Parallelization of the Conjugate Gradient Method

1st Hakan Hasan
Sofia University "St. Kliment Ohridski"
Sofia, Bulgaria
hthasan@uni-sofia.bg

2nd Roberto Meroni
Universitat Politecnica de Catalunya
Barcelona, Spain
roberto.meroni@estudiantat.upc.edu

3rd Bice Marzagora
Politecnico di Milano
Milano, Italy
bice.marzagora@mail.polimi.it

***Abstract*—This project aimed to enhance the performance and scalability of the Conjugate Gradient iterative solver, by parallelizing and optimizing its execution on the MeluXina supercomputer.**

Two distinct parallelized and optimized versions has been developed. One version was crafted using CUDA to harness the power of GPU acceleration, focusing on optimizing data parallelism and memory management to improve computational efficiency on GPU architectures. The other version employed a combined approach of MPI and OpenMP, where MPI was used to manage communication between different compute nodes, and OpenMP facilitated multi-threading within each node, aiming to maximize the use of available CPU resources.

This dual-strategy approach allowed us to explore and compare the benefits of both GPU-based and CPU-based parallel computing, tailoring each version to leverage the specific architectural features of the MeluXina supercomputer.

I. CUDA

Most of the functions in the Conjugate Gradient solver involve a massive number of simple arithmetic operations that can be executed independently across data elements, making the GPU architecture and CUDA's programming model a natural choice for enhancing the performance of the algorithm.

A. Preconditioner

Preconditioning transforms the original system into a new system that is easier to solve with iterative methods and has the same solution. The objective is to reduce the condition number, so that the computation will take fewer iterations.

In our implementation, a basic preconditioning method is used (Jacobi Preconditioner), which offers a good balance between computational simplicity and effectiveness, as it does not significantly increase the computational burden while still offering benefits in terms of convergence rates. In this implementation, it reduced the number of iterations of more than 10% with little-to-none overhead.

In fact, the main steps of this preconditioning method are:

- Extracting the diagonal, where each thread performs a copy operation on a different diagonal element. This occurs only during the initialization phase.
- Apply the preconditioner, where each thread divides an element of a vector by the corresponding element of the diagonal.

Both of these functions are embarrassingly parallel and perfectly suitable to be executed on the GPU.

It has to be mentioned that more advanced preconditioning methods like *ILU* or *Incomplete Cholesky* could have been applied, and that the choice of the preferred method should take into account the structure of the matrix.

B. Initialization and copy into the GPU memory

The necessary data, including the matrix A and the vector b, are copied into the GPU memory. Most of the variables used by the solver are allocated and computed directly on the GPU, as they serve no purpose on the CPU.

Whenever possible, data transfers to GPU memory are performed asynchronously. This approach is particularly significant when copying the matrix, given its potential for large sizes.

C. CUDA Streams

Not all speed improvements are achieved by exploiting data parallelism. Some functions do not depend on the functions called previously in the sequential version and can be executed in parallel. This is managed with CUDA Streams, which provide a simple and intuitive way to describe the flow of operations and the dependencies within the algorithm.

D. AXPBY

The operation is highly parallelizable. Each thread calculates the multiplication and addition for a specific vector element.

E. GEMV

For the General Matrix-Vector multiplication (*gemv*), the function from the *cuBLAS* library is used.

This solution is very simple to implement and offers high performances.

F. Dot Product

1) First Step and copying into shared memory: To take advantage of faster memory access in shared memory within a block, the result of the element-by-element multiplication is directly stored into the shared memory.

2) First reduction (within tiles): This implementation of the dot product is different from how usually it is implemented on CUDA.

Instead of executing directly the reduction across the whole block, the blocks are divided into tiles. The reduction first occurs within each tile, so that only threads belonging to the same tile need to synchronize with each other (contrary to the usual versions where you need to synchronize across the whole block).

The finer-grain synchronization is made possible by *Cooperative Groups*, a recent feature from CUDA that allows asynchronous operations within a warp. The reduction uses sequential addressing.

During the reduction, even if each thread is storing data, only half of the threads perform the computation. There is no need to synchronize over these threads, so *coalesced_threads()* allows to select only the active threads (the threads entering the clause $if(id < stride)$).

3) Copy into thread registers: At the end of the first reduction, the first two threads of each tile are storing the results. The final step of the first reduction (sum the first two data of each tile) directly stores the result in one of the registers of the first 32 threads.

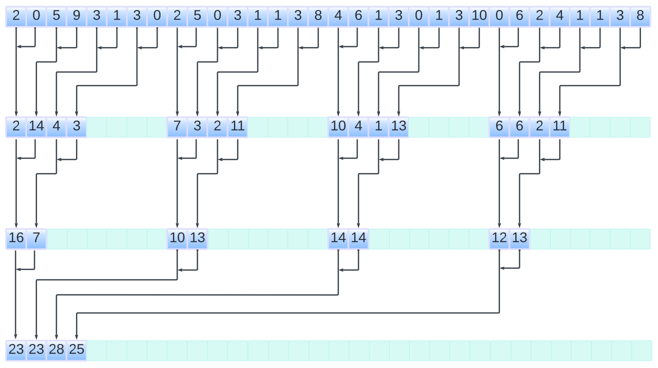


Fig. 1. Reduction within tiles (simplified)

4) Second reduction (within blocks): Now, each block has the necessary data in the registers of the first 32 threads (a warp). Within a warp, *__shfl_down_sync()* allows data exchange directly from register to register. This is faster than shared memory, which requires a load, a store and an extra register to hold the address.

The warp performs the reduction, and the result is holded by the first thread.

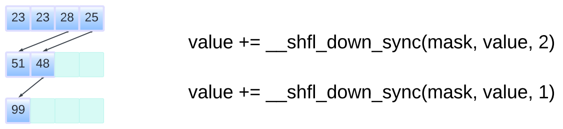

23 23 28 25
51 48
99
value += __shfl_down_sync(mask, value, 2)
value += __shfl_down_sync(mask, value, 1)

Fig. 2. Finalize reduction within a block (shuffle)

5) Final reduction within devices: Now, each block has its reducted value stored into the first thread. *atomicAdd()* sums the result of each block reduction into *d_result*, the variable allocated in global memory.

The function *atomicAdd()* is already optimized and efficiently performs reductions, which is relevant in cases of large numbers of blocks.

G. Multiple GPUs implementation (NCCL)

Each MeluXina GPU node consists of four NVIDIA A100-40 GPUs, where each device has 40GB of on-board High Bandwidth Memory.

In order to apply the algorithm on matrices of size greater than 65536, which occupies 33GB, it is necessary to use more than one GPU.

One approach could be to use MPI to divide the matrix, compute each part of the matrix on different nodes and gather the results through message passing. This would require the data to be copied from the GPU memory to the CPU memory, and then again from the CPU memory to the memory of another GPU device.

Taking advantage of the MeluXina architecture, where GPUs belonging to the same node are connected with each other through NVLink 3 (more than 7x faster than InfiniBand), it is possible to skip the CPU memory step and transmit data directly from GPU global memory to another GPU global memory within the same node. This allows to solve matrices up to size of 131072 (129GB). For matrices of larger size, it will be necessary to use more than one GPU node and add MPI communication.

This implementation uses the NVIDIA Collective Communications Library (NCCL), which provides an API conceptually very similar to MPI. Since this introduces some overheads, the multiple GPUs implementation will be used only when required ($size > 65536$), while for smaller sizes all the computation will be executed on a single GPU.

1) Memory Allocation and Initialization: Each GPU receives a subset of the matrix A through an asynchronous Memcpy, so that the data transfer can be executed in parallel with computational operations. Cases where the matrix is not exactly divisible by the number of GPUs are handled by padding.

2) Extracting A diagonal: The diagonal of the matrix is necessary to compute the preconditioner, but A is distributed across different GPUs. Each GPU identifies the diagonal elements within its subset of A and the complete diagonal is obtained by each GPU through *ncclAllGather*.

3) Main Loop: The vector p is calculated only by the first device, and is then sent to the other GPUs through *ncclBroadcast*.

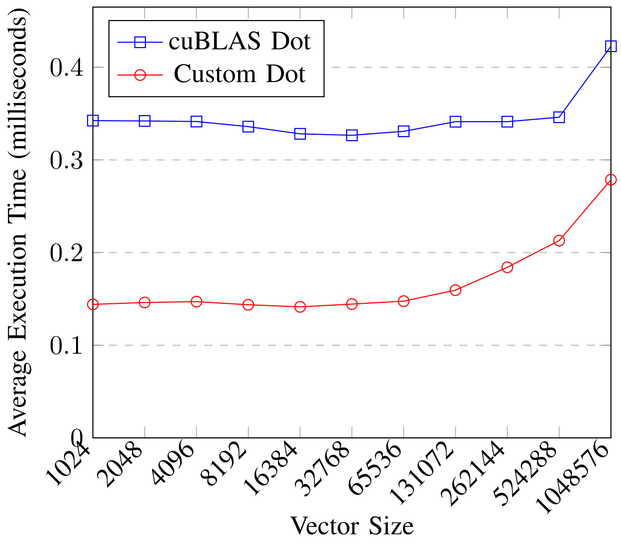
Each GPU computes *gemv* on its subset of the matrix, and the results are then gathered together and collected in each local memory.

The rest of the computations in the loop are done only by the first device. Even if this might seem a waste of resources, involving the other GPUs in this phase would only add communication overheads and not obtain any optimization in performance, since the threads of a single GPU are enough to handle all the remaining operations that can be done in parallel.

H. Results

1) cuBLAS Dot vs Custom Dot Performance: Testing the custom Dot Product function against the function from the *cuBLAS* library *cuBLASDdot()*, the custom Dot function seems to be faster.

The test has been run with 1000 randomly generated vectors, with size ranging from 1024 to 1048576 (averages are reported in the graph). Before the actual execution, both functions have been "warmed-up" with preliminary executions (without the warm-up the cuBLAS function was performing even worse).



2) Performance of conjugate_gradients function: For the smallest matrix size (1024), the parallel implementation is actually slower than the sequential. This could be due to the overhead of data transfer between the CPU and GPU or initialization overhead that does not pay off for small matrices.

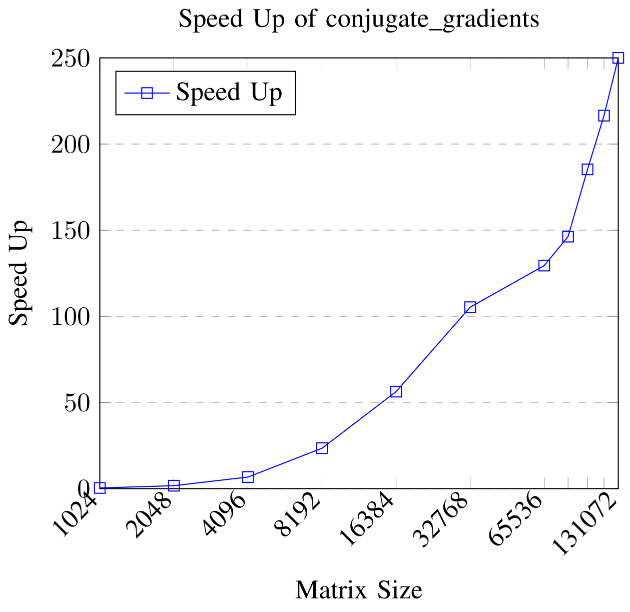
TABLE I
COMPARISON OF SEQUENTIAL AND PARALLEL PROCESSING TIMES OF THE CONJUGATE GRADIENTS SOLVER

Matrix Size	Sequential Time (s)	Parallel Time (s)	Speed Up
1024	0.317732	0.80412	0.395
2048	1.415385	0.829786	1.706
4096	5.754851	0.855679	6.725
8192	23.087489	0.984721	23.446
16384	91.245612	1.620782	56.297
32768	385.393505	3.660107	105.296
65536	1530.148246	11.818751	129.468
81920	2391.296125	16.348761	146.268
98304	3591.374915	19.382558	185.289
114688	4892.271062	22.598382	216.488
131072	7134.581354	28.538812	249.995

As the matrix size increases, the parallel implementation's efficiency becomes more pronounced. For instance, a matrix of size 8192 sees over 20 times speedup, and at size 16384, more than 50 times. This trend continues with increasing matrix sizes.

The most considerable speedups are observed in the largest matrix sizes tested. For example, for the 32768 matrix size, the solver is over 100 times faster on the GPU than on the CPU. At the extreme end, with a matrix size of 131072, the parallel version achieves a speedup of approximately 250 times, reducing the computational time from over 7134 seconds to less than 30 seconds.

This data effectively demonstrates the significant advantages of using GPU acceleration for large-scale computations in high-performance computing scenarios. The larger the problem size, the greater the benefit from parallel processing capabilities of GPUs.



II. MPI and OpenMP

In this section, a hybrid parallelization approach is employed, integrating MPI (Message Passing Interface) and OpenMP (Open Multi-Processin). This strategy is utilized to parallelize the algorithm and examine its scalability characteristic.

A. Parallelizing matrix and vector read operation

MPI was used to improve read file operations: the process begins with each MPI process identifying its unique rank and the total number of participating nodes. Each process is responsible for dynamically allocating memory sufficient for its assigned segment of the matrix, based on the calculated local number of rows and columns. This step ensures that each node is assigned a specific segment of the matrix to read and then process.

In scenarios where the matrix rows are not evenly divisible by the number of MPI processes, the function dynamically adjusts by allocating the remaining rows to the last process. The use of the fseek function to navigate to the appropriate file position for each process is pivotal for facilitating independent and non-overlapping read operations.

This implemented function successfully divides matrix reading tasks among MPI processes, it was implemented to minimizing I/O bottlenecks and improving scalability.

B. Parallelization method

The Conjugate Gradient algorithm is inherently well-matched for parallel processing of data. By employing MPI, the algorithm divides matrices and vectors into horizontal segments, assigning each MPI rank the task of executing computations relevant to its segment. This approach is complemented by leveraging OpenMP for enhanced parallelism, which applies multithreading to the for-loops responsible for matrix-vector multiplication, linear combinations of vectors, and dot product operations.

As shown in Algorithm 1, MPI and OpenMP features used are: *MPI_Allreduce* for dot products *MPI_Allgatherv* for matrix-vector product *omp parallel for* and/or *omp simd* in matrix-vector product *omp parallel for reduction* for sub-vector dot products.

C. AXPBY

The for-loop is simply marked for parallel execution using the *#pragma omp parallel for* directive, as this operation is inherently parallel and does not necessitate data communication between threads or processes.

Algorithm 1 Parallel Conjugate Gradients Method

```
1: Input: Matrix $A$ (local), vector $b$ (local), tolerance rel_error, max_iters
2: Initialize local residual $r_{\text{local}} = b$
3: Initialize local search direction $p_{\text{local}} = b$
4: $bb \leftarrow$ DOTP($b$, $b$, local_size)    ▷ Local dot product of $b$ and $b$
5: MPI_ALLREDUCE($bb$, $bb$, SUM)    ▷ Reduce $bb$ across all processes
6: $rr \leftarrow bb$
7: MPI_ALLGATHERV($p_{\text{local}}$, $p$)    ▷ Gather initial search directions from all processes
8: for num_iters = 1 to max_iters do
9:     $Ap_{\text{local}} \leftarrow$ GEMVP($A$, $p$, local_size)    ▷ Local matrix-vector product
10:    $tmp \leftarrow$ DOTP($p_{\text{local}}$, $Ap_{\text{local}}$, local_size)    ▷ Local dot product
11:    MPI_ALLREDUCE($tmp$, $tmp$, SUM)    ▷ Reduce dot product result across all processes
12:    $alpha \leftarrow rr/tmp$
13:    $x \leftarrow x + \alpha p_{\text{local}}$    ▷ Update solution vector
14:    $r_{\text{local}} \leftarrow r_{\text{local}} - \alpha Ap_{\text{local}}$    ▷ Update local residual
15:    $tmp \leftarrow$ DOTP($r_{\text{local}}$, $r_{\text{local}}$, local_size)    ▷ Local dot product
16:    MPI_ALLREDUCE($tmp$, $tmp$, SUM)    ▷ Reduce new residual norm across all processes
17:    $rr_{\text{new}} \leftarrow tmp$
18:    if $\sqrt{rr_{\text{new}}/bb} <$ rel_error then
19:        break
20:    $beta \leftarrow rr_{\text{new}}/rr$
21:    $p_{\text{local}} \leftarrow r_{\text{local}} + \beta p_{\text{local}}$ ▷ Update local search direction
22:    $rr \leftarrow rr_{\text{new}}$
23:    MPI_ALLGATHERV($p_{\text{local}}$, $p$)    ▷ Gather updated search directions from all processes
```

D. GEMV

For the General Matrix-Vector multiplication (gemv) OpenMP directives are utilized. The *#pragma omp parallel for* directive spreads the computation across multiple threads, with each handling a row of the matrix. This capitalizes on the independent nature of row computations, aligning with data parallelism principles in matrix operations. The inner loop leverages the *#pragma omp simd reduction(+:y_val)* directive for vectorization, employing SIMD instructions to speed up the dot product calculations.

E. Dot Product

The computation of the dot product is parallelized using OpenMP, where the *reduction(+:sub_prod)* clause ensures that each thread's partial sum of sub-products is correctly reduced into the *sub_prod* variable.

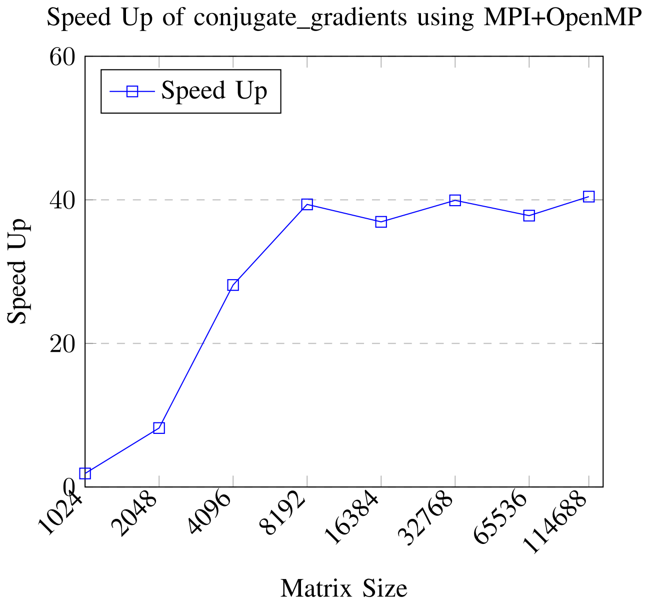
After the parallel region, the function employs MPI's *MPI_Allreduce* function to aggregate the sub-products from all threads into the *result* variable across all processes. This approach combines the shared-memory parallelism of OpenMP with the distributed-memory capabilities of MPI, allowing for efficient computation of dot products.

F. Results

The performance of the algorithm is analyzed across a range of matrix sizes from 1024 to 114688, using a setup with 64 tasks and 4 CPUs per task.

TABLE II
COMPARISON OF SEQUENTIAL AND PARALLEL PROCESSING TIMES

Matrix Size	Sequential Time (s)	Parallel Time (s)	Speed Up
1024	0.317732	0.168662	1.883
2048	1.415385	0.172340	8.213
4096	5.754851	0.197508	28.137
8192	23.087489	0.586480	39.366
16384	91.245612	2.470454	36.934
32768	385.393505	9.651943	39.929
65536	1530.148246	40.479755	37.801
114688	4892.271062	120.943841	40.450



The trend encapsulated in Table II indicates that the algorithm scales well with increasing matrix sizes, with the speed-up generally increasing as the matrix size increases. It is observed that beyond a matrix size of about 8000, the speed-up factor tends to plateau. This phenomenon can be attributed to inherent characteristics of parallel computing environments and the specific challenges associated with scaling such algorithms. Amdahl's Law offers a theoretical foundation for this behavior, suggesting that the serial portion of the algorithm inherently limits the overall speed-up achievable through parallelization.