-GPU execution: memory allocation and work distribution. Memory allocation determines the data mapping to GPUs, and therefore conditions all work distribution schemes and communication phases in the application. Unified Virtual Memory simplifies the codification of memory allocations, but its effects on performance depend on how data is used by the devices and how the devices' driver is going to orchestrate the data transfers across the system.

In this paper we present a multi-GPU and Unified Virtual Memory (UM) implementation of the NAS Multi-Zone Parallel Benchmarks which alternate communication and computation phases offering opportunities to overlap these phases. We analyse the programmability and performance effects of the introduction of the UM support.

Our experience shows that the programming efforts for introducing UM are similar to those of having a memory allocation per GPU. On an evaluation environment composed of 2 x IBM Power9 8335-GTH and 4 x GPU NVIDIA V100 (Volta), our UM-based parallelization outperforms the manual memory allocation versions by 1.10x to 1.85x. However, these improvements are highly sensitive to the information forwarded to the devices' driver describing the most convenient location for specific memory regions. We analyse these improvements in terms of the relationship between the computational and communication phases of the applications.

© 2021 The Author(s). Published by Elsevier Inc. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

1. Introduction

GPU-based computing systems have become the de facto standard High Performance Computing (HPC) solution for many computational domains. Machine Learning, BioInformatics, Scientific Computing and many more are domains that have clear examples of representative applications like TensorFlow [1], Caffe [21], Smith-Waterman [17], Alya [10] that provide support for GPU-based systems. One particular aspect of GPU-based systems is their programmability. Depending on the nature of the application, many changes have to be introduced, usually involving a partial redesign, implementation and tuning. These efforts have been accepted mainly due to the impressive performance levels that GPU-based systems can deliver.

Some multi-GPU systems have incorporated a transparent management of the memory allocation across both the GPUs and CPUs. In particular, CUDA Unified Virtual Memory [19,20,26,27] allows the programmer to allocate memory and offload computations to the GPUs with no regard of where the memory has been physically allocated. In addition to a page fault mechanism, the GPU driver allocates memory and moves data based on computation demand. Programmability is greatly improved at the cost of some potential performance loss due to the interference of the driver activity (e.g.: page faults, update of TLB structures) that might cause a serialization of computation and communication. CUDA includes support to guide the driver to help avoid some of these overheads. This support is based on run-time primitives that inform the run-time about when and how the computation will use data and its associated memory. In general, the complexity of these primitives resides on knowing where to place them within the application

* Corresponding author.
E-mail addresses: marc@ac.upc.edu (M. González), enricm@ac.upc.edu (E. Morancho).

https://doi.org/10.1016/j.jpdc.2021.08.001
0743-7315/© 2021 The Author(s). Published by Elsevier Inc. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

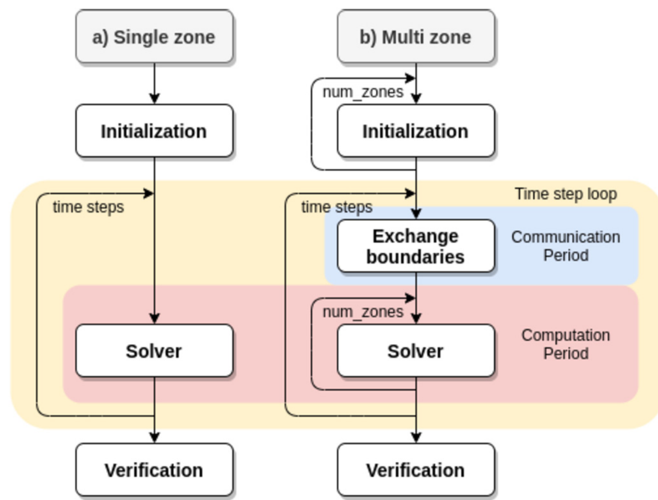


Fig. 1. Flow graphs of single-zone and multi-zone BT, SP and LU.

sources. The programmer has to ensure that they get processed sufficiently in advance to generate the desired effect.

The main contribution of this work is the evaluation of the Unified Virtual Memory support for multi-GPU systems and scientific codes. We use the multi-zone NPB [6] benchmark suite, a standard for numerical applications and for evaluating parallel systems. Our evaluation analyzes both performance and programmability. Performance is evaluated mainly in terms of speedup. Programmability is evaluated in terms of code complexity related to memory allocation and optimization of data transfers between the devices. In particular, we focus on the use of the Unified Virtual Memory primitives for memory allocation, guiding the placement of data structures and pre-fetch operations. The proposal has been implemented with CUDA 10.0 and executed on a IBM Power9 8335-GTG and 4 GPUs NVIDIA V100 (Volta). We have observed speedup factors that range from 1.10x to 1.85x of our UM-based parallelization with respect to a multi-GPU parallelization that uses manual allocation per GPU.

This paper is organized as follows: Section 2 describes the structure of the parallelism in the NPB-MZ suite, Section 3 describes the evaluation analysis of a Unified Virtual Memory version of the NPB-MZ benchmark suite, Section 4 discusses prior works related to this paper and, finally, Section 5 discusses the main conclusions.

2. GPU parallelization of NPB-MZ benchmarks

2.1. NPB-MZ benchmark suite

The Multi-Zone NPB (NPB-MZ) [6] re-implements the NAS Parallel Benchmarks (NPB) [4] to expose a coarse level of parallelism. In this work we focus just on the three pseudo applications of the suites: BT, SP and LU. Both suites define several input classes for each benchmark (classes S, W, A, B, ... determine a sequence of increasing size of the input 3D volume) and include a verification mechanism for the results.

The flow graph of the original NPB applications is shown in Fig. 1-a). The benchmarks perform a time step loop where at each iteration the solver updates the input volume. They differ on the solver: Block Tri-diagonal (BT), Scalar Penta-diagonal (SP) or Lower-Upper Gauss-Seidel (LU).

NPB-MZ implementations (BT-MZ, SP-MZ and LU-MZ) divide the input 3D volume into a 2D tiling through x and y dimensions (Fig. 2). We refer to the tiles as zones; each zone has four adjacent zones (neighbourhood assumes a toroidal topology).

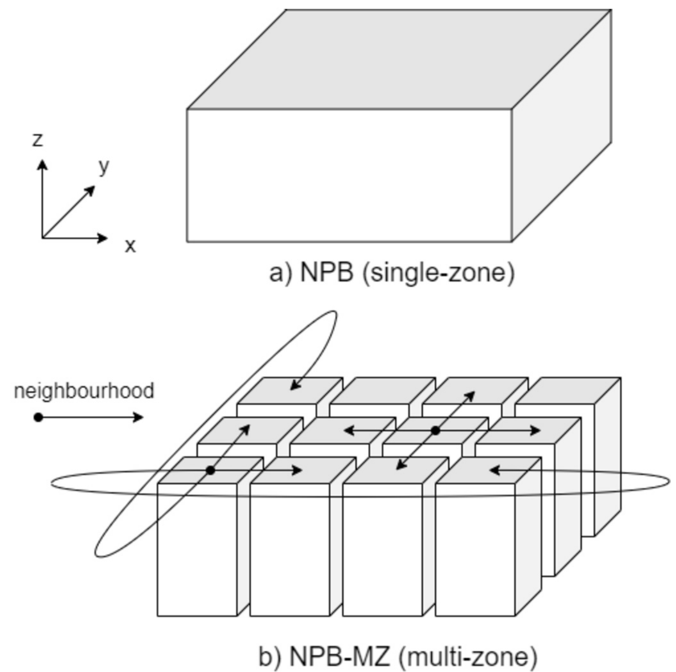


Fig. 2. a) NPB input volume. b) Tiling performed by NPB-MZ and neighbourhood relation between zones.

The flow graph of NPB-MZ benchmarks (Fig. 1-b) has an additional loop level within the time step loop with respect to the single-zone flow graph. This additional loop traverses the zones and applies the solver to each zone; as these iterations are independent, this loop exposes a coarse level of parallelism. However, before applying the solver on the next time step, the boundary values of each zone must be exchanged with its neighbour zones.

Table 1 shows, for some input classes, the overall size of the 3D volume (in terms of number of points and Gigabytes) for all benchmarks. The Table also shows the number of zones created by each NPB-MZ benchmark. We observe that LU-MZ always creates 16 zones; however, in SP-MZ and BT-MZ, the larger the class, the larger the number of zones. This difference impacts the performance trends of the benchmarks in terms of scalability.

Table 1 shows the sizes of the zones created by each NPB-MZ benchmark. For each input class, LU-MZ and SP-MZ creates equally-sized zones; however, BT-MZ creates zones with a wide variety of sizes (the ratio between the largest size and the smallest size is about 20). This difference will impact the performance trends of the benchmarks in terms of load balancing.

Table 1 shows the number of time steps performed by each benchmark at each input class.

2.2. Execution periods and kernel execution order

As shown in Fig. 1-b, the flow graph of NPB-MZ benchmarks is the same. First, data structures are allocated and initialized. Next, the time-step loop solves the equations; the number of time-step iterations depends on both the benchmark and the input class (Table 1). Each iteration is divided into two periods, *Communication Period* and *Computation Period*, that are executed sequentially. Finally, the correctness of the results is checked by a verification procedure.

Fig. 3 zooms into the time-step loop to show the sequence of functions (computational phases) called by each period. In NPB-MZ applications:

Table 1
Characterization of NPB-MZ benchmarks: overall size (number of points and memory requirements) and, for each benchmark, number of zones, zone size and number of time steps.

Input class	3D volume x × y × z (points)	Memory (GB)	Num. zones (x × y) LU	SP & BT	Zone size (points per zone) LU	SP	BT	Time steps LU	SP	BT
B	304 × 208 × 17	≈ 0.2	4 × 4	8 × 8	67.184	16.786	from 2.992 to 59.976	250	400	200
C	480 × 320 × 28	≈ 0.8	4 × 4	16 × 16	268.800	16.800	from 2.912 to 60.648	250	400	200
D	1.632 × 1.216 × 34	≈ 13	4 × 4	32 × 32	4.217.088	65.892	from 11.968 to 243.236	300	500	250
E	4.224 × 3.456 × 92	≈ 250	4 × 4	64 × 64	83.939.328	327.888	from 59.248 to 1.203.452	300	500	250

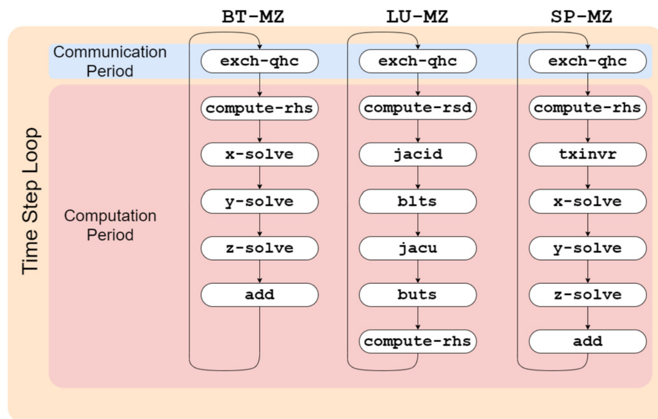


Fig. 3. Time-step loop of NPB-MZ benchmarks. Sequence of functions, computational phases, called at each time-step iteration.

- The *Communication Period* propagates information from each zone to its neighbour zones (boundary exchange). Zones are processed sequentially. For each zone, its adjacent 2D surfaces must be retrieved from its neighbour zones in order to update the surfaces and to send them back.
- The *Computation Period* updates each zone's contents by sequentially executing several functions. Each function processes all zones; inside each function, zones can be processed in parallel.

2.3. Sources of parallelism

The NPB-MZ suite exhibits several levels of parallelism. This section details the sources of the available parallelism at each period.

2.3.1. Communication Period: intra-zone parallelism

Algorithm 1 depicts the pseudo-code of three versions of the boundary-exchange procedure.

1) The sequential CPU version, a), processes zones one after the other; no inter-zone parallelism is available. However, some intra-zone parallelism is exposed; next versions will exploit it.

2) The single-GPU version, b), exploits the intra-zone parallelism by the definition of CUDA kernels.

3) The multi-GPU version, c), must introduce data-transfer statements because neighbour zones may be mapped to different GPU's. So, before a GPU starts processing a zone, if its neighbour zones are mapped to other GPU's, their adjacent surfaces must be copied to the GPU. After the computation, the results must be sent back to the GPU's where the zones are mapped.

2.3.2. Computation Period: inter-zone parallelism

At each iteration of the time-step loop, the *Computation Period* calls a sequence of procedures (Fig. 3); each procedure performs a computational phase of the solver. The serial implementation of each computational phase traverses all zones. However, as the computational phase of each zone is independent from the other zones, this loop exposes parallelism: inter-zone parallelism.

This parallelism can be exploited by an OpenMP implementation (Algorithm 2-a). A parallel zone is created and a pool of threads (one for each GPU) competes to process the zones. The runtime procedures `get-task()` and `commit-task()` are in charge of assigning each zone to only one thread.

2.3.3. Computation Period: intra-zone parallelism

Each computational phase of the *Computation Period* (Fig. 3) is composed of several loop nests that traverse all zone points. These loops expose a new level of parallelism: intra-zone parallelism. This parallelism can be exploited by several CUDA kernels, one for each loop nest. Each loop nest is transformed to a CUDA kernel implemented within C++; its iteration space is mapped into the block grid and GPU thread blocks. For this process, all CUDA kernels have been coded with special focus in avoiding warp divergence and maximizing the memory coalesced accesses at the warp level. Both the block grid and thread blocks have been defined in accordance to the memory layout of matrix-based data structures. This corresponds to practices observed in previous works [2,7] and conforming to best CUDA practices applied to both the NPB and NPB-MZ benchmark suites. Algorithm 2-b depicts both the transformation from the sequential code to the CUDA kernel and the CUDA kernel invocation.

2.4. Evaluated NPB-MZ implementations

2.4.1. Manual memory allocation

We have implemented a multi-GPU parallel version of the NPB-MZ using manual memory allocation. Then, according to the zone-to-GPU mapping, the data structures of each zone are allocated into a particular GPU during the *Initialization Period*. This mapping is constant; cannot be changed during the time-step loop. Therefore, this approach needs to know in advance what work distribution scheme is applied at the inter-zone parallel level. In addition, manual data transfers have to be implemented in the *Communication Period* to exchange boundary values according to the adjacencies of the zones.

2.4.2. Unified Virtual Memory (UM)

When UM is introduced, the restrictions observed in the manual version are no longer applicable. It is not necessary to know in advance which GPU will process which zone during the *Computation Period*. Similarly, explicit data transfers during the *Communication Period* are no longer needed. The devices' driver automatically moves the required data among the GPUs. When the *Communication Period* is executed, accessing adjacent zones residing on different GPUs will cause the driver to migrate or replicate the memory pages containing border elements. Similarly, when the *Computation Period* is executed, pages that were moved in the previous *Communication Period* are transferred back to the device where the zone must be processed. In conclusion, programmability is greatly simplified with respect to manual memory allocation.

We have designed two more UM-based versions, one making usage of the primitives referring to preferred page location, the other using primitives for data pre-fetch.

Algorithm 1 Code scheme for the exchange of boundary values. a) depicts the sequential host version. b) depicts the single-GPU version where intra-zone parallelism is coded in the form of a kernel invocation. c) depicts the multi-GPU version with zone transfer statements.

a) CPU

```
! CPU version

do zone = 1, num_zones
  east_zone  = adjacency_east[zone]
  north_zone = adjacency_north[zone]
  copy_face(tmpEast, mesh[east_zone])
  copy_face(tmpNorth, mesh[north_zone])
  compute_border(mesh[zone], tmpEast, tmpNorth)
  copy_face(mesh[east_zone], tmpEast)
  copy_face(mesh[north_zone], tmpNorth)
```

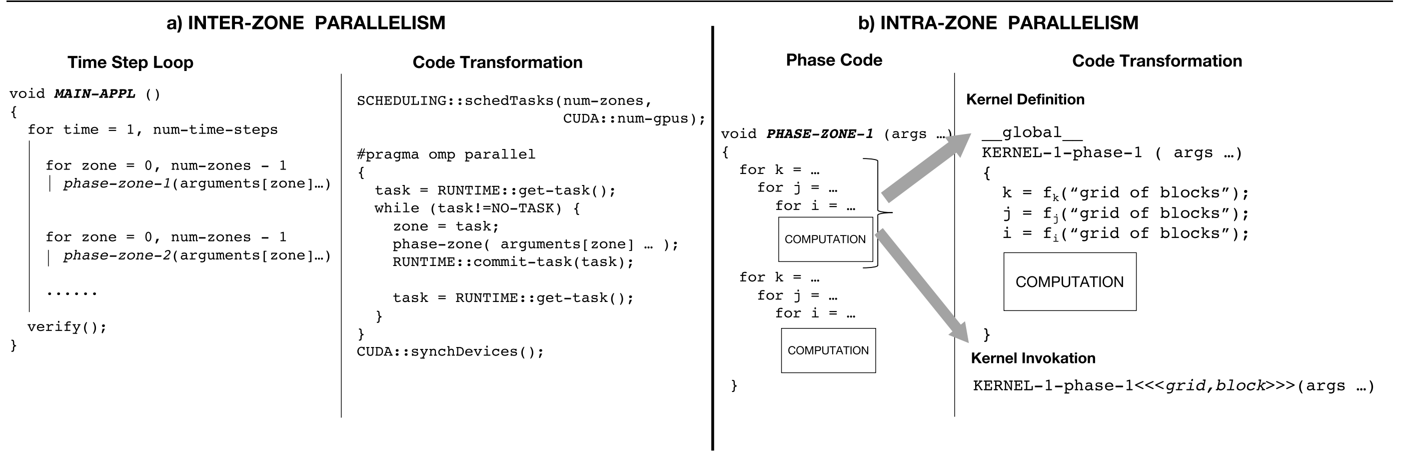
b) Single-GPU

```
! Single-GPU version

do zone = 1, num_zones
  east_zone  = adjacency_east[zone]
  north_zone = adjacency_north[zone]
  CUDA::compute_border<<< grid, block, shared >>>
    (mesh[zone], mesh[east_zone], mesh[north_zone])
```

c) Multi-GPU

```
! Multi-GPU version

do zone = 1, num_zones
  east_zone  = adjacency_east[zone]
  north_zone = adjacency_north[zone]
  east_gpu   = zone_gpu_mapping[east_zone]
  north_gpu  = zone_gpu_mapping[north_zone]
  zone_gpu   = zone_gpu_mapping[zone]
  CUDA::copy_zone
    (east_gpu, mesh[east_zone], zone_gpu, tmp1)
  CUDA::copy_zone
    (north_gpu, mesh[east_zone], zone_gpu, tmp2)
  CUDA::compute_border<<<grid, block, shared>>>
    (mesh[zone], tmp1, tmp2)
  CUDA::copy_zone
    (zone_gpu, tmp1, east_gpu, mesh[east_zone])
  CUDA::copy_zone
    (zone_gpu, tmp2, east_gpu, mesh[north_zone])
```

Algorithm 2 a) Inter-Zone Parallelism: The left side shows the code skeleton for the time-step loop. Its body is composed of as many loops (*zone-phase* loops) as the number of phases. Each *zone-phase* loop traverses the zones and applies a phase over a zone on each iteration. The right side shows the OpenMP implementation that distributes the iterations of the *zone-phase* loops (e.g.: the zones) to the GPUs. An OpenMP parallel region is defined and one thread controls one CPU, executing the body of the parallel region. The *while* loop invokes the runtime system to acquire work. b) Intra-Zone Parallelism: Each phase is coded as a subroutine where iterative structures implement a computation over the multi-dimensional matrices that represent one zone. Each iterative structure is converted into a CUDA kernel (Kernel Definition), and substituted by a kernel CUDA call (Kernel Invocation). Each kernel operates only over a single zone.



Unified Virtual Memory with Advises: The UM support is open to some guidance coming from the programmer. In particular, it is possible to advise the run-time system about access patterns and the placement of memory regions. In particular, the programmer can inform whether a memory region is going to be read, written or both. Also, the programmer can advise the run-time about which device should host a memory region (e.g.: zone), thus specify which GPU has to allocate the physical memory associated to the memory region. We have generated versions of the NPB.MZ applications using this set of functionalities. Algorithm 2-b exposes the code structure to orchestrate the computation of a particular zone into a specific GPU. This code has been modified so that it includes the advises to the CUDA run-time system. We forward the type of access (read or write) and the preferred location, in this case the GPU that is going to process the zone (e.g.: we use the thread id in OpenMP to identify the GPU in charge of processing a zone).

Unified Virtual Memory with Data Pre-fetch: The UM run-time also supports Pre-fetching. We have adapted the *Communication Period* to use this functionality so that pre-fetch is applied to transfer in advance adjacent zones, and also to transfer back these zones to where they originally have been mapped according to the scheduling applied at the inter-zone level of parallelism. Pre-fetching actions have been inserted within the code exposed in Algorithm 1-c (e.g.: *copy-zone* calls become asynchronous, and the original loop has been unrolled to overlap border computation with communication).

2.5. Programmability assessment

This subsection points out some initial conclusions regarding the overall programmability improvements offered by the use of Unified Virtual Memory. In general, there are two main aspects that are directly affected by the introduction of Unified Virtual Memory. On the one hand, how the application is conditioned by the data structure placement across the different device memory subsystems. On the other hand, this placement determines the communication patterns. For the NPB-MZ benchmark suite, these two aspects correspond to the zone placement and the zone transfers.

2.5.1. Zone placement

In non-UM versions, zone allocation is performed at a per-device level. This means that the programmer has to design the GPU parallelization under a static and fixed zone-to-GPU mapping,

assigning the zone processing in advance and keeping it constant through the execution. In contrast, UM versions require a single and global zone allocation that immediately becomes visible to all devices including the host. Thus, the programmer is relieved of the burden of redesigning the application to support a dynamic zone-to-GPU mapping. However, if UM advises are introduced (which is highly recommended) to guide the devices' driver for memory placement issues, then programmability is again affected by coding efforts to explicitly define the zone-to-GPU mapping, type of accesses (e.g.: read/write, only read, only write) and forward this information to the devices' driver. Consequently, only if UM advises are avoided, programmability is greatly improved but, in general, UM advises improve the performance of applications. In terms of coding efforts, the amount of new code lines introduced to activate the UM support is not significant. Complexity is related to memory advises, and becomes similar to that of a manual memory allocation per device. The reason for that is that in both cases the programmer needs to deploy an explicit description of the memory placement, whether in the form of memory allocation runtime calls or in the form of memory placement advises.

2.5.2. Zone transfers

During the *Communication Period*, adjacent zones have to be transferred to the GPU in charge of processing a particular zone. For non-UM versions, this translates to explicit data transfer operations involving one or two entire zones. In addition, temporary buffers have to be allocated in all devices to store the zones transferred from other devices during the *Communication Period*. For UM versions this changes drastically. With UM, the programmer is relieved of introducing explicit zone transfers. The devices' driver moves data under computation demand. Therefore, as GPU kernels reference border data of adjacent zones, data is moved accordingly. This corresponds to a significant programmability improvement. We will see in the next section that it also introduces significant performance improvements. During the *Communication Period* and under UM, entire zones are no longer transferred. Instead, only memory pages containing border elements are transferred and this results in less data transfers and shorter communication times with respect to non-UM versions.

3. Evaluation

This section is divided into two parts. Firstly, we profile the execution of NPB-MZ applications in terms of different metrics in order to understand and justify the observed performance. Secondly, we study overall performance in terms of strict speedup numbers with respect to a manual CUDA parallelization that does not rely on UM.

All experiments have been conducted on a system composed of 2 x IBM Power9 8335-GTH @ 2.4 GHz with 512 GB of main memory and 4 x GPU NVIDIA V100 (Volta) with 16 GB HBM2. All applications have been coded and compiled within the CUDA10.0 framework [19,20,26,27].

3.1. Performance characterization

This subsection characterizes the three NPB-MZ applications in terms of execution-time distribution, number of data transfers between GPUs and amount of data transferred. We have implemented four versions of each application. First, a non UM version where memory allocation, kernels and data transfers have been manually coded (labelled *application name*-**man**). Then, an initial UM version obtained from the *man* version where memory is allocated using the UM interface (labelled *app-name*-***um***). Finally, two optimized versions that use advise and pre-fetch primitives:

Table 2
Execution time distribution for SP-MZ-man. Input class D (1024 equally sized zones). Static scheduler over 4 GPUs.

SP-MZ-man (D) — **4 GPUs, static scheduler**

Kernel	Aggregated: Exec time (%)	Aggregated: Exec time (s)	Per call: Avg. time (ms)	Per call: Min. time (ms)	Per call: Max. time (ms)
x_solve_kernel	28.18	5.34	0.13	0.12	0.14
z_solve_kernel	27.16	5.14	0.12	0.12	0.14
y_solve_kernel	22.22	4.21	0.10	0.10	0.11
compute_rhs_kernel_2	7.27	1.38	0.03	0.03	0.04
[CUDA memcpy PtoP]	3.40	0.64	0.06	0.04	0.09
txinvr_kernel	3.23	0.61	0.01	0.01	0.02
compute_rhs_kernel_1	2.74	0.52	0.01	0.01	0.01
Other	5.80	1.10	-	-	-

app-name-***um-advise*** (just uses CUDA advises) and *app-name*-***um-all*** (uses both CUDA advises and pre-fetch). All performance data has been collected using the profiling tool provided by NVIDIA [22,23].

3.1.1. SP-MZ

Table 2 depicts the contribution of the most time-consuming kernels to the execution time of *SP-MZ-man* application running input class D over 4 GPUs. For each kernel, the table shows its contribution to the overall execution time (in terms of percentage and time), and the average, minimum and maximum execution times of its invocations. In this version, 4 kernels take almost 85% of total execution time: *x_solve_kernel*, *y_solve_kernel*, *z_solve_kernel* and *compute_rhs_kernel_2*; they belong to the *Computation Period*: The execution times of the kernels range from 97 µs to 143 µs for the solver kernels, and from 30 µs to 40 µs for the *compute_rhs_kernel_2*. Observe that *compute_rhs_kernel_1* kernel represents 2.74% of execution time with bursts that range from 8 µs to 14 µs, averaging 12.08 µs.

The border-computation kernels (*copy_north_south_face* and *copy_east_west_face*, not shown in table) take 2.43%, a very small part of total execution; the execution bursts of both kernels range from 4.48 µs to 10.43 µs (averaging 6.05 µs and 4.91 µs respectively). Regarding the communications, peer-to-peer transfers (*CUDA_memcpy_PtoP* kernel) take about 3.40% of total execution time. Consequently, the execution is totally dominated by the *Computation Period*, and in particular by the solver kernels. Also, we observe how the computational kernels do not present a significant variance between their Min. Time, Max. Time and Avg. Time.

The data provided in the previous paragraph will be relevant in our comparison versus the execution under UM support in order to understand the effects of UM in the execution of the SP-MZ application. Table 3 shows time distribution in the same manner as we have described it for the *man* version but now for the *um* version. The first thing to notice is the different time distribution among the kernels. The kernels *compute_rhs_kernel_1* and *copy_north_south_face_kernel* take almost 60% of total execution time with very different execution bursts with respect to the *man* version. These kernels now expose a Max. time in their execution bursts of 14,06 ms and 12,08 ms. These changes are due to the CUDA driver handling of the page faults occurring during the execution of these kernels. Fig. 4 shows the list of kernels and their execution order for the SP-MZ application. The kernels are executed along the Initialization, Communication, Computation and Verification periods. This order is important to understand the influence of UM support on the execution time distribution among the kernels. Notice that the *compute_rhs_kernel_1* is executed immediately after the *Communication Period*. This means that pages that were moved across the

Table 3
Execution time distribution for SP-MZ-um. Input class D (1024 equally sized zones). Static scheduler over 4 GPUs.

SP-MZ-um (D) 4 GPUs, static scheduler	Exec time (%)	Exec time (s)	Avg. time (ms)	Min. time (ms)	Max. time (ms)
Kernel	Aggregated		Per call		
compute_rhs_kernel_1	37.64	20.06	0.47	0.01	14.06
copy_north_south_face_kernel	20.85	11.11	0.26	0.01	12.08
x_solve_kernel	10.07	5.37	0.13	0.12	0.14
z_solve_kernel	9.69	5.16	0.12	0.12	0.13
y_solve_kernel	7.93	4.23	0.10	0.10	0.11
exact_rhs_kernel_init	4.55	2.42	2.37	0.39	7.31
initialize_kernel	3.57	1.90	0.93	0.01	4.17
Other	5.70	3.04	-	-	-

Period	Kernel	Execution Order
Initialization	exact_rhs_kernel_init	1st
	exact_rhs_kernel_x	2nd
	exact_rhs_kernel_y	3rd
	exact_rhs_kernel_z	4th
	initialize_kernel	5th
Communication	copy_east_west_face_kernel	1st
	copy_north_south_face_kernel	2nd
Computation	compute_rhs_kernel_1	1st
	compute_rhs_kernel_2	2nd
	txinvr_kernel	3rd
	x_solve_kernel	4th
	z_solve_kernel	5th
	y_solve_kernel	6th
	add_kernel	7th
Verification	error_norm_kernel	1st
	rhs_norm_kernel	2nd

Fig. 4. Kernel along Initialization, Communication, Computation and Verification periods for SP-MZ.

different GPUs according to the zone adjacency during *Communication Period*, now have to be moved back to the GPU where the *Computation Period* will actually happen. This corresponds to the execution of the *compute_rhs_kernel_1*. The same happens with both *copy_north_south_face_kernel* and *copy_east_west_face_kernel*, but this time when switching between the computation and the communication periods. At this point, all pages associated to adjacent zones but residing in different GPUs have to be moved where the border computation happens. These observations justify that the computational kernels *x_solve_kernel*, *y_solve_kernel*, *z_solve_kernel* and *compute_rhs_2_kernel* now only correspond to a 25% of total execution time. For overall execution time comparison, at the end of this section both the *Computation Period* and *Communication Period* as well as the total execution time are addressed and compared for all versions of the SP-MZ application (see Fig. 6).

The introduction of the UM support also affects the *Initialization Period*. Notice how the kernels in *Initialization Period* have changed their behaviour: *exact_rhs_kernel_init* and *initialize_kernel* account for 8% of total execution time with Max. Time execution bursts of 7 ms and 4 ms respectively. The reason for this change is the same as the one we have observed for the computational kernels. Memory is touched for the first time in the GPUs with the execution of these initializing kernels.

Fig. 5 details significant metrics regarding the execution under the UM support. The uppermost part of the figure details event counting for version *SP-MZ-um*. Memory thrashes refer to driver activity related to ping-pong page movements between GPUs. Remote mappings from/to device refer to the action of page replication from and to a GPU, but with no redefinition of the preferred location of a page. Transfers from/to device, refer to actual data transfers that have occurred per GPU. These counters and the data on top of their series describe how much data has been transferred and how many driver bursts of execution were needed. All data corresponds to an aggregation of 40 iterations of the execution of the application. For *SP-MZ-um*, memory thrashes are present in all GPUs and in the range of a thousand for GPUs 0 and 1, and one hundred for GPUs 2 and 3 (the chart shows data in logarithmic scale). Remote mapping counters describe an imbalance between the GPUs: 0 and 1 account for most of these events. Given that work is scheduled under a static scheme where all GPUs receive the same number of zones to compute (including their borders), this is clearly a symptom of wrong page placement. The communications imposed by border computation mislead the driver in setting what should be the correct location for each zone: replicas are generated for pages that contain data belonging to the zone borders. Notice that data transfers will constantly occur while executing the Computation and Communication Period, as those two are entangled due to the movement and replication of pages containing border data. These transfers are in the range 10.000 and 30.000 and the total amount of data transferred per GPU is between 2 GB and 3 GB.

If advises are forwarded to the device driver, then the performance of the UM support changes drastically. Table 4 shows time distribution for the *SP-MZ-um-advise* application. In general, we observe that we recover the time distribution seen for the *SP-MZ-man* version. This implies that the interference of the driver to move data across the devices has been minimized up to a point so that communications happening inside the kernel executions do not alter the overall computational weight of each kernel. Notice also how the differences between Avg. Time, Min. Time and Max. Time is only significant for kernels in the *Initialization Period*. All kernels in the *Computation Period* do not present significant interference coming from the device driver activity: notice the *compute_rhs_kernel_1* which now is not in the 3 most heavy computational kernels and it does present some variance in its execution bursts, but not in the range of what has been observed in version *SP-MZ-um* (see Table 3). The kernels in the *Communication Period* do present differences in their Avg. Time, Min. Time and Max. Time. This is reasonable as these kernels require driver activity to move the zone pages where the zone border resides. Regarding the UM profile, we observe how all data transfers now are in the range of one thousand (logarithmic scale in Fig. 5). No memory thrashes appear and remote mappings are in the range of one hundred per GPU. This indicates that the driver has distributed the zones over the GPUs according to the advises and the pattern of usage: pages containing border data are replicated without generating any thrashing.

Nevertheless, the *SP-MZ-um* benefits from the utilization of UM. Notice that in the *SP-MZ-man* version, peer-to-peer transfers account for 3,4% (see Table 2) of overall execution time. Now all these data transfers are automatically done by the driver according to the memory footprint the GPUs define along the zone and border computation.

Fig. 6 shows the execution times for one iteration of both the Compute and the Communication periods (column graphs, left axis) and the total execution time (line graph, right axis). For the *SP-MZ-man* version, the partial execution time for Communication and Computation periods are 28.83 ms and 115.67 ms on each iteration of the application (for class D there are 300 iterations).

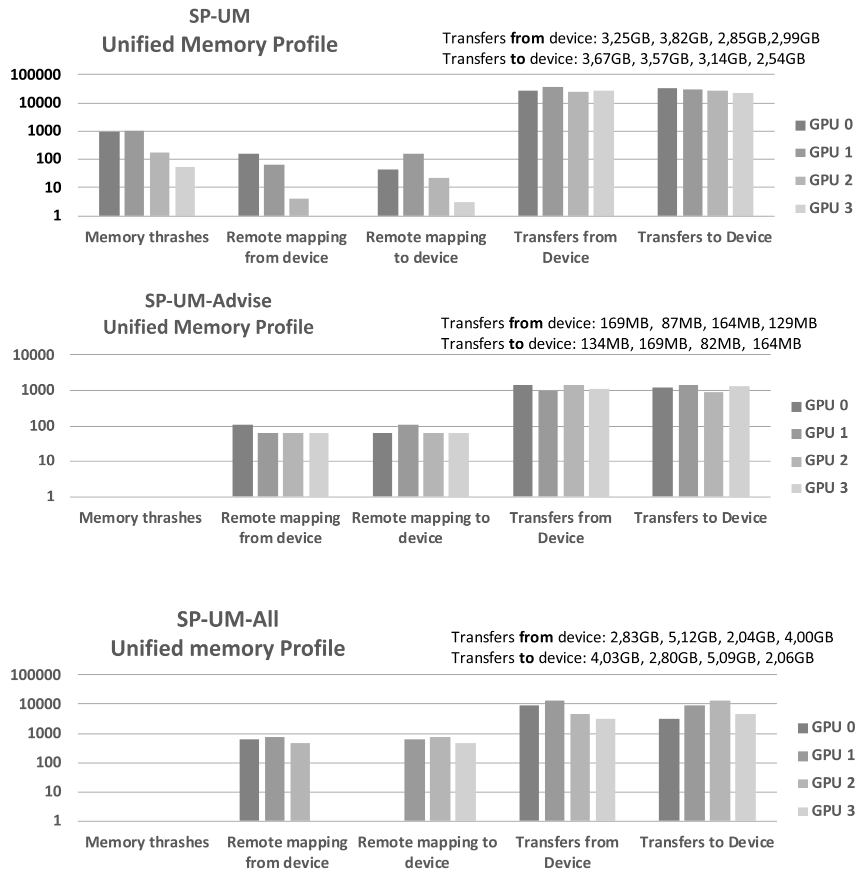


Fig. 5. Statistics of the driver activity related to UM events for SP-MZ (40 iterations). Input class D (1024 equally sized zones). Static scheduler over 4 GPUs. Memory thrashes refer to driver activity related to ping-pong page movements between GPUs. Remote mappings refer to the action of page replication from and to a GPU, but with no redefinition of the preferred location of a page. Transfers from and to, refer to actual data transfers that have occurred per GPU.

Table 4
Execution time distribution for SP-MZ-um-advise. Input class D (1024 equally sized zones). Static scheduler over 4 GPUs.

SP-MZ-um-advise (D)
4 GPUs, static scheduler

Kernel	Exec time (%)	Exec time (s)	Avg. time (ms)	Min. time (ms)	Max. time (ms)
	Aggregated		Per call		
x_solve_kernel	22.72	5.35	0.13	0.12	0.14
z_solve_kernel	21.88	5.16	0.12	0.12	0.13
y_solve_kernel	17.91	4.22	0.10	0.10	0.12
exact_rhs_kernel_init	10.54	2.48	2.43	0.40	10.75
initialize_kernel	7.88	1.86	0.91	0.01	4.52
compute_rhs_kernel_2	5.82	1.37	0.03	0.03	0.04
compute_rhs_kernel_1	3.73	0.88	0.02	0.01	5.45
copy_north_south_face_kernel	3.58	0.84	0.02	5.57	4.59
Other	5.75	1.36	-	-	-

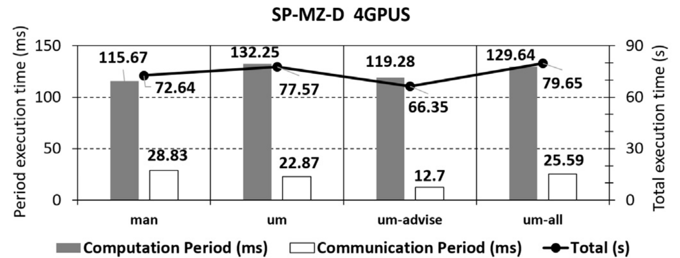


Fig. 6. Execution time for SP-MZ versions *man*, *um*, *um-advise* and *um-all*. Input class D (1024 equally sized zones). Static scheduler over 4 GPUs. Left axis: partial execution time of an iteration of the Computation and the Communication Period. Right axis: total execution time.

Total execution time is 72.64 s. For the *SP-MZ-um* version, the partial execution time for Communication and Computation periods are 22.87 ms and 132.25 ms on each iteration of the application. Total execution time is 77.57 s. So, the *SP-MZ-um* is slower than *SP-MZ-man* but *Communication Period* executes faster while the *Computation Period* slower. The reason for that is that the SP-MZ application processes many zones (1024 in input class D) but of a small size. Therefore, the amount of communication needed to move an entire zone or just the memory pages that contain border elements is similar given the memory layout of zones (e.g.: 3D matrices structured in contiguous memory). Yet, the total amount of communication to transfer an entire zone is larger than just transferring the border elements. The *SP-MZ-man* always transfers a whole zone whenever this zone is bordered by another residing in a different GPU. In contrast, the *SP-MZ-um* version moves less data, but induces more overhead from driver activity (e.g.: pages faults, update of TLB structures). The *SP-MZ-um-advise* version takes advantages of the fact that now only pages containing border data are moved and with much less driver activity. This explains the execution times observed in this version: 12.7 ms and 119.28 ms for Communication and Computation periods, and 66.35 s for total execution time.

The UM support includes pre-fetch advises to guide the driver. One option is to introduce them in the *Communication Period* to try to overlap the border computation with transfers containing border data of other zones. *SP-MZ-um-all* version includes this optimization. But the pre-fetch advises only accept a description of

Table 5
Execution time distribution for LU-MZ-man. Input class D (16 equally sized zones). Static scheduler over 4 GPUs.

LU-MZ-man (D) — 4 GPUs, static scheduler

Kernel	Aggregated: Exec time (%)	Aggregated: Exec time (s)	Per call: Avg. time (ms)	Per call: Min. time (ms)	Per call: Max. time (ms)
jacld_blts_kernel	22.54	5.14	0.01	0.01	0.02
[CUDA memcpy PtoP]	21.42	4.88	3.72	2.32	5.25
jacu_buts_kernel	20.72	4.72	0.01	0.01	0.02
rhs_kernel_z	11.71	2.67	3.88	3.83	4.23
rhs_kernel_y	10.84	2.47	3.59	3.50	3.70
rhs_kernel_init	5.74	1.31	1.90	1.82	1.97
Other	7.03	1.60	0.00	0.00	0.00

Period	Kernel	Execution Order
Communication	copy_east_west_face_kernel	1st
	copy_north_south_face_kernel	2nd
Computation	ssor_kernel1	1st
	jacld_blts_kernel	2nd
	jacu_buts_kernel	3rd
	ssor_kernel2	4th

Fig. 7. Kernel execution order along Communication and Computation periods for LU-MZ.

the data in terms of initial address and total amount of data to be pre-fetched. This is not suitable for the border layout as this one occupies many non-consecutive pages in memory. For this version, the partial execution time for Communication and Computation periods are 25.59 ms and 129,64 ms on each iteration of the application. Total execution time is 79.65 s. Clearly the pre-fetch advises are generating again too much communication and driver activity. The bottom graph of Fig. 5 details the amount of data and the driver activity. This version moves a similar amount of data as the *SP-MZ-um* version with similar remote mappings and data transfers. Therefore, the driver does not capture the communication pattern given by the memory page usage that both the Computation and Communication period expose. Notice that in terms of programmability, it would significant increase complexity for the programmer to identify the access patterns during the border computation and generate an accurate description of which data is going to be used at memory page level so that this can be translated at run-time into the appropriate calls to the pre-fetch CUDA primitives so we did not implement this.

3.1.2. LU-MZ

The LU-MZ application operates with a reduced set of zones (16) but of a greater size than the SP-MZ application (as detailed in Table 1). This is important as we will show that the relation between the total amount of data and the amount of border data is going to be an essential parameter to understand the effect of the UM support. Table 5 shows the execution time distribution among the most time consuming application kernels. Notice that peer-to-peer transfers (*CUDA_memcpy_PtoP*) take up to 22% of total execution time. This is related to the zone transfers during the *Communication Period*. The *Computation Period* is dominated by 3 kernels (*jacld_blts_kernel*, *jacu_buts_kernel* and *rhs_kernel_z*) accounting for about 56% of total execution time.

Fig. 7 shows the execution order between the computational kernels but only for those in the *Computation Period* and *Communication Period*. As in the previous section, this order as well as the repetition of the two periods along the iterative process of the application is essential to understand the changes caused by the use of UM. For LU-MZ, there are 10 initializing kernels and 2 verifying kernels that have been omitted in the list.

Table 6
Execution time distribution for LU-MZ-um. Input class D (16 equally sized zones). Static scheduler over 4 GPUs.

LU-MZ-um (D) — 4 GPUs, static scheduler

Kernel	Aggregated: Exec time (%)	Aggregated: Exec time (s)	Per call: Avg. time (ms)	Per call: Min. time (ms)	Per call: Max. time (ms)
copy_north_south_face_kernel	29.58	8.53	13.01	2.00	48.02
jacld_blts_kernel	19.63	5.66	0.01	0.01	0.84
jacu_buts_kernel	16.37	4.72	0.01	0.01	0.03
rhs_kernel_z	9.68	2.79	4.06	3.93	4.23
rhs_kernel_y	8.98	2.59	3.77	3.60	3.97
rhs_kernel_init	4.92	1.42	2.06	1.85	10.09
erhs_kernel_init	2.79	0.80	50.27	38.15	64.48
ssor_kernel2	2.45	0.71	1.08	1.00	1.19
Other	5.60	1.62	-	-	-

Table 7
Execution time distribution for LU-MZ-um-advise. Input class D (16 equally sized zones). Static scheduler over 4 GPUs.

LU-MZ-um-advise (D) — 4 GPUs, static scheduler

Kernel	Aggregated: Exec time (%)	Aggregated: Exec time (s)	Per call: Avg. time (ms)	Per call: Min. time (ms)	Per call: Max. time (ms)
jacld_blts_kernel	26.12	5.16	0.01	0.01	0.31
jacu_buts_kernel	23.98	4.74	0.01	0.01	0.02
rhs_kernel_z	14.08	2.78	4.04	3.92	4.17
rhs_kernel_y	13.04	2.57	3.74	3.55	3.95
rhs_kernel_init	7.18	1.42	2.06	1.83	9.95
erhs_kernel_init	3.67	0.72	45.26	31.44	57.24
ssor_kernel2	3.56	0.70	1.07	1.01	1.20
rhs_kernel_x	3.20	0.63	0.92	0.89	0.95
Other	5.18	1.02	-	-	-

Table 6 shows the execution-time distribution for the *LU-MZ-um* version. First, note the lack of peer-to-peer transfers because all data transfers are triggered automatically by the devices' driver intervention. Second, the most time-consuming kernel is *copy_north_south_face_kernel*, a kernel that computes part of the border computation. This increase is justified by the fact that along its computation, the devices miss the data of the adjacent zones which reside in a different device. Thus, the relation between the original computation within the kernel and the amount of overhead and data transfers changes completely as well as its overall execution time and computational weight. If we check for the kernels that have greater disparities between their Avg. Time, Min. Time and Max. Time we see that this kernel has average execution times closer to its minimum value and far away from its maximum value. This indicates that the driver activity (e.g.: pages faults, update of TLB structures) is huge compared to the actual amount of computation. Notice that the aggregate time for the other kernels remains very similar to those observed in the *LU-MZ-man* version. For overall execution time comparison, at the end of this section both the *Computation Period* and *Communication Period* as well as the total execution time are addressed and compared for all versions of the LU-MZ application (see Fig. 6).

If advises are introduced, then the problems observed in the previous paragraph disappear. Table 7 shows the new execution time distribution. In this version no kernel presents significant disparities between the Avg. Time, Min. Time and Max. Time. Also the distribution of execution time and its aggregate value are the

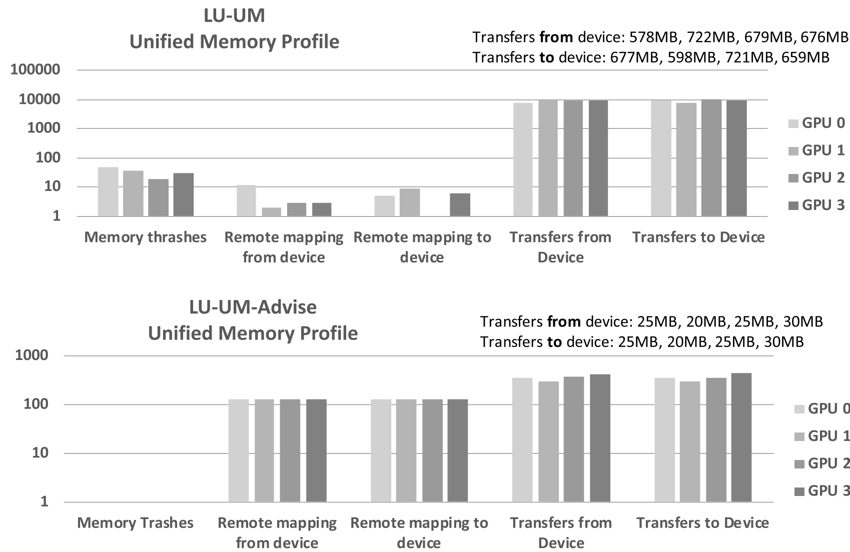


Fig. 8. Statistics of the driver activity related to UM events for LU-MZ. Input class D (16 equally sized zones). Static scheduler over 4 GPUs. Memory thrashes refer to driver activity related to ping-pong page movements between GPUs. Remote mappings refer to the action of page replication from and to a GPU, but with no redefinition of the preferred location of a page. Transfers from and to, refer to actual data transfers that have occurred per GPU.

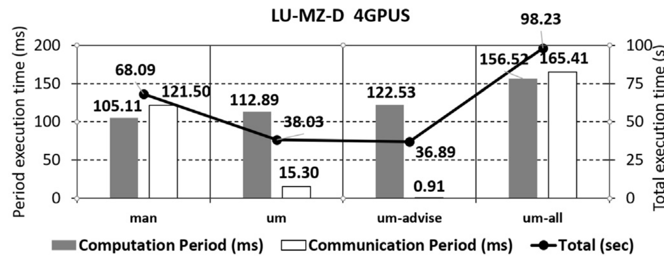


Fig. 9. Execution time for LU-MZ versions *man*, *um*, *um-advise* and *um-all*. Input class D (16 equally sized zones). Static scheduler over 4 GPUs. Left axis: partial execution time of an iteration of the Computation and the Communication Period. Right axis: total execution time.

same as in the *LU-MZ-man* version, indicating that the amount of overhead introduced by the driver is no longer significant. Fig. 8 shows the accounting for both *LU-MZ-um* and *LU-MZ-um-advise* versions. For the *LU-MZ-um* version we observe memory thrashes in the range close to one hundred per GPU, that disappear in the *LU-MZ-um-advise* version and are substituted by remote memory mappings. This indicates that the driver avoids memory page faults and instead activates the replication of shared memory pages. Notice how the total amount of data transferred per GPU drops from 500 MB-700 MB to 20 MB-30 MB (data on top of the transfers from and to the devices).

The effects of UM support in performance are shown in Fig. 9. For the *LU-MZ-man* version, the partial execution time for communication and computation periods are 121.50 ms and 105.11 ms on each iteration of the application (for class D there are 250 iterations); the total execution time is 68.09 s. For the *LU-MZ-um* version, the partial execution time for communication and computation periods are 0.18 ms and 124.05 ms on each iteration of the application; total execution time is 37.34 s. The impressive improvements generated by the UM support are justified by one essential aspect of the LU-MZ application. In this application there are just 16 zones but of huge size (in contrast, SP-MZ presents many zones but of much more smaller size). So, the ratio between the data transfers associated to border elements and their computation is very favourable to the UM versions. While the *LU-MZ-man* version moves entire zones to perform border computations, the *LU-MZ-um* version transfers just the memory pages that strictly contain the border elements. This explains the drop in the execution time for the *Communication Period*. Compared with the introduction of advises, the *LU-MZ-um-advise* version moves less data too with less overhead coming from the driver activity (e.g.: pages faults, update of TLB structures). This explains the execution times observed in this version: 0.20 ms and 122.53 ms for Communication and Computation periods; total execution time is 36.25 s. We also include the numbers for the *LU-MZ-um-all* version (includes pre-fetch operations) as it has been done with the SP-MZ application. The performance for this version is even worse than that observed with the *LU-MZ-man* version, mainly due to entire zones being moved by the pre-fetch operations, misleading the driver and generating the same amount of communication as in the *LU-MZ-man* version plus additional driver overheads.

3.1.3. BT-MZ

The BT-MZ application presents very similar responses to the activation of the UM support, as those observed in the SP-MZ and LU-MZ applications. Therefore we skip the profiling results for this application. The conclusions are the same: the *BT-MZ-um-advise* version runs faster mainly due to the fact that it requires less data transfers and does not incur significant driver interventions. Fig. 10 in its uppermost chart shows the execution time for the Computation and Communication periods as well as total execution time for all studied versions. For the *BT-MZ-man* version, the partial execution time for the Communication and Computation periods are 747.50 ms and 28.50 ms on each iteration of the application (for class D there are 500 iterations); total execution time is 194 s. For the *BT-MZ-um* version, the partial execution time for communication and computation periods are 762 ms and 25.50 ms on each iteration of the application; total execution time is 196.88 s. For the *BT-MZ-um-advise* version, the partial execution time for communication and computation periods are 757 ms and 10.50 ms on each iteration of the application. Total execution time is 192 s. The introduction of advises reduces the execution time for the *Communication Period*, but introduces driver activity along the *Computation Period*. Combining both we explain why the UM versions perform similar to the manual version.

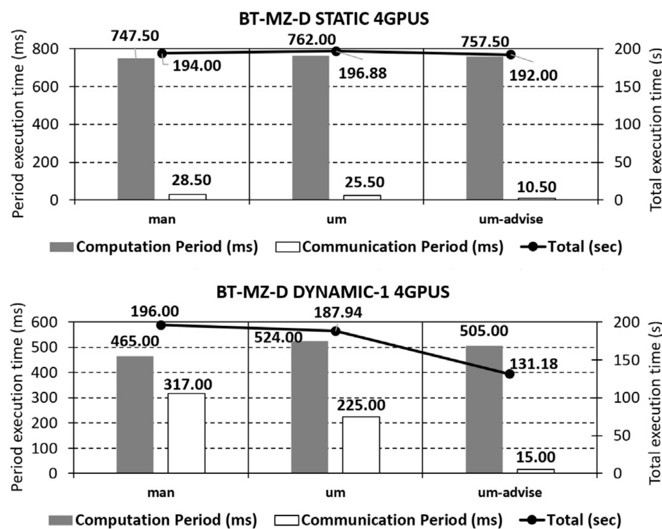


Fig. 10. Execution time for BT-MZ versions *man*, *um* and *um-advise*. Input class D (1024 non-equally sized zones). Static scheduler (top graph) and Dynamic scheduler (bottom graph) over 4 GPUs. Left axis: partial execution time of an iteration of the Computation and the Communication Period. Right axis: total execution time.

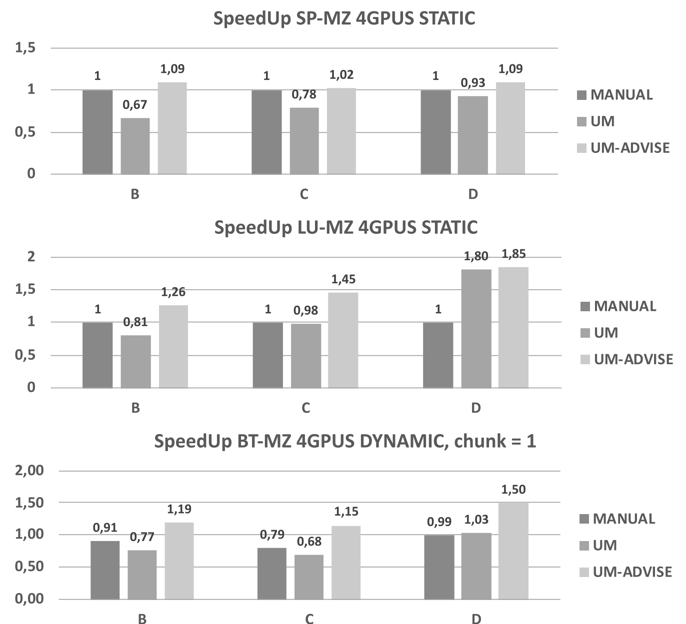


Fig. 11. Overall performance for SP-MZ, LU-MZ and BT-MZ applications for versions *man*, *um*, *um-advise*. STATIC and DYNAMIC,1 schedulers over 4 GPUs. Input classes B, C, D.

However, BT-MZ is unique within the NPB-MZ suite: its input is composed of non equally sized zones, which leads to load unbalance at its inter-zone parallelism level. If zones are distributed under a static scheduling, then GPUs receive a different amount of total work. UM allows for a straightforward implementation of other work scheduling schemes with no need of re-coding the application. Changing the scheduler in the code in Algorithm 2-a for the inter-zone parallelism changes the zone distribution among GPUs without having to perform memory allocation operations and track where does each zone reside. We have implemented a dynamic scheduling which is applied to the *Computation Period* and tries to balance the *Computation Period* at the cost of introducing some synchronization overhead. Notice that the CPUs orchestrating the execution on the GPUs will eventually synchronize to obtain work (e.g.: zones) to offload it to the devices. We have implemented a variant of this type of scheduling so that it memorizes the zone-to-GPU mapping after the first time the *Computation Period* is executed. Every time after this initial step, the *Computation Period* is executed guided by the mapping obtained in its first iteration. Fig. 10 in its lowermost chart depicts the execution time for the Computation and Communication periods as well as the total execution time for all studied versions under a dynamic schedule with chunk equal to 1. We can see that the *Computation Period* improves by 1.45x and 1.60x. Although, notice how the *Communication Period* for the manual version now performs worse. The reason for this is the dynamic scheduling completely changes the distribution of zones compared with static assignment. Adjacency is no longer maximized within the zone-to-GPU mapping (static scheduler does maximize adjacency [12]), so many more entire zone transfers have to be performed. In contrast, UM versions just move the memory pages that contain the border elements, not the entire zone. But the *BT-MZ-um* suffers from a significant driver intervention that is solved by the introduction of advises. In the *BT-MZ-um-advise* version the *Communication Period* recovers its original performance levels and the *Computation Period* is well balanced among GPUs.

3.1.4. Qualitative conclusions

We have seen that UM support has a positive impact on performance (almost a 1.10 speedup factor for SP-MZ, 1.85 for LU-MZ and 1.5 for BT-MZ), but requires the programmer to encode essential information regarding the specific location and access type (read/write) to the data structures. This introduces programmability issues as the programmer has to code the application so that work distribution schemes (in this case a static/dynamic scheduler) are translated into data placement events. Although, we observe that the original non UM multi-GPU code also has to deal with a similar problem, it is limited to the memory allocation stage, where for each GPU the programmer needs to manually code its memory allocation, therefore statically binding which GPU will process which zone. In addition, significant improvements come from the fact that with UM all data transfers happen automatically with no intervention of the programmer. Moreover, with UM only the strictly necessary memory pages are moved across the system, improving the original manual versions where entire zones are transferred.

Finally, UM allows the introduction of more convenient scheduling schemes that solve the work imbalance that appears at the inter-zone parallelism level. The efforts for implementing dynamic schedulers on the manual version are much harder than in the UM versions, giving much less performance in return.

3.2. Overall performance

In this subsection we address the overall performance within the NPB-MZ suite. We show the speedup achieved by the UM versions with respect to the manual version.

3.2.1. SP-MZ

The uppermost chart of Fig. 11 depicts the performance for SP-MZ application with input classes B, C and D executed by 4 GPUs using a static scheduling. Three versions are compared: *SP-MZ-man*, *SP-MZ-um* and *SP-MZ-um-advise*. The usage of UM produces an improvement of almost 10% only when advises are introduced. The version *SP-MZ-um* presents some slowdown mainly because the devices' driver has insufficient information about the optimal location of the memory pages for each zone. As discussed in Section 3.1, memory pages associated to border elements are repeatedly transferred between the GPUs due to the border computation. In particular for the SP-MZ application, the relation between the

total amount of zone elements and border elements explains the magnitude of the UM effect. If this relation implies that moving the memory pages containing border elements almost corresponds to transferring the entire zone, then the performance effects of UM are minimum. This is the case for the SP-MZ application where the zone dimensions generate this relation (in SP-MZ we have many zones but of small size, see Table 1).

3.2.2. LU-MZ

The middle chart of Fig. 11 depicts the performance for LU-MZ application with input classes B, C and D executed by 4 GPUs using a static schedule. We observe that only the UM versions that use advises expose a significant improvement. As it has been mentioned in the previous section, the version *LU-MZ-um* presents some slowdown mainly because the devices' driver has insufficient information about the most convenient location of the memory pages for each zone. For *LU-MZ-um-advises* speedups are in the range of 1.25, 1.45 and 1.85. We observe that the larger the input class the greater the effect of the UM support.

In general, the relation between the overall size of each zone and the size of its borders determines the impact of the UM support. The main advantage of the UM versions is that communication is limited to the border elements. While non-UM versions transfer the whole zones, UM versions avoid transferring the whole zones because they transfer just the memory pages strictly containing border elements. If the memory layout of the whole zone and the memory layout of the border elements are such that to move the border elements at page level we are almost transferring the whole zone, then the benefits of UM are minimum. This explains why in smaller input classes we observe less improvement than in larger input classes.

3.2.3. BT-MZ

The BT-MZ application presents a significant load imbalance in its inter-zone level of parallelism. The mesh zones are of very different size, and this imbalance increases with the input class. UM allows for an immediate implementation of appropriate work-distribution schemes without requiring explicit memory transfers between GPUs.

The bottom-most chart of Fig. 11 depicts the performance for BT-MZ application with input classes B, C and D executed by 4 GPUs using a variant of a dynamic scheduling with chunk=1. The variant is based on a memorizing capability of the scheduler that retains the mapping of zones to devices for subsequent executions [5,25,28]. The speedup numbers are obtained comparing the fastest version of the *BT-MZ-man* execution under a static schedule versus executions of the *BT-MZ-man*, *BT-MZ-um* and *BT-MZ-um-advise* versions under the dynamic schedule with the memorize capability.

Notice that the dynamic execution of the *BT-MZ-man* version is not taking any advantage of the new scheduling. With the dynamic scheduling the zone adjacency is no longer maximized in terms of zone and GPU affinity. Thus, many more entire zone transfers occur during the *Communication Period* of the application. This has been studied in Section 3.1.3. In Fig. 10 and for input class D we observe how for this version the dynamic scheduler increases the *Communication Period* by almost a 12x factor. In contrast, for the same version, the dynamic scheduling reduces the time of the *Computation Period* by a factor of 1.60x (from 747 ms to 465 ms). Version *BT-MZ-um* behaves similarly but due to different reasons. Its *Computation Period* is reduced by similar factor (from 762 ms to 524 ms, 1.45x) given the new work distribution scheme, but its *Communication Period* is increased by almost 10x. This is explained by the driver activity that is not able to capture what should be the most appropriate location for zone memory pages, given the amount of page movements associated to the border computation. The *BT-MZ-um-advise* version solves this issue, exposing speedup factors of 1.19, 1.15 and 1.50 for classes B, C and D respectively. For this version the improvements are justified by the correct placement of zones and by the more efficient load balance.

4. Related work

Several studies have analyzed both performance and programmability of applications using the Unified Virtual Memory model. In [16], the impact of UM in applications is generally studied in several applications of different domains. At the programmability level, there have been studies that translate high level constructs, such as OpenMP directives into CUDA code using UM: in [18] the programming model support for UM in OpenMP is evaluated through an extended LLVM compiler that translates OpenMP directives into CUDA code.

At the operating system level, there have been proposals to manage and guide the page placement and optimize overheads and communications. Heterogeneous Memory Management (HMM) for the Linux kernel [11,26] provides mechanisms to mirror the CPU page table on the devices and to integrate the device memory subsystem into the virtual memory address space.

CPU to GPU interconnects are another factor that directly impacts the performance of data movement. There have been evaluations of the interconnect network on modern GPU systems [13,14]. In [24] a micro-benchmark framework is developed a to evaluate the raw bandwidth performance with UM.

Some works apply advanced features of UM in the scope of Deep-Learning frameworks. One example is OC-DNN [3], an extended Caffe framework that uses UM to support the training of out-of-core batch sizes. They use memory advises to trigger data evictions and prefetch directives to trigger migrations. In general these techniques are found useful in optimizing training performance but incorrect use can lead to performance degradation.

Memory over-subscription in GPU memory requires efficient page eviction to make space for newly requested pages. [9] proposed two pre-eviction policies using a tree-based neighbourhood pre-fetching technique to select candidate pages. [15] introduced a memory management framework named ETC (Eviction, Throttling and Compression) that includes eager page pre-eviction, memory-aware throttling to avoid memory thrashing and data compression at page level. These optimization techniques target future GPU designs that require hardware modifications to be effective.

Work balance and loop scheduling have been studied for many types of applications and for both shared and distributed memory architectures. Adaptive loop schedulers have been proposed based on information gathered at runtime. In [5,25,28] loop schedulers combine information gathering (e.g.: runtime execution times or actual sizes of data structures) with the ability to memorizing the work assignment produced by the scheduler itself. Specifically for NAS-MZ benchmarks, [8] describes a feedback scheduler based on execution times to determine thread distribution and work assignment for NUMA shared memory architectures.

We conclude this section with works that have ported the NPB or NPB-MZ suites to GPU systems. Dümmler and Rünger [7] evaluated NPB-MZ benchmarks on hybrid CPU+GPU architectures. They decomposed the workloads and, using a static scheduling, distributed them among the CPU's or the GPU. Their evaluations show a significant performance improvement with respect to both pure GPU and pure CPU implementations. But this work does not include any study on the usage of Unified Virtual Memory.

In [2], the authors compare a CUDA implementation of the NPB benchmarks with implementations in OpenCL and OpenACC. This work outlines some good CUDA programming practices applied to the NPB benchmark suite. These practices have been used in

our parallelization of the NPB-MZ suite. In particular, all CUDA kernels have been coded with special focus in avoiding warp divergence and maximizing the memory coalesced accesses at the warp level. Loop nests have been transformed to CUDA kernels where both the block grid and thread blocks have been defined in accordance to the memory layout of matrix-based data structures.

In [12] a multi-GPU parallelization is described and evaluated but without making usage of the UM support. Instead, this work focus on the relation between the work schedulers, data mapping on the devices and the cost of communications between the devices for the NPB-MZ benchmark suite.

5. Conclusions

In this paper we have shown how the Unified Virtual Memory support improves both programmability and performance of multi-GPU applications. Programmability is significantly improved due to two main changes that come from the use of UM. First, memory allocation can be coded once for all GPUs, avoiding specific memory allocation on a per device manner. Second, all necessary data transfers among the devices can be left to the device driver, avoiding manual data transfers for communication phases in the application. For this specific aspect, programmability is affected as we have observed that best performance levels are achieved when the programmer forwards information to the run-time system about the most convenient location for application data structures. This requires similar efforts as to those of per-GPU memory allocation.

In terms of performance, the main improvements come from the communication optimization that UVM can introduce. For programmers, moving data structures between the devices is naturally performed with no regard of how these data structures will be used. So, sometimes, entire regions of memory are transferred between GPUs. In contrast, UM is able to select the memory pages that are strictly necessary for a computation to be performed. This can lead to more optimized communications. We have observed another positive aspect of the UM support. For applications with significant load imbalance between the devices, UM allows for an immediate implementation of appropriate work distribution schemes with no regard on whether memory allocation has been performed on each device. In particular, we have implemented a dynamic scheduler that outperforms a static scheduler using the per-GPU memory allocation scheme.

The current GPU implementation of the NPB-MZ suite can be optimized with data halo regions for border computations. During the communication phases only fine grain parallelism is exploited and data transfers are interleaved with computational bursts. These data transfers can be optimized with packing/unpacking actions of border data so that the communication overheads are minimized. The study of the UM support over this type of computations remains as future work.

For the NPB-MZ benchmark suite, we have observed speedup factors that range from 1.10x to 1.85x of our UM-based parallelization relative to our non-UM manual memory allocation versions on a system composed of 2 x IBM Power9 8335-GTH and 4 x GPU NVIDIA V100 (Volta).

Declaration of competing interest

The authors declare having no conflict of interest of any kind.

Acknowledgments

This work was supported by the Spanish Ministry of Science and Technology (PID2019-107255GB).

References

[1] M. Abadi, et al., TensorFlow: large-scale machine learning on heterogeneous distributed systems, http://download.tensorflow.org/paper/whitepaper2015.pdf, 2015.
[2] G. Araujo, D. Griebler, M. Danelutto, L. Fernandes, Efficient NAS parallel benchmark kernels with CUDA, in: 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, 2020.
[3] A.A. Awan, C.-H. Chu, H. Subramoni, X. Lu, D.K. Panda, OC-DNN: exploiting advanced unified memory capabilities in CUDA 9 and Volta GPUs for out-of-core DNN training, in: Proceedings of IEEE 25th International Conference on High Performance Computing (HiPC), IEEE, 2018, pp. 143–152.
[4] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, S. Weeratunga, The NAS parallel benchmarks, Int. J. High Perform. Comput. Appl. 5 (3) (1991) 63–73, https://doi.org/10.1177/109434209100500306.
[5] J.M. Bull, Feedback guided dynamic loop scheduling: algorithms and experiments, in: D. Pritchard, J. Reeve (Eds.), Euro-Par'98 Parallel Processing, 1998, pp. 377–382.
[6] R.F.V. der Wijngaart, H. Jin, NAS Parallel Benchmarks, Multi-Zone Versions, Tech. Rep. NAS-03-010, NASA Ames Research Center, Jul 2003.
[7] J. Dümmler, G. Rünger, Execution schemes for the NPB-MZ benchmarks on hybrid architectures: a comparative study, in: M. Bader, A. Bode, H. Bungartz, M. Gerndt, G.R. Joubert, F.J. Peters (Eds.), Procs. of the Intl. Conf. on Parallel Computing, ParCo 2013, in: Advances in Parallel Computing, vol. 25, IOS Press, 2013, pp. 733–742.
[8] A. Duran, M. Gonzàlez, J. Corbalán, Automatic thread distribution for nested parallelism in OpenMP, in: Proceedings of the 19th Annual International Conference on Supercomputing, 2005, pp. 121–130.
[9] D. Ganguly, Z. Zhang, J. Yang, R. Melhem, Interplay between hardware prefetcher and page eviction policy in CPU-GPU unified virtual memory, in: Proceedings of the 46th International Symposium on Computer Architecture (ISCA), ACM, 2019, pp. 224–235.
[10] G. Giuntoli, J. Grasset, A. Figueroa, C. Moulinec, M. Vázquez, G. Houzeaux, S. Longshaw, S. Oller, Hybrid CPU/GPU FE2 multi-scale implementation coupling Alya and Micropp, 2019.
[11] J. Glisse, Redhat heterogeneous memory management, in: Proceedings of Linux Plumbers Conference, 2018.
[12] M. Gonzalez, E. Morancho, Multi-GPU parallelization of the NAS multi-zone parallel benchmarks, in: IEEE Transactions on Parallel and Distributed Systems, vol. 32, IEEE, 2020, pp. 229–241.
[13] Z. Jia, M. Maggioni, B. Staiger, D.P. Scarpazza, Dissecting the NVIDIA Volta GPU architecture via microbenchmarking, arXiv preprint, arXiv:1804.06826, 2018.
[14] A. Li, S.L. Song, J. Chen, X. Liu, N. Tallent, K. Barker, Tartan: evaluating modern GPU interconnect via a multi-GPU benchmark suite, in: IEEE International Symposium on Workload Characterization (IISWC), IEEE, 2018, pp. 191–202.
[15] C. Li, R. Ausavarungnirun, C.J. Rossbach, Y. Zhang, O. Mutlu, Y. Guo, J. Yang, A framework for memory oversubscription management in graphics processing units, in: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS19), ACM, 2019, pp. 49–63.
[16] W. Li, G. Jin, X. Cui, S. See, An evaluation of unified memory technology on Nvidia GPUs, in: IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, IEEE, 2015, pp. 1092–1098.
[17] S. Manavski, G. Valle, CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman string alignment, BMC Bioinform. 9 (Suppl 2) (2008) S10, https://doi.org/10.1186/1471-2105-9-S2-S10.
[18] A. Mishra, L. Li, M. Kong, H. Finkel, B. Chapman, Benchmarking and evaluating unified memory for OpenMP GPU offloading, in: Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC, ACM, 2017, p. 6.
[19] NVIDIA, NVIDIA. Tesla P100 white paper, 2016.
[20] NVIDIA, NVIDIA. CUDA C Programming Guide, 2019.
[21] NVIDIA, GPU-accelerated Caffe, https://www.nvidia.com/en-gb/data-center/gpu-accelerated-applications/caffe/, 2020.
[22] NVIDIA, CUDA profiler users guide, https://docs.nvidia.com/cuda/pdf/CUDA_Profiler_Users_Guide.pdf, November 2020.
[23] NVIDIA, CUDA ToolKit documentation, https://docs.nvidia.com/cuda/profiler-users-guide/index.html, November 2020.
[24] C. Pearson, A. Dakkak, S. Hashash, C. Li, H. Chung, J. Xiong, W.M. Hwu, Evaluating characteristics of CUDA communication primitives on high-bandwidth interconnects, in: Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering, ACM, 2019, pp. 209–218.
[25] C.D. Polychronopoulos, D.J. Kuck, Guided self-scheduling: a practical scheduling scheme for parallel supercomputers, IEEE Trans. Comput. C-36 (12) (1987) 1425–1439.
[26] N. Sakharnykh, Unified memory on Pascal and Volta, in: GPU Technology Conference (GTC), 2017, http://on-demand.gputechconf.com/gtc/2017/presentation/s7285-nikolay-sakharnykh-unified-memory-on-pascal-and-volta.pdf.
[27] N. Sakharnykh, Everything you need to know about unified memory, in: GPU Technology Conference (GTC), 2018.

[28] Yong Yan, Canming Jin, Xiaodong Zhang, Adaptively scheduling parallel loops in distributed shared-memory systems, IEEE Trans. Parallel Distrib. Syst. 8 (1) (1997) 70–81.

Marc Gonzalez Tallada received the degree in computer science in 1996 and the PhD degree in computer science in 2003, both from the Universitat Politècnica de Catalunya (UPC), Spain. In 2001, he joined the Department of Computer Architecture at UPC, where he is currently an associate professor. His research interests are related to programming models and compilers for High Performance Computing technologies.

Enric Morancho received the degree in computer science in 1992 and the PhD degree in computer science in 2002, both from the Universitat Politècnica de Catalunya (UPC), Spain. In 1993, he joined the Department of Computer Architecture at UPC, where he is currently an associate professor. His research interests include processor micro-architecture, memory hierarchy, awareness of architecture in programming and operating systems.