14th World Congress on Computational Mechanics (WCCM)
ECCOMAS Congress 2020
Virtual Congress: 11-15 January 2021
F. Chinesta, R. Abgrall, O. Allix and M. Kaliske (Eds)

NUMA-AWARE STRATEGIES FOR THE HETEROGENEOUS EXECUTION OF SPMV ON MODERN SUPERCOMPUTERS

Xavier Álvarez-Farré[1], Andrey Gorobets[2], F. Xavier Trias[1] and Assensi Oliva[1]

[1] Technical University of Catalonia,
Heat and Mass Transfer Technological Center,
Carrer Colom 11, 08222 Terrassa (Barcelona), Spain
{xavier.alvarez.farre, francesc.xavier.trias, asensio.oliva}@upc.edu

[2] Keldysh Institute of Applied Mathematics,
Miusskaya Sq. 4, 125047 Moscow, Russia
andrey.gorobets@gmail.com

Key words: High-performance computing, sparse algebra, SpMV, NUMA, heterogeneous computing

Abstract. The sparse matrix-vector product is a widespread operation amongst the scientific computing community. It represents the dominant computational cost in many large-scale simulations relying on iterative methods, and its performance is sensitive to the sparse pattern, the storage format and kernel implementation, and the target computing architecture. In this work, we are devoted to the efficient execution of the sparse matrix-vector product on (potentially hybrid) modern supercomputers with non-uniform memory access configurations. A hierarchical parallel implementation is proposed to minimise the number of processes participating in distributed-memory parallelisation. As a result, a single process per computing node is enough to engage all its hardware and ensure efficient memory access on manycore platforms. The benefits of this approach have been demonstrated on up to 9,600 cores of MareNostrum 4 supercomputer, at Barcelona Supercomputing Center.

1 INTRODUCTION

Large sparse matrices arise in the numerical resolution of partial differential equations. The sparse pattern of these matrices (*i.e.*, the distribution of the non-zero coefficients) depends on the spatial discretisation of a computational domain and the numerical method employed. The sparse matrix-vector product (SpMV) is, therefore, a widespread operation amongst the scientific computing community. Indeed, SpMV kernel represents the dominant computational cost in many large-scale simulations relying on iterative methods.

The performance of the SpMV kernel is sensitive to the sparse pattern (application), the storage format and kernel design (software), and the computing architecture (hardware). It is one of such memory-bound kernels with a very low arithmetic intensity, that is the ratio of float-

ing point operations (FLOP) to memory traffic in bytes (around 1:8 FLOP per byte), and often leads to irregular, uncoalesced memory accesses reducing the memory access efficiency. To top it off, in distributed-memory parallel processing, vector elements and matrix rows are distributed among a group of processes and this induces data exchanges between them. Therefore, the efficient execution of SpMV requires a fine-tuning process (*e.g.*, right choice of the sparse matrix storage format, proper workload balancing, reordering of unknowns to reduce matrix bandwidth, optimizing memory access to minimize cache misses).

Significant effort is devoted to the optimization of the SpMV for different applications and state-of-the-art computing environments. For instance, the introduction of the graphics processing unit (GPU) architecture into high-performance computing (HPC) systems motivated the research of new sparse matrix storage formats and SpMV implementations [1, 2] as reviewed by Filippone et al. [3]. The continuous evolution of central processing units (CPUs) also motivates the research for efficient SpMV kernels on such architectures [4, 5, 6].

In this work, we are devoted to the implementation of the SpMV kernel on hybrid systems with non-uniform memory access (NUMA) configurations, which were ignored in our previous implementations [7]. Ironically, when processors became faster than memory and started suffering from serious data starvation, vendors featured multi-core architectures and exacerbated the problem. Ever since multiple processors had to fight for memory access. The aim for NUMA configurations, introduced three decades ago [8], was to alleviate this conflict by providing separate memory banks and controllers for each processor or group of processors. Namely, NUMA allows for faster access to local memory at the expenses of slower access to remote memory. To deal with such configurations, some authors rely on message-passing interface (MPI) (*i.e.*, assigning at least one MPI rank per NUMA node, even one per core) which leads to a compact data placement and ensures an efficient memory usage [9, 10, 11, 6]. However, these approaches do not exploit the underlying shared-memory paradigm at their best. Instead, they increase the number of processes participating in data exchanges and the global size of the messages. Other authors have proposed NUMA-aware implementations which depend on runtime data migrations [12]. Instead, our approach relies on a predictive parallel initialisation by first-touch policy, similar to that of [13, 5], but it is also compatible with heterogeneous computing.

2 NUMA-AWARE, HIERARCHICAL PARALLELISATION OF THE SPMV KERNEL

2.1 Overview of modern supercomputers

A quick look at the world's fastest supercomputers reveals the huge variety of computing architectures competing in the exascale race. Modern supercomputers consist of multiple (potentially hybrid) nodes and usually introduce NUMA configurations (*cf.* Figure 1). Hybrid nodes combine different architectures, such as manycore CPU and GPU, among others. An efficient distributed-memory (DM) multiple instruction, multiple data (MIMD) parallelisation is required to engage the nodes of an HPC system. Within the nodes, different computing units

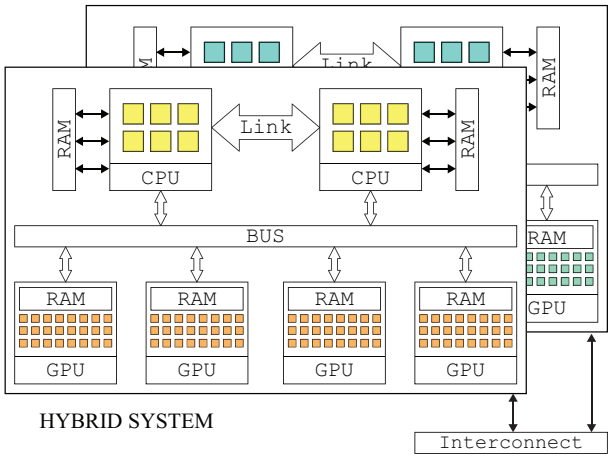

Figure 1: Example of HPC system configuration.

feature different parallel paradigms. The CPU, so-called *host*, consists of a pool of cores packed into NUMA nodes, that is, different CPU sockets, even groups of cores within sockets, with separate memory banks and controllers. The non-uniform memory access allows for faster access to local memory at the expenses of slower access to remote memory. A fine-tuned, NUMA-aware shared-memory (SM) MIMD parallelisation is required to engage all the CPU cores, ensuring thread affinity and local memory access. On the other hand, massively-parallel processors of different architectures such as MIC or GPU, also known as *accelerators*, integrate an on-chip memory space separated from the host. Nowadays, GPU is the most common kind of accelerator. To deal with such processors, the algorithms must be compatible with the stream processing (SP) paradigm.

This section is devoted to the execution of the SpMV kernel, $y \leftarrow \mathcal{A}x + y$, on modern supercomputers. We describe a NUMA-aware, hierarchical parallel implementation based on a multilevel workload distribution divided into three levels: 1) inter-node, 2) intra-node, and 3) intra-unit.

2.2 Multilevel workload distribution

Let us consider a sparse matrix, $\mathcal{A} \in \mathbb{R}^{m \times n}$, representing the linear map $f : U \to V$, where $U \in \mathbb{R}^n$ and $V \in \mathbb{R}^m$. The workload distribution is fulfilled with a multilevel decomposition, a distribution technique very suitable for hierarchical parallel implementations [14, 15, 7]. This technique aims at distributing the computational load in two levels at least. Roughly, the first level assigns the load per computing node and the second level the load per computing unit; further levels may be required to target processors with complex NUMA configurations [16]. Figure 2 shows a row-major, multilevel decomposition of a generic sparse matrix and the vector spaces it maps; this figure will be described throughout the rest of this section.

The first-level (inter-node) partitioning, represented in Figure 2 by a grey division, distributes

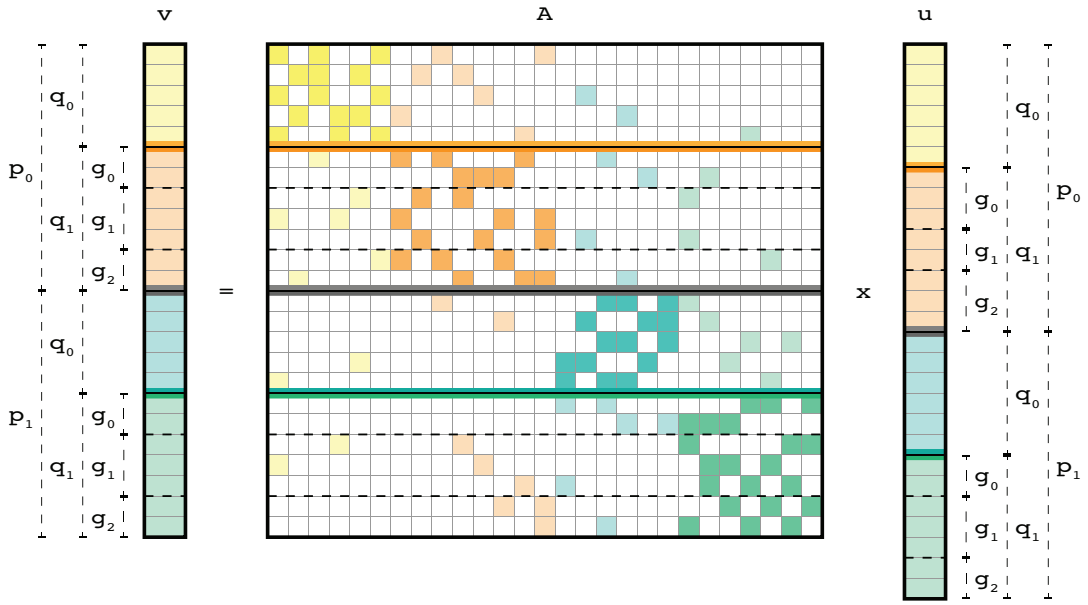

Figure 2: Representation of a row-major, multilevel decomposition for SpMV. Vector elements and matrix rows are divided into first-level chunks, denoted as `p0` and `p1`, for two cluster nodes, which are further divided into second-level chunks, denoted as `q0` and `q1`, for two computing units. Some of the second-level chunks are organized into third-level intervals (those assigned to processors with NUMA configurations), denoted as `g0`, `g1` and `g2`.

the initial workload among P computing nodes. Specifically, U and V are divided into P chunks of elements and indexed contiguously. Likewise, $\mathcal{A}$ is divided into P chunks of rows, matching the division of the output space, V.

The second-level (intra-node) partitioning, represented in Figure 2 by coloured divisions, distributes the local workload among Q computing units. This partitioning must conform to the actual performance of each unit for the sake of load balancing (*e.g.*, using a partitioning tool such as METIS [17]). This second-level partitions of vectors and matrices are stored in a set of *subvector* and *submatrix* objects. Besides, because different computing units feature different parallel paradigms, we introduce an abstract object called *device*, outlined in Listing 1. It declares pure-virtual methods for both data management and kernel execution, and its derivatives are specialised for different parallel implementations (*e.g.*, OpenMP, OpenCL, CUDA). They also can manipulate the submatrices for different sparse storage formats according to the problem requirements. Therefore, within a computing node, a set of devices is in charge of operating on sets of second-level subvectors and submatrices.

Listing 1: Outline of the abstract object device, a base class for architecture-specific implementations. Each computing unit is assigned a specialised instance of the devices object, depending on its architecture.

```
class device
{
  public:
    void allocate(subvector&, ...) = 0;
    void allocate(submatrix&, ...) = 0;
```

```
    void free(subvector&) = 0;
    void free(submatrix&) = 0;
    ...
    void spmv(submatrix&, subvector&, subvector&) = 0;
    ...
};
```

The third-level (intra-unit) partitioning, represented in Figure 2 by a dashed line, is rather an arrangement of the second-level partition into G intervals, assigned to G NUMA groups. For a given group, g, a pair of integers, *offset* and *size*, define the interval g. In contrast with the first- and second-level decomposition in which data is explicitly distributed, this implicit third-level arrangement exploits NUMA configurations at its best. Using thread affinity, threads are bound to NUMA groups. The data is initialised through a predictive first-touch so that each third-level interval resides in the thread's local memory bank, as in [5].

In comparison with other common implementations which assign an MPI rank to each computing unit [9, 10, 11, 6], this approach minimises the number of processes participating in MPI exchanges, as well as the global size of the messages. It takes full advantage of the intra-node topology and the shared-memory parallel processors and minimises inter-node communications. Moreover, this advantage will only strengthen as the memory hierarchies of modern supercomputers become more complex, increasing the number of accelerators and NUMA groups per node.

2.3 Overlapping execution of SpMV

In this multilevel decomposition approach, partitions may be identified by the pair of integers (p,q), that is, the first- and second-level ID. The elements in (p,q) partition become a set of contiguous indices, the *own* set of (p,q). By the same token, non-own elements become the *outer* set of (p,q).

Any non-zero entry, $(\mathcal{A})_{ij}$, represents a coupling between the ith element in V and the jth element in U. In distributed parallel processing, there is no guarantee that the ith and jth elements of vector spaces V and U are located in the same memory space, and hence the parallel SpMV kernel induces data exchanges between processes. For instance, the fifth row in Figure 2, located in (p_0,q_0), is coupled with an element of (p_0,q_1) and an element of (p_1,q_1). The communication stage affects the performance and limits the scalability of the operation, so necessary in large-scale applications. Therefore, the DM MIMD parallelisation of the SpMV must minimise the communication overhead.

The multithreaded overlapping execution scheme shown in Figure 3 aims at hiding the communications under computations. In this regard, the own set is further organised into *inner* and *interface* categories. The inner set consists of those elements of the own set which are coupled with own elements only. Conversely, interface set consists of those elements coupled with an element of the outer set. The outer elements of U required by the interface couplings are de-

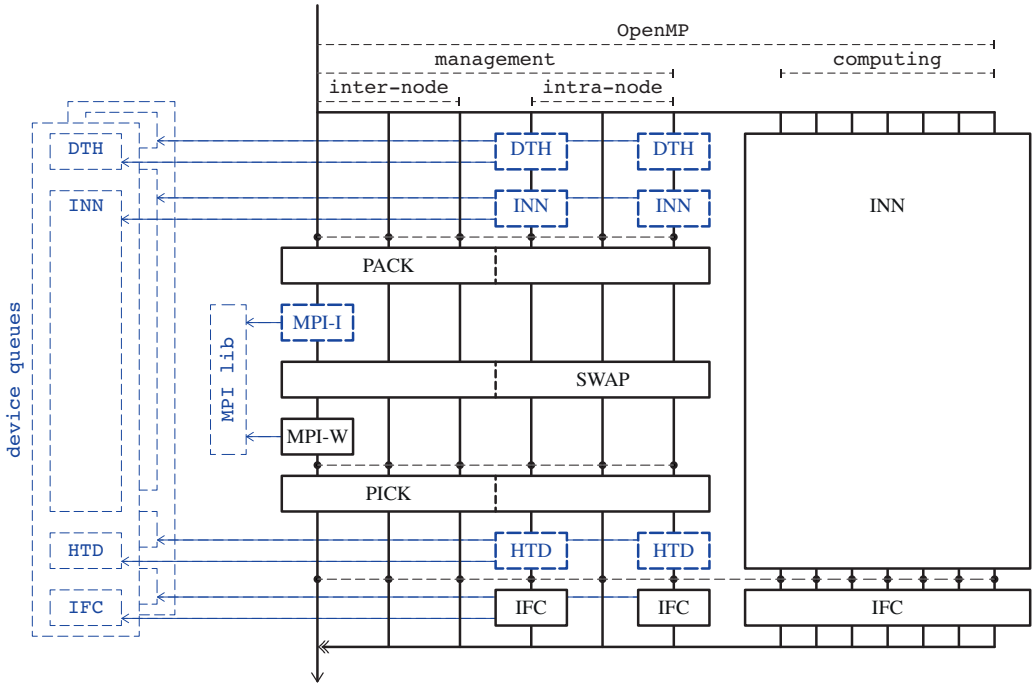

Figure 3: Overlapping execution diagram using a flat OpenMP parallel region. Dashed boxes denote non-blocking calls (*i.e.*, threads enqueue the job into the queue of a devices and continue without waiting). Definition of the boxes: INN, kernel execution for the inner set; IFC, kernel execution for the interface set; DTH, device-to-host copy of the interface; HTD, host-to-device copy of the halo; PACK, filling of the MPI send buffer with interface; SWAP, intra-node halo exchanges between devices; PICK, gathering of the halo from MPI receive buffers; MPI-I, launching of non-blocking send and receive calls; MPI-W, waiting at MPI barrier.

noted as *halo*. Given that the inner calculations are independent of the halo, this allows us to perform computations for inner elements simultaneously with communications, needed for the interface elements only. Thus, in a synchronous implementation, the execution time is:

$$t_{syn} = t_{inn} + t_{upd} + t_{ifc}.$$

Following the overlapping scheme, the execution time reduces to:

$$t_{ovl} = max(t_{inn}, t_{upd}) + t_{ifc}.$$

In our previous implementation [15, 7], we used nested OpenMP regions for distributing roles between groups of threads. This approach was inefficient on NUMA configurations because it was not able to ensure locality due to the dynamic scheduling and the undefined thread affinity within nested regions.

The new version of multithreaded execution shown in 3 avoids nested parallelism for computations. Instead, a flat OpenMP region assigns threads a fixed role, either computing or management. Computing threads are assigned a static third-level interval of the workload and

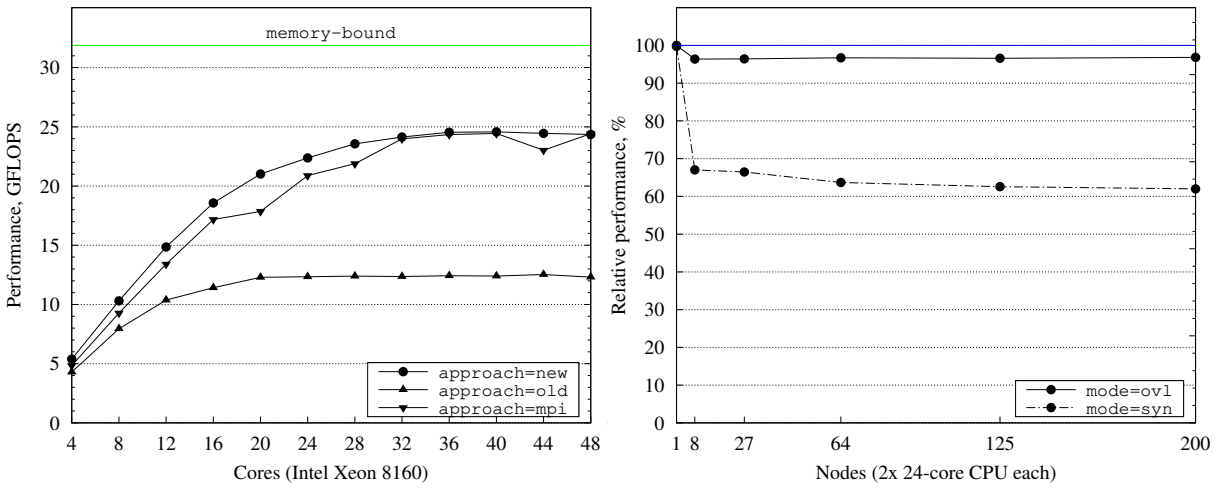

Figure 4: Comparison of different parallel modes of the SpMV kernel on a single node of MareNostrum 4 supercomputer (left) for a workload of 17 million rows. Weak scaling of different execution modes of the SpMV kernel on up to 200 nodes of MareNostrum 4 for a constant workload of 17 million cells per node (right).

perform the calculations on the hostside, that is, the CPU cores. These threads must be distributed evenly among NUMA groups and first-touch its chunk to ensure locality. Management threads are in charge of operating the second-level partitions assigned to accelerators through devices queues and participating in parallel processing of communications.

3 PERFORMANCE ANALYSIS

In this paper, the performance on multiprocessor nodes has been enhanced through a NUMA-aware, hierarchical parallelisation. The benefits of the new version have been tested on MareNostrum 4 supercomputer at the Barcelona Supercomputing Center. Its nodes with two Intel Xeon 8160 CPUs (24 cores, 2.1 GHz, 6 DDR4-2666 memory channels, 128 GB/s memory bandwidth, 33MB L3 cache) are interconnected through the Intel Omni-Path network (12.5 GB/s).

The sparse matrix used in this study arises from the symmetry-preserving discretisation [18] of the Laplacian operator on unstructured hex-dominant meshes of 17 million cells. Therefore, the majority of rows contain seven non-zero coefficients. The sparse matrix storage format used in this analysis is the classic ELLPACK [19], although we expect the benefits of this implementation approach are independent of the storage format.

The single-node results of the SpMV for different parallel execution approaches are shown in 4 (right). The MPI-only approach, which leads to the most compact data placement, is not different from the OpenMP mode with NUMA placement and thread binding to cores (denoted as `approach=new`). In this OpenMP mode threads are equally distributed among the two CPU sockets. The previous version of implementation (`approach=old`) with dynamic loop-based parallelism performs notably worse and due to remote memory accesses. Indeed, the

maximal performance achieved by new and old approaches is 24.15 and 12.53 GFLOPS, respectively, which corresponds to 75% and 39% of the theoretically achievable performance.

The weak scaling results of the SpMV kernel on up to 200 nodes, or 9,600 cores, are shown in 4 (right). The new approach is used to compare both the overlap and the synchronous execution modes. The plot shows the weak scaling in terms of relative performance compared to a single node for the constant workload of 17 million mesh cells per node. The performance of the overlap and synchronous modes drops to 96% and 67%, respectively, already at 384 cores due to the presence of communications. However, the communication overheads remain nearly constant until 9,600 cores.

The heterogeneous capabilities of this implementation approach, and further scalability results, have been shown in [7, 16] in application to algebra-based, CFD simulations.

4 CONCLUSIONS

In this work, a hierarchical parallel implementation approach of the SpMV on modern supercomputers has been described. The OpenMP parallelisation has been significantly improved in comparison with the previous version. A NUMA-aware strategy with proper thread binding and predictive data initialisation has increased the performance on dual-CPU nodes of the MareNostrum 4 supercomputer by a factor $\times 1.9$ times compared with dynamic loop-based parallel versions. Besides, the parallel performance has been demonstrated on up to 9,600 cores. The results show that our overlapping execution strategy can effectively hide most of the communication overheads with adequate loads per node.

It should be noted that the execution pattern described for the SpMV kernel can be used for any routine in numerical simulation codes that require a halo update and can be decomposed into inner and interface parts.

In our future work, we plan to focus on further accelerating the communications. For instance, we will study a NUMA-aware management of inter- and intra-node data exchanges. Specifically, the placement of exchange buffers and the binding of management threads considering the topology of the computing node.

ACKNOWLEDGEMENTS

The work of A. Gorobets has been funded by the Russian Science Foundation, project 19-11-00299.

The work of X. Álvarez-Farré, F. X. Trias and A. Oliva has been financially supported by the ANUMESOL project (ENE2017-88697-R) by the Spanish Research Agency (Ministerio de Economía y Competitividad, Secretaría de Estado de Investigación, Desarrollo e Innovación), and the FusionCAT project (001-P-001722) by the Government of Catalonia (RIS3CAT FEDER).

The studies of this work have been carried out using the MareNostrum 4 supercomputer of the Barcelona Supercomputing Center (projects IM-2020-2-0029 and IM-2020-3-0030); the TSUBAME3.0 supercomputer of the Global Scientific Information and Computing Center at Tokyo Institute of Technology; the Lomonosov-2 supercomputer of the shared research facilities of HPC computing resources at Lomonosov Moscow State University; the K-60 hybrid cluster of the collective use center of the Keldysh Institute of Applied Mathematics. The authors thankfully acknowledge these institutions for the compute time and technical support.

REFERENCES

[1] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis – SC '09*, (New York, New York, USA), p. 1, ACM Press, 2009.

[2] J. L. Greathouse and M. Daga, "Efficient Sparse Matrix-Vector Multiplication on GPUs Using the CSR Storage Format," in *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 769–780, IEEE, nov 2014.

[3] S. Filippone, V. Cardellini, D. Barbieri, and A. Fanfarillo, "Sparse Matrix-Vector Multiplication on GPGPUs," *ACM Transactions on Mathematical Software*, vol. 43, pp. 1–49, mar 2017.

[4] M. Almasri and W. Abu-Sufah, "CCF: An efficient SpMV storage format for AVX512 platforms," *Parallel Computing*, vol. 100, dec 2020.

[5] C. Alappat, J. Laukemann, T. Gruber, G. Hager, G. Wellein, N. Meyer, and T. Wettig, "Performance Modeling of Streaming Kernels and Sparse Matrix-Vector Multiplication on A64FX," *Proceedings of PMBS 2020: Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems, Held in conjunction with SC 2020: The International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–7, 2020.

[6] G. Xiao, K. Li, Y. Chen, W. He, A. Y. Zomaya, and T. Li, "CASpMV: A Customized and Accelerative SpMV Framework for the Sunway TaihuLight," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 1, pp. 131–146, 2021.

[7] X. Álvarez, A. Gorobets, F. Trias, R. Borrell, and G. Oyarzun, "HPC^2–A fully-portable, algebra-based framework for heterogeneous computing. Application to CFD," *Computers & Fluids*, vol. 173, pp. 285–292, sep 2018.

[8] W. J. Bolosky, R. P. Fitzgerald, and M. L. Scott, "Simple but effective techniques for NUMA memory management," *Operating Systems Review (ACM)*, vol. 23, no. 5, pp. 19–31, 1989.

[9] X. Guo, M. Lange, G. Gorman, L. Mitchell, and M. Weiland, "Developing a scalable

hybrid MPI/OpenMP unstructured finite element model," *Computers & Fluids*, vol. 110, pp. 227–234, 2015.

[10] F. Witherden, B. C. Vermeire, and P. E. Vincent, "Heterogeneous computing on mixed unstructured grids with PyFR," *Computers & Fluids*, vol. 120, pp. 173–186, oct 2015.

[11] B. A. Page and P. M. Kogge, "Scalability of Hybrid Sparse Matrix Dense Vector (SpMV) Multiplication," in *2018 International Conference on High Performance Computing & Simulation (HPCS)*, pp. 406–414, IEEE, jul 2018.

[12] D. Merrill and M. Garland, "Merge-Based Parallel Sparse Matrix-Vector Multiplication," in *SC16: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 678–689, IEEE, nov 2016.

[13] S. Chen, J. Fang, D. Chen, C. Xu, and Z. Wang, "Adaptive Optimization of Sparse Matrix-Vector Multiplication on Emerging Many-Core Architectures," in *2018 IEEE 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pp. 649–658, IEEE, jun 2018.

[14] G. Oyarzun, R. Borrell, A. Gorobets, F. Mantovani, and A. Oliva, "Efficient CFD code implementation for the ARM-based Mont-Blanc architecture," *Future Generation Computer Systems*, vol. 79, pp. 786–796, feb 2018.

[15] A. Gorobets, S. Soukov, and P. Bogdanov, "Multilevel parallelization for simulating compressible turbulent flows on most kinds of hybrid supercomputers," *Computers & Fluids*, vol. 173, pp. 171–177, sep 2018.

[16] X. Álvarez-Farré, A. Gorobets, and F. X. Trias, "A hierarchical parallel implementation for heterogeneous computing. Application to algebra-based CFD simulations on hybrid supercomputers," *Computers & Fluids*, vol. 214, p. 104768, jan 2021.

[17] D. Lasalle and G. Karypis, "Multi-threaded Graph Partitioning," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pp. 225–236, IEEE, may 2013.

[18] F. X. Trias, O. Lehmkuhl, A. Oliva, C. D. Pérez-Segarra, and R. W. C. P. Verstappen, "Symmetry-preserving discretization of NavierStokes equations on collocated unstructured grids," *Journal of Computational Physics*, vol. 258, pp. 246–267, feb 2014.

[19] D. R. Kincaid, T. C. Oppe, and D. M. Young, "ITPACKV 2D User 's Guide," tech. rep., Center for Numerical Analysis, University of Texas, 1989.