High Performance Linear Algebra

Lecture 10: Distributed BLAS

Ph.D. program in High Performance Scientific Computing

Fabio Durastante **Pasqua D'Ambra** **Salvatore Filippone**

January 22, 2026 — 16.00:18.00

Dipartimento
di Matematica
Università di Pisa

Last time on High Performance Linear Algebra

Up to now we have seen:

- How to implement basic MPI programs in Fortran,
- Environement setup for MPI development,
- Point-to-point and collective communication,
- How to distribute vectors across MPI ranks,

We are now ready to implement distributed linear algebra routines, we will start from the Level-1 BLAS.

Table of Contents

2 Distributed Level-1 BLAS

Last time we had implemented a distributed vector type

We have constructed the following **distributed vector type**:

```
type :: mpi_ddistributed_vector
  integer :: n_local      ! number of local elements
  integer :: n_global     ! total number of global elements
  integer :: comm         ! MPI communicator
  real(real64), allocatable :: data(:)  ! local data array
contains
  procedure, pass(this) :: dinit
  final :: dfinalize
end type mpi_ddistributed_vector
```

We have now to **enrich this type** with the Level-1 BLAS operations.

Level 1: the scaling operation

The first Level-1 BLAS operation we are going to implement is the **vector scaling**:

$$\mathbf{y} \leftarrow \alpha\,\mathbf{y}, \quad \alpha \in \mathbb{R}, \quad \mathbf{y} \in \mathbb{R}^N.$$

This operation is **embarrassingly parallel**, since each element of the vector can be scaled independently from the others.

We can therefore implement this operation as a **method** of the distributed vector type, as before we add the method signature to the type definition:

```
type :: mpi_ddistributed_vector
  ...
contains
  ...
  procedure, pass(this) :: dscal_dist
end type mpi_ddistributed_vector
```

Level 1: the scaling operation

Then we can implement the method using the BLAS `dscal` routine on the local data:

```
subroutine dscal_dist(this, alpha)
  implicit none
  class(mpi_ddistributed_vector), intent(inout) :: this
  real(real64), intent(in) :: alpha
  call dscal(this%n_local, alpha, this%data, 1)
end subroutine dscal_dist
```

We remind that the `dscal` routine has the **following signature**:

```
dscal(n, alpha, x, incx)
```

where `n` is the number of elements to scale, `alpha` is the scaling factor,

Compiling and linking the distributed BLAS library

To compile and link the distributed BLAS library we need to **link against both the MPI library and the BLAS library**. The compilation command is:

```
mpifort -c -I<path_to_blas_include> mpi_ddistributed_vector.f90 -o
↪  mpi_ddistributed_vector.o
```

and the linking command is:

```
mpifort mpi_ddistributed_vector.o -L<path_to_blas_lib>
↪  -l<blas_library_name> -o libmpi_ddistributed_blas.a
```

As usual, it is convenient to create a `Makefile` or even better a `CMakeLists.txt` to automate the compilation and linking process. For the latter we need to look for both MPI and BLAS using the `find_package()` command.

Compiling and linking the distributed BLAS library

This is an example of a `CMakeLists.txt` file to compile and link the distributed BLAS library:

```
cmake_minimum_required(VERSION 3.10)
project(mpi_ddistributed_blas LANGUAGES Fortran)
find_package(MPI REQUIRED COMPONENTS Fortran)
find_package(BLAS REQUIRED)

add_library(mpi_ddistributed_blas mpi_ddistributed_vector.f90)
target_link_libraries(mpi_ddistributed_blas MPI::MPI_Fortran
↪  BLAS::BLAS)
```

Level 1: the axpy operation

The second Level-1 BLAS operation we are going to implement is the **axpy operation**:

$$\mathbf{y} \leftarrow \alpha \mathbf{x} + \mathbf{y}, \quad \alpha \in \mathbb{R}, \quad \mathbf{x}, \mathbf{y} \in \mathbb{R}^N.$$

This operation is again **embarrassingly parallel**, since each element of the vectors can be updated independently from the others.
The implementation pathway is the same as before, we add the method signature to the type definition:

```
type :: mpi_ddistributed_vector
  ...
contains
  ...
  procedure, pass(this) :: daxpy_dist
end type mpi_ddistributed_vector
```

Level 1: the axpy operation

Then we can implement the method using the BLAS `daxpy` routine on the local data, but before we have **two checks** to perform:

- Check that the vector `x` is distributed over the same communicator as `this`,
- Check that the local sizes of the two vectors are the same.

The communicator check can be performed using the `MPI_Comm_compare` routine:

```
MPI_Comm_compare(comm1, comm2, result, ierr)
```

and then check that `result == MPI_IDENT`.

The local size check is simply an `if` statement on the `n_local` attribute of the two vectors.

Level 1: the axpy operation


```
subroutine daxpy_dist(this, alpha, x)
  implicit none
  class(mpi_ddistributed_vector), intent(inout) :: this
  real(real64), intent(in) :: alpha
  class(mpi_ddistributed_vector), intent(in) :: x
  integer :: ierr, are_comm_compatible
  if (this%n_local /= x%n_local) then
      write(error_unit, *) "Error: Local sizes do not match in daxpy_dist"
      call MPI_Abort(this%comm, 1, ierr)
  end if
  call MPI_Comm_compare(this%comm, x%comm, are_comm_compatible, ierr)
  if (are_comm_compatible /= MPI_IDENT) then
      write(error_unit, *) "Error: Communicators do not match in daxpy_dist"
      call MPI_Abort(this%comm, 1, ierr)
  end if
  call daxpy(this%n_local, alpha, x%data, 1, this%data, 1)
end subroutine daxpy_dist
```

Level 1: the scalar product

The third Level-1 BLAS operation we are going to implement is the **scalar product**:

$$\alpha = \mathbf{x}^\top \mathbf{y} = \sum_{i=1}^{N} x_i y_i, \quad \mathbf{x}, \mathbf{y} \in \mathbb{R}^N.$$

This operation is **not embarrassingly parallel**, since we need to **reduce** the contributions from all the processes to compute the final result.

We can therefore implement this operation as a **method** of the distributed vector type, as before we add the method signature to the type definition:

```
type :: mpi_ddistributed_vector
  ...
contains
  ...
  procedure, pass(this) :: ddot_dist
end type mpi_ddistributed_vector
```

Level 1: the scalar product

Then we can implement the method using the BLAS `ddot` routine on the local data, but before we have **two checks** to perform:

- Check that the vector `x` is distributed over the same communicator as `this`,
- Check that the local sizes of the two vectors are the same.

And we have to perform a **global reduction** of the local contributions to compute the final result, but we have a decision to make:

- ? Do we want to return the result to all processes?
- ? Do we want to return the result only to the root process?

Level 1: the scalar product

The prototype to **delegate** the choice of the reduction strategy to the user is:

```
subroutine ddot_dist(this, x, result, rank)
  implicit none
  class(mpi_ddistributed_vector), intent(in) :: this
  class(mpi_ddistributed_vector), intent(in) :: x
  integer, intent(in), optional :: rank
  real(real64), intent(out) :: result
end subroutine ddot_dist
```

If the user provides the `rank` argument, then the result is returned only to the specified rank, otherwise it is returned to all ranks.
We can then implement the method using the BLAS `ddot` routine on the local data and then performing the reduction using `MPI_Reduce` or `MPI_Allreduce`.

Level 1: the scalar product

The two checks are the same as before, and we don't rewrite them here, then we compute the local dot product:

```
local_dot = ddot(this%n_local, this%data, 1, x%data, 1)
```

and then we perform the reduction:

```
if (present(rank)) then
    call MPI_Reduce(local_dot, result, 1, MPI_REAL8, MPI_SUM, rank,
    ↪  this%comm, ierr)
else
    call MPI_Allreduce(local_dot, result, 1, MPI_REAL8, MPI_SUM,
    ↪  this%comm, ierr)
end if
```

Level 1: the norm operation

The fourth Level-1 BLAS operation we are going to implement is the **vector norm**:

$$\|\mathbf{x}\|_2 = \sqrt{\mathbf{x}^\top \mathbf{x}} = \left(\sum_{i=1}^{N} x_i^2 \right)^{1/2}, \quad \mathbf{x} \in \mathbb{R}^N.$$

This operation is similar to the scalar product, since we need to **reduce** the contributions from all the processes to compute the final result.

We can therefore implement this operation as a **method** of the distributed vector type, as before we add the method signature to the type definition:

```
type :: mpi_ddistributed_vector
  ...
contains
  ...
  procedure, pass(this) :: dnrm2_dist
end type mpi_ddistributed_vector
```

Level 1: the norm operation

Then we can implement the method using the BLAS `dnrm2` routine on the local data, and then performing a global reduction of the local contributions to compute the final result, using the same strategy as before.

The prototype of the method is:

```
subroutine dnrm2_dist(this, result, rank)
  implicit none
  class(mpi_ddistributed_vector), intent(in) :: this
  integer, intent(in), optional :: rank
  real(real64), intent(out) :: result
end subroutine dnrm2_dist
```

Level 1: the norm operation

The local norm computation is performed using the `dnrm2` routine:

```
local_norm = dnrm2(this%n_local, this%data, 1)**2
```

and then we **perform the reduction**:

```
if (present(rank)) then
    call MPI_Reduce(local_norm, result, 1, MPI_REAL8, MPI_SUM, rank,
    ↪  this%comm, ierr)
else
    call MPI_Allreduce(local_norm, result, 1, MPI_REAL8, MPI_SUM,
    ↪  this%comm, ierr)
end if
```

Finally we take the square root of the result to obtain the final norm, if we are on the root process (or on all processes if no root was specified).

Level 1: the norm operation

The final step is to take the square root of the result:

```
if (present(rank)) then
  if (myrank == rank) then
    result = sqrt(global_sum)
  end if
else
  result = sqrt(global_sum)
end if
```

With this we have completed the implementation of the Level-1 BLAS operations for our distributed vector type.

Performance considerations for Level-1 BLAS

The Level-1 BLAS operations we have implemented are all **memory-bound**, since they require a lot of data movement compared to the number of floating-point operations performed.

In a distributed memory setting, the performance of these operations is further limited by the **communication overhead** introduced by the MPI routines used for data distribution and reduction.

Let us try to analyze the performance of the

- axpy,
- dot product,
- norm.

What models can we use?

Distributed axpy performance model

The distributed axpy operation requires:

- Reading the local parts of vectors $\mathbf{x}$ and $\mathbf{y}$ from memory,
- Writing the updated local part of vector $\mathbf{y}$ to memory.

Therefore, the total data movement is:

$$\text{Data Movement} = 2 \cdot n_{\text{local}} \cdot \text{sizeof(real64)}.$$

The number of floating-point operations is:

$$\text{FLOPs} = 2 \cdot n_{\text{local}}.$$

Thus, the operational intensity is:

$$I = \frac{\text{FLOPs}}{\text{Data Movement}} = \frac{2 \cdot n_{\text{local}}}{2 \cdot n_{\text{local}} \cdot \text{sizeof(real64)}} = \frac{1}{\text{sizeof(real64)}}.$$

This has not changed from the shared memory case.

The communication cost in distributed axpy

In addition to the memory access costs, the distributed axpy operation incurs a communication cost due to the distribution of vectors across MPI processes.

However, since the axpy operation is **embarrassingly parallel**, there is no need for inter-process communication during the computation itself.

Therefore, the **communication cost is negligible** for the axpy operation, and the *performance is primarily determined by the memory bandwidth of the local memory*.

Let us run some experiments to confirm this.

axpy performance bechmark

We can write a single benchmark program to test the performance of the distributed axpy operation for both **strong** and **weak scaling**.

We read the type of experiment from the command line arguments:

```
if (rank == 0) then
  call get_command_argument(1, scaling_type)
  call get_command_argument(2, arg_buffer)
  read(arg_buffer, *) n_size_input

  if (trim(scaling_type) /= 'strong' .and. trim(scaling_type) /= 'weak') then
      write(error_unit, '(A)') "Error: scaling_type must be 'strong' or 'weak'"
      call MPI_Abort(MPI_COMM_WORLD, 1, ierr)
  end if
end if
```

axpy performance bechmark

Then we set the global vector size based on the scaling type:

```
! Broadcast arguments to all ranks
call MPI_Bcast(scaling_type, 20, MPI_CHARACTER, 0, MPI_COMM_WORLD, ierr)
call MPI_Bcast(n_size_input, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr)

! Compute global size based on scaling type
if (trim(scaling_type) == 'strong') then
  n_global = n_size_input
else  ! weak scaling
  n_global = n_size_input * nprocs
end if
```

Recall: **strong scaling** means fixed problem size, **weak scaling** means fixed problem size per process.

axpy performance bechmark

2 Distributed Level-1 BLAS

We are now ready to create the distributed vectors and run the benchmark:

```
! Initialize vectors and data
call x%dinit(MPI_COMM_WORLD, n_global)
call y%dinit(MPI_COMM_WORLD, n_global)
x%data = 1.0_real64
y%data = 2.0_real64
call MPI_Barrier(MPI_COMM_WORLD, ierr)
call y%daxpy_dist(alpha, x) ! Warmup run
call MPI_Barrier(MPI_COMM_WORLD, ierr)
! Timing runs
start_time = MPI_Wtime()
do i = 1, num_trials
  call y%daxpy_dist(alpha, x)
end do
end_time = MPI_Wtime()
elapsed_time = (end_time - start_time) / num_trials
```

axpy performance bechmark

Finally we gather the results:

```
! Gather timing statistics
call MPI_Reduce(elapsed_time, max_time, 1, MPI_REAL8, MPI_MAX, 0, MPI_COMM_WORLD, ierr)
call MPI_Reduce(elapsed_time, min_time, 1, MPI_REAL8, MPI_MIN, 0, MPI_COMM_WORLD, ierr)
call MPI_Reduce(elapsed_time, avg_time, 1, MPI_REAL8, MPI_SUM, 0, MPI_COMM_WORLD, ierr)
```

Which we can then print from the root process:

```
avg_time = avg_time / nprocs
write(output_unit, *)
write(output_unit, '(A, I12)') "Global vector size:     ", n_global
write(output_unit, '(A, I12)') "Local vector size:      ", x%n_local
write(output_unit, '(A, F12.6, A)') "Average time:           ", avg_time * 1000, " ms"
write(output_unit, '(A, F12.6, A)') "Min time:               ", min_time * 1000, " ms"
write(output_unit, '(A, F12.6, A)') "Max time:               ", max_time * 1000, " ms"
write(output_unit, '(A, F12.2, A)') "Throughput:             ", &
    (n_global * 3.0_real64 * 8.0_real64) / (avg_time * 1.0e9), " GB/s"
write(output_unit, '(A, F12.2, A)') "Performance:            ", &
    (n_global * 2.0_real64) / (avg_time * 1.0e9), " GFLOP/s"
```

axpy performance experiment setup

We run the benchmark on the AMELIA cluster at IAC-CNR, which is the same we used to measure network performance.

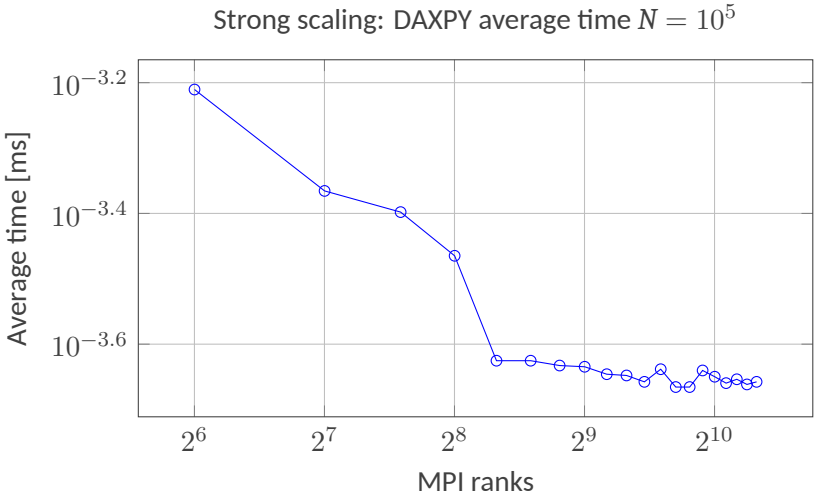
```
#!/bin/bash
#SBATCH --job-name=axpy_scaling_64ppn
#SBATCH --nodes=20
#SBATCH --ntasks-per-node=64
#SBATCH --time=00:30:00
#SBATCH --partition=prod-gn
#SBATCH --mem=900Gb
#SBATCH --output=axpy_%j.out
#SBATCH --error=axpy_%j.err

# Load Intel oneAPI modules
module load intel/oneapi/intel_MKL-2023.2.0 intel/oneapi/intel_MPI-2023.2.0
```

axpy performance experiment setup

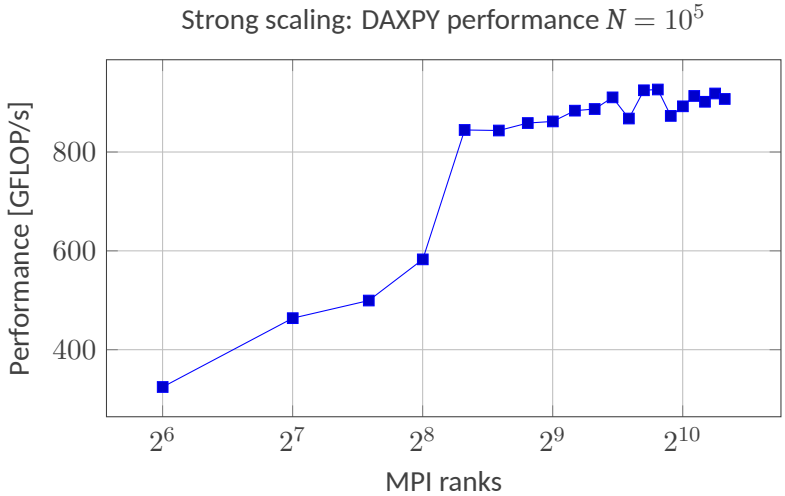
For the **strong scaling** we launch the benchmark:

```
N_GLOBAL_STRONG=100000   # fixed total vector size
echo " STRONG SCALING TEST"
echo " Global vector size = ${N_GLOBAL_STRONG}"
for NODES in $(seq 1 20); do
    NTASKS=$((NODES * 64))
    echo
    echo "Strong scaling run:"
    echo "  Nodes     : $NODES"
    echo "  MPI ranks: $NTASKS"
    echo "  N_global : $N_GLOBAL_STRONG"
    mpirun -np $NTASKS ./mpi_axpy_scaling strong $N_GLOBAL_STRONG
    echo "----------------------------------------"
done
```

axpy performance experiment: strong scaling



axpy performance experiment: strong scaling



Strong scaling analysis

The strong scaling results show that the execution time **decreases** as we increase the number of MPI ranks, which is the expected behavior. However, the **rate of decrease** slows down significantly after 256 ranks.

Key observations:

- From 64 to 256 ranks: execution time decreases from 0.616 ms to 0.343 ms ($\approx 45\%$ reduction),
- From 256 to 1280 ranks: execution time decreases from 0.343 ms to 0.220 ms ($\approx 36\%$ reduction).

This indicates that we are approaching the **communication and memory bandwidth limits** of the system.

Strong scaling efficiency

The performance per rank (in GFLOP/s) shows a **significant improvement** from 64 to 320 ranks, increasing from ≈ 324 GFLOP/s to ≈ 845 GFLOP/s.

Beyond 320 ranks, the performance **stabilizes** around 900 GFLOP/s with minor fluctuations.

This behavior is typical for memory-bound operations:

- Initial improvement due to better cache utilization and memory bandwidth saturation,
- Plateau effect due to the inherent memory bandwidth limitation of individual compute nodes.

Strong scaling saturation

The saturation in performance suggests that we have **reached the memory bandwidth limit** of the underlying hardware at approximately 900 GFLOP/s.

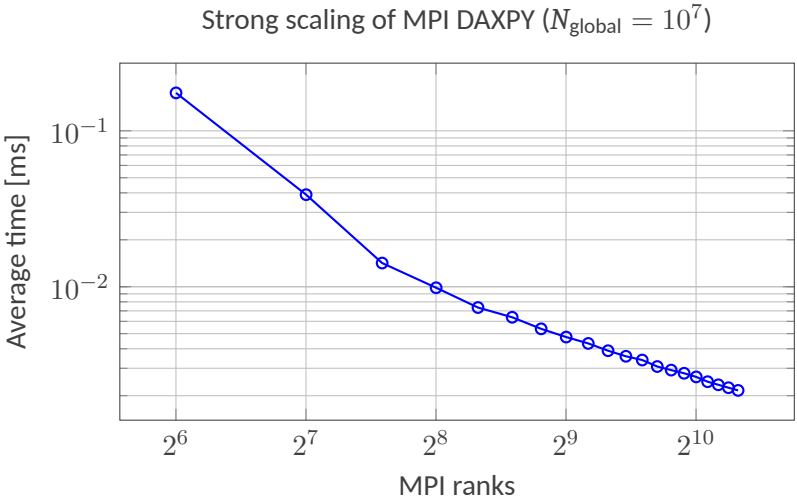
This is consistent with the memory-bound nature of the axpy operation, where:

- The operational intensity is very low ($I = 1/8$ for 64-bit floats),
- The computation is limited by memory bandwidth, not floating-point performance.

Further improvements would require either:

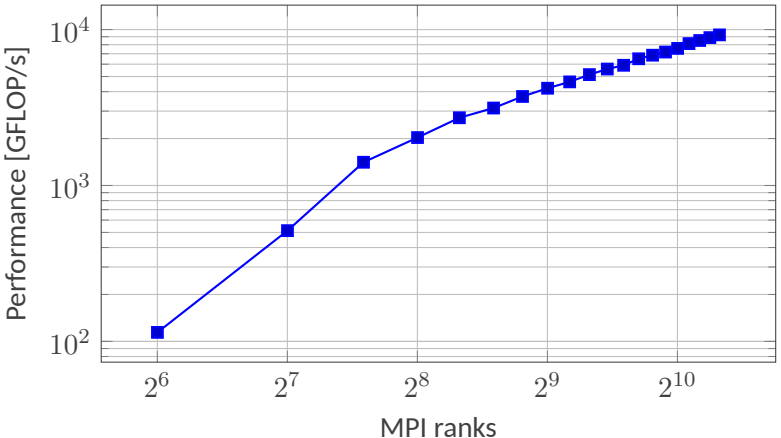
- Increasing the problem size (to improve cache reuse), and let's do it!

axpy performance experiment: strong scaling



axpy performance experiment: strong scaling

Strong scaling performance of MPI DAXPY ($N_{\text{global}} = 10^7$)

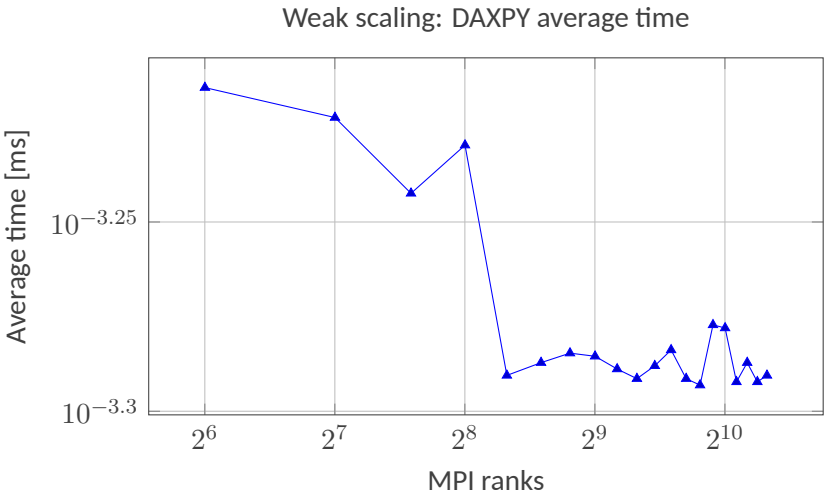


axpy performance experiment: weak scaling setup

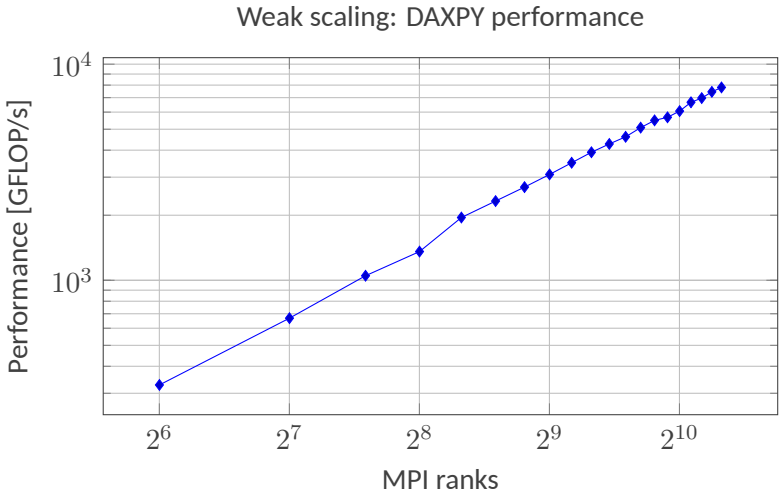
For the **weak scaling** we launch the benchmark:

```
N_LOCAL_WEAK=1562   # local size per MPI rank
echo " WEAK SCALING TEST"
echo " Local vector size per rank = ${N_LOCAL_WEAK}"
for NODES in $(seq 1 20); do
    NTASKS=$((NODES * 64))
    N_GLOBAL_WEAK=$((NTASKS * N_LOCAL_WEAK))
    echo
    echo "Weak scaling run:"
    echo "  Nodes    : $NODES"
    echo "  MPI ranks: $NTASKS"
    echo "  N_global : $N_GLOBAL_WEAK (${N_LOCAL_WEAK} per rank)"
    mpirun -np $NTASKS ./mpi_axpy_scaling weak $N_LOCAL_WEAK
    echo "----------------------------------------"
done
```

axpy performance experiment: weak scaling



axpy performance experiment: weak scaling



Weak Scaling Results Analysis

The weak scaling results for the distributed axpy operation demonstrate that as we increase the number of MPI ranks while maintaining a constant local vector size per rank, the execution time remains relatively stable at approximately 0.5 ms.

- From 64 to 1280 ranks: execution time fluctuates between 0.51 and 0.61 ms, showing excellent stability,
- The performance in GFLOP/s scales nearly linearly with the number of ranks, indicating efficient distributed memory utilization,
- This demonstrates that the **embarrassingly parallel** nature of the axpy operation is preserved in the distributed implementation.

This behavior is ideal for weak scaling scenarios, where the problem size grows proportionally with the number of processes. The stable execution time confirms that there is no significant communication overhead in the distributed axpy operation.

Weak Scaling Performance Insights

The weak scaling results for the distributed axpy operation highlight several key insights:

- The constant local computation time across all ranks confirms that the distributed axpy implementation has negligible communication overhead,
- The linear increase in total GFLOP/s (from ≈ 328 to ≈ 7807 GFLOP/s) demonstrates perfect computational scaling,
- The local memory bandwidth utilization remains constant, indicating that each rank operates independently without interference from inter-process communication.

Overall, the weak scaling results validate that the distributed axpy operation is well-suited for large-scale parallel computations, with minimal communication overhead and excellent scalability properties.

Scaling of the dot product and norm operations

We can now investigate the scaling behavior of the **dot product** and **norm** operations, which involve global reductions and are therefore expected to exhibit different scaling characteristics compared to the axpy operation.

The dot product and norm operations require communication between MPI processes to aggregate local results, which can introduce significant overhead, especially as the number of processes increases.

We will analyze both **strong** and **weak scaling** for these operations to understand their performance limits.

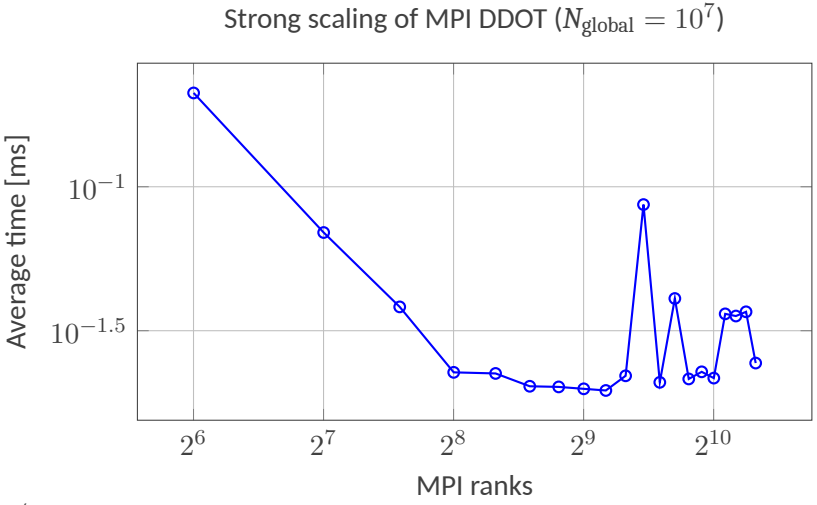
Writing the benchmark

2 Distributed Level-1 BLAS

The initial part of the benchmark program is similar to the axpy benchmark, decide the scaling type and set the global vector size accordingly. The main difference is in the timing section, where we replace the axpy call with either the dot product or norm call:

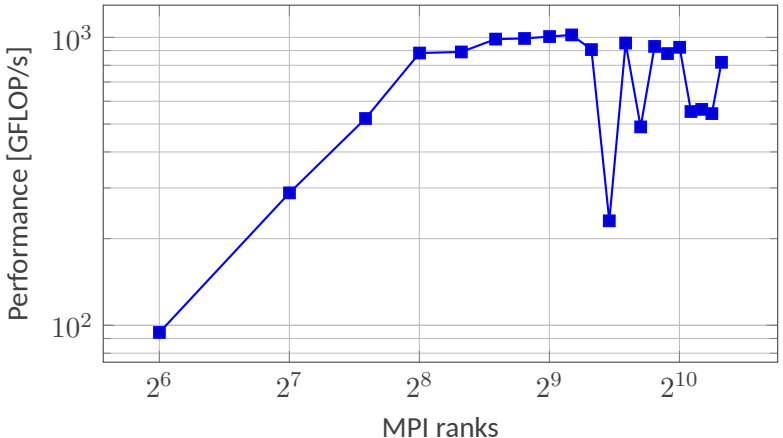
```
call x%dinit(MPI_COMM_WORLD, n_global)
call y%dinit(MPI_COMM_WORLD, n_global)
x%data = 1.0_real64
y%data = 2.0_real64
call MPI_Barrier(MPI_COMM_WORLD, ierr)
call x%ddot_dist(y, dot_value)
call MPI_Barrier(MPI_COMM_WORLD, ierr)
start_time = MPI_Wtime()
do i = 1, num_trials
  call x%ddot_dist(y, dot_value)
end do
end_time = MPI_Wtime()
elapsed_time = (end_time - start_time) / num_trials
```

ddot performance experiment: strong scaling



ddot performance experiment: strong scaling

Strong scaling performance of MPI DDOT ($N_{\text{global}} = 10^7$)



Strong Scaling Analysis: MPI DDOT ($N_{\text{global}} = 10^7$)

The strong scaling results for the distributed dot product reveal interesting behavior patterns:

- **Initial strong scaling (64–576 ranks):** Execution time decreases from 0.212 ms to 0.020 ms, showing nearly ideal scaling behavior.
- **Peak performance (512–576 ranks):** Approximately 1000 GFLOP/s, indicating efficient local computation and reduction.
- **Performance degradation (704+ ranks):** Significant variability and performance drops appear, suggesting communication overhead dominates.

The critical observation is that performance becomes unstable beyond 576 ranks, with execution time fluctuating between 0.020 ms and 0.087 ms.

Understanding the Scaling Breakdown

Why does performance degrade at high rank counts?

- Local vector size: $n_{\text{local}} = \frac{10^7}{N_{\text{ranks}}}$
- At 1280 ranks: $n_{\text{local}} \approx 7800$ elements
- Too small for effective cache utilization

Communication cost dominates:

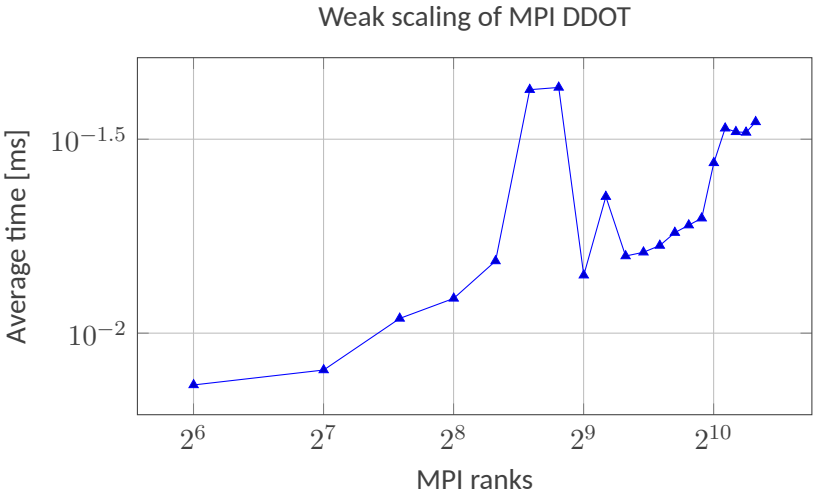
- MPI reduction becomes a bottleneck
- Synchronization overhead exceeds computation time
- Network latency not fully hidden by computation

Key insight: Unlike embarrassingly parallel axpy, dot product requires global synchronization via MPI_Allreduce, which becomes increasingly expensive as rank count grows.

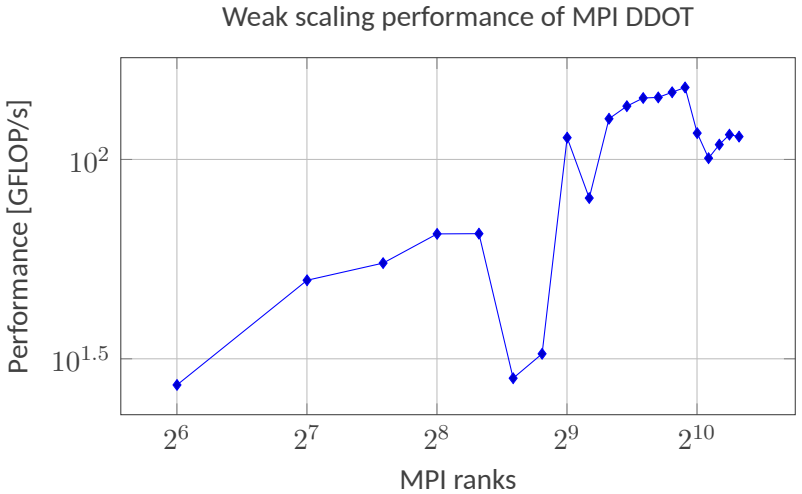
Recommendations for Practical Use

- **Optimal configuration:** 256–576 ranks with problem size $N \geq 10^6$ per rank
  - Maintains local vector size large enough for efficient computation
  - Communication overhead remains manageable
- **Weak scaling preferred:** For dot product and norm operations, weak scaling (constant work per rank) provides better performance predictability
- **Avoid extreme decomposition:** Do not distribute to more ranks than necessary; overhead grows quadratically with rank count for global reductions

ddot performance experiment: weak scaling



ddot performance experiment: weak scaling



Weak Scaling of Distributed DDOT with MPI_Allreduce

Setup: Each MPI rank maintains a fixed local vector of 1562 elements. The global dot product requires `MPI_Allreduce` to aggregate local contributions across all ranks.

Execution time behavior:

- Scales well up to ≈320 ranks with gradual time increase
- Significant spikes at 384–448 and 1024+ ranks indicate communication bottlenecks
- Variability suggests network congestion and synchronization overhead

Performance scaling:

- Peak performance ≈150 GFLOP/s around 960 ranks
- Drops align with execution time spikes, confirming communication dominates
- Unlike axpy, performance is latency-bound, not compute-bound

Weak Scaling of Distributed DDOT with MPI_Allreduce

Key insight: At high rank counts, each rank's local vector is too small to hide the cost of the global all-reduce operation. The computation time becomes negligible compared to synchronization overhead.

Communication avoiding algorithms

To mitigate these issues, research has moved towards **communication-avoiding algorithms** that try to reduce the number of global synchronizations required, for example by restructuring computations to perform more local work before communicating, or by reformulating algorithms to reduce the frequency of reductions.

Conclusion and next steps

We have

- ✔ Implemented Level-1 BLAS operations for distributed vectors using MPI,
- ✔ Analyzed performance characteristics for axpy, dot product, and norm,
- ✔ Identified scaling limits due to communication overhead in reduction operations

Next steps:

- Explore Level-2 and Level-3 BLAS operations in distributed settings.