Vectorized MPI Reduction Optimization: enhancing performance through AVX

Antonio Genovese, Simone Giglio, Davide Iovino
Department of Computer Science
University of Salerno
Italy

***Abstract*—This paper implements a novel approach to distributed computing using MPI reduction combined with vectorization techniques. In high-performance computing, MPI reduction is a commonly used operation to efficiently aggregate data across multiple processes. However, traditional implementations of MPI reduction can suffer from performance limitations due to the sequential nature of the operation. In this work, we propose a vectorization-based approach to enhance the efficiency of MPI reduction by leveraging the capabilities of modern SIMD (Single Instruction, Multiple Data) architectures.**

We first introduce a vectorized implementation of the MPI reduction algorithm, where the reduction operation is performed in parallel across multiple data elements within each process. This allows us to exploit the data parallelism inherent in SIMD architectures, achieving a speedup compared to traditional sequential reduction implementations. We also investigate the impact of different vectorization strategies on the overall performance of the algorithm, considering factors such as vector length and alignment requirements.

To evaluate the effectiveness of our approach, we conduct a series of experiments on a parallel computing system with multiple nodes. Our results demonstrate that the vectorized MPI reduction outperforms traditional approaches in terms of both computation time and scalability. Furthermore, we analyze the trade-offs between computational efficiency and memory usage, highlighting the importance of optimizing vectorization for specific hardware configurations.

I. Introduction

Motivation. Parallel and distributed computing has become essential for addressing the computational challenges posed by large-scale scientific simulations, data analytics, and machine learning applications. Message Passing Interface (MPI) is a widely adopted standard for implementing distributed algorithms, allowing efficient communication and coordination among multiple processes. One common operation is MPI reduction, which combines data from multiple processes to produce a single result. However, the performance of traditional MPI reduction implementations can be limited due to the sequential nature of the operation. Therefore, there is a need for innovative approaches that can enhance the efficiency of MPI reduction and leverage the capabilities of modern hardware architectures. So, in this paper we will implement [1] an extension to the MPI_Reduce that brings an array in input for each MPI process and produces a single result using the vectorization,

Related work. Several studies have focused on optimizing MPI reduction to improve its performance. One common approach is to use parallel algorithms that exploit the underlying parallelism of the reduction operation. For example, tree-based algorithms, such as binomial and logarithmic reduction, have been proposed to reduce the communication overhead and exploit parallelism. These algorithms partition the processes into a hierarchy and perform the reduction operation in a structured manner. In the last release of OpenMPI (v4.1), AVX support was added for MPI collective reduction operations [2].

II. Background: MPI Reduction

MPI. MPI is a standard that includes point-to-point message-passing, collective communications, group and communicator concepts, process topologies, environmental management, process creation and management, one-sided communications, extended collective operations, external interfaces, I/O, some miscellaneous topics, and multiple tool interfaces [3]. It follows a message-passing model, where processes communicate by sending and receiving messages allowing programmers to express parallelism by dividing a problem into smaller tasks that can be executed concurrently, leveraging the power of multiple processors or computing nodes. Each process works on a subset of the data and can exchange information with other processes through point-to-point communication or collective operations.

Reduction. Reduction operations play a fundamental role in parallel computing, enabling the aggregation of data from multiple processes or threads into a single result. In various scientific, engineering, and data analysis applications, reduction operations are employed to compute global quantities such as sums, products, minimum or maximum values, or even logical operations across distributed data. Traditionally, reduction operations have been implemented using sequential algorithms, where the root process sequentially collects and combines the contributions from other processes. While this approach ensures correctness, it can limit the scalability and performance of parallel computations, particularly when dealing with large-scale datasets or a high number of processes.

SIMD and AVX. Vectorization (Single Instruction Multiple Data) is a technique that aims to exploit data parallelism by performing multiple computations simultaneously on vectorized data. Modern processors support SIMD instructions, which enable the execution of the same operation on multiple data elements in parallel. Those instructions can significantly

accelerate computations, as they reduce the number of cycles needed to process amounts of data.

One technologies that implements the vectorization in modern processors is Intel Advanced Vector Extensions (AVX), an extension to the x86 instruction set architecture that introduces wider vector registers and new instructions to perform SIMD operations [4].

AVX provides 256-bit wide vector registers, allowing for the simultaneous processing of 8 single-precision floating-point numbers or 4 double-precision floating-point numbers. This increased vector width enables higher computational throughput and improved parallelism, making it well-suited for data-intensive operations in parallel computing.

III. Proposed Method

Overview. In this chapter, we present our proposed method for enhancing the efficiency of MPI reduction through vectorization techniques. Our approach aims to leverage the capabilities of modern SIMD architectures to achieve higher computational throughput and improved performance in parallel computing environments.

Vectorization-Based MPI Reduction Algorithm. We implemented the solution proposed by Zhong et al. [1] but with a different use of intrinsic for the reduction implementation. Instead of the use of AVX-512 in our implementation we were limited to the AVX2 instruction set due to hardware lackness.

Reducing with AVX-512 pseudo code.

```
procedure ReductionOp(in_buf,inout_buf,count,
    type)
   types_per_step = vector_length / (8 x
       sizeof_type)
   #pragma unroll
   for k = 0 to count with increment of
       types_per_step do
      _mm512_loadu from in_buf + offset
      _mm512_loadu from inout_buf + offset
      _mm512_reduction_op
      _mm512_storeu to inout_buf + offset
      Update left_over and offset
  if ( left_over != 0 ) then
     Update types_per_step >>= 1
     if ( types_per_step <= left_over) then
         _mm256_loadu from in_buf + offset
         _mm256_loadu from inout_buf + offset
         _mm256_reduction_op
         _mm256_storeu to inout_buf + offset
         Update left_over and offset
  if ( left_over != 0 ) then
     Update types_per_step >>= 1
     if ( types_per_step <= left_over) then
         _mm128_loadu from in_buf + offset
         _mm128_loadu from inout_buf + offset
         _mm128_reduction_op
         _mm128_storeu to inout_buf + offset
         Update left_over and offset
  if (left_over != 0 ) then
     while ( left_over != 0 ) do
         Set case_value
         Switch(case_value) : {8 Cases}
         Update left_over
```

Starting from this pseudo-code we have produced our version in AVX2. Therefore, after scattering the data across all the available processors the algorithm continues reducing the array until its size is of 8 elements, following the main idea of the Zhong et al. propose. Then we gather all the data from the processors in order to have the result in the root one.

Our reduction with AVX2 pseudocode. The full implementation is visible at: https://github.com/sgiglio11/HPC_Submission

```
procedure reduce_array(array, length,
    reduction_operator) {
 bool isVectorsEven = false;

 for(; length != 8; length >>= 1){
  if(isVectorsEven)
   length += 4;

  isVectorsEven = false;

  #pragma omp unroll
  for (int i=0, j=0; i<length; i+=16, j+=8){
   if(i+8 < length) {
    __m256 vect1 = _mm256_loadu(array+i);
    __m256 vect2 = _mm256_loadu(array+(i+8));

    __m256 result;

    //case 0 add, case 1 mul, case 2 min, case
        3 max
    switch(reduction_operator){
     case 0:result=_mm256_add_ps(vect1,vect2);
     break;
     case 1:result=_mm256_mul_ps(vect1,vect2);
     break;
     case 2:result=_mm256_min_ps(vect1,vect2);
     break;
     case 3:result=_mm256_max_ps(vect1,vect2);
     break;
    }

    _mm256_storeu(array + j, result);
   } else {
    move(array + j, array+i, 8 * sizeof(float)
        );
    isVectorsEven = true;
   }
  }
 }
 return reduce_arr_to_flt(array, 8,
     reduction_operator);
}
```

Moreover in this implementation we have considered factors such as vector length and alignment requirements producing aligned and unaligned versions where the main difference is how the allocation of data is done. The difference between the pseudo-code showed and the not one it's only about the procedures used to load and store the data.

Implementation Details. Just pay attention to when the numbers of vectors into the array are odd, in that case we need to manage the last one until we can put beside of it another one to reduce them. Moreover they use the #pragma

unroll clause that enables the loop unrolling. It replicates the body of a loop a number of times and adjusts the loop control accordingly. For the execution we used OpenMPI library into C language. Moreover, for the use of pragma clauses we have also used OpenMP while about the vectorization we have used Intel Intrinsics that allows us to use C-style functions that provide access to other instructions without writing assembly code.

To conclude, the final version of our implementations uses the showed algorithms in combination with the MPI_Gather to process the final data and obtain a single number in output.

Processing final data with MPI_Gather pseudocode.

```
void reduce_array_mpi_vec(array, length,
    reduction_operator, my_rank, nproc,
    root_results, own_result) {
    own_result = reduce_array(array, length,
        reduction_operator);
    MPI_Gather(
        own_result,
        1,
        MPI_Type,
        root_results,
        1,
        MPI_Type,
        0,
        MPI_COMM_WORLD
    );

    if(my_rank == 0) {
        own_result = 0.0;
        own_result = reduce_arr_to_flt(
            root_results, nproc,
            reduction_operator);
    }
}
```

IV. Experimental Results

Experimental setup. For our benchmarks we have used one node composed by 12 vCPU Intel (R) Core (TM) i7-8750H CPU @ 2.20 GHz and 16GB of RAM SODIMM 2400MHz. So, due to the limitation of out hardware set we will see a decrement into the performances when we will use 16 MPI processes that must be switched across the available vCPU; this operation means a waste of time in our case.

For our benchmarks we have used the directives of the paper *Reproducible MPI Benchmarking Is Still Not As Easy As You Think* [5] in order to have an accurate estimate of our performances. We have compared the sequential version to the standard one which uses only the MPI standards procedures, It runs 75 times the execution and then computes the arithmetic mean. Lastly, We have chosen a number quite big in order to obtain execution times around one second.

Strong scalability. Strong scaling concerns the speedup for a fixed problem size with variation of the number of processors, and it is governed by Amdahl's law. For our experiments we have chosen to run the strong scalability experiments' with an array size of $2^{29} = 536870912$ while the number of processors changes from 2^1 to 2^4.

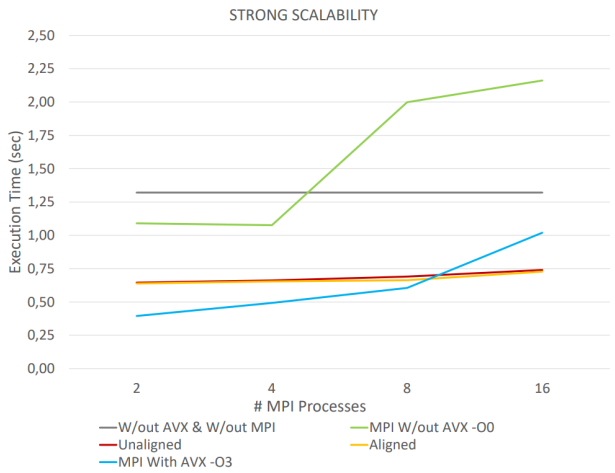

Fig. 1. Strong Scalability (execution time) with fixed size of $2^{29} = 536870912$.

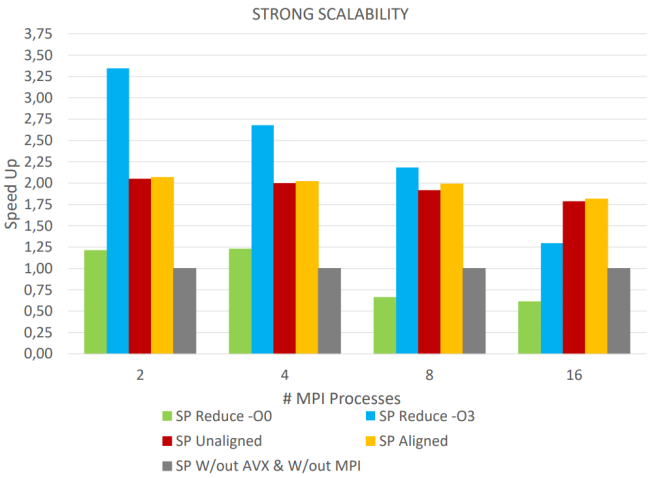

Fig. 2. Strong Scalability (speed up) with fixed size of $2^{29} = 536870912$.

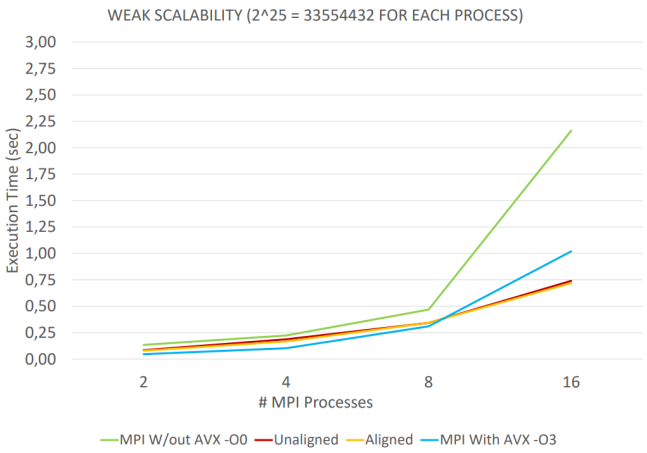

Fig. 3. Weak Scalability with fixed size of $2^{25} = 33554432$ for each process.

Weak scalability. Weak scaling concerns the speedup for a scaled problem size respect to the number of processors, and it is governed by Gustafson's law. In that case we have fixed $2^{25} = 33554432$ about the array size for each process used.

Comments. As we can see from the Fig. 1, we have an improvement using our reduction compared to the MPI one, without compiler optimizations (-O0). This behavior is justified by the SIMD improvement that we obtained using Vectorization due to the multiple arithmetic operations done for a single clock cycle. On the other hand if we compare the optimized one (-O3) to the our version we can see that the compiler optimizations achieve higher perfomances, due to the use of vectorization in combination with other optimizations, like loop unrolling and interchange, code transformations, and so on... Moreover, seeing the Strong Scalability (speed up) from the Fig. 2, we notice that the optimized version of MPI_Reduce (-O3) drops in performance increasing the number of MPI processes, furthermore this one achieve an improvement compared to the not optimized one, without AVX and MPI. Whereas, about the different versions of the vectorization we can't see a lot of difference, maybe to the minimum influence of the array alignment compared to the message passing effort. So, we can conclude that the vectorization is a lot important for the optimization of this use case because it represents the biggest part of the speedup obtained across these different versions presented in this paper.

V. Conclusions

In conclusion, this study has examined the application of vectorization in the implementation of a new reduction operation in MPI. Through our research, we have demonstrated the potential benefits of utilizing vectorization techniques to enhance the performance and efficiency of reduction operations in MPI-based parallel computing. The results indicate that by leveraging vectorization, we can achieve significant improvements in both computation time and resource utilization. This optimization approach enables more efficient data processing and communication, leading to faster execution times and enhanced scalability for parallel applications that heavily rely on reduction operations. Moreover, our findings highlight the importance of carefully designing and implementing vectorized reduction algorithms to fully exploit the computational capabilities of modern processors and parallel computing architectures. By effectively utilizing SIMD (Single Instruction, Multiple Data) instructions and data-level parallelism, we can leverage the inherent parallelism within reduction operations to achieve higher throughput and increased computational efficiency. Link to github repository: https://github.com/sgiglio11/HPC_Submission

References

[1] Dong Zhong, Qinglei Cao, George Bosilca, Jack Dongarra. *Using long vector extensions for MPI reductions.* 2022.

[2] OpenMPI. *Major User-Noticeable Changes Across Major Release Series* URL: https://www.open-mpi.org/software/ompi/major-changes.php

[3] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 4.0* URL: https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf 2021.

[4] Intel. *Intel® Intrinsics Guide v3.6.6* URL: https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html

[5] Sascha Hunold, Alexandra Carpen-Amarie. *Reproducible MPI Benchmarking Is Still Not As Easy As You Think* URL: http://hunoldscience.net/paper/mpi_measure_sahu_2016.pdf