The Journal of Supercomputing (2025) 81:283
https://doi.org/10.1007/s11227-024-06794-3

An autotuning approach to select the inter-GPU communication library on heterogeneous systems

Jesús Cámara[1] · Javier Cuenca[2] · Victor Galindo[2] · Arturo Vicente[2] · Murilo Boratto[3]

Accepted: 28 November 2024

Abstract

In this work, an automatic optimisation approach for parallel routines on multi-GPU systems is presented. Several inter-GPU communication libraries (such as CUDA-Aware MPI or NCCL) are used with a set of routines to perform the numerical operations among the GPUs located on the compute nodes. The main objective is the selection of the most appropriate communication library, the number of GPUs to be used and the workload to be distributed among them in order to reduce the cost of data movements, which represent a large percentage of the total execution time. To this end, a hierarchical modelling of the execution time of each routine to be optimised is proposed, combining experimental and theoretical approaches. The results show that near-optimal decisions are taken in all the scenarios analysed.

Keywords Autotuning · Communication libraries · Multi-GPU · Heterogeneous computing

Jesús Cámara, Javier Cuenca, Victor Galindo, Arturo Vicente and Murilo Boratto have contributed equally to this work.

✉ Jesús Cámara
jesus.camara@infor.uva.es

Javier Cuenca
jcuenca@um.es

Victor Galindo
victor.galindog@um.es

Arturo Vicente
arturo.vicentej@um.es

Murilo Boratto
murilo.boratto@fieb.org.br

[1] Department of Informatics, University of Valladolid, Valladolid, Spain

[2] Department of Engineering and Technology of Computers, University of Murcia, Murcia, Spain

[3] Supercomputing Center for Industrial Innovation, SENAI CIMATEC, Salvador, Bahia, Brazil

Published online: 12 December 2024

1 Introduction

Nowadays, the heterogeneity of computing systems makes the efficient execution of scientific applications extremely challenging. These systems typically consist of a set of hybrid nodes composed of multiple processing units, such as multicore CPUs and one or more accelerators (typically GPUs) with different computational capabilities and memory hierarchies. Therefore, several aspects should be considered to make an efficient use of all these processing units. Firstly, it is necessary to have a good workload scheduling policy according to their computational capacities [1]. In this way, given a specific problem to solve, this policy implicitly entails the selection of which processing units will be used, with a greater or lesser workload, and which of these units will be discarded. Another important decision to take would be the communication library to use for data movements among processing units, mainly GPUs, since they will handle the most considerable amount of data according to their capacities. There are different frameworks/libraries that can be used for data communication between GPUs. Also, several approaches could be considered. For example, the use of data intra-node copies between host and devices plus traditional MPI communications between CPUs. Another option could be the use of CUDA-Aware MPI [2], which allows any GPU to send the data directly to any other GPU without CPU intervention. Likewise, the NCCL library [3] could be considered to perform this direct data transfer between GPUs.

In this sense, taking into account the different execution scenarios where the application can be executed, the selection of the best approach or the most appropriate library becomes complicated. That is, the search space depends on several factors: the platform capabilities, the type of communication operations involved in the application, and the data volume transferred in these communications. Therefore, the use of an autotuning framework is crucial to avoid the application developer the selection of both the workload scheduling policy and the inter-GPU communication library.

In recent years, a number of techniques to automatically improve the performance of software oriented to heterogeneous platforms have emerged [4]. In this way, we can find global autotuning frameworks, such as CLTune [5], Orio [6] or ATF [7], where specific techniques or methods can be introduced, such as in [8]. In general terms, given a routine, the optimisation problem of finding the values of the adjustable parameters that minimise the execution time can be addressed through the use of metaheuristic methods, such as OpenTuner [9], or by using heuristics tailored for each specific routine to optimise [10]. On the other hand, regarding the operating mode of the autotuning engine, two approaches can be raised: a pseudo-theoretical point of view, estimating theoretical execution times based on real times measured by executing basic operations [11, 12], or an experimental point of view, where different alternatives are explored by analysing the behaviour observed after several executions [10, 13]. Furthermore, due to the bottleneck that inter-GPU communications can represent in this execution context, multiple approaches and software developments have been carried out aimed at

studying and improving the performance of such communications [14], but are not focused on the selection of the most appropriate inter-GPU communication library to efficiently execute a given routine on a specific heterogeneous platform.

In this work, a hierarchical autotuning approach is developed in order to model the execution time of any high-level routine using the balanced aggregation of the execution time models of its lower-level subroutines. Thus, the training phase can be focused only on the lowest-level routines: the calculation and communication kernels. For each of these kernels, this training provides its performance map, that is, a topographical view of its execution time across a set of values of its adjustable parameters. By moving in ascending hierarchical order, these performance maps are used to obtain an overview of the performance behaviour of each higher-level routine, according to its execution model and without further experimentation. This autotuning approach gives rise to the main contributions of this work:

- Each multi-GPU routine is provided with an automatic adjustment of its configuration parameters: the selection of which GPUs to use, the workload distribution among them and the inter-GPU communication library/framework.
- The proposed hierarchical method leads to a significant reduction in the experimentation required for the whole process. It also allows a certain degree of fault tolerance and high scalability for both software and hardware.

The rest of the paper is organised as follows. In Sect. 2, the main ideas of the autotuning methodology are described. Then, a proof of concept is presented in Sect. 3, showing how this methodology can be applied to different types of routines and execution platforms. Finally, in Sect. 4, the main conclusions derived from this work are presented.

2 Automatic optimisation methodology

The main goal of the proposed methodology is to automatically select near-optimal values for the adjustable parameters, APs, of a hybrid parallel routine, R, to be executed on a heterogeneous platform, P. This platform usually consists of a set of hybrid nodes, which are composed of several processing units, PUs, of different types: multicore CPUs and computational accelerators, typically GPUs. In this context, the study will focus on the following APs:

- The selection of PUs of P to be used.
- The workload to be distributed in a balanced way among the selected PUs.
- The library to be used for each inter-PU communication operation involved in R.

A hierarchical approach is applied to decompose the global optimisation problem of the execution time of R on P into a set of subproblems. These subproblems correspond to the partial execution times of both each calculation kernel, K, within R, on each PU, and each communication operation, C, within R, on each subset of PUs.

To simplify the description of the proposed method, let us assume that R, which solves a problem of size n, can be schematised as a sequence of S steps, where the i-th step consists, in turn, of two phases. In a first phase, a communication operation, C_i, belonging to the communication library, L_i, is performed to transfer n_i data from a subset of sender PUs to a subset of receiver PUs. Thus, the j-th receiver PU will have received n_{ij} data at the end of this phase. In the second phase, all the receiving PUs execute the K_i calculation kernel with the received data. Likewise, it can be assumed that both the input data and the output results are stored in a specific PU, referred to as the root PU. Therefore, according to the scheme shown in Fig. 1, the autotuning process of R will be carried out following three main phases:

1. *Modelling*: The execution time of R is modelled as the addition, for all the steps, of the communication operation time and the maximum calculation time:

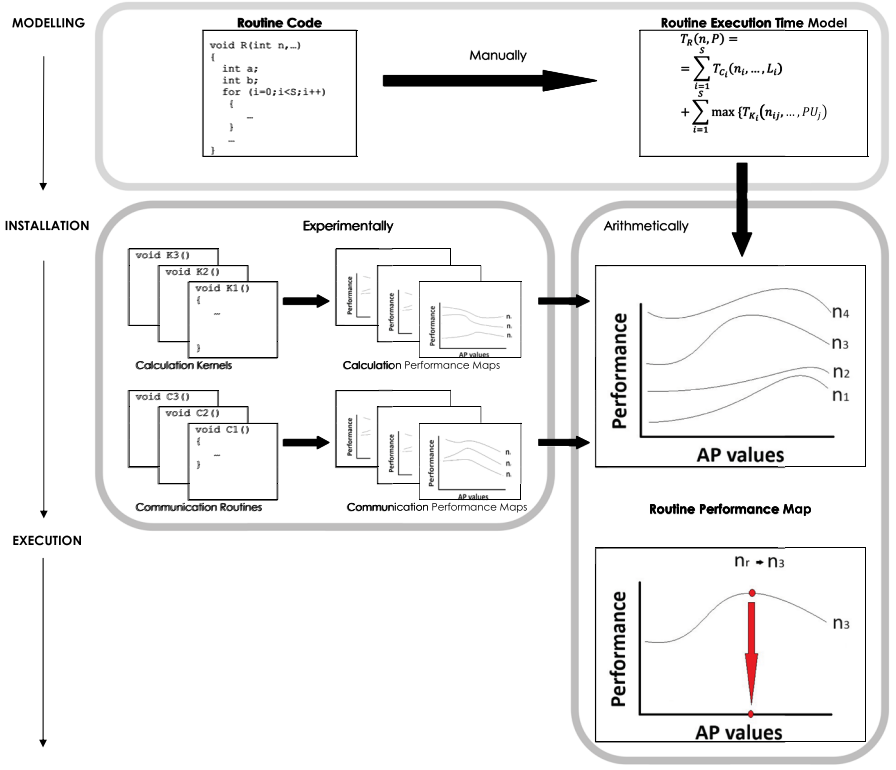

Fig. 1 High-level overview of the autotuning process of a hybrid parallel routine, R, on a heterogeneous platform, P

$$T_R(n,P) = \sum_{i=1}^{S} T_{C_i}(n_i, \text{sender\_}PU\text{\_set}_i, \text{receiver\_}PU\text{\_}set_i, L_i) + \sum_{i=1}^{S} \max_j \left\{ T_{K_i}(n_{ij}, PU_j) \right\} \quad (1)$$

2. *Installation:* The performance map, PM, of a routine is defined as the performance values of this routine for a given set of problem sizes when different combinations of AP values are used. In this way, the autotuning engine is in charge of experimentally obtaining the PM of each basic subroutine of R. That is:

   - The experimental PM of each basic calculation kernel, K_i, on each PU for a given range of training problem sizes.
   - The experimental PM of each basic communication routine, C_i, from each available communication library, L_i, on different subsets of PUs for a given range of training message sizes.

   Then, by using the data from these basic PMs, the whole PM of R is generated arithmetically, without further experimentation, for a given range of training problem sizes, according to its execution time model (Eq. 1). Finally, the information from all the PMs is stored for further use both for executing R and installing other routines.
3. *Execution:* Given a specific problem size, n_r, the optimisation process is carried out directly by traversing the search space defined by the PM of R (previously built during the installation phase), in order to find the best combination of AP values for n'_r, the training problem size closest to n_r. In this way, the overhead introduced is minimal, since no experimentation is required. Moreover, as n_r increases, since the query time for the best AP values is constant, it becomes even more negligible compared to the total execution time of the routine.

It is also important to note that this methodology allows for a certain degree of fault tolerance. In this way, if any of the PUs and/or any of the communication libraries are no longer available in the execution platform, all the training information from the rest of the available PUs and libraries remains usable. The decisions to be made for a high-level routine are re-adapted, without the need for retraining.

Likewise, there is a good degree of scalability, since when a different PU is added to the platform, it is only necessary to perform the training of the basic routines on this new hardware to extend the PM of the routine. All the performance information previously stored for the rest of the PUs in the platform remains equally useful. Moreover, following this hierarchical approach, if a new higher-level routine needs to be optimised, no further experimentation is required. All the performance information previously obtained from lower-level routines can be used to build arithmetically its whole PM, by means of its execution time model.

3 Experimental study

This section tests the usefulness of the proposed methodology in a number of execution scenarios. To do so, two different computational platforms are used. On the one hand, a small platform with a high degree of heterogeneity both in the configuration of the compute nodes and in the PUs (CPUs and GPUs). On the other hand, a larger and more powerful platform, but with less heterogeneity.

The small platform is **HETEROSOLAR**, a heterogeneous cluster located at the University of Murcia (Spain). This platform consists of a set of nodes of different characteristics connected via a Gigabit Ethernet network. The compute nodes used for the experimental study are:

- *venus*: with 2 Intel Xeon E5-2620 (hexa-core) CPUs at 2.40 GHz and 2 GPUs: a NVIDIA GeForce GT 640 and a NVIDIA PNY Quadro P2200.
- *saturno*: with 4 Intel Xeon E7530 (hexa-core) CPUs at 1.87 GHz and 1 NVIDIA Tesla K20c GPU.

The powerful platform is **OGBON**, from the SENAI CIMATEC Supercomputing Center in Brazil. This platform is made up of 78 compute nodes interconnected via an Infiniband network using NVIDIA's UCX [15]. In turn, each node consist of 1 Intel Xeon Gold 6240 CPU with 18 cores at 2.60 GHz, 384 GB RAM and 4 NVIDIA Tesla V100-SXM2-32GB GPUs interconnected via NVLINK [16].

The following subsections describe the study carried out on these platforms with several parallel routines that have different space-temporal partitioning schemes for both the computations and the communications.

3.1 Proof of concept: a parallel tiled matrix multiplication

This subsection describes how the proposed methodology is applied to a multi-GPU parallel tiled matrix multiplication routine, PTMM. In this routine, both the input matrices, A and B, and the resulting matrix, C, are handled by means of square tiles of a given size, $t \times t$. In this way, each tile of C will be the result of matrix-multiplying a tile row of A by a tile column of B.

Thus, given a multi-GPU platform, P, the execution time model of PTMM for solving a n^2 problem, using the communication library L, and considering that the input data are stored in one of its GPUs (referred to as the root GPU) and where GPU_i is in charge of calculating n_i rows of C, can be modelled as:

$$\begin{aligned} T_{\text{PTMM\_ABC}}(n^2, t, P, L) = {} & T_{\text{scatter\_}A}(n^2, n_i \times n, \text{rootGPU}, \text{GPUset}, L) + \\ & T_{\text{broadcast\_}B}(n^2, \text{rootGPU}, \text{GPUset}, L) + \max_i \left\{ T_{\text{TMM\_}A_iBC_i}(n_i \times n, t, \text{GPU}_i) \right\} + \\ & T_{\text{gather\_}C}(n^2, n_i \times n, \text{rootGPU}, \text{GPUset}, L) \end{aligned} \tag{2}$$

That is, the execution time of PTMM can be modelled as the addition of the time required to scatter matrix A from root GPU, $T_{\text{scatter\_}A}$, the time to broadcast matrix

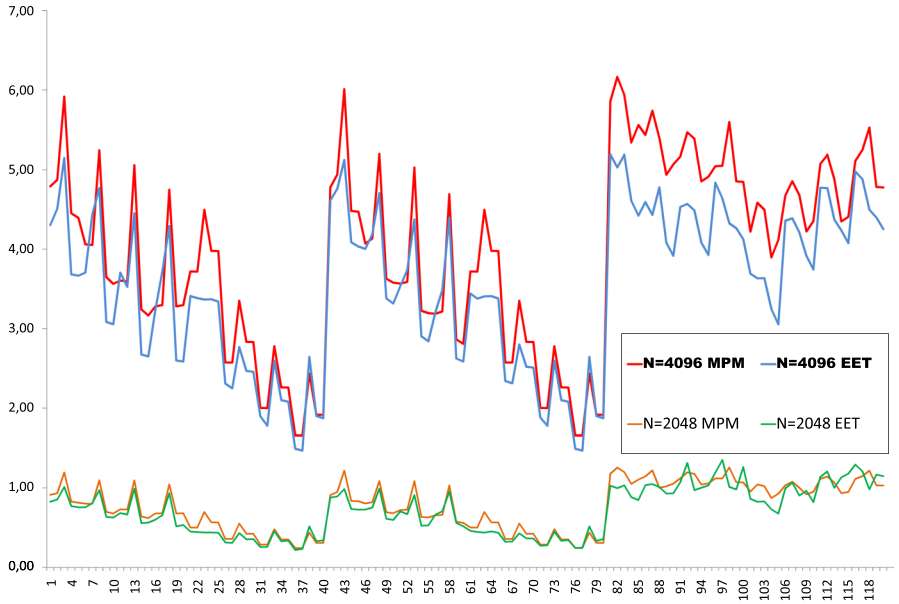

Fig. 2 PTMM: Comparison of the execution time (in seconds) predicted by the model performance map (MPM) and the experimental execution time (EET), for $n = \{2048, 4096\}$ and 120 different combinations of the AP values, on HETEROSOLAR

B also from root GPU, $T_{\text{broadcast\_}B}$, the maximum of the computation times spent by the set of GPUs to calculate their corresponding portions of C, $T_{\text{TMM\_}A_iBC_i}$, and the time to gather these portions of C back to the root GPU, $T_{\text{gather\_}C}$. Then, in the installation phase, as described in Sect. 2, the autotuning engine analyses each part of this execution time model as follows:

- Communication submodels (T_{scatter}, $T_{\text{broadcast}}$, T_{gather}): An experimental training is performed with a set of messages sizes to obtain the PMs of these three basic communication routines[1] for each available communication library, varying both the set of GPUs used and the root GPU where the input data and results are located.
- Calculation submodel (T_{TMM}): An experimental training is performed with a set of problem sizes to get the PM of the basic calculation routine on each GPU.

Next, the performance data of these basic routines is used to arithmetically generate the whole PM of PTMM according to its execution time model (Eq. 2) for a given range of training problem sizes. Finally, the information of all the PMs is stored for further use when the PTMM routine is executed, and also when other routines are installed.

[1] Although the algorithmic issues of gather and scatter operations are usually equivalent, it could be preferable to model them separately in order to capture possible behaviour differences.

Table 1 Comparison of the performance obtained with the values selected by the autotuning engine (AUT) versus the optimum (OPT) and the average obtained by hand (HAN) when executing the PTTM routine on HETEROSOLAR using different root GPUs (venus GPU_0 (v0), venus GPU_1 (v1), and saturno GPU_0 (s0))

Scenario		AUT Parameter Values				OPT	HAN
Root GPU	Problem Size	Selected GPUs (workload factor)	Communication Library			Performance Difference (%)	
			Scatter A	Broadcast B	Gather C		
v0	1024	v0(×4), v1(×1)	MPI	MPI	MPI	0	95
v0	2048	v0(×4), v1(×1)	CA-MPI	CA-MPI	CA-MPI	−5	98
v0	3072	v0(×4), v1(×1)	CA-MPI	CA-MPI	CA-MPI	0	65
v0	4096	v0(×4), v1(×1)	CA-MPI	CA-MPI	CA-MPI	0	97
v1	1024	v1(×1), v0(×4)	MPI	MPI	MPI	0	95
v1	2048	v1(×1), v0(×4)	CA-MPI	CA-MPI	CA-MPI	−2	95
v1	3072	v1(×1), v0(×4)	CA-MPI	CA-MPI	CA-MPI	0	52
v1	4096	v1(×1), v0(×4)	CA-MPI	CA-MPI	CA-MPI	0	92
s0	1024	s0(×1), v0(×1)	MPI	NCCL	MPI	−8	65
s0	2048	s0(×1), v0(×1)	MPI	NCCL	MPI	−4	57
s0	3072	s0(×1), v0(×1)	MPI	NCCL	MPI	−3	42
s0	4096	s0(×1), v0(×1)	MPI	NCCL	MPI	−2	32

The proposed execution time models are validated by comparing the execution time predicted by the model performance map (MPM), with the experimental execution time (EET) obtained on a real computing platform across a wider range of scenarios. Figure 2 shows an example for the parallel matrix multiplication routine (PTMM) on HETEROSOLAR, for $n = \{2048, 4096\}$. A set of 120 different combinations of AP values have been considered for this comparative graph: communication library (inter-CPU MPI, CUDA-Aware MPI, NCCL, and two hybrid combinations of them for the three communication operations inside PTMM), number of GPUs (2 and 3), root GPU (venus GPU_0, venus GPU_1, and saturno GPU_0), and computational load balancing between GPUs (overloading the most powerful one by up to 4 times compared to the others). As can be seen, MPM tends to behave quite similarly to EET, capturing the trends along the different execution scenarios proposed. In any case, it is worth remembering that the goal is not an exact modelling of the execution time but the design of a good decision-making tool.

In order to demonstrate the usefulness of the autotuning engine proposed in this work, Table 1 shows a experimental performance comparison for different execution scenarios (root GPU and problem size), on HETEROSOLAR, depending on the AP values. The experimental execution time obtained with the AP values selected by using the performance map, AUT, is compared with the optimal execution time using a perfect oracle that could always get the optimal AP values, OPT, and also versus an estimation of the execution time that could be obtained by a routine

developer who decides such values by hand, HAN.[2] As can be seen, automatically made decisions allow for optimal or near-optimal performance in most execution scenarios, despite the variability of the best AP values when changing the starting scenario (the root GPU and the size of the problem to be solved). Moreover, if we take a look at the average performance that would be obtained with a selection by hand of the AP values, the usefulness of the proposed methodology is even more evident, as it achieves from 50% to about 100% performance improvement. Analysing these results more closely, it can be observed how, despite having three GPUs available, the autotuning engine only decides to use different pairs of GPUs depending on where the initial data are located. When using the two venus GPUs, GPU_0 workload factor is equal to $\times 4$, that is, it chooses to overload this GPU by up to 4 times the load assigned to GPU_1, due to the difference in computing power between them. However, when the initial data are in saturno (s0 is the root GPU), the overhead of sending data to venus entails that it is no longer appropriate to use venus GPU_1, and that, in addition, the workload assigned to venus GPU_0 is only the same amount of work as that assigned to the saturno GPU.

Regarding the communication libraries to be used, it can be seen how the decision taken again depends on both the root GPU and the problem size. Thus, when the root GPU is one of the venus GPUs, the standard MPI library between CPUs is chosen for small problem sizes. However, when the problem size increases, CUDA-Aware MPI is selected. These decisions change significantly when the root GPU is the saturno GPU. In this scenario, the NCCL library is used to send matrix B, and the standard MPI library is used between CPUs for matrices A and C. The reasons for the selection of different communication libraries, depending on the type of communication operation and the senders and receivers involved, seem to be related to the way they work internally. In any case, it is beyond the scope of this work to look deeper into these causes, since the main objective is to automatically make near-optimal decisions for a set of execution scenarios, without the cost of a more in-depth study of the underlying architectures.

To achieve the automatic decision-making capability at execution time shown in Table 1, the autotuning engine took around 14 seconds during the installation phase to build the whole PM of the PTMM routine according to its execution time model (Eq. 2). Concretely, for the set of installation sizes $\{512, 1536, 2560, 3584, 4608\}$, the times required to build the basic calculation PM for the TMM routine and to obtain the basic communication PMs (broadcast, scatter and gather routines) were 11801 and 1691 milliseconds, respectively. At this point, it is important to note that all the information collected in these basic PMs will be reused in the installation of any other higher-level routines that make use of these basic routines, without additional experimental cost.

Consider now the behaviour in the other heterogeneous platform, OGBON, using two compute nodes with a total of 8 GPUs. Table 2 shows how the autotuning engine manages to make good decisions, even if, at any given time, the availability of the computational resources changes. For instance, if the NCCL library becomes

[2] For each execution scenario, HAN time has been calculated as the average execution time among those obtained with the different considered combinations of AP values.

Table 2 Comparison of the performance obtained by the autotuning engine (AUT) versus the optimum (OPT) when executing the PTMM routine on OGBON with different problem sizes and using the available GPUs and Libraries

Scenario		AUT parameter values				OPT
Problem	Available resources	Communication library				Perf.
Size	GPUs—Libraries	GPUs	Scatter A	Broadcast B	Gather C	Diff (%)
2048	8—MPI CA-MPI NCCL	4	NCCL	NCCL	NCCL	0
2048	7—MPI CA-MPI NCCL	4	NCCL	NCCL	NCCL	0
2048	8—MPI CA-MPI	4	CA-MPI	CA-MPI	CA-MPI	−9
2048	8—MPI	2	MPI	MPI	MPI	−6
4096	8—MPI CA-MPI NCCL	8	NCCL	NCCL	NCCL	0
4096	7—MPI CA-MPI NCCL	4	NCCL	NCCL	NCCL	0
4096	8—MPI CA-MPI	4	CA-MPI	CA-MPI	CA-MPI	0
4096	8—MPI	2	MPI	MPI	MPI	−9
8192	8—MPI CA-MPI NCCL	8	NCCL	NCCL	NCCL	0
8192	7—MPI CA-MPI NCCL	4	NCCL	NCCL	NCCL	0
8192	8—MPI CA-MPI	4	CA-MPI	CA-MPI	CA-MPI	0
8192	8—MPI	2	MPI	MPI	MPI	−1
16384	8—MPI CA-MPI NCCL	8	NCCL	NCCL	NCCL	0
16384	7—MPI CA-MPI NCCL	4	CA-MPI	CA-MPI	CA-MPI	−11
16384	8—MPI CA-MPI	4	CA-MPI	CA-MPI	CA-MPI	0
16384	8—MPI	3	MPI	MPI	MPI	−1
32768	8—MPI CA-MPI NCCL	8	NCCL	NCCL	NCCL	0
32768	7—MPI CA-MPI NCCL	7	NCCL	NCCL	NCCL	−5
32768	6—MPI CA-MPI NCCL	6	NCCL	NCCL	NCCL	0
32768	5—MPI CA-MPI NCCL	4	CA-MPI	CA-MPI	CA-MPI	−9
32768	8—MPI CA-MPI	4	CA-MPI	CA-MPI	CA-MPI	0
32768	8—MPI	8	MPI	MPI	MPI	0

unavailable, or if one of the GPUs stops working, the decisions are re-adapted to provide near-optimal performance, thus offering some fault tolerance. Going into the details of these results, for small problem sizes ($n = 2048$) the extra cost of inter-node communication does not compensate the computing power of using the 8 available GPUs, so it is often better to use only the 4 GPUs of a single node. A similar situation arises for medium problem sizes ($n = 4096, 8192, 16384$) when only 7 GPUs are available. However, for larger problem sizes ($n = 32768$) it is profitable to continue using all GPUs if, at least, 6 of them are available. On the other hand, if at any given time the NCCL library is not available, the second best choice would be CUDA-Aware MPI, but as it has a higher inter-node communication cost, again the best choice is usually to use a single node. Finally, if CUDA-Aware MPI is also not available, the higher relative cost of communications using MPI between CPUs means that the best decisions generally involve selecting considerably fewer GPUs, mainly for smaller problem sizes. Although not shown in Table 2, it is worth

noting that the average improvement in performance compared to an assumed decision making by hand is about 65%, as in HETEROSOLAR.

3.2 Experimental results with other routines

This subsection summarises the results obtained when the proposed methodology is applied to routines with different space-temporal schemes for both calculation and communication, which implies a more complex decision-making process.

In the PLU routine, a LU factorisation is carried out using the Gaussian reduction algorithm. First, the input singular square matrix, A, is scattered from the root GPU to all GPUs. After that, a set of n steps are performed. In step j, the row j of A is broadcasted. Then, this row is used by each GPU to update its part of the j-th column of A (progressing in the calculation of the resulting matrix, L) and to reduce its set of rows of A starting from column $j+1$ (progressing in the calculation of the resulting matrix, U). Finally, each GPU sends its part of both result matrices to the root GPU. Thus, given an execution platform P, the execution time model for the PLU routine to solve a n^2 problem, assigning $n_i \times n$ data to the i-th GPU, being L the communication library used, can be modelled as:

$$\begin{aligned} T_{\text{PLU\_}A}(n^2, P, L_{\text{com}}) = & T_{\text{scatter\_}A}(n^2, n_i \times n, \text{root GPU, GPUset}, L) + \\ & \sum_{j=1}^{n} T_{\text{broadcast\_}row\_A}(n-j, \text{root GPU}_j, \text{GPUset}, L) + \\ & \sum_{j=1}^{n} \max_i \left\{ T_{\text{update\_}L}(n_i, \text{GPU}_i) + T_{\text{reduce\_}U}(n_i \times (n-j), \text{GPU}_i) \right\} + \\ & T_{\text{gather\_}LU}(n^2, n_i \times n, \text{root GPU, GPUset}, L) \end{aligned} \tag{3}$$

The PHR routine performs a data frequency analysis using a multi-GPU platform. To this end, the input data, D, are distributed among the different GPUs. Then, each GPU calculates its partial histogram and, finally, the results are grouped by means of a reduction operation of the partial histograms. Thus, given an execution platform P, the execution time model for the PHR routine to calculate the histogram of N_H data from a total of N_D input data, assigning N_{Di} data to the i-th GPU, being L the communication library used, can be modelled as:

$$\begin{aligned} T_{\text{PHR}}(N_D, N_H, P, L) = & T_{\text{scatter\_}D}(N_D, N_{Di}, \text{root GPU, GPUset}, L) + \\ & \max_i \left\{ T_{\text{hist}}(N_{Di}, N_H, \text{GPU}_i) \right\} + T_{\text{red\_add\_}H}(N_H, \text{root GPU, GPUset}, L) \end{aligned} \tag{4}$$

Finally, PNB is a multi-GPU routine for the resolution of the n-body problem using the all-pairs algorithm. This routine takes as input parameters the position and mass of the N bodies and returns as output the attractive force applied on each of these bodies. First, the masses and positions are distributed among the GPUs. After that, d iterations are carried out, being d the number of GPUs. In the j-th iteration, GPU_j distributes its subsets of positions and masses to all GPUs and, then, each GPU updates the partial calculation of the forces acting on its assigned subset of bodies.

Table 3 Execution of PLU on HETEROSOLAR for different problem sizes and root GPUs (venus GPU_0 (v0), venus GPU_1 (v1), saturno GPU_0, (s0))

Scenario		AUT parameter values			
Root GPU	Problem size	Selected GPUs (workload factor)	Communication library		
			Scatter A	Broadcast row	Gather LU
v0	1024	v0(×3)–v1(×1)	CA-MPI	MPI	CA-MPI
v0	2048	v0(×3)–v1(×1)	CA-MPI	MPI	CA-MPI
v0	3072	v0(×3)–v1(×1)	MPI	MPI	MPI
v0	4096	v0(×3)–v1(×1)	MPI	MPI	MPI
v1	1024	v1(×1)–v0(×1)	CA-MPI	MPI	CA-MPI
v1	2048	v1(×1)–v0(×3)	CA-MPI	MPI	CA-MPI
v1	3072	v1(×1)–v0(×3)	MPI	MPI	MPI
v1	4096	v1(×1)–v0(×3)	MPI	MPI	MPI
s0	1024	s0(×1)–v0(×1)	MPI	MPI	MPI
s0	2048	s0(×1)–v0(×1)	MPI	MPI	MPI
s0	3072	s0(×1)–v0(×1)	MPI	MPI	MPI
s0	4096	s0(×1)–v0(×1)	MPI	MPI	MPI

Table 4 Execution of PLU on OGBON for different problem sizes

Scenario		AUT parameter values			
Available GPUs	Problem size	Communication library			
		GPUs	Scatter A	Broadcast Row	Gather LU
8	2048	4	NCCL	CA-MPI	NCCL
8	4096	4	NCCL	CA-MPI	NCCL
8	8192	4	NCCL	CA-MPI	NCCL
8	16384	4	NCCL	CA-MPI	NCCL
8	32768	4	NCCL	CA-MPI	NCCL

Finally, the results obtained for all the forces are gathered. Thus, given an execution platform P with d GPUs, the execution time model for the PNB routine to solve the N-body problem, whose coordinates and masses are $\{X, Y\}$ and M, respectively, assigning N_i bodies to the i-th GPU, being F_X and F_Y the resulting forces, and L the communication library used, can be modelled as:

$$\begin{aligned} T_{\text{PNB}}(N,P,L) = & T_{\text{scatter\_}XYM}(N, N_i, \text{rootGPU}, \text{GPUset}) + \\ & \sum_{j=0}^{d-1} T_{\text{broadcast\_}X_jY_jM_j}(N_j, \text{GPU}_j, \text{GPUset}, L) + \max_i \left\{ T_{nb}(N_i, N_j, GPU_i) \right\} + \\ & T_{\text{gather\_}F_XF_Y}(N, N_i, \text{rootGPU}, \text{GPUset}, L) \end{aligned} \tag{5}$$

Table 5 Execution of PHR on HETEROSOLAR using different root GPUs (venus GPU_0 (v0), venus GPU_1 (v1), and saturno GPU_0 (s0)), for different problem and histogram sizes

Scenario			AUT parameter values		
Root GPU	Problem size	Histogram size	Selected GPUs (workload factor)	Communication library	
				Scatter D	Reduction H
v0	8 M	8 M	v0(×1)–v1(×1)	NCCL	NCCL
v0	64 M	8 M	v0(×1)–v1(×1)	CA-MPI	NCCL
v0	8 M	64 M	v0(×1)–v1(×1)	MPI	NCCL
v0	64 M	64 M	v0(×1)–v1(×1)	CA-MPI	NCCL
v1	8 M	8 M	v0(×1)–v1(×1)	NCCL	NCCL
v1	64 M	8 M	v0(×1)–v1(×1)	CA-MPI	NCCL
v1	8 M	64 M	v0(×1)–v1(×1)	MPI	NCCL
v1	64 M	64 M	v0(×1)–v1(×1)	CA-MPI	NCCL
s0	8 M	8 M	s0(×1)–v0(×1)	NCCL	NCCL
s0	64 M	8 M	s0(×1)–v0(×1)	CA-MPI	NCLL
s0	8 M	64 M	s0(×1)–v0(×1)	MPI	NCCL
s0	64 M	64 M	s0(×1)–v0(×1)	CA-MPI	NCCL

Table 6 Execution of PHR on OGBON for different problem and histogram sizes

Scenario			AUT parameter values		
Available GPUs	Problem size	Histogram size	Communication library		
			GPUs	Scatter D	Reduction H
8	8 M	8 M	4	NCCL	NCCL
8	64 M	8 M	4	NCCL	NCCL
8	256 M	8 M	4	CA-MPI	NCCL
8	8 M	64 M	4	NCCL	NCCL
8	64 M	64 M	4	NCCL	NCCL
8	256 M	64 M	4	CA-MPI	NCCL
8	8 M	256 M	4	NCCL	NCCL
8	64 M	256 M	4	NCCL	NCCL
8	256 M	256 M	4	CA-MPI	NCCL

Tables 3, 4, 5, 6, 7, 8 show the decisions taken for the AP values of these routines with different execution scenarios. As can be seen, these decisions can be quite different from those taken for the PTMM routine (Tables 1, 2). That is, as detailed in Sect. 2, after the experimental training phase of the basic calculation and communication routines, it is necessary to arithmetically design the performance map of each routine based on its execution time model. Although not shown in these tables, it is worth noting that near-optimal decisions are taken in most of the execution

Table 7 Execution of PNB on HETEROSOLAR for different problem sizes and root GPUs (venus GPU_0 (v0), venus GPU_1 (v1), and saturno GPU_0, (s0))

Scenario		AUT parameter values			
Root GPU	Problem size	Selected GPUs (workload factor)	Communication Library		
			Scatter *XYM*	Broadcast	Gather $F_X F_Y$
v0	4096	v0(×4)–v1(×1)	MPI	MPI	MPI
v0	8192	v0(×4)–v1(×1)	MPI	MPI	MPI
v0	16384	v0(×4)–v1(×1)	MPI	MPI	MPI
v0	32768	v0(×4)–v1(×1)	MPI	MPI	MPI
v1	4096	v1(×1)–v0(×2)	MPI	MPI	MPI
v1	8192	v1(×1)–v0(×2)	MPI	MPI	MPI
v1	16384	v1(×1)–v0(×2)	MPI	MPI	MPI
v1	32768	v1(×1)–v0(×2)	MPI	MPI	MPI
s0	4096	s0(×1)–v0(×1)	NCCL	MPI	NCCL
s0	8192	s0(×1)–v0(×1)	NCCL	MPI	NCCL
s0	16384	s0(×1)–v0(×1)	NCCL	MPI	NCCL
s0	32768	s0(×1)–v0(×1)	NCCL	MPI	NCCL

Table 8 Execution of PNB on OGBON for different problem sizes

Scenario		AUT parameter values			
Available GPUs	Problem size	Communication library			
		GPUs	Scatter *XYM*	Broadcast	Gather $F_X F_Y$
8	32768	8	NCCL	NCCL	CA-MPI
8	65536	8	NCCL	NCCL	CA-MPI
8	131072	8	NCCL	NCCL	CA-MPI
8	262144	8	NCCL	NCCL	CA-MPI

scenarios and the average improvement in performance compared to an assumed decision making by hand is about 35%.

4 Conclusions and future work

This work has faced the challenge of properly selecting the communication library and the workload distribution when running parallel routines on multi-GPU systems. To do so, an autotuning approach based on a hierarchical view of the software is used.

Given a routine, a execution time model is designed as a guide to build the performance map of that routine, that is, a topographical view of its execution time through a set of values of its adjustable parameters. This process is carried out

hierarchically, taking the performance maps of the most basic routines as a starting point. In this way, the required experimental effort is done at installation time, since it focuses on building these basic maps.

This accomplishes two goals: first, it eliminates the experimentation phase in the autotuning process of the main routine, which significantly reduces the corresponding overhead. Secondly, to reuse all the implicit information gathered in the performance maps of the basic routines, as no additional cost is incurred to create maps for other higher-level routines that use them.

The results obtained show that this hierarchical approach leads to routine performance maps that can be used as tools to make near-optimal decisions on the values of their adjustable parameters. All this regardless of the specific topology of the execution platform and the software implementation used within the routine to be handled.

By treating each processing unit of the execution platform as a black box, with a corresponding performance map for each routine, the extension of this methodology to other computational environments with other types of accelerators, such as FPGAs, is greatly facilitated. In addition, it is intended to include the necessary functionality to decide the best available numerical library for each accelerator, in the same way as has been done for the communication library.

Following this research line, our medium-term goal is to integrate the proposed hierarchical autotuning approach into PARCSIM [17], a full-featured simulator for running numerical software on heterogeneous parallel platforms. The autotuning engine would substantially enhance the functionality of PARCSIM, since currently this tool needs to generate all possible combinations of the AP values for a given input scenario (problem size and execution platform) in order to determine the best way to execute a routine. In contrast, with the proposed hierarchical autotuning engine this would not be necessary, since it allows to directly select the best execution configuration by making use of the performance information stored during the installation phase.

Finally, as future work, other performance metrics, such as energy consumption or memory usage, could be considered for optimisation by the autotuning engine using a similar approach. For example, regarding energy consumption, the execution time model of the routine would be transformed into the corresponding energy consumption model by simply replacing the terms related to the execution times of the basic routines by their respective in energy consumption. Afterwards, in the installation phase, the energy maps of each basic routine would be obtained experimentally and then, using the model, the map of the routine to be optimised would be constructed arithmetically.

Acknowledgements This work is supported by Grant PID2022-136315OB-I00 and Grant PID2022-142292NB-I00, both funded by MCIN/AEI/10.13039/501100011033/ and by "ERDF A way of making Europe", EU. The authors thank the SENAI CIMATEC Supercomputing Center for Industrial Innovation for making the OGBON Supercomputer available.

Funding Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature.

References

1. Agullo E, Cámara J, Cuenca J, Giménez D (2020) On the autotuning of task-based numerical libraries for heterogeneous architectures. In: Proceedings of the 2019 International Conference on Parallel Computing. Adv Parallel Comput 36:157–166
2. Kraus J (2013) An Introduction to CUDA-Aware MPI. https://developer.nvidia.com/blog/introduction-cuda-aware-mpi. Accessed 20 Nov 2024
3. Luehr N (2016) Fast multi-GPU collectives with NCCL. https://developer.nvidia.com/blog/fast-multi-gpu-collectives-nccl. Accessed 20 Nov 2024
4. Schoonhoven RA, Werkhoven B, Batenburg KJ (2023) Benchmarking optimization algorithms for auto-tuning gpu kernels. IEEE Trans Evolut Comput 27(3):550–564. https://doi.org/10.1109/TEVC.2022.3210654
5. Nugteren C, Codreanu V (2015) CLTune: a generic auto-tuner for OpenCL kernels. In: 2015 IEEE 9th International Symposium on Embedded Multicore/Many-Core Systems-on-Chip (MCSoC), IEEE Computer Society, USA, pp 195–202
6. Hartono A, Norris B, Sadayappan P (2009) Annotation-based empirical performance tuning using Orio. In: 2009 IEEE International Symposium on Parallel & Distributed Processing, pp 1–11
7. Rasch A, Schulze R, Steuwer M, Gorlatch S (2021) Efficient auto-tuning of parallel programs with interdependent tuning parameters via auto-tuning framework (ATF). ACM Trans Archit Code Optim 18(1):1–26
8. Lim R, Norris B, Malony A (2017) Autotuning gpu kernels via static and predictive analysis. In: 2017 46th International Conference on Parallel Processing (ICPP), pp 523–532
9. Ansel J, Kamil S, Veeramachaneni K, Ragan-Kelley J, Bosboom J, O'Reilly U-M, Amarasinghe S (2014) OpenTuner: an extensible framework for program autotuning. In: 23rd International Conference on Parallel Architectures and Compilation Techniques. ACM, Edmonton, Canada, pp 303–316
10. Cuenca J, García L, Giménez D, Herrera F (2017) Guided installation of basic linear algebra routines in a cluster with manycore components. Concurr Comput Pract Exp 29(15):e4112
11. Cámara J, Cuenca J, García L, Giménez D (2014) Auto-tuned nested parallelism: a way to reduce the execution time of scientific software in NUMA systems. Parallel Comput 40(7):309–327
12. Rico-Gallego JA, Díaz-Martín JC, Manumachu RR, Lastovetsky AL (2019) A survey of communication performance models for high-performance computing. ACM Comput Surv 51(6):1–36
13. Manumachu RR, Lastovetsky AL (2019) Design of self-adaptable data parallel applications on multicore clusters automatically optimized for performance and energy through load distribution. Concurr Comput Pract Exp 31(4):e4958
14. Ranganath K, Abdolrashidi A, Song S, Wong D (2019) Speeding up collective communications through inter-GPU re-routing. Comput Archit Lett 18(2):128–31
15. UCX: Unified Communication X. https://openucx.org/documentation/
16. NVIDIA NVLINK. https://www.nvidia.com/en-us/data-center/nvlink/
17. Cámara J, Cano J, Cuenca J, Saura-Sánchez M (2022) PARCSIM: a parallel computing simulator for scalable software optimization. J Supercomput 78(15):17231–17246