Parallelization of the Conjugate Gradient Solver for Dense Linear Systems on the Meluxina Supercomputer

Melanie Tonarelli
Politecnico di Milano
Milan, Italy
melanie.tonarelli@mail.polimi.it

Edoardo Carrà
Politecnico di Milano
Milan, Italy
edoardo.carra@mail.polimi.it

Henrique Gil
Università della Svizzera Italiana
Lugano, Switzerland
henrique.dias.gil@usi.ch

***Abstract*—In this study, we investigate diverse parallelization techniques to improve the efficiency of the Conjugate Gradient method for solving dense linear systems. We address implementation challenges, outlining strategies to overcome them. Our implementations are rigorously tested on various node types of the MeluXina supercomputer.**

***Index Terms*—High-Performance Computing (HPC), Conjugate Gradient (CG), Performance Evaluation, Scalability, Hardware acceleration.**

I. Introduction

The Conjugate Gradient (CG) method, crucial for solving symmetric positive-definite linear systems iteratively, finds significance in scientific and engineering applications. As computational challenges grow, there's a demand for faster algorithms, prompting a shift to High-Performance Computing (HPC). Parallelizing iterative algorithms, like CG, becomes essential for significant speedup and scalability. In this context, our project aims to parallelize the CG method, enhancing its performance on modern architectures, particularly leveraging the MeluXina supercomputer. The objective is to improve efficiency and scalability, making CG suitable for high-performance computing.

The CG pseudo algorithm is displayed above:

Algorithm 1 Conjugate Gradient

Initialize $x^{(0)}$, set $r^{(0)} \leftarrow b - Ax^{(0)}$, $p^{(0)} \leftarrow r^{(0)}$
for $k = 0, 1, 2, \ldots$ until convergence **do**
  $\alpha_k \leftarrow \frac{{r^{(k)}}^T r^{(k)}}{{p^{(k)}}^T A p^{(k)}}$
  $x^{(k+1)} \leftarrow x^{(k)} + \alpha_k p^{(k)}$
  $r^{(k+1)} \leftarrow r^{(k)} - \alpha_k A p^{(k)}$
  Check for convergence
  if converged **then**
    break
  end if
  $\beta_k \leftarrow \frac{{r^{(k+1)}}^T r^{(k+1)}}{{r^{(k)}}^T r^{(k)}}$
  $p^{(k+1)} \leftarrow r^{(k+1)} + \beta_k p^{(k)}$
end for=0

II. Methodology

A. Problem Analysis

The computational efficiency of the CG method faces significant **bottlenecks**, notably in matrix-vector multiplication (GEMV) and irregular memory access patterns. Moreover, large matrices strain memory resources, leading to potential access delays and exhaustion. Consequently, distributing the matrix across multiple computing nodes becomes essential.

Recognizing these hurdles, we conducted a thorough analysis of data dependencies between operations, which served as a fundamental step in understanding and addressing inefficiencies. Moreover, the inherent complexities of distributed systems, such as communication overhead, latency, load imbalance, and global synchronization requirements, necessitated careful consideration.

B. Parallel Programming Model

In parallelizing the CG method on MeluXina, we employed various strategies to address the identified bottlenecks.

For shared-memory parallelization within a node, **OpenMP** and **CUDA** were utilized. **OpenMP** facilitated communication between thread and **CUDA**, tailored for GPUs, optimized matrix-vector multiplication and provided explicit memory control.

In the distributed-memory context, **MPI** handled communication overhead, load imbalance, and synchronization challenges. For accelerators, NCCL addressed efficient GPU communication.

Hybrid approaches combined strengths, such as **MPI + OpenMP**, **MPI + CUDA** and **NCCL + CUDA**, leveraging both shared and distributed-memory paradigms. These strategies targeted MeluXina's architecture for enhanced CG method performance.

III. Implementation Details

The project, named **Linear Algebra for MeluXina (LAM)**, is a scientific library focusing on software engineering principles. It emphasizes a balance between performance, abstraction, and usability, guided by the **zero-overhead abstraction principle**. The design employs a structured class hierarchy for

modularity, maintainability, and code consistency. An abstract base class at the top of the hierarchy serves as the interface for all implementations, defining essential methods for standardized parallelization strategies. Moreover, the design encourages easy extension by creating new classes inheriting from the base class, promoting code **reusability** and facilitating the integration and comparison of parallelization strategies. Additionally, each implementation is generic, supporting both **single and double precision** for optimization purposes.

A. OpenMP

The OpenMP implementation introduces finer-grained parallelism to key operations like `gemv`, `axpby`, and `dot` within the CG algorithm. Parallel regions strategically placed in critical sections leverage data and code dependency analysis. Notably, the approach specifically addresses the issue of **false sharing** in multi-core architectures by aligning memory allocations with thread affinity, minimizing the risk of performance degradation. For instance, the parallelization of matrix-vector multiplication (`gemv`) efficiently distributes the workload among threads by operating on blocks of matrix rows, as it demonstrated in the code snippet below.

```
#pragma omp parallel for reduction(+:result)
for(size_t i = 0; i < size; i++)
{
    result += x[i] * y[i];
}
```

Listing 1: Dot product

```
#pragma omp parallel for
for(size_t i = 0; i < size; i++)
{
    y[i] = alpha * x[i] + beta * y[i];
}
```

Listing 2: axpby

```
#pragma omp parallel for
for(size_t r = 0; r < rows; r++)
{
    FloatingType y_val = 0.0;
    for(size_t c = 0; c < cols; c++)
    {
        y_val += alpha * A[r * cols + c] * x[c];
    }
    y[r] = beta * y[r] + y_val;
}
```

Listing 3: gemv

The problem-solving process begins with a "**first-touch policy**" to parallelize the initialization of data structures, optimizing for **Non-Uniform Memory Access (NUMA)** architecture. In NUMA, memory pages can be local or remote to a CPU. The Linux default first-touch policy allocates memory based on the first write access, crucial for NUMA systems. As the OpenMP implementation uses static scheduling, parallel initialization ensures each thread uses memory local to its CPU, preventing inefficiencies across sockets. This approach can result in a significant speedup, potentially up to **5 times** compared to the base OpenMP implementation. If you don't parallelize the initialization, the memory will end up local to the main thread which will be worse for threads that are on different sockets. (Figure 1)

This approach has the potential to yield a significant speedup, up to approximately **5 times** if compared to the base OMP implementation.

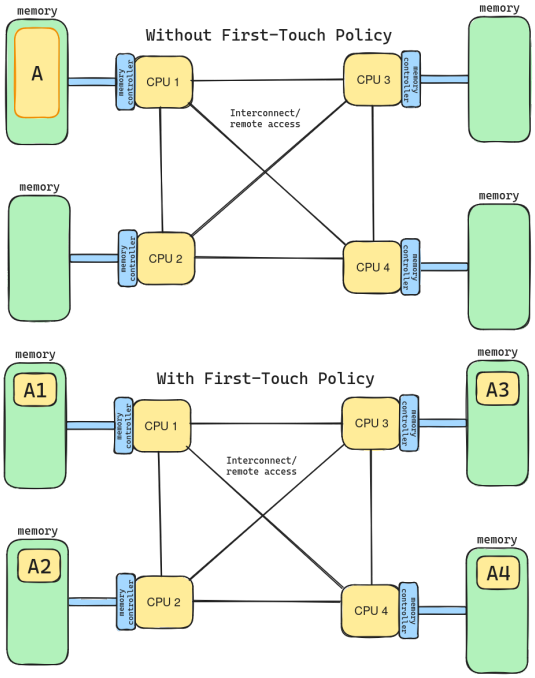

Fig. 1: First-touch policy

B. OpenMP + MPI

The MPI with OpenMP implementation integrates the Message Passing Interface (MPI) for inter-process communication with OpenMP for intra-process parallelism. This approach aims to effectively distribute computational tasks across distributed-memory systems while harnessing shared-memory parallelism within each node.

Key features of this implementation include:

- **MPI Communication**: Utilization of MPI directives facilitates seamless data exchange among distributed processes. Collective operations, such as global reduction, are employed efficiently to enable computations spanning multiple nodes. Our initial code dependency and data analysis critically guided the identification of key communication points:
  - For GEMV, each node independently performs the operation using a block of rows from the matrix, generating a portion of the overall multiplication. To synchronize the results, nodes utilize the MPI `Allgatherv` operation, ensuring the seamless exchange of portions of the vector while maintaining the necessary offset.
  - The dot product is efficiently computed by allowing each node to process a portion of the two vectors in-

dependently. Subsequently, a collective `Allreduce` operation is employed to consolidate the local results from all ranks, achieving a synchronized and accurate final result.
  - The `axpby` operation, involving the linear combination of vectors, does not necessitate any inter-process communication, as it can be executed independently on each node.
- **OpenMP Parallelism**: Intra-node parallelism is optimized using OpenMP constructs, leveraging multi-core architectures for enhanced performance.
- **Data Distribution**: The workload is evenly distributed among MPI processes, ensuring balanced execution across nodes. Each process operates on a localized portion of the matrix and vectors, minimizing communication overhead while maximizing parallelism. Each block of the matrix is loaded independently by the respective process, leveraging the **distributed file system/matrix assembler**.

C. CUDA

CUDA is NVIDIA's parallel computing platform, designed for harnessing the parallel processing power of GPUs in various applications. Firstly, let us introduce a set of general optimization techniques, consistently applied across all our CUDA implementations.

Efficient utilization of global memory and bandwidth is crucial, requiring organized and coalesced memory access aligned with **DRAM bursts**, as shown in Figure 2.

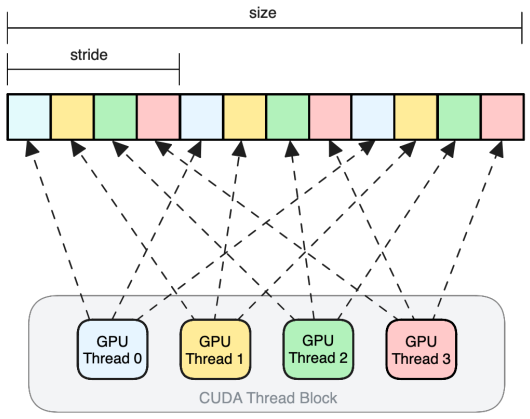

Fig. 2: Coalesced Memory Access Pattern

Furthermore, shared memory plays a pivotal role in reducing redundant global memory accesses and addressing challenges related to non-coalesced memory access. Shared memory, although beneficial, introduces concerns such as bank conflicts, which can lead to serialized access and diminish the advantages of parallelism. Adopting **bank-stride access patterns** helps mitigate these conflicts; such approach is particularly important in the implementation of sum reduction.

Minimizing CPU-GPU data transfer is another optimization, achieved by storing intermediate data directly on the GPU. Grouping matrix and vector data transfers at the outset of the computation enhances memory bandwidth utilization. For multiple GPUs, **stream** utilization offers improved parallelism and scalability by overlapping kernels and memory copy operations from different streams, effectively hiding memory latency and optimizing resource usage. Ultimately, utilizing page-locked (or "pinned") memory enables increased bandwidth between the host and the device.

We will now discuss the CUDA kernels implemented for some common operation in parallel computing and linear algebra.

In the sum reduction function, a common parallel computing operation, we employed a divide-and-conquer strategy. The input array is divided into chunks matching the CUDA thread block size. Threads concurrently calculate partial sums using a tree-based approach within each block, storing results in **shared memory**.To mitigate bank-conflicts, we have opted for sequential addressing, employing consecutive thread indices, as opposed to interleaved addressing. This process is recursively performed until a single number, the final result, remains. Additionally, we optimized by unrolling the last warp, utilizing a function designed for efficient reduction within a warp and capitalizing on its parallel processing capabilities.

The sum reduction is specifically used in both the scalar product between vectors $x, y \in \mathbb{R}^n$ and `gemv`. Since the implementation of the dot product consists of subsequent sum reductions over the array of the element-wise product $x[i] * y[i]$, we will focus on `gemv`. In handling dense matrices, our approach involves linearized storage, representing the matrix as a one-dimensional array for optimized memory access. Each block manages a matrix row independently, avoiding synchronization complexities. Threads access data with a constant stride aligning with the block size, ensuring efficient memory access. Results are stored in shared memory for subsequent sum reduction, enhancing performance. This strategy scales effectively for larger matrices, leveraging GPU parallel processing capabilities.

The vector addition operation in the CG algorithm is straightforward, and detailed analysis is unnecessary. Key considerations have been addressed previously.

In the context of **multiple GPUs** (confined to a single node), the `gemv` operation is the only parallelized one as it dominates the iteration, while other operations are executed by a single GPU, minimizing unnecessary inter-GPU communication. Therefore, The matrix is evenly divided among GPUs, each handling a designated set of rows. During `gemv`, peer-to-peer communication, which exploits **NVLink** interconnects between `gemv`, is utilized to gather results on the primary GPU. Additionally, streams are leveraged for concurrent **asynchronous** execution of memory copy operations and kernels from different streams, providing an extra layer of parallelization in the `gemv` operation.

D. MPI+CUDA

The hybrid MPI+CUDA implementation enables the distribution of computations across multiple GPUs situated on distinct nodes. This implementation leverages **CUDA-aware** support within the MPI library, enabling the direct exchange of GPU buffers through MPI directives and eliminating the need for GPU-CPU communication.

Each MPI process is associated with a specific GPU device. Analogously to the MPI+OMP implementation, each block of the matrix is loaded/assembled independently by the respective process and copied asynchronously into the device. In contrast to the MPI+OMP implementation, the only operation that is distributed among the devices is the `gemv`. This leads to a different communication pattern: while `dot` and `axpby` can be carried on the device associated to the process with rank 0, the `gemv` requires the first device to broadcast the direction p to all other devices and subsequently gather all the partial results of the `gemv`.

E. MPI+NCCL+CUDA

In our implementation, we optimize multi-GPU computations using the NVIDIA Collective Communications Library (**NCCL**) in conjunction with MPI. The latter is used for the initial setup and matrix assembly/loading, while NCCL is used for inter-GPU communication. NCCL is equipped with the capability to **discover the underlying devices' network topology**, enabling efficient utilization of the fastest path between a pair of devices, while avoiding the CPU-GPU communication. Although an **initial setup overhead** of approximately **3-4 seconds** exists, the subsequent gains in communication speed and efficiency justify this brief cost.

In a manner similar to the MPI+CUDA implementation, each MPI process is responsible for at most one GPU, and the `gemv` operation is carried out on the first rank. To minimize communication overhead, all other operations are exclusively computed by a single device within a specific node, enhancing overall performance. This approach is valuable for distributed computing across nodes and scenarios involving multiple GPUs within a node. Overall, CUDA + NCCL integration, coupled with thoughtful parallelization and communication management, enhances the effectiveness of multi-GPU computations.

IV. Experimental Setup

A system overview of MeluXina is shown in table I.

TABLE I: Compute Modules Overview

Compute Module	CPU	Accelerator	RAM
Cluster	2x AMD EPYC Rome 7H12 64c @2.6GHz	-	512GB
Accelerator - GPU	2x AMD EPYC Rome 7452 32c @2.35GHz	4x NVIDIA Ampere 40GB HBM	512GB

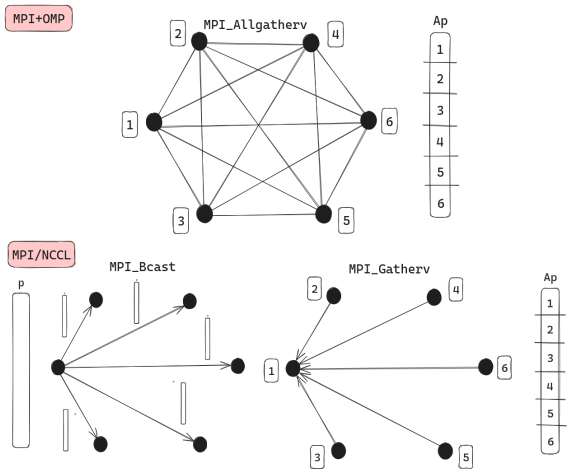

Fig. 3: Data communication comparison between MPI+OMP implementation and MPI/NCCL with CUDA implementation.

In the MPI+OMP implementation, with n as the number of processes and M as the size in Bytes of the vector, the data communication involves $\frac{n(n-1)}{2}$ bidirectional transmissions. Each transmission is of size $\frac{M}{n}$ Bytes in both directions. Contrastingly, in the MPI/NCCL with CUDA implementation, the communication requirements are simplified. This approach necessitates $2(n-1)$ unidirectional transmissions. The first $n-1$ transmissions are of size M Bytes, and the subsequent ones are of $\frac{M}{n}$ Bytes.

The depicted computing nodes are interconnected through an InfiniBand network, a high-speed interconnect technology prevalent in data centers and HPC environments. This technology facilitates low-latency and high-bandwidth communication between servers and storage systems, contributing to the overall efficiency and performance of the computing infrastructure. Additionally, the accelerator-GPU nodes feature NVLink 3 interconnects (1,555GB/s memory bandwidth) between GPUs, enhancing communication efficiency within the node.

Each implementation was tested with different matrix sizes, different numbers of MPI processes, threads and GPUs. The test both used a matrix **loaded from a file** or **generated** by the program. Loading the matrix from a file is useful for testing the solver with a specific matrix and right-hand side, while, on the other hand, generate the matrix allows testing the solver with any matrix sizes.

V. Performance Analysis

The performance analysis is organized into three key segments. Initially, we delve into the efficiency assessment of individual linear algebra operations employed within the CG method. Subsequently, we shift our focus to the program's performance when utilizing a single computing node. Lastly, we direct our attention to the performance aspects associated with computations spanning multiple nodes. This structured approach provides a comprehensive examination of the program's efficiency at different levels of computational complexity and parallelization.

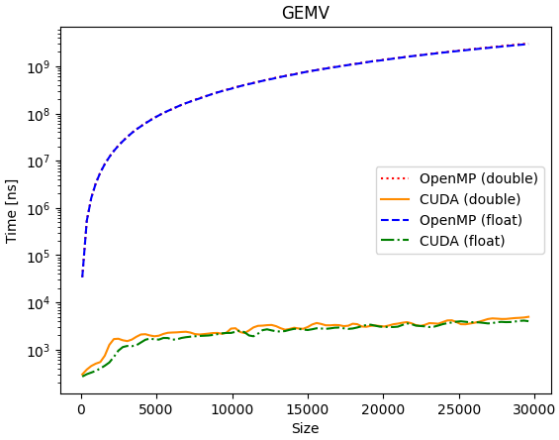

Fig. 4: Matrix-vector multiplication

A. Linear Algebra Operations

Extensive tests demonstrate the remarkable efficiency of GPU/CUDA processing in contrast to CPU/OpenMP performance, particularly evident in handling large datasets. Furthermore, it is observed that single precision computations outpace their double precision counterparts in terms of speed.

In Figure 4, we scrutinize the execution time, measured in nanoseconds, of the most resource-intensive operation throughout the entire iteration of the CG method: the matrix-vector multiplication (`gemv`). In our analysis, we evaluate performance using small-sized matrices to assess the scalability of our program, even with diminutive datasets. As anticipated, the CUDA implementation outperforms the OpenMP counterpart, affirming the expected efficiency gains associated with GPU acceleration. When working with larger-sized matrices, our evaluation is centered exclusively on the CUDA implementation, comparing the performance of a single GPU against multiple GPUs within a computing node. Despite the consistent communication overhead, the multi-GPU implementation sets itself apart by introducing finer-grain parallelism and leveraging increased memory availability for storing more data. This proves especially advantageous in handling larger matrices.

B. Single Node Results

Firstly, we conducted a comparison between the duration of a single iteration in the CG method and the duration of the GEMV operation. The results demonstrated that GEMV accounts for nearly **90%** of the total time. This empirical confirmation aligns with our initial considerations, justifying our focus on parallelizing GEMV across nodes.

Through extensive testing, we observed interesting trends in the performance of our implementations.

- The OpenMP approach exhibits remarkable speedup with small-sized matrices, showcasing its effectiveness in handling parallel tasks within a single node. Referring to table II, the OpenMP implementation exhibits different behaviors as the number of threads increases. On the one hand, the basic implementation degrades when increasing the number of threads. This is due to the complexity of the underlying NUMA topology and the non-locality of the data when using threads mapped to different NUMA nodes. On the other hand, the implementation using the First-Touch policy scales much better thanks to the **data locality**.

Matrix Size	32	64	128	256
500 FT	0.017	0.024	0.115	0.116
500	0.022	0.071	0.072	0.261
1000 FT	0.033	0.042	0.104	0.227
1000	0.043	0.097	0.373	0.421
10000 FT	7.044	3.886	2.526	2.568
10000	7.076	7.493	10.049	8.343
20000 FT	28.062	15.655	10.943	5.382
20000	28.36	29.758	33.118	40.369
30000 FT	63.237	34.066	16.764	22.653
30000	63.558	67.414	75.44	90.655
40000 FT	111.972	57.756	32.281	29.497
40000	112.323	119.151	130.917	154.645

TABLE II: Comparison of execution times (in seconds) with and without FIRST-TOUCH (FT) policy for different matrix sizes (on the left) and OMP threads

- On the other hand, employing a single GPU within a node proves highly efficient, leading to a rapid increase in speedup. This is especially evident for larger matrices, as it is shown in Table III. Nevertheless, it's crucial to consider the spatial limitations inherent in this approach, given that each GPU has 40GB in our case. Consequently, the scalability of this approach becomes constrained, posing limitations on its applicability for matrices exceeding a certain size.
- The introduction of multiple GPUs within a node leads to a decline in speedup, primarily attributed to increased communication overhead between GPUs, offsetting the gains achieved through parallelization. However, it's essential to highlight that the approach still yields improvement as the matrix size scales up. The strategy remains valuable, offering enhanced parallel processing capabilities, especially when dealing with substantial datasets. The inherent benefits of employing multiple GPUs shine through as the matrix size increases, providing ample space for data storage and computation — an aspect not achievable with a single GPU.

To provide a clear overview of these findings, we present the performance speedup in Table III.

N	OpenMP	CUDA with 1 GPU	CUDA with 4 GPUs
500	3.88	1.02	0.838
1000	9.09	35.87	4.29
5000	10.2	76.4	14.7
10000	14.01	109.25	35.8

TABLE III: Performance Speedup of CG Method for Matrices of Size NxN

C. Multiple Nodes Results

Strong scaling examines the speedup for a consistent problem size in relation to the number of processors, following **Amdahl's law**. Figure 6 illustrates the **speedup** of each implementation in comparison to its respective single-node counterpart. Notably, the speedup of OpenMP + MPI experiences a decline due to escalating **data traffic** between compute nodes. In contrast, the GPU implementation exhibits an **upward trend** in speedup.

Weak scaling evaluates the speedup for a scaled problem size in relation to the number of processors, following Gustafson's law. Figure 5 illustrates the real efficiency trend for each multi-node implementation relative to the single-node scenario. Notably, efficiency decreases with an increasing number of processors or nodes. This reduction is attributed to the growing problem size, causing elevated data transfer latency between nodes due to the constrained bandwidth of the communication network.

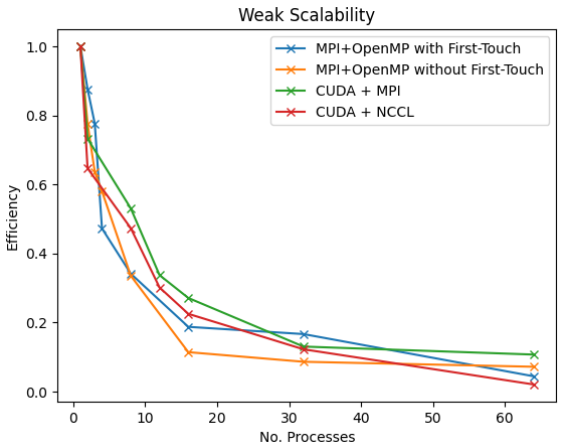

Fig. 5: Weak Scalability Analysis

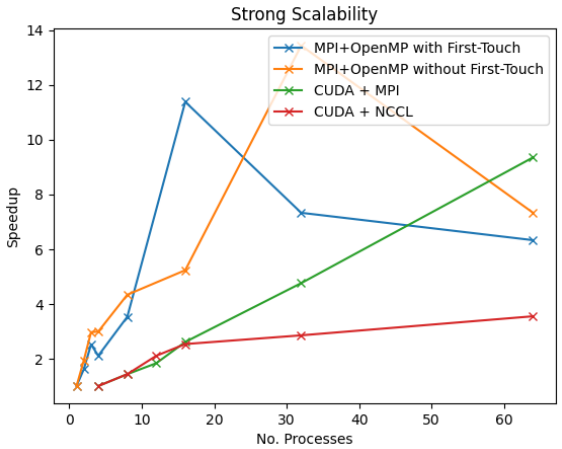

Fig. 6: Strong Scalability Analysis using a matrix of size 200000x200000

The tables IV and V emphasize the **dominance of the MPI+CUDA** implementation that consistently exhibits superior performance, showcasing its scalability.

However, it is noteworthy to observe a peculiar behavior in the MPI+NCCL implementation, as indicated in IV. The execution times suggest a deviation from the expected scalability trend. Further investigation into this unusual behavior may be warranted to identify potential factors influencing the performance of MPI+NCCL in the specified scenarios.

Finally, when compared to the average duration of a single iteration in the sequential implementation, the MPI+OMP, MPI+CUDA, and NCCL+CUDA implementations exhibit a speedup of several thousand times.

Nodes/devices	MPI+OMP	MPI+CUDA	MPI+NCCL
1	15.233	1.935	3.315
2	12.656	3.081	3.117
8	8.511	0.43	13.381
16	4.606	0.326	17.46
32	5.036	0.295	16.698

TABLE IV: Comparison of execution times (in seconds) of MPI+OMP, MPI+CUDA, and MPI+NCCL implementations with different number of nodes/GPUs, on a matrix 30000x30000

Nodes/devices	MPI+OMP	MPI+CUDA	MPI+NCCL
100000	298	979	3974
120000	397	1354	-
140000	476	1150	-
160000	397	2600	-
200000	826	4140	-

TABLE V: Comparison of the average of a single iteration of the CG (in milliseconds) on different **generated** matrix of size NxN, using 32 nodes with MPI+OMP and 32 GPUs with MPI+CUDA and NCCL+CUDA

VI. Conclusion

In conclusion, our exploration of parallelization techniques for the CG method reveals insights into enhancing its efficiency in solving dense linear systems. We focused on MeluXina's architecture, employing OpenMP, CUDA, MPI, and NCCL for shared and distributed-memory models. Hybrid approaches combining these models demonstrated the strengths of both paradigms.

Performance analyses, including scalability, highlighted efficiency and limitations. For small matrices, leveraging GPU computational power yielded notable speedup of up to **100 times**. Conversely, for larger matrices, a multi-node approach using MPI and CUDA proved more effective despite increased communication overhead, showcasing scalability.

Future improvements should target scalability challenges, addressing communication overhead and optimizing inter-node communication. Our study emphasizes tailoring parallelization strategies to the computational task, highlighting the matrix size's impact on the most suitable approach.