The Journal of Supercomputing
https://doi.org/10.1007/s11227-024-06011-1

Toward HPC application portability via C++ PSTL: the Gaia AVU-GSR code assessment

Giulio Malenza[1] · Valentina Cesare[2] · Marco Aldinucci[1] · Ugo Becciani[2] · Alberto Vecchiato[3]

Accepted: 19 February 2024

Abstract
The computing capacity needed to process the data generated in modern scientific experiments is approaching ExaFLOPs. Currently, achieving such performances is only feasible through GPU-accelerated supercomputers. Different languages were developed to program GPUs at different levels of abstraction. Typically, the more abstract the languages, the more portable they are across different GPUs. However, the less abstract and co-designed with the hardware, the more room for code optimization and, eventually, the more performance. In the HPC context, portability and performance are a fairly traditional dichotomy. The current C++ Parallel Standard Template Library (PSTL) has the potential to go beyond this dichotomy. In this work, we analyze the main performance benefits and limitations of PSTL using as a use-case the Gaia Astrometric Verification Unit-Global Sphere Reconstruction parallel solver developed by the European Space Agency Gaia mission. The code aims to find the astrometric parameters of $\sim 10^8$ stars in the Milky Way by iteratively solving a linear system of equations with the LSQR algorithm, originally GPU-ported with the CUDA language. We show that the performance obtained with the PSTL version, which is intrinsically more portable than CUDA, is comparable to the CUDA one on NVIDIA GPU architecture.

Keywords High-performance computing · Standard parallelism · GPU programming · Astrometry

1 Introduction

Following technological capabilities, scientific computing data rapidly increase in size in a broad range of applications, quickly approaching ~10–100 PB of input data. To analyze such large datasets, Peta- and ExaFLOPs performance is needed. Effectively exploiting this performance on large scientific applications (beyond deeply optimized small benchmarks) requires re-engineering and co-design of

Extended author information available on the last page of the article hardware and software. Calculations should be performed on growing numbers of nodes, and the architecture of each node is becoming increasingly heterogeneous [1]. While the research on alternative computing architectures is ongoing, today, the mainstream Exascale architecture can be sketched as a large cluster of nodes interconnected with a fast Mellanox/Cray network, where each node is an Intel/AMD/ARM multicore accelerated with multiple NVIDIA/AMD GPUs. Despite this blatant simplicity, each configuration of networking/multicore/GPUs may require re-coding the applications to reach an acceptable performance and efficiency, whereas moving from multicore to multicore+GPUs clusters also requires a significant code redesign.

A fast way to port to GPU existing code bases designed to run on CPU is provided by adopting high-level and directive-based languages, such as OpenMP [2] and OpenACC [3], which can allow GPU offload with minimal application redesigns [4]. This approach reduces code porting time but might result in poor performance since, without a code rearrangement thought for running on GPU, significant bottlenecks stemming from poor management of host-to-device (H2D) and device-to-host (D2H) data transfers, as well as inefficient memory access patterns, might occur. Typically, the application should be redesigned to massively exploit data parallelism and this might require a significant effort. Once redesigned, the coding of the application can also be made using a portable programming framework, such as OpenMP and OpenACC, providing a portable solution on different architectures. However, these portable frameworks typically underperform against low-level and architecture-specific languages, such as CUDA, leaving much room for optimization and fine-tuning architecture-dependent parameters, which can entail significant performance boosts. The performance vs portability dichotomy traditionally affected and still affects the high-performance computing realm [5, 6].

A potently new trade-off between performance and portability for coding GPU-accelerated applications can be provided by the class of programming frameworks where parallel algorithms are first-class programming constructs. Among them, it is worthwhile to mention C++ PSTL [7] (Parallel Standard Template Library), SYCL [8] (Standard Parallel Programming for C++), Kokkos [9], Fastflow [10], and HIP [11] (Heterogeneous-Compute Interface for Portability). Differently from programming frameworks designed to support the transition from sequential to parallel code, such as OpenMP and OpenACC, typically based on parallelization of loops [4], they allow programmers to reason about the properties of (data) parallel algorithms and their memory access patterns, and eventually to design and optimize abstract enough (thus portable) efficient code. In this work, we specifically focus on C++ PSTL. This parallel programming model is fascinating for portability because it does not require explicitly inserting any external directive not included in the standard of the language, which should be supported by any compiler vendor for supported architectures.

Starting from C++11, the first set of parallel constructs and concurrency topics have been introduced to the C++ standard. Further iterations of the C++ standards have progressively introduced new concepts and refined the specifications. In particular, from C++17 onward, the algorithms of the STL were extended by introducing execution policies to express and exploit parallel computations [12, 13].[1] A compiler that generates C++17 executable code is `nvc++` [14]. This compiler, part of the NVIDIA HPC SDK toolbox, can generate assembly code for GPUs or CPUs with multithreading by adding the compiler flag `-stdpar=gpu` or `-stdpar=multicore`. In recent studies (e.g., [14]), NVIDIA has demonstrated that clean and portable codes can be written without significant performance losses compared to CUDA codes.

We apply the C++ PSTL approach to a use-case: the Astrometric Verification Unit-Global Sphere Reconstruction (AVU-GSR) parallel solver [15]. This solver finds the astrometric parameters for $\sim 10^8$ stars in the Milky Way by solving a system of linear equations with the iterative LSQR algorithm [16, 17], which performs two matrix-by-vector products per iteration. The original code is written in CUDA for optimal performance on NVIDIA GPUs. In this code, we did not employ an implementation of the LSQR algorithm from an external library (e.g., BLAS,[2] LAPACK,[3] Intel oneMKL,[4] cuSPARSE,[5] or MAGMA[6]), to have full control of all control knobs to optimize the execution of LSQR according to the specific structure of the coefficient matrix of the system of equations (see also [18]). This custom implementation also allowed further optimization of the communications between the MPI processes and a better use of the memory and, thus, the possibility of solving larger systems.

Our work aims to demonstrate that the C++ PSTL code version does not significantly degrade performance concerning the CUDA version of the code by testing the two versions on three different HPC infrastructures. We chose this code as a use-case due to its compute-bound rather than MPI communications-bound nature.

The outline of this paper is as follows. After a description of the usages of the C++ PSTL to offload HPC applications to the GPU from the literature (Sect. 2), we briefly present the structure of the Gaia AVU-GSR code (Sect. 3) and of its previous parallel versions: on CPUs with MPI + OpenMP (Sect. 3.1) and on GPUs with MPI + OpenACC (Sect. 3.2). In Sect. 4, we describe the two versions of the code that are compared in this work: a new optimized version of the MPI + CUDA code (Sect. 4.1), and the new implementation of the code in C++ PSTL (Sect. 4.2). Then, we present some performance tests to compare the efficiency of the CUDA and C++ codes on different infrastructures and the weak scaling properties of the two codes on Leonardo CINECA supercomputer (Sect. 5). To execute these performance tests, we do not run systems that take as input real data but only simulated data. This is an obliged choice since an amount of real data to test the weak scalability up to a sufficiently large number of nodes (256 on Leonardo) has not been produced yet.

[1] https://en.cppreference.com/w/cpp/17.

[2] https://www.netlib.org/blas/.

[3] https://www.netlib.org/lapack/.

[4] https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html.

[5] https://docs.NVIDIA.com/cuda/cusparse/.

[6] https://icl.utk.edu/magma/.

However, real and simulated data are distributed in the same way in the system of equations and, thus, simulated data are as representative to study the weak scalability of the code as the real data. Section 6 concludes the paper and presents future directions for this work.

2 Related works

Developing applications in C++ using the PSTL compiled to offload computations to GPUs is a new approach in the HPC scenario. To the authors' knowledge, there are very few preceding examples of this approach.

Lin et al. [14] ported many representative HPC mini-applications employing standard C++17 to the GPU. These mini-applications are both compute- and memory-bound to span all possible cases. With proper benchmarks, they compared the performance of these mini-applications with other porting of the same codes performed in OpenMP, CUDA, and SYCL. The C++17 applications resulted in comparable performances with their previous porting versions on different platforms with diverse architectures, which indicates the high portability of this method besides its good performance.

Malenza et al. [19] ported to GPU with C++11 a mini-application built from the open-source software OpenFOAM[7] which computes the Gauss-Green gradient. The performance of this mini-application was tested on an ARM-multicore architecture with NVIDIA GPUs, obtaining a speedup from 1.66x to 5.11x compared to the same application running on the CPU. Given this promising result, the authors aim to port other sections of the OpenFOAM software to GPU with the same approach.

Asahi et al. [20] built a mini-application of a kinetic plasma simulation code based on the Vlasov equation. The mini-application is written in standard C++ and runs on multi-CPU and multi-GPU systems. They demonstrate that this method does not impair the readability and productivity of the mini-application and provides a performant portable solution with a certain speedup over a previous version written in Kokkos on Intel Icelake, NVIDIA V100, and A100 GPU architectures.

Among other works which exploit standard C++ to offload their codes to GPU, notice:

1. the work of Latt, et al. [12], who ported to GPU with C++ the Palabos software library for complex fluid flow simulations;
2. the work of Bhattacharya, et al. [21], which investigated different portable parallel solutions (Kokkos, SYCL, OpenMP, C++ standard) for high energy physics use cases on accelerators (such as GPUs) architectures and compared them according to a set of metrics, concluding that the C++ standard could provide the best solution. A follow-up is the work of Atif, et al. [22], that also considered the Alpaka language as a possible portable parallel solution;

[7] https://www.openfoam.com.

3. the work of Gomez et al. [13], who described the C++ porting of the ExaHyPE code, a solver engine for hyperbolic partial differential equations for complex wave phenomena;
4. the work of Kang, et al. [23], who wrote cuGraph primitives in standard C++ and tested algorithms using these primitives over 1000 GPUs.

Unlike the related works that adopt this method in mini-applications, we apply it to a complete and real-world scientific application.

3 The Gaia AVU-GSR code

The Astrometric Verification Unit-Global Sphere Reconstruction (AVU-GSR) parallel solver [15] is a code developed by the European Space Agency (ESA) mission Gaia [24],[8] launched in late 2013 and expected to end in mid-2025, whose aim is to map the positions and proper motions of ~1% ($\sim 10^9$) stars in our Galaxy. The AVU-GSR code finds these astrometric parameters for $\sim 10^8$ of these stars, the so-called "primary" stars [25], besides the attitude and the instrumental settings of the Gaia satellite and the Parametrized Post Newtonian (PPN) γ global parameter, by solving an overdetermined system of linear equations [15]:

$$\mathbf{A} \times \vec{x} = \vec{b}. \tag{1}$$

In Eq. (1), the coefficient matrix $\mathbf{A}$ is large and sparse and contains $\sim (10^{11}) \times (5 \times 10^8)$ elements. This matrix has high sparsity: of the 5×10^8 elements per row, only 24 at most are different from zero, and to fit it in the (distributed) main memory, it is encoded as a dense matrix $\mathbf{A}_\mathrm{d}$, which only contains the nonzero coefficients of $\mathbf{A}$. The dense matrix $\mathbf{A}_\mathrm{d}$ has at most $\sim (10^{11}) \times (24)$ elements (~19 TB), reducing the problem by 7 orders of magnitude. The known terms array $\vec{b}$ is as long as the number of rows of the matrix $\mathbf{A}$ ($\sim 10^{11}$ elements, ~800 GB) and the solution array $\vec{x}$ is as long as the number of columns of the original sparse matrix $\mathbf{A}$ ($\sim 5 \times 10^8$ elements, ~4 GB). These numbers refer to a case for a system at the end of the Gaia mission, that is, with a complete dataset.

The solution $\vec{x}$ has to be iteratively found in the least-squares sense with the LSQR algorithm [16, 17], which represents ~95% of the computation time of the entire AVU-GSR code.

Before the LSQR iterations start, the data are either imported in binary format, if we consider real data, or generated within the code, if we consider simulated data. Then, the system is preconditioned to improve the convergence speed of the LSQR. The system is de-preconditioned after the LSQR convergence (for more details see [15, 18, 26]. Since the system is overdetermined, several constraint equations are set at the bottom of the system.

8 https://sci.esa.int/web/gaia.

The majority of the computation time of the LSQR algorithm and of the entire AVU-GSR code basically consists in the call of the *aprod* function in the modes 1 and 2 (see Algorithm 1), where *aprod* 1 provides the iterative estimate of the known terms array $\vec{b}$:

$$\vec{b}^i += \mathbf{A} \times \vec{x}^{i-1}, \tag{2}$$

and *aprod* 2 provides the iterative estimate of the solution array $\vec{x}$:

$$\vec{x}^i += \mathbf{A}^T \times \vec{b}^i, \tag{3}$$

computing two matrix-by-vector products.

Algorithm 1 LSQR algorithm implemented for the Gaia mission

```
Require: $\mathbf{A}$, $\vec{b}$
    Each MPI process executes the aprod 2 function to compute $\vec{x}^0 += \mathbf{A}^T \times \vec{b}$
    MPI_Allreduce($\vec{x}^0$, MPI_SUM)
    while convergence || maximum iteration reached do
        Each MPI process executes the aprod 1 function to compute $\vec{b}^i += \mathbf{A} \times \vec{x}^{i-1}$
        MPI_Allreduce($\vec{b}^i$, MPI_SUM)
        Each MPI process executes the aprod 2 function to compute $\vec{x}^i += \mathbf{A}^T \times \vec{b}^i$
        MPI_Allreduce($\vec{x}^i$, MPI_SUM)
    end while
```

The solution is iterated in a while loop up to the algorithm's convergence, in the least-squares sense, or when a maximum number of iterations, set at runtime, is reached.

The AVU-GSR code was firstly parallelized on CPUs with a hybrid MPI + OpenMP approach [15] (Sect. 3.1), and then the LSQR part was ported to GPUs by replacing OpenMP firstly with OpenACC [18, 27, 28] (Sect. 3.2) and then with CUDA [26, 29] (Sect. 4.1). The MPI part is common to all versions of the code.

Algorithm 1 summarizes the main steps of the LSQR part of the AVU-GSR code common to all code implementations. After setting the initial condition with the *aprod* 2 function, the solution $\vec{x}$ is reduced with a `MPI_SUM` operation among the MPI processes with the `MPI_Allreduce()` collective and blocking communication operation. Then, the LSQR while loop starts and, after the end of each *aprod* region, a `MPI_Allreduce()` operation is performed. This function sums the partial results of the known terms array $\vec{b}$ found by each MPI process with *aprod* 1 and of the solution array $\vec{x}$ found by each MPI process with *aprod* 2.

The left part of Fig. 1 schematically represents how the coefficients are distributed in the original sparse matrix **A**. The rows of **A** ($\sim 10^{11}$ in the final Gaia

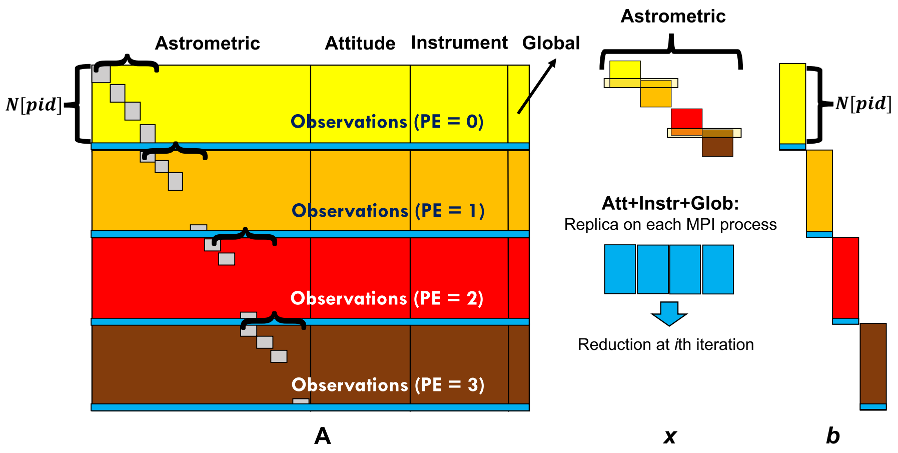


Fig. 1 Parallelization scheme of the system of equations (Eq. 1) on four MPI processes in a single node of a computer cluster. *Left panel*: coefficient matrix **A**. *Middle panel*: unknowns array $\vec{x}$. *Right panel*: known terms array $\vec{b}$. Different colors (yellow, orange, red, and brown) refer to different MPI processes or processing elements (PE). The block-diagonal part on the left side of the coefficient matrix illustrates its nonzero astrometric section. In the middle panel, the four square blocks diagonally placed and labeled as "Astrometric" represent the astrometric part of the solution array distributed among the MPI processes. All the light blue parts are replicated on all the MPI processes. These are the constraints equations (the narrow light blue bands at the end of each process in the coefficient matrix and the known terms array) and the attitude, instrumental, and global portions of the solution array (the four light blue aligned blocks, labeled as "Att+Instr+Glob"). At each iteration i, the replicated portions of $\vec{b}$ and $\vec{x}$ are reduced

dataset) are the system equations, and they represent the observations of the primary stars. Each star is observed $\sim 10^3$ times on average. After these rows, an additional number of constraint equations is set. The columns of **A** ($\sim 5 \times 10^8$ in the final Gaia dataset) represent the number of unknowns to solve.

The matrix is vertically structured in four sections: astrometric, attitude, instrumental, and global. The astrometric section represents ~90% of the entire matrix. The nonzero astrometric coefficients are organized in a block-diagonal structure, where each block represents a single star (gray blocks in the left part of Fig. 1). Each row has a limited number of nonzero astrometric parameters N_{Astro} between 0 and 5. We always set N_{Astro} to 5 for our simulations.

In the considered modelization, the attitude part counts $N_{\text{Att}} = 12$ nonzero parameters per row, structured in $N_{\text{Axes}} = 3$ blocks (representing the attitude axes) of $N_{\text{ParAxis}} = 4$ coefficients. Between two consecutive blocks, N_{DFA} elements are equal to zero, where N_{DFA} is the number of degrees of freedom carried by each attitude axis.

The instrumental part has a number of nonzero coefficients, N_{Instr}, between 0 and 6, which do not follow a regular pattern. The global part only contains $N_{\text{Glob}} = 1$ global coefficient, the γ parameter of the PPN formalism. In our simulations, we always set $N_{\mathrm{Instr}} = 6$ and we do not consider the global section of the system, i.e., having 23 nonzero elements per row of $\mathbf{A}_{\mathrm{d}}$.

Figure 1 represents a system of equations parallelized over four MPI processes in one compute node. The computation assigned to each MPI process is highlighted in yellow, orange, red, and brown. The computation replicated on each MPI process is marked in light blue. The computation referred to a horizontal section of the coefficient matrix, i.e., a portion of the total number of observations, is related to a single MPI process. Instead, the computation of the constraint equations is replicated in each process. This replication does not carry a significant overhead since the constraint equations represent a negligible fraction of the total number of equations. This solution avoids a complicated reorganization of the code. Given this schema, after *aprod* 1, only the fraction of the $\vec{b}$ array related to the constraints equations has to be reduced.

Given the regular block-diagonal organization of the astrometric section, this part is distributed among the MPI processes. This operation was less intuitive for the other three sections, which show a less regular structure and are, thus, replicated in each process. This replication does not entail a substantial loss of performance given that the attitude + instrumental + global sections only represent ~10% of the entire matrix. Given this schema, after *aprod* 2, only the fraction of the $\vec{x}$ array related to the attitude + instrumental + global sections has to be reduced.

In this work, we only compare the performances of the CUDA and C++ PSTL codes. However, a brief description of the other two code versions (OpenMP and OpenACC) is provided below to give a more profound background and show the code versions from which the currently employed versions were derived.

3.1 The OpenMP parallelization

The MPI + OpenMP version ran in production on CINECA infrastructure Marconi100 from 2014 to 2022. The computation related to each MPI process (colored portions in Fig. 1) is further parallelized on the CPU over the OpenMP threads. Listing 1 shows the astrometric part of *aprod* 1. *Aprod* 1 performs the matrix-by-vector product $\mathbf{A}_{\mathrm{d}}[i \times N_{\mathrm{par}} + j] \times x[\mathrm{offsetMi}[i] + j]$ and saves the result in the scalar variable *sum*. The index i iterates along the observations within a single MPI process with rank *pid*, $N[pid]$, and the index j iterates along the number of nonzero astrometric coefficients per row, where $N_{\mathrm{Astro}} = 5$. The for loop that iterates on i is parallelized with the `#pragma omp for` OpenMP directive, enclosed within a `#pragma omp parallel` region. Then, the result *sum* is cumulated in the known terms array, $b[i]$, as the index i advances. The variable `offsetMi[i]` depends on the array $\vec{M}_{\mathrm{i}}$, called "matrix index array," whose elements at even positions are the indexes of the first nonzero astrometric coefficients of each row of the original matrix $\mathbf{A}$.

The odd indexes of $\vec{M}_\mathrm{i}$ contain the same information for the attitude indexes. The variable `offset` is an offset local to the MPI process *pid*.

The elements of $\mathbf{A}_\mathrm{d}$ at line 10 are not read contiguously in memory when each *j*-loop concludes. At the beginning of every *j*-loop, the element of $\mathbf{A}_\mathrm{d}$ "jumps" $N_\mathrm{par} = 23$ elements, since this code section only refers to the astrometric vertical portion of the coefficient matrix (see Fig. 1), and the attitude + instrumental + global sections have to be jumped. The attitude, instrumental, and global sections of *aprod* 1 and the four sections of *aprod* 2 follow an analogous logic (see Algorithms 2 and 3 of [26] to see the complete pseudocodes of *aprod* 1 and 2 in the MPI + OpenMP, MPI + OpenACC, and MPI + CUDA versions of Gaia AVU-GSR code).

```
1  int main(int argc, char **argv)
2  {
3  #pragma omp parallel private(pid, sum) shared(N,x,Ad,Mi,b)
4      {
5  #pragma omp for
6          for(long i = 0; i < N[pid]; i++) {
7              sum = 0.0;
8              offsetMi = Mi[2*i] - offset;
9              for(long j = 0; j < NAstro; j++){
10                 sum = sum + Ad[i*Npar + j]*x[offsetMi + j];
11             }
12             b[i] += sum;
13         }
14     }
15     exit(EXIT_SUCCESS);
16 }
```

Listing 1: OpenMP aprod 1 astrometric section.

```
1  int main(int argc, char **argv)
2  {
3  #pragma acc parallel private(pid, sum) present(N,x,Ad,Mi,b)
4      {
5  #pragma acc loop
6          for(long i = 0; i < N[pid]; i++) {
7              sum = 0.0;
8              offsetMi = Mi[2*i] - offset;
9              for(long j = 0; j < NAstro; j++){
10                 sum = sum + Ad[i*Npar + j]*x[offsetMi + j];
11             }
12             b[i] += sum;
13         }
14     }
15     exit(EXIT_SUCCESS);
16 }
```

Listing 2: OpenACC aprod 1 astrometric section.

```
__global__ void aprod1_Kernel_astro(double* b_dev, const double*
    Ad_dev, const double* x_dev, const u_int32_t* MiAstro_dev,
    const u_int64_t Npar, const u_int32_t Nobs, const u_int32_t
    offset, const u_int16_t NAstro)
{
    double sum = 0.0;
    u_int32_t offsetMi = 0;
    u_int32_t i = blockIdx.x * blockDim.x + threadIdx.x;

    // Check if current CUDA thread is inside matrix borders
    while (i < Nobs) {
        sum = 0.0;
        offsetMi = MiAstro_dev[i] - offset;
        for(u_int32_t j = 0; j < NAstro; j++) {
            sum = sum + Ad_dev[i*Npar + j] * x_dev[offsetMi + j];
        }
        b_dev[i] = b_dev[i] + sum;

        i += gridDim.x*blockDim.x;
    }
}

int main(int argc, char **argv)
{
    aprod1_Kernel_astro<<<gridDim, blockDim>>>(b_dev, Ad_dev, x_dev
    , MiAstro_dev, Npar, N[pid], offset, NAstro);
    exit(EXIT_SUCCESS);
}
```

Listing 3: CUDA aprod 1 astrometric section.

```
void aprod1_Kernel_astro(double* b, const double* Ad, const double*
     x, const i_int* MiAstro, const l_int& Npar, const i_int& Nobs,
     const i_int& offset, const s_int& NAstro)
{
    const auto& start=std::views::iota(0,nobs).begin();
    const auto& end=std::views::iota(0,nobs).end();

    std::for_each(POL, start, end,[=](const auto i){
            double sum{ZERO};
            const i_int offsetMi = MiAstro[i] - offset;
            for(i_int j = 0; j < NAstro; ++j) {
                sum += Ad[i*Npar + j] * x[offsetMi + j];
            }

        b[i] = b[i] + sum;
        });
}

int main(int argc, char **argv)
{
    aprod1_Kernel_astro(b, Ad, x, MiAstro, Npar, Nobs, offset,
   NAstro);
    exit(EXIT_SUCCESS);
}
```

Listing 4: C++ PSTL aprod 1 astrometric section.

3.2 The OpenACC parallelization

The first GPU porting of the AVU-GSR code was performed with OpenACC [18, 27, 28]. It consisted in a preliminary study, to explore the feasibility of a GPU porting of the application, which required a minimal code rearrangement. The code runs on multiple GPUs per node, where the MPI processes are assigned to the GPUs of the node in a round-robin fashion. The optimal way to run the OpenACC code is to set the number of MPI processes per node equal to the number of GPUs of the node. This feature is also shared with the CUDA and C++ PSTL codes.

Listing 2 shows the astrometric part of *aprod* 1 ported with OpenACC. We can see that the structure of the code is the same as for OpenMP, with the difference that the `#pragma omp parallel` and `#pragma omp for` directives are now replaced by `#pragma acc parallel` and `#pragma acc loop` directives, respectively. The H2D and D2H data transfers were explicitly managed through specific directives.

With this first-order parallelization approach, the speedup over the OpenMP version was of ~1.5 [18, 28].

4 The CUDA and C++ implementations

4.1 The CUDA parallelization

The OpenACC porting paved the way to a more extensive optimization, where OpenACC was replaced by CUDA [26, 29]. This new porting required a substantial redesign. The grid of blocks of GPU threads where the different GPU regions are running has been customized to match the topology of the matrix-by-vector operations to solve. In this work, we compare the performance of our novel C++ PSTL version of the code (Sect. 4) with a version of the CUDA code that was further optimized compared to the original [26, 29]. Below, we describe this new optimized version, which preserves the general structure of the original CUDA code.

Listing 3 shows the astrometric part of *aprod* 1 ported with CUDA, composed by the definition of a CUDA kernel later called from the program's main function. We define the global index of the GPU thread within the grid of blocks of threads, i =`blockIdx.x` × `blockDim.x + threadIdx.x`, where `blockIdx.x` is the block index inside the grid, `blockDim.x` is the block size in threads unit, and `threadIdx.x` is the thread index local to each block. The GPU thread index i is directly mapped to the index of the observation local to the MPI process *pid*.

The "dev" suffix to the arrays name indicates that the arrays are allocated on the GPU device. The for loop-statement `for (`i `= 0;` $i < N[pid]$`);` i`++` is replaced by the while loop `while (`$i < N[pid]$`)`. The thread index i cannot be larger than $N[pid]$, since it would cause a memory overflow. The kernel runs on a grid of threads having size `gridDim` × `blockDim`, where `gridDim` is the number of blocks in the grid, and `gridDim` and `blockDim` are passed as parameters within the angle brackets "<<<>>>" in the kernel call inside the main of the code. This grid is generally smaller than the maximum size of the problem, $N[pid]$, and, thus, the problem is divided in tiles of size `gridDim` × `blockDim` which are covered by all the GPU threads until the thread index $i < N[pid]$. The quantities `gridDim` and `blockDim` were empirically found kernel by kernel to customize each of them. This approach provides better performance compared to a single tile case, as adopted in the original CUDA code of [26, 29], where the number of blocks of threads was set such that the thread index i can cover the entire `for` $(i =$ `0`; $i < N[pid]$); i++ for loop, and the number of threads in a block was always set to 1024 (the maximum value allowed on a NVIDIA V100 GPU present on Marconi100).

In Listing 3, the quantity `MiAstro_dev` represents only the astrometric part of the $\vec{M}_i$ array, that is, its even indexes. This provides a slight optimization in GPU programming since, with `MiAstro_dev`, the indexes can be contiguously read in memory, whereas in $\vec{M}_i$ a jump of one element must be performed for every memory access. An analogous `MiAtt_dev` array was defined in the code to contain the attitude (odd) indexes of $\vec{M}_i$.

The unsigned int types `u_int16_t`, `u_int32_t`, and `u_int64_t` were used to guarantee greater portability compared to correspondent signed int types.

Besides porting the code with CUDA, we also dealt better with the H2D and D2H data transfers, and more code regions were GPU-ported compared to the OpenACC version. With these optimizations, the percentages of time fraction in one LSQR iteration due to GPU computation, data transfers, and CPU computation are $\gtrsim$90%, ~3 %, and ~3% [26, 29]. Moreover, the speedup over the CPU OpenMP code increases from ~1.5 [18], as obtained for the OpenACC code, to ~ 14, [26, 29] for the original CUDA code [26, 29] and even further for this current optimized CUDA version. These speedups are calculated for the same problem, which maps differently depending on whether the code runs on the CPU and the GPU. The speedup of ~14 presents an increasing trend with the memory occupied by the system and with the number of employed GPU resources [26, 29].

4.2 The C++ parallelization

The main issue with the CUDA code is its reduced portability since CUDA is limited to run on GPUs with NVIDIA architecture. Moreover, some optimization parameters need to be tuned even to run on different NVIDIA GPUs. On the one hand, this is not particularly problematic for the Gaia code, which is close to its mission end and likely needs to migrate only once from Marconi100 to Leonardo. On the other hand, the class of applications based on the LSQR algorithm will definitely benefit from more portable and comparably performant solutions in the perspective of running on different (pre-)Exascale platforms. For this purpose, the code was rewritten in C++20 PSTL and compiled with the `nvc++` compiler.

Originally, the STL mainly consisted of three components: iterators, containers, and algorithms. Algorithms can be subdivided into different classes. There are algorithms to iterate and transform container elements (`std::for_each` and `std::transform`), algorithms useful to perform summary operations (`std::reduce`), algorithms to select containers elements (`std::search` and std::find), algorithms to copy and fill containers (std::copy and std::fill), algorithms to sort containers (std::sort), and so on. The main CUDA functions of LSQR were rewritten using these algorithms.

As stated in Sect. 3, the system comprises four parts: the astrometric, the attitude, the instrumental, and the global ones. The CUDA version of the code treats the aprod 1 and 2 computation of each of these four parts in a different function. Listings 3 and 4 show how the astrometric section of *aprod* 1 transform from CUDA to C++ PSTL.

To provide another example, we show how the dscal function, employed in several points of the code to scale an array to a specific factor "sign $*$ val", was transformed from CUDA to C++. The original CUDA code is:

```
__global__ void dscal (double* array_dev, const double val, const
    u_int32_t N, const int sign)
{
    u_int32_t i = blockIdx.x * blockDim.x + threadIdx.x;

    while (i < N) {
            array_dev[i] = sign*(array_dev[i]*val);
            i += gridDim.x*blockDim.x;
            }
}
```

Listing 5: CUDA dscal routine

where "array_dev" can be either $\vec{b}_{\text{dev}}$ or $\vec{x}_{\text{dev}}$.
This function was rewritten in standard C++ code in the following way:

```
inline void dscal(double* array, const double val, const int32_t& N
    , const int sign){
    std::transform(POL,
                        array,
                        array+N,
                        array,
                        [=](auto arrayi){
                            return sign*(arrayi*val);
                        });
}
```

Listing 6: C++ dscal routine

In Listings 4 and 6, POL is a macro specifying the execution policy, defined in the header <execution>, which can be:

- std::execution::seq → The sequenced policy (since C++17). This policy forces the execution of an algorithm to run sequentially on the CPU.
- std::execution::unseq → The unsequenced policy (since C++20). This policy executes the calling algorithm using vectorization on the calling thread.

- `std::execution::par` → The parallel policy (since C++17). This policy tells the compiler that the algorithm could be run in parallel.
- `std::execution::par_unseq` → The parallel unsequenced policy (since C++20). This policy allows the algorithm to be run in parallel on multiple threads each able to vectorize the calculation.

In the C++ version, H2D and D2H data transfers cannot be dealt with explicit directives, as in OpenACC and CUDA. These are handled indirectly via the Unified Memory mechanism. Limiting data transfers is fundamental to saving performance. On the Gaia AVU-GSR code, the system matrix, which represents most of the data, is transferred to GPUs before the start of the iteration phase. During iterations, only a couple of vectors are updated between GPUs. For this reason, we do not expect a significant overhead deriving from H2D and D2H data movements.

5 Results

To compare the performance of the CUDA and C++ codes, we perform two classes of tests: (1) we compare their efficiency on a single node of three different infrastructures, Leonardo, Karolina (IT4I), and EPITO@HPC4AI [30], whose hardware and software features are detailed in Table 1; (2) we investigate their weak scalability up to 256 nodes (1024 GPUs) on the Leonardo CINECA supercomputer.

To evaluate the efficiency, we measure the average time (that includes every code section, e.g., MPI communications, GPU kernels execution, H2D and D2H data transfers, and CPU execution per iteration) of 100 iterations of the LSQR for a system that occupies 95% of the total GPU memory of the node (244 GB on Leonardo, 304 GB on Karolina, and 76 GB on EPITO). We take the maximum value among all the MPI processes to estimate the average iteration time. To increase the statistical significance of the measurement, we execute three runs per code and infrastructure and estimate the average iteration time as the mean of the mean values resulting from the three simulations. The maximum average iteration time error is the standard deviation of the values obtained from the three simulations.

The histogram in Fig. 2 shows the efficiency of the C++ PSTL code on Leonardo, Karolina, and EPITO clusters, calculated as the ratio of the maximum average iteration time of the CUDA and C++ codes to compare the performance of the C++ code concerning the CUDA code. The efficiency of the C++ code is of 0.808 ± 0.003, 0.6417 ± 0.0008, and 0.691 ± 0.003 on EPITO, Karolina, and Leonardo, where the error bars on the efficiencies are calculated by propagating the errors on the average iteration time of the CUDA and C++ codes, appearing at numerator and denominator, respectively. The larger efficiency on EPITO could be because there are only 2 GPUs per node on this infrastructure; the unified memory mechanism works better than on a Leonardo and Karolina node, where there are 4 and 8 GPUs, respectively.

To investigate the weak scalability of the codes, we measure the total time of a complete simulation, run for 100 LSQR iterations, also considering the time before

Table 1 Hardware features and software specifications of the infrastructures employed for our analyses

Name	Hardware	Software
Leonardo	1 x CPU Intel Xeon 8358 32 cores, 2.6 GHz	gcc−12.2.0
	4 x GPU NVIDIA A200, 256 GB HBM2	NVHPC 23.5
	512 (8 x 64) GB RAM DDR4 3200 MHz	HPCX (OMPI 4.1.5rc2)
	4x 200 Gb/s IB port	CUDA 11.8
	4x PCIe Gen 4.0 CPU-GPU, 32GB/s	
	NVLink GPU-GPU, 600 GB/s	
Karolina	2x AMD EPYC^{TM} 7763, 64 cores, 2.45 GHz	gcc−12.2.0
	8 NVIDIA A100 GPUs, 320 GB HBM2	NVHPC 23.5
	1024 GB DDR4 3200MT/s RAM	HPCX (OMPI 4.1.5)
	4x 200 Gb/s IB port	CUDA 12.0
	NVLink GPU-GPU, 600 GB/s	
	8x PCIe Gen 4.0 CPU-GPU	
EPITO @HPC4AI	1 ARM Ampere Altra CPU (80 cores)	gcc−12.2.0
	2 NVIDIA A100 GPUs, 80GB HBM2	NVHPC 23.5
	2x 100Gb/s IB port	HPCX (OMPI 4.1.5rc2)
	2x PCIe Gen 4.0 CPU-GPU	CUDA 11.7

and after LSQR of the CUDA and C++ codes from 1 to 256 nodes (from 4 to 1024 GPUs) on Leonardo. The systems occupy 95% of the total GPU memory of one node multiplied by the number of nodes (244 GB on one node, 488 GB on two nodes, up to 62464 GB = 62.464 TB on 256 nodes). As already mentioned in the introduction, we do not have such an amount of real data, and thus, we employed simulated data. Yet, simulated data are as representative as real data to investigate the weak scalability of the code since both real and simulated data are distributed in the same way in the system of equations, following the schema of Fig. 1.

We again run three simulations per selected number of GPUs to increase the statistical significance of the measurement. We also measure the total time of the simulations due to the MPI communications alone. We take the maximum value from all the MPI processes for the total time and communication time. As before, each time measurement is the mean of the times (either total or of MPI communications alone) resulting from the three simulations for each number of GPUs, and its uncertainty is estimated as the standard deviation of the same three values.

Figure 3 shows the weak scaling of the CUDA (blue curves) and C++ (red curves) codes on Leonardo, where Fig. 3a and 3b represent the maximum MPI communication and maximum total times, respectively, as a function of the number of nodes (bottom axis) and of GPUs (top axis). Figure 3c represents instead the computation-over-communication ratio, given by the ratio between the maximum computation time and the maximum MPI communication time. We estimate the maximum computation time as the difference between the maximum total and maximum communication time. The uncertainty on the computation-over-communication ratio is calculated by propagating the errors on the numerator and the denominator. The dashed lines in each panel represent the ideal cases: the values of the represented

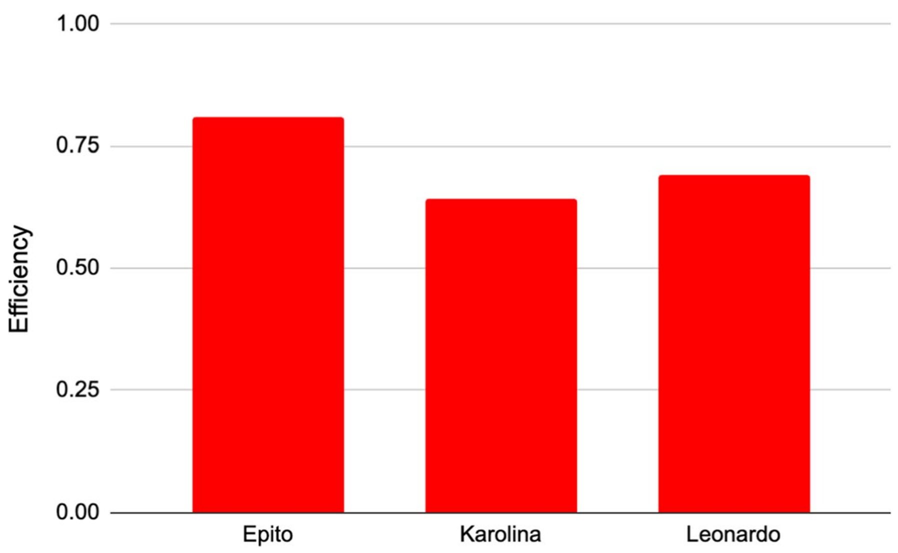
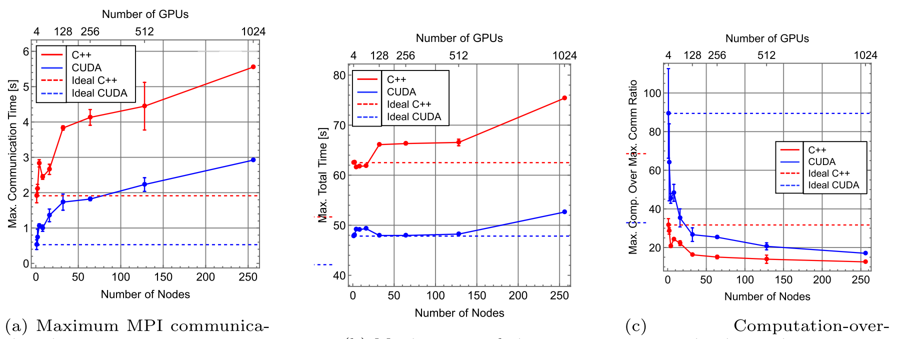


Fig. 2 Efficiency of the C++ code on EPITO, Karolina, and Leonardo infrastructures. The efficiencies are calculated as the ratio of the maximum average iteration time of the CUDA and C++ codes to compare the performance of the C++ code concerning the CUDA code

quantities measured on one node. The code is compute-bound; computation dominates communication even on 256 nodes (1024 MPI processes) when the number of MPI communications substantially increases. The computation-over-communication ratio passes from 32 ± 3 to 12.5744 ± 0.0008, for the C++ code, and from 89 ± 23 to 17.02 ± 0.13, for the CUDA code, from 1 to 256 nodes. The MPI communication time grows as the number of nodes increases, but it seems to follow a logarithmic trend, as expected. In this way, the code remains compute-bound even on 256 nodes, which is sufficient to solve a system with a size produced at the end of the Gaia mission.

Figure 4a represents the maximum average iteration time for the C++ (red) and CUDA (blue) codes as a function of the number of nodes (bottom axis) and of GPUs (top axis). Figure 4b reflects what is shown in Fig. 4a but in terms of the efficiency. As in Fig. 2, the efficiency of the C++ code is calculated as the ratio between the maximum average iteration time of the CUDA and C++ codes. The bars of the histogram related to 1 node correspond to the Leonardo bar of histogram 2. The efficiency of the C++ code is nearly constant along the entire range of nodes (GPUs), with a maximum of 0.717 ± 0.002, on 4 nodes (16 GPUs), and a minimum of 0.6142 ± 0.0005, on 256 nodes (1024 GPUs).

In summary, computation is mainly given by three matrix-by-vector products (two within the LSQR cycle and one before it), and communication is mainly given by three `MPI_Allreduce` operations (two within the LSQR cycle and one before it). The MPI communications are subdominant concerning the total, also for a large number of MPI processes, because, despite the global synchronization operations, the reduced data are much smaller than the problem assigned per node, which



(a) Maximum MPI communication time.

(b) Maximum total time.

(c) Computation-over-communication ratio.

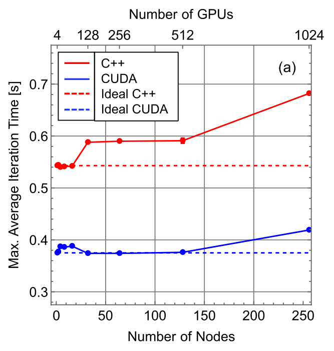
Fig. 3 Weak scaling properties of the C++ (red) and CUDA (blue) codes from 1 to 256 nodes (i.e., from 4 to 1024 GPUs) of Leonardo, as a function of the number of nodes (bottom) and of GPUs (top axis). Error bars are present in each panel, but some are not visible since they are smaller than the point markers. The dashed lines in each panel are shown as a reference, and they represent the ideal cases, i.e., the quantities measured on one node for the C++ (dashed red) and the CUDA (dashed blue) codes

guarantees good weak scalability of both CUDA and C++ codes up to 256 nodes (1024 GPUs) on Leonardo.

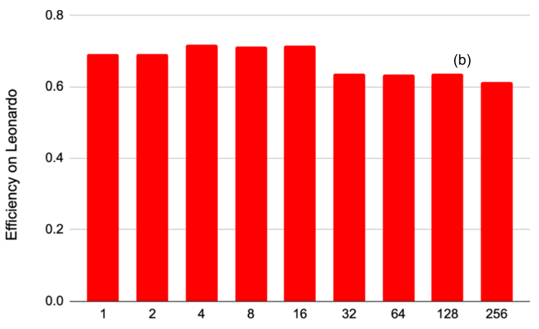
6 Conclusions

We present a new approach in perspective of the performance portability of HPC applications suitable for large-scale systems. The approach is based on C++ PSTL, which provides a good trade-off between performance and portability. We apply this methodology to a use-case, that is, the Gaia AVU-GSR solver, previously parallelized on the CPU with MPI+OpenMP and then ported to the GPU firstly by replacing OpenMP with OpenACC and then directly by coding with CUDA (that is the version currently used for production). Since the Gaia AVU-GSR solver implements a LSQR iterative algorithm, widely employed in HPC codes [31–37], and has a computation-dominated, instead of communication-dominated, nature, we considered this use case paradigmatic for showcasing the benefits of PSTL for a broad class of scientific applications.

The primary objective of this work was to demonstrate that the C++ PSTL version of the application is not significantly slower than its CUDA counterpart. For this, we compared the performance of the CUDA and C++ codes on a single node of three different accelerated architectures, EPITO (ARM+NVidia), Karolina (AMD+NVIDIA), and Leonardo (Intel+NVIDIA), and we explored their weak scalability on the Leonardo supercomputer. The two codes present comparable performance and scaling behaviors (using C++ we achieved about 70% of the CUDA performance), which indicates that C++ PSTL is a suitable approach to parallelize the Gaia AVU-GSR code in perspective of the future Gaia Data Releases. More generally, it points to C++ PSTL as a more portable and comparably performant solution for other HPC applications.



(a) Maximum average iteration time.



(b) Efficiency of the C++ PSTL code.

Fig. 4 Figure 4a: maximum average iteration time of the C++ (red) and CUDA (blue) codes from 1 to 256 nodes (i.e., from 4 to 1024 GPUs) of Leonardo, as a function of the number of nodes (bottom axis) and of GPUs (top axis). The error bars on the maximum average iteration time, calculated as the standard deviation from the three simulations, are smaller than the point markers. The dashed lines are shown as a reference and represent the ideal cases, i.e., the maximum average iteration time measured on one node for the C++ (dashed red) and the CUDA (dashed blue) codes. Figure 4b: efficiency of the C++ PSTL code calculated as the ratio between the maximum average iteration time of the CUDA and C++ codes, from 1 to 256 nodes (i.e., from 4 to 1024 GPUs) of Leonardo

Unlike the CUDA code, where H2D and D2H data transfers are explicitly handled by the programmer, in C++ PSTL, they are automatically managed by the compiler with the Unified Memory mechanism. We structured the Gaia AVU-GSR code to minimize data transfers during iteration. The results of [26, 29] show that data transfers only represent the ~3% of one iteration. For this reason, using Unified Memory in C++ PSTL did not produce a relevant overhead, which could be even more reduced by further developing the C++ PSTL approach on the AVU-GSR code in the future.

We have to point out that rewriting the code in C++ PSTL to make it more portable while maintaining its performance does not necessarily come for free. Comparing Listing 4 with Listings 1 and 2, which show the aprod 1 function in C++ PSTL, OpenMP, and OpenACC, we can see that the parallelization with C++ PSTL might be less intuitive than the one of OpenMP and OpenACC, obtained with high-level directives, by a non-expert programmer. As for CUDA (Listing 3), the time-to-solution to obtain a version of the code in C++ PSTL might be longer than in OpenMP and OpenACC since a code rearrangement is required. However, the advantages of this effort can be seen in the long term. C++ PSTL porting produces a single version of the code that can run on different platforms without significantly losing performance, as will be tested in future work. Moreover, it is essential to note that, being C++ PSTL a standard, it guarantees more straightforward code maintainability.

Since with C++ PSTL we do not significantly loose in performance compared to CUDA, in the near future, we aim to bring this analysis to a next level, i.e., to test the performance portability of the C++ PSTL Gaia AVU-GSR code on different GPU architectures such as NVIDIA and AMD. Moreover, we aim to extend this study to other code versions, which have to be either optimized or built, parallelized with OpenACC, OpenMP that allows GPU offload, HIP, SYCL, and Kokkos. We plan to test the scalability properties of these codes compiled with different compilers (i.e., *llvm* and *clang* besides *nvc*) on several architectures, such as RISC-V, and accelerators, such as AMD GPUs, running for example on Setonix, located at the Pawsey Supercomputing Center and ranked in the Top500 list. A more detailed study can be performed, evaluating the FLOPs of each operation on the CPU and GPU and how each MPI communication operation would increase in time as the number of nodes increases. This would provide a theoretical scalability study to be compared with the experimental results.

Besides scalability, we will also aim to investigate the numerical stability of the system solution at increasing model sizes executed on an increasing number of compute resources. Moreover, we plan to compare the energy consumption of the diverse application versions on different platforms to address "Green Computing" (i.e., the ability to build infrastructures and applications to execute calculations that involve large data volumes without excessively increasing the energetic consumption), another important target of HPC, besides performance portability. For the latter point, Setonix would be a suitable platform, being classified as the world's fourth "greenest" supercomputer as ranked in the Green500[9] list.

Acknowledgements We thank the referees for improving this work. The results reported in this work are part of foreground knowledge of the project *"Investigation of Scalability, Numerical Stability, and Green Computing of LSQR-based applications involving Big Data in perspective of Exascale systems: the ESA Gaia mission case study"* minigrant awarded to Dr. Valentina Cesare by INAF. We acknowledge EuroHPC JU for the access to the *Karolina* supercomputer at IT4Innovations (Czech Republic). We acknowledge the CINECA award under the ISCRA initiative, for the availability of high-performance computing resources and support. We acknowledge the University of Torino for the access to *EPITO@HPC4AI* cluster. This work has been also partially supported by the Spoke 1 "FutureHPC & BigData" of the ICSC–Centro Nazionale di Ricerca in High Performance Computing, Big Data and Quantum Computing and hosting entity, funded by European Union-Next GenerationEU; by the Italian Space Agency (ASI) (Grant No. 2018-24-HH.0) as part the Gaia mission; by EuroHPC JU under the project EUPEX (grant n. 101033975).

Funding Open access funding provided by Università degli Studi di Torino within the CRUI-CARE Agreement.

[9] https://www.top500.org/lists/green500/2023/11/.

References

1. Carpenter P, Utz U-H, Narasimhamurthy S, Suarez E (2022) Heterogeneous high performance computing. Zenodo. https://doi.org/10.5281/zenodo.6090425
2. Dagum L, Menon R (1998) Openmp: an industry-standard api for shared-memory programming. IEEE Comput Sci Eng 5(1):46–55. https://doi.org/10.1109/99.660313
3. Farber R (2016) Parallel programming with OpenACC, 1st edn. Morgan Kaufmann Publishers Inc., San Francisco
4. Aldinucci M, Cesare V, Colonnelli I, Martinelli AR, Mittone G, Cantalupo B, Cavazzoni C, Drocco M (2021) Practical parallelization of scientific applications with OpenMP, OpenACC and MPI. J Parallel Distrib Comput 157:13–29. https://doi.org/10.1016/j.jpdc.2021.05.017
5. Reed DA, Gannon D, Dongarra JJ (2022) Reinventing high performance computing: challenges and opportunities. arXiv:abs/2203.02544, https://doi.org/10.48550/arXiv.2203.02544
6. Amaral V, Norberto B, Goulão M, Aldinucci M, Benkner S, Bracciali A, Carreira P, Celms E, Correia L, Grelck C, Karatza H, Kessler C, Kilpatrick P, Martiniano H, Mavridis I, Pllana S, Respício A, Simão J, Veiga L, Visa A (2019) Programming languages for data-intensive HPC applications: a systematic mapping study. Parallel Comput. https://doi.org/10.1016/j.parco.2019.102584
7. open-std.org. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3724.pdf. Accessed 15-01-2024 (2013)
8. Group TKSW (2021) SYCL 2020 Specification (revision 4). Rev. 8. https://registry.khronos.org/SYCL/specs/sycl-2020/pdf/sycl-2020.pdf
9. Edwards HC, Trott CR, Sunderland D (2014) Kokkos: enabling manycore performance portability through polymorphic memory access patterns. J Parallel Distrib Comput 74(12):3202–3216. https://doi.org/10.1016/j.jpdc.2014.07.003
10. Aldinucci M, Ruggieri S, Torquati M (2010) Porting decision tree algorithms to multicore using FastFlow. In: Balcázar JL, Bonchi F, Gionis A, Sebag M (eds) Proceedings of European Conference in Machine Learning and Knowledge Discovery in Databases (ECML PKDD). LNCS, vol 6321. Springer, Barcelona, pp 7–23. https://doi.org/10.1007/978-3-642-15880-3_7
11. AMD (2021) AMD HIP Programming Guide. Rev. 1210. https://raw.githubusercontent.com/RadeonOpenCompute/ROCm/rocm-4.5.2/AMD_HIP_Programming_Guide.pdf
12. Latt J, Coreixas C, Marson F, Thyagarajan K, Santana Neto JP, S S, Brito G (2021) Porting a scientific application to GPU using C++ standard parallelism. https://doi.org/10.13140/RG.2.2.27117.92647
13. Gomez U, Brito Gadeschi G, Weinzierl T (2023) GPU offloading in ExaHyPE through C++ standard algorithms, pp 2302–09005 https://doi.org/10.48550/arXiv.2302.09005, arXiv:2302.09005 [cs.MS]
14. Lin W-C, Deakin T, McIntosh-Smith S (2022) Evaluating iso c++ parallel algorithms on heterogeneous hpc systems. In: 2022 IEEE/ACM International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS), pp 36–47. https://doi.org/10.1109/PMBS56514.2022.00009
15. Becciani U, Sciacca E, Bandieramonte M, Vecchiato A, Bucciarelli B, Lattanzi MG (2014) Solving a very large-scale sparse linear system with a parallel algorithm in the gaia mission. In: 2014 International Conference on High Performance Computing Simulation (HPCS), pp 104–111. https://doi.org/10.1109/HPCSim.2014.6903675
16. Paige CC, Saunders MA (1982) Lsqr: an algorithm for sparse linear equations and sparse least squares. ACM Trans Math Softw (TOMS) 8(1):43–71. https://doi.org/10.1145/355984.355989
17. Paige CC, Saunders MA (1982) Algorithm 583: Lsqr: sparse linear equations and least squares problems. ACM Trans Math Softw (TOMS) 8(2):195–209. https://doi.org/10.1145/355993.356000
18. Cesare V, Becciani U, Vecchiato A, Lattanzi MG, Pitari F, Raciti M, Tudisco G, Aldinucci M, Bucciarelli B (2022) The Gaia AVU-GSR parallel solver: preliminary studies of a LSQR-based application in perspective of exascale systems. Astron Comput 41:100660. https://doi.org/10.1016/j.ascom.2022.100660. arXiv:2212.11675 [astro-ph.IM]
19. Malenza G, et al (2022) Analysis of openfoam performance obtained using modern c++ parallelization techniques. https://hdl.handle.net/20.500.11767/130796
20. Asahi Y, Padioleau T, Latu G, Bigot J, Grandgirard V, Obrejan K (2022) Performance portable vlasov code with c++ parallel algorithm. In: 2022 IEEE/ACM international workshop on

performance, portability and productivity in HPC (P3HPC), pp 68–80. https://doi.org/10.1109/P3HPC56579.2022.00012

21. Bhattacharya M, Calafiura P, Childers T, Dewing M, Dong Z, Gutsche O, Habib S, Ju X, Kirby M, Knoepfel K, Kortelainen M, Kwok M, Leggett C, Lin M, Pascuzzi VR, Strelchenko A, Viren B, Yeo B, Yu H (2022) Portability: a necessary approach for future scientific software. https://doi.org/10.48550/arXiv.2203.09945, arXiv:2203.09945 [physics.comp-ph]
22. Atif M, Battacharya M, Calafiura P, Childers T, Dewing M, Dong Z, Gutsche O, Habib S, Knoepfel K, Kortelainen M, Kwok KHM, Leggett C, Lin M, Pascuzzi V, Strelchenko A, Tsulaia V, Viren B, Wang T, Yeo B, Yu H (2023) Evaluating portable parallelization strategies for heterogeneous architectures in high energy physics. https://doi.org/10.48550/arXiv.2306.15869, arXiv:2306.15869 [hep-ex]
23. Kang S, Hastings C, Eaton J, Rees B (2023) cugraph c++ primitives: vertex/edge-centric building blocks for parallel graph computing. In: 2023 IEEE international parallel and distributed processing symposium workshops (IPDPSW), pp 226–229 . https://doi.org/10.1109/IPDPSW59300.2023.00045
24. Gaia Collaboration, Vallenari A, Brown AGA, Prusti T, et al (2023) Gaia Data Release 3. Summary of the content and survey properties. Astron Astrophys 674, 1 https://doi.org/10.1051/0004-6361/202243940, arXiv:2208.00211 [astro-ph.GA]
25. Vecchiato A, Bucciarelli B, Lattanzi MG, Becciani U, Bianchi L, Abbas U, Sciacca E, Messineo R, De March R (2018) The global sphere reconstruction (GSR). Demonstrating an independent implementation of the astrometric core solution for Gaia. Astron Astrophys 620:40. https://doi.org/10.1051/0004-6361/201833254, arXiv:1809.05145 [astro-ph.IM]
26. Cesare V, Becciani U, Vecchiato A, Lattanzi MG, Pitari F, Aldinucci M, Bucciarelli B (2023) The MPI + CUDA Gaia AVU-GSR parallel solver toward next-generation Exascale infrastructures. Publ Astron Soc Pac 135(1049):074504. https://doi.org/10.1088/1538-3873/acdf1e. arXiv:2308.00778 [astro-ph.IM]
27. Cesare V, Becciani U, Vecchiato A, Lattanzi MG, Pitari F, Raciti M, Tudisco G, Aldinucci M, Bucciarelli B (2021) Gaia AVU-GSR parallel solver towards exascale infrastructure. In: Astronomical Data Analysis Software and Systems XXXI, Astronomical Society of the Pacific Conference Series. Astronomical Society of the Pacific Conference Series, vol 527, p 457 (in Press)
28. Cesare V, Becciani U, Vecchiato A, Pitari F, Raciti M, Tudisco G (2022) The Gaia AVU-GSR parallel solver: preliminary porting with OpenACC parallelization language of a LSQR-based application in perspective of exascale systems. INAF Technical Reports 163. https://doi.org/10.20371/INAF/TechRep/163
29. Cesare V, Becciani U, Vecchiato A (2022) The MPI+CUDA Gaia AVU-GSR parallel solver in perspective of next-generation Exascale infrastructures and new green computing milestones. INAF Technical Reports 164. https://doi.org/10.20371/INAF/TechRep/164
30. Aldinucci M, Rabellino S, Pironti M, Spiga F, Viviani P, Drocco M, Guerzoni M, Boella G, Mellia M, Margara P, Drago I, Marturano R, Marchetto G, Piccolo E, Bagnasco S, Lusso S, Vallero S, Attardi G, Barchiesi A, Colla A, Galeazzi F (2018) HPC4AI, an AI-on-demand federated platform endeavour. In: ACM computing frontiers, Ischia, Italy. https://doi.org/10.1145/3203217.3205340
31. Naghibzadeh S, van der Veen A-J (2017) Radioastronomical least squares image reconstruction with iteration regularized krylov subspaces and beamforming-based prior conditioning. In: 2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP). IEEE, pp 3385–3389.https://doi.org/10.1109/ICASSP.2017.7952784
32. Joulidehsar F, Moradzadeh A, Doulati Ardejani F (2018) An improved 3d joint inversion method of potential field data using cross-gradient constraint and lsqr method. Pure Appl Geophys 175(12):4389–4409. https://doi.org/10.1007/s00024-018-1909-7
33. Liang S-X, Jiao Y-J, Fan W-X, Yang B-Z (2019) 3d inversion of magnetic data based on lsqr method and correlation coefficient self constrained. Prog Geophys 34(4):1475–1480. https://doi.org/10.6038/pg2019CC0275
34. Liang S-X, Wang Q, Jiao Y-J, Liao G-Z, Jing G (2019) Lsqr-analysis and evaluation of the potential field inversion using lsqr method. Geophys Geochem Explor 43(2):359–366. https://doi.org/10.11720/wtyht.2019.1261
35. Bin G, Wu S, Shao M, Zhou Z, Bin G (2020) Irn-mlsqr: an improved iterative reweight norm approach to the inverse problem of electrocardiography incorporating factorization-free preconditioned lsqr. J Electrocardiol 62:190–199. https://doi.org/10.1016/j.jelectrocard.2020.08.017

36. Jaffri NR, Shi L, Abrar U, Ahmad A, Yang J (2020) Electrical resistance tomographic image enhancement using mrnsd and lsqr. In: Proceedings of the 2020 5th International Conference on Multimedia Systems and Signal Processing, pp 16–20. https://doi.org/10.1145/3404716.3404722
37. Guo H, Zhao H, Yu J, He X, He X, Song X (2021) X-ray luminescence computed tomography using a hybrid proton propagation model and lasso-lsqr algorithm. J Biophotonics 14:202100089. https://doi.org/10.1002/jbio.202100089

Authors and Affiliations

Giulio Malenza[1] · Valentina Cesare[2] · Marco Aldinucci[1] · Ugo Becciani[2] · Alberto Vecchiato[3]

✉ Giulio Malenza
giulio.malenza@unito.it

Valentina Cesare
valentina.cesare@inaf.it

Marco Aldinucci
marco.aldinucci@unito.it

Ugo Becciani
ugo.becciani@inaf.it

Alberto Vecchiato
alberto.vecchiato@inaf.it

[1] Department of Computer Science, University of Turin, Corso Svizzera 185, 10149 Turin, TO, Italy

[2] Astrophysical Observatory of Catania, INAF, Via Santa Sofia 78, 95123 Catania, CT, Italy

[3] Astrophysical Observatory of Turin, INAF, Via Osservatorio 20, 10025 Pino Torinese, TO, Italy