EUMaster4HPC Challenge: Parallelizing the Conjugate Gradient Method

Daniele Ferrario [1] Luca Guffanti [1]
Ekkehard Steinmacher [2]

[1]Politecnico di Milano [2]Università della Svizzera italiana

Objective

EXPLORE AND **BENCHMARK** DIFFERENT PARALLELIZATION TECHNIQUES TO **ACCELERATE** THE EXECUTION OF THE **CONJUGATE GRADIENT (CG) ALGORITHM** ON **CPU, GPU AND FPGA** USING THE **MeluXina** SUPERCOMPUTER.

MeluXina features:

CPU CLUSTER: 573 Nodes, each with two 2.6 GHz AMD Rome CPUs, for a total of 256 HT cores and 512 GB of RAM.

GPU CLUSTER: 200 Nodes, each with two 2.35 GHz AMD Rome CPUs, for a total of 128 HT cores plus 4 NVIDIA A100-43 GPUs and 1.92 TB of local SSD.

FPGA CLUSTER: 20 Nodes, each with two 2.35 GHz AMD Rome CPUs, for at total of 128 HT cores plus 2 Intel Stratix 10MX 16GB FPGA. [1]

Mathematical Foundation - The Conjugate Gradient method

The conjugate gradient method is an **iterative solver** for a system of linear equations. Consider the system

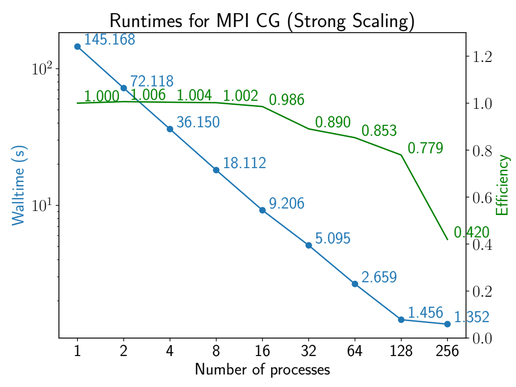
$$\mathbf{A}x = b\,, \tag{1}$$

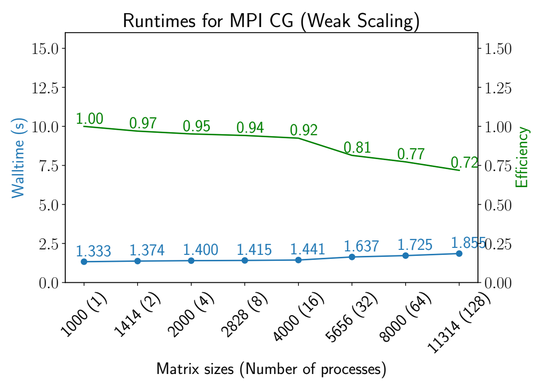
where $\mathbf{A} \in \mathbb{R}^{n\times n}$, $n \in \mathbb{N}$ is a **symmetric positive definite matrix**, and $x \in \mathbb{R}^n$ and $b \in \mathbb{R}^n$ are vectors. $\mathbf{A}$ and b are known. The system is solved for x. The Conjugate Gradient method iteratively produces better approximations of the exact solution $\tilde{x}$. It stops either when the residual $||r||_2 = ||b - \mathbf{A}x||_2 < \epsilon \in \mathbb{R}$ or when a predefined number of iterations is reached.

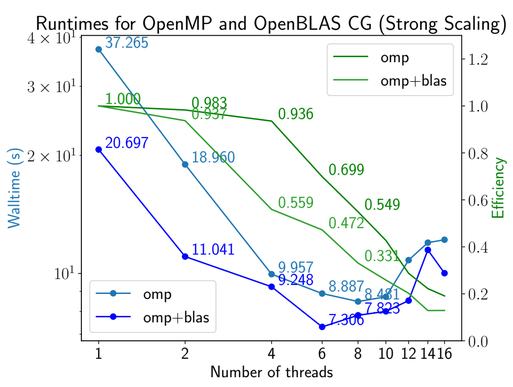
CG algorithm scheme. Here $(\cdot,\cdot)$ refers to the euclidean dot product.

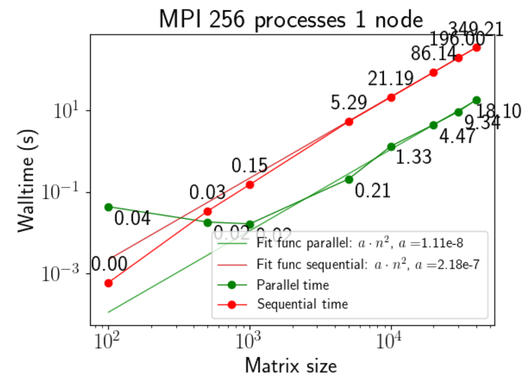
Choose x_0, compute $r_0 \leftarrow b - \mathbf{A}x_0$, set $d_0 \leftarrow r_0$ ▷ mat-vec product: Use `Allgatherv` and/or `omp parallel for`
while $||r_k||_2 > \epsilon$ **and** $k \leq max\_iters$ **do**
  Compute step size $\alpha_k \leftarrow \frac{(r_k,r_k)}{(d_k,\mathbf{A}d_k)}$ ▷ dot product: Use `omp parallel for reduction`
  Update solution $x_{k+1} \leftarrow x_k + \alpha_k d_k$ ▷ vec addition: Use `omp parallel for`
  Update residual $r_{k+1} \leftarrow r_k - \alpha_k \mathbf{A}d_k$
  Compute scalar $\beta_k \leftarrow \frac{(r_{k+1},r_{k+1})}{(r_k,r_k)}$
  Update conjugate direction $d_{k+1} \rightarrow r_{k+1} + \beta_k d_k$
  Update iteration count $k \leftarrow k+1$
end while
if converged **then** x_k approximates $\tilde{x}$[2]

OpenMP, OpenBLAS & MPI







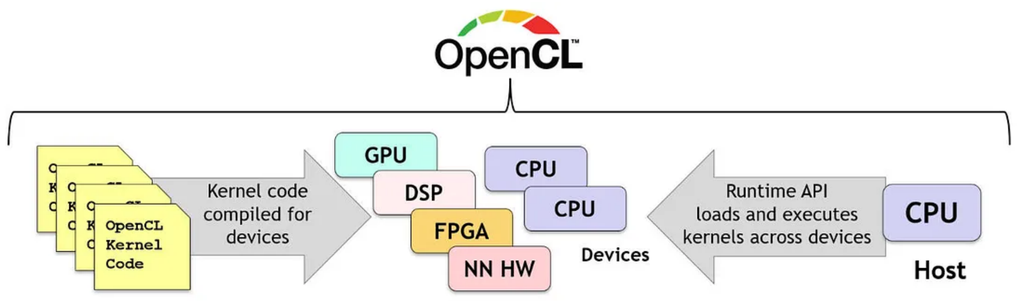


Scaling behavior: The problem size is $n = 10000$ and the tolerated error $\epsilon = 10^{-9}$.

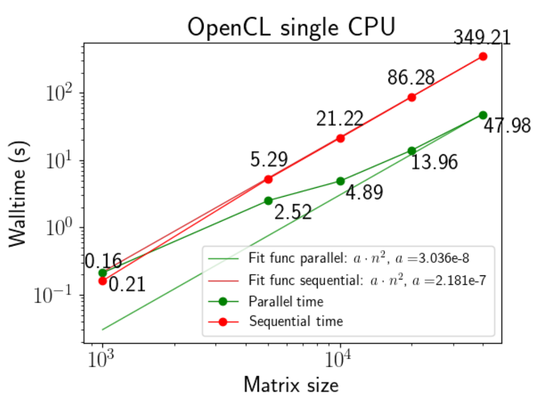
- Top left and top right: Strong and weak scaling of the MPI parallel program.
- Bottom left: Strong scaling of the OpenMP and the OpenMP+OpenBLAS parallel programs.
- Bottom right: Runtime as a function of n for the MPI parallel and sequential CG programs. The complexity of the CG problem is $\mathcal{O}(n^2)$. The average speedup of the MPI program when compared to the sequential program is $\bar{s} == \frac{a_{\text{seq}}}{a_{\text{par}}} \approx 19.64$.

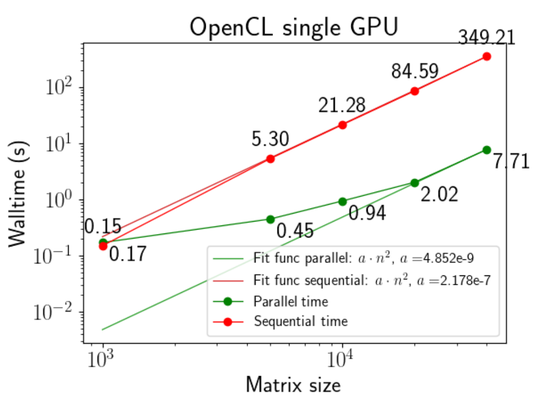
Roofline model: The operational intensity of the MPI program is: $I \approx 39.0$ FLOPs/byte. However the ridge point of the cluster is $P \approx 13.3$ FLOPs/byte. Hence $I > I$, so this program is **compute bound**.

OpenCL



OpenCL framework focuses on **heterogeneous computing**: with little to no changes to the kernels, it's possible to run the same code on CPUs, GPUs and FPGAs from different vendors.

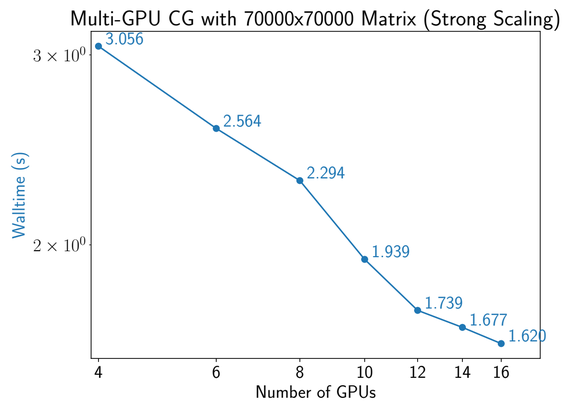


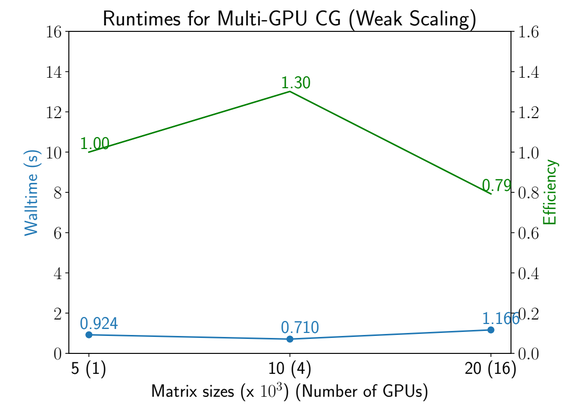


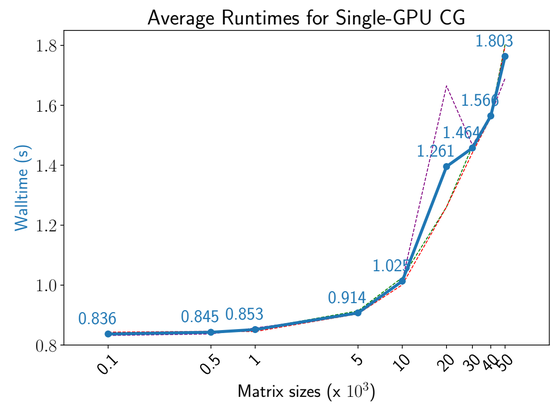
Scaling behavior: Runtime as a function of n for the OpenCL parallel program ran on either a single CPU or GPU and compared with the sequential program. The average speedup of the OpenCL program on a CPU is $\bar{s} \approx 7.17$. For OpenCL ran on a GPU it is $\bar{s} \approx 44.89$.

CUDA

GPU acceleration proved to be the best performing, with a **300x speedup** on a 70000x70000 system on a 16 GPU system with respect to a single-core execution. It's **performance** is 2.96 TFLOPs.





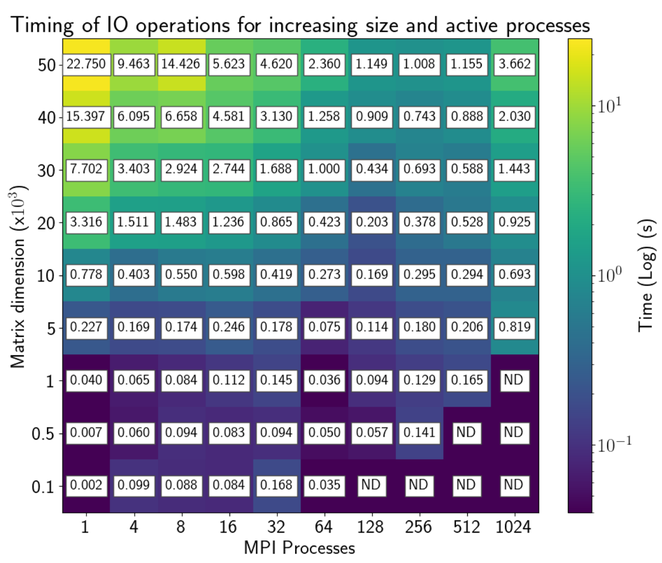


Scaling behavior:

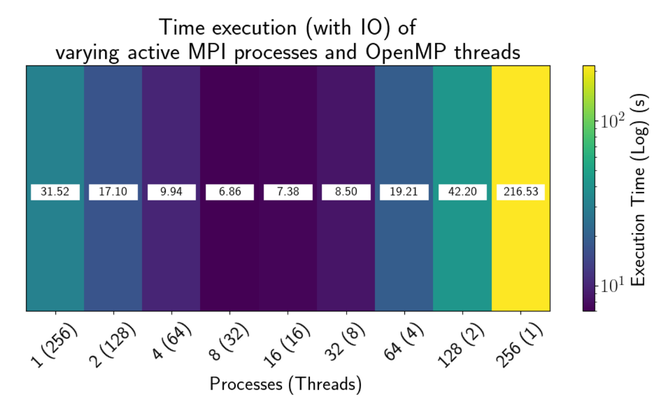
- Top left and right: Strong and weak scaling of the Multi-GPU parallel program.
- Bottom left: Runtime as a function of n.

The GPU implementations are **memory bound**.

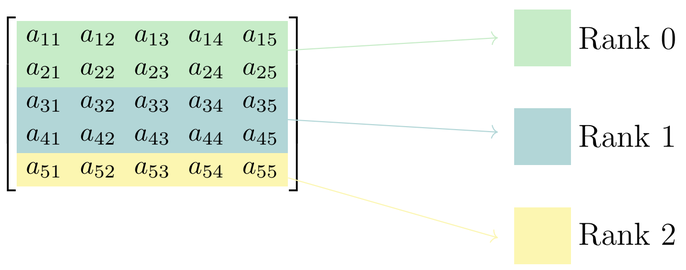
IO Optimization, Comparison & Matrix distribution



IO operations significantly **slow down** the program, and to overcome this issue we used **MPI_IO primitives for parallel file reading**. This greatly **improved** performance.



Increasing the number of processes enhances the MPI+OpenMP solver performance on one single node. Its growth, however, leads to escalated IO overhead, significantly impacting the overall execution time.



Scattering the matrix as evenly as possible between ranks proves vital to reach high speeds: the matrix-vector product, which is the most computationally-intensive task, is distributed. Vectors, on the other hand, are stored completely in each process memory.

Summary

Solver strategy	Solving time w.o. IO	Speedup
Sequential	86.14	1×
OMP	20.4503	4.21×
OpenCL CPU	13.95	6.17×
MPI-OMP 8P x 32 Th	2.83	30.48×
OpenCL Single GPU	2.02	42.65×
CUDA Single GPU	1.66	51.85×
MPI 256P	1.29	66.65×
CUDA-MPI 16 GPUs	1.228	70.14×
MPI Distributed IO 256P	0.95	90.67×

Runtime of the CG-algorithm and speedups for the explored parallelization techniques with $n = 20000$ and $\epsilon = 10^{-6}$.

Acknowledgements

The authors would like to thank Dr. Ezhilmathi Krishnasamy for his expert guidance and the *MeluXina* supercomputer team not only for their excellent support but also for the providence of the computing resources.

References

[1] (2024, Mar.) Meluxina: System overview. [Online]. Available: https://docs.lxp.lu/system/overview/

[2] (2024, Mar.) Wikipedia: Conjugate gradient method. [Online]. Available: https://en.wikipedia.org/wiki/Conjugate_gradient_method

This project has received funding from the European High-Performance Computing Joint Undertaking under grant agreement No. 101051997
EUROHPC SUMMIT 2024, Antwerp — EUMaster4HPC See more: https://github.com/Ferraah/ParallelCG
daniele6.ferrario@mail.polimi.it luca2.guffanti@mail.polimi.it ekkehard.steinmacher@usi.ch