# Performance Analysis of Raspberry Pi 4B (8GB) Beowulf Cluster: STREAM Benchmarking

Dimitrios Papakyriakou
PhD Candidate
Department of Electronic Engineering
Hellenic Mediterranean University
Crete, Greece

Ioannis S. Barbounakis
Assistant Professor
Department of Electronic Engineering
Hellenic Mediterranean University
Crete, Greece

## ABSTRACT
This study presents a detailed performance analysis of a 24-node Beowulf cluster built with Raspberry Pi 4B devices, equipped with 8GB of RAM, running a 64-bit operating system utilizing the STREAM Benchmark which is a widely recognized tool for evaluating memory bandwidth performance in high-performance computing (HPC) environments. Unlike typical processor benchmarks that focus on computing power, STREAM a memory bandwidth benchmark focuses on how quickly data can be transferred between the memory and the processor, which is a critical performance factor in high-performance computing (HPC) systems like Beowulf clusters.

Fundamental memory operations *Copy***,** *Scale***,** *Add***,** and *Triad*, are utilized to assess how efficiently the cluster handles memory-intensive workloads across increasing MPI process counts.

Additionally, MPI-based communication benchmarks assess the inter-node message-passing performance, providing deeper insights into memory bandwidth utilization under distributed computing conditions. The findings offer valuable insights on the perspectives of using Raspberry Pi clusters for HPC applications in education, research, and prototyping. Furthermore, recommendations for performance optimizations and system enhancements are proposed to improve scalability**,** efficiency**,** and communication overhead in such low-cost HPC clusters.

## Keywords
Raspberry Pi 4 Beowulf cluster, Cluster, Message Passing Interface (MPI), MPICH, Memory Performance, Low-cost Clusters, Parallel Computing, ARM Architecture, STREAM Benchmark

## 1. INTRODUCTION
In recent years, the development of single-board computers (SBCs) with enhanced processing capabilities has made it possible to explore low-cost alternatives for building high-performance computing (HPC) clusters. The Raspberry Pi 4B, equipped with an ARM Cortex-A72 quad-core processor and 8GB of RAM, has emerged as a popular choice for building Beowulf clusters for parallel and distributed computing experiments. Unlike traditional HPC systems that rely on costly and power-intensive hardware, SBC-based clusters offer a unique opportunity to investigate performance scalability at a fraction of the cost.

The performance of (HPC) systems is influenced by several critical factors, including memory bandwidth, latency, communication efficiency, and computational capabilities. Stream variations, which measure sustained memory bandwidth and access patterns, provide insights into how effectively a system can handle data-intensive tasks. Memory bandwidth, the rate at which data is transferred between memory and processors, directly impacts the speed of computation, especially in memory-bound applications. Latency, the time taken to access or transfer data, is equally critical, as it affects overall system responsiveness and the efficiency of parallel processing.

The STREAM benchmark consists of four simple yet effective tests:

*Copy* – Measures how fast data can be duplicated evaluating raw memory transfer speed (load/store performance).

*Scale* – Similar to *Copy*, but with an extra multiplication step. Tests the ability to perform simple arithmetic (multiplication) while moving data.

*Add* – Measures the bandwidth when adding two arrays element-wise. In other word, tests how quickly two arrays can be added together, assessing the system's performance in basic vector addition, requiring more memory operations.

*Triad* – A combination of all three, representing a real-world scientific workload. Measures bandwidth for the most complex of the four operations, combining addition and scaling and represents a more realistic computational pattern found in many scientific and engineering applications.

By running these tests across multiple processing units, STREAM helps identify memory bottlenecks, test memory bandwidth scaling, and assess the overall efficiency of the system's data transfers. For a Raspberry Pi 4B Beowulf cluster, this benchmarking is particularly useful in understanding how well the system handles parallel workloads and whether communication between nodes is a limiting factor.

In essence, STREAM provides a simple yet powerful way to evaluate whether a system's memory bandwidth is a performance bottleneck, making it invaluable for researchers optimizing cluster performance.

This study focuses on evaluating the performance of a 24-node Beowulf cluster composed of Raspberry Pi 4B nodes, with an emphasis on memory performance, inter-node communication, and variations in memory-intensive operations, providing insights into how effectively the cluster handles large data streams.

The findings aim to illuminate the performance trends and bottlenecks associated with memory-intensive workloads in Raspberry Pi-based clusters. Additionally, recommendations for optimizing performance through software and hardware

enhancements are discussed to guide future developments in cost-effective, distributed computing systems.

Traditional high-performance computing (HPC) systems primarily utilize CPUs for double-precision floating-point calculations, which are critical for scientific simulations, numerical analysis, and large-scale computations. However, modern supercomputing architectures have evolved to incorporate a heterogeneous mix of processing units, including Graphics Processing Units (GPUs) and Field-Programmable Gate Arrays (FPGAs). GPUs excel at massively parallel operations due to their thousands of cores optimized for concurrent data processing, making them ideal for machine learning, matrix operations, and simulations. FPGAs, on the other hand, offer reconfigurable hardware that can be tailored to specific computational tasks, providing significant improvements in power efficiency and performance for specialized workloads. This shift towards heterogeneous computing enables supercomputing systems to achieve higher throughput, reduced latency, and improved energy efficiency, paving the way for breakthroughs in areas such as artificial intelligence, climate modeling, and computational biology [1], [2].

The Raspberry Pi (RPi) 4 Model B with 8GB of RAM, as shown in "Figure 1," is utilized in the Beowulf cluster. It features a 64-bit quad-core ARMv8 Cortex-A72 CPU running at 1.5 GHz, offering three times the processing power of each predecessor RPi 3B+ model [3], [4].

The affordability of the Raspberry Pi was a key factor in exploring its potential as a cost-effective solution for constructing a high-performance computing cluster and evaluating its capability to operate efficiently in parallel. clustering mode.

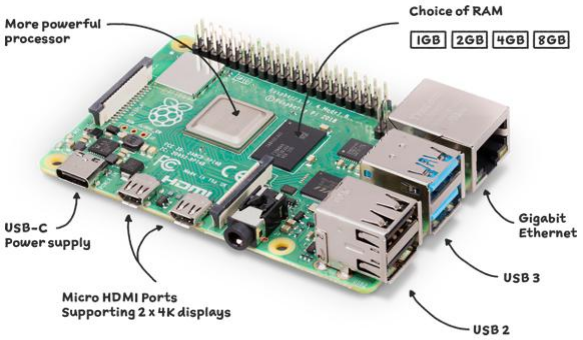


**Figure 1: Single Board Computer (SBC) - Raspberry Pi 4 Model B [3].**

# 2. SYSTEM DESCRIPTION

## 2.1 Hardware Equipment

The Beowulf cluster consists of 24 Raspberry Pi 4 devices, as depicted in "Figure 2". One Raspberry Pi 4B (8GB) serves as the master (or head) node, tasked with managing job allocation and resource distribution, while the remaining 23 Raspberry Pis function as worker nodes, executing tasks as directed by the master node.

The nodes are organized into four stacks, each containing six Raspberry Pis, and connected through one Gigabit switches (TL-SG1024D), enabling a maximum network throughput of 1000 Mbps per node. This network configuration facilitates the creation of a cohesive computing system resembling a supercomputer. The entire cluster is powered by two switch-mode power supplies, each rated at 60 amps with a 5V output boosted to 5.80V to compensate for voltage drops along the wiring.

Moreover, the master (or head) node comprises a Samsung (1TB) 980 PCI-E 3 NVMe M.2 SSD external disk whereas the slave (or worker) nodes host each one of a (256 GB) Patriot P300P256GM28 NVME M.2 2280 external disk.


**Figure 2: Deployment of the Beowulf Cluster with (24) RPi-4B (8GB)**

## 2.2 Software Tools

The Operating System used to setup the RPi's in the cluster is the latest "Debian GNU/Linux 12 (bookworm)" which is the latest official supported Operating System (OS - 64 bits) with Kernel version 6.6.62+rpt-rpi-v8 and the CPU architecture and capabilities of the system "Figure 3","Figure 4".

The second software package required for the setup was the Message Passing Interface (MPI), specifically the MPICH implementation. MPICH is a highly efficient and widely adaptable implementation of MPI, which is one of the most commonly used frameworks for passing message in parallel computing. It is important to note that MPI itself is not a library but a standardized framework for designing message-passing libraries, as recommended by the MPI Forum. Among the prominent MPI implementations available for use on Raspberry Pi are OpenMPI and MPICH. For this project, MPICH was selected. Originally an acronym for Message Passing Interface Chameleon, MPICH adheres to the MPI standard and supports applications written in C, C++, and FORTRAN.

The third software package installed was the GNU Compiler Collection (GCC) Fortran compiler, which is well-suited for high-performance computing due to its optimization and multi-threading capabilities. As the default compiler suits many HPC environments, GCC is crucial for compiling and optimizing parallel computing applications.

The fourth essential software package was OpenBLAS, a highly optimized Basic Linear Algebra Subprograms (BLAS) library. OpenBLAS provides efficient implementations for performing linear algebra operations, which are foundational for many scientific and engineering computations.

Finally, the STREAM benchmark software package is needed to be downloaded and compiled accordingly.

```
4. rpi4B-ma-00_150
pi@rpi4B-ma-00:~ $ cat /etc/debian_version
12.8
pi@rpi4B-ma-00:~ $ cat /etc/os-release | grep -i 'PRETTY_NAME'
PRETTY_NAME="Debian GNU/Linux 12 (bookworm)"
pi@rpi4B-ma-00:~ $ uname -a
Linux rpi4B-ma-00 6.6.62+rpt-rpi-v8 #1 SMP PREEMPT Debian 1:6.6.62-1+rpt1 (2024-
11-25) aarch64 GNU/Linux
pi@rpi4B-ma-00:~ $ inxi -m | grep total
  RAM: total: 7.71 GiB used: 694.7 MiB (8.8%) gpu: 76 MiB
pi@rpi4B-ma-00:~ $ cat /proc/cpuinfo | grep -i 'Raspberry'
Model           : Raspberry Pi 4 Model B Rev 1.5
pi@rpi4B-ma-00:~ $ uname -r
6.6.62+rpt-rpi-v8
pi@rpi4B-ma-00:~ $ cat /proc/version
Linux version 6.6.62+rpt-rpi-v8 (serge@raspberrypi.com) (gcc-12 (Debian 12.2.0-1
4) 12.2.0, GNU ld (GNU Binutils for Debian) 2.40) #1 SMP PREEMPT Debian 1:6.6.62
-1+rpt1 (2024-11-25)
pi@rpi4B-ma-00:~ $
```

**Figure 3: OS release, Kernel Version, RAM memory and RPi HW version.**

```
pi@rpi4B-ma-00:~ $ lscpu
Architecture:             aarch64
  CPU op-mode(s):         32-bit, 64-bit
  Byte Order:             Little Endian
CPU(s):                   4
  On-line CPU(s) list:    0-3
Vendor ID:                ARM
  Model name:             Cortex-A72
    Model:                3
    Thread(s) per core:   1
    Core(s) per cluster:  4
    Socket(s):            -
    Cluster(s):           1
    Stepping:             r0p3
    CPU(s) scaling MHz:   33%
    CPU max MHz:          1800.0000
    CPU min MHz:          600.0000
    BogoMIPS:             108.00
    Flags:                fp asimd evtstrm crc32 cpuid
Caches (sum of all):
  L1d:                    128 KiB (4 instances)
  L1i:                    192 KiB (4 instances)
  L2:                     1 MiB (1 instance)
NUMA:
  NUMA node(s):           2
  NUMA node0 CPU(s):      0-3
  NUMA node1 CPU(s):
```

**Figure 4: CPU architecture and System Capabilities.**

## 2.3 Design

The RPi cluster architecture diagram is depicted in "Figure 5". It comprises 24 Raspberry Pi 4B devices, each equipped with 8GB of memory [4], interconnected through a 24-port Gigabit Ethernet switch (1000 Mbps). Among the 24 nodes, one serves as the master (or head) node, while the remaining 23 function as worker nodes. The network configuration uses static IP addressing, ensuring that each node has a unique and fixed IP address. Communication between the master node and the worker nodes is conducted exclusively through secure shell (SSH) connections.

The master node is equipped with a Samsung 980 PCIe 3.0 NVMe M.2 SSD (1TB), capable of theoretical maximum write speeds of 3000 MB/s and read speeds of 3500 MB/s. The worker nodes were upgraded to use Patriot P300P256GM28 NVMe M.2 SSDs (256GB), with maximum write and read speeds of 1100 MB/s and 1700 MB/s, respectively. The Raspberry Pi 4B's support for external booting allowed the SSDs to be mounted via USB 3.0 ports, which offer a theoretical transfer speed of 4.8 Gbps (600 MB/s), significantly outperforming USB 2.0's maximum transfer rate of 480 Mbps (60 MB/s).

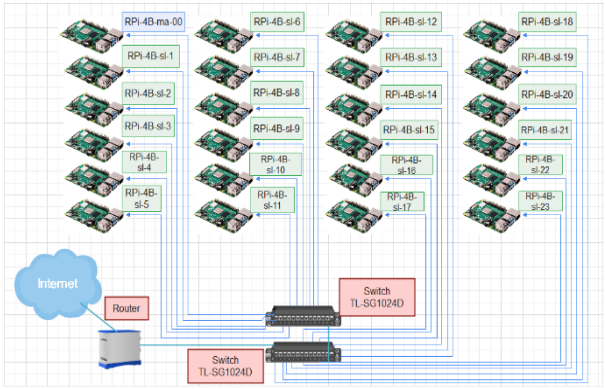


**Figure 5: RPi-4B Beowulf cluster architecture diagram [9].**

By leveraging the superior read and write speeds of the NVMe SSDs, this phase of testing anticipated substantial performance improvements over the microSD-based setup. While the USB 3.0 interface imposes some bandwidth limitations, the NVMe SSDs' capabilities far exceed these constraints, ensuring a notable enhancement in cluster performance.

## 3. STREAM variations and advanced memory performance.

The STREAM benchmark is a widely used tool for measuring the sustainable memory bandwidth and performance of memory subsystems in high-performance computing (HPC) systems [5], [6]. It provides insight into how efficiently a system can move data between the CPU and memory, which is critical for many scientific and engineering workloads that are memory-bound. The purpose of STREAM benchmark is to:

- To measure the memory bandwidth of a system under a realistic workload.
- To test the system's ability to handle large, vector-style operations that involve significant data movement.
- To evaluate memory hierarchy performance, such as cache utilization and main memory throughput.

The core concept of STREAM benchmark relies on three pillars such as [7], [8]:

- *Memory Bandwidth*: Refers to the rate at which data can be transferred between memory and the CPU. High memory bandwidth is essential for workloads that require frequent data access, such as numerical simulations or machine learning.
- *Large Data Arrays*: STREAM operates on arrays much larger than the system's cache to ensure the measurement reflects main memory performance rather than cache performance
- *Simplicity*: The benchmark consists of simple, loop-based operations that are easy to implement and optimize, making it suitable for a wide variety of systems.

STREAM measures the performance of four basic vector operations, which are representative of common computational workloads. The Key Metrics are addressed below:

- *Copy:* Copies one array into another, tests memory bandwidth for read and write operations following the formula $C = A$

- *Scale:* Scales an array by a constant factor, tests computational throughput and memory bandwidth following the formula $B = scala \times A$.
- *Add:* Performs element-wise addition of two arrays into a third, and tests memory bandwidth for multiple data streams following the formula $C = A + B$.
- *Triad:* Combines scaling and addition operations and tests the system's ability to handle multiple, simultaneous memory operations following the formula $C = A + scalar \times B$.

These vector operations allow STREAM to measure memory bandwidth in practical scenarios, such as computational workloads involving linear algebra, data analytics, or numerical simulations.

The output of the STREAM across the cluster provides the performance for four memory bandwidth operations:

- *Copy:* Measure $a[i] = b[i]$
- *Scale:* Measure $a[i] = scalar * b[i]$
- *Add:* Measure $a[i] = b[i] + c[i]$
- *Triad:* Measure $a[i] = b[i] + scalar * c[i]$

The STREAM benchmarking methodology is based on the following key parameters:

- *Array Size:* Arrays must be significantly larger than the system's cache (typically hundreds of megabytes) to ensure measurements reflect main memory performance. The size is controlled via DSTREAM_ARRAY_SIZE parameter.
- *Iterations:* Each kernel is executed multiple times (default value is 10 to minimize noise and measure stable performance.
- *Best time:* The shortest execution time across iterations is used to calculate memory bandwidth for each kernel, ensuring results are not skewed by transient delays.
- *Output Metrics:* The output metrics are the below:
  Bandwidth (MB/s): Memory throughput for each operation.
  Execution Times (s): Average, minimum, and maximum times for each kernel.

The STREAM Benchmark does not need to be installed on each worker node individually. It only needs to be compiled and executed on the master node, provided that:

- *Shared File System:* The worker nodes have access to the same directory where the stream_mpi binary is located, typically through a shared file system like NFS (Network File System).
- *MPI Environment:* The MPI environment is properly configured so that the master node can distribute the benchmark workload to the worker nodes.

As a general recommendation in terms of requirements for selecting DSTREAM_ARRAY_SIZE parameter are the following:

- *Stream benchmark requirements:* A general recommendation is that the total size of all three arrays ( $a, b, and\ c$ should be at least 4 times the size of the CPU cache (L2 and L3 combined). For Raspberry Pi 4B, the L2 cache is 1MB and there is no L3 cache, so each array should be much larger than 4MB.
- *Memory Allocation per Node:* Each Raspberry Pi 4B has 8GB of RAM. Avoid using 100% of the memory for the benchmark to leave room for the operating system and other processes. A safe usage is $75\% - 80\%$ of total memory per node.
- *Distribution Execution Considerations:* When running STREAM in an MPI environment, the total array size will be distributed across nodes. The DSTREAM_ARRAY_SIZE represents the size per process.

Each array ( $a, b, and\ c$) needs STREAM_ARRAY_SIZE × size of (double) bytes. Since a double is 8 bytes the memory usage per array is calculated as:

( $Memory\ usage\ per\ array = X \times 8\ bytes$)

The total memory usage for all three arrays is given by:

( $Total\ Memory\ usage\ for\ the\ three\ arrays = 3 \times X \times 8\ bytes$).

As an example, we have for one RPi node:

Usage memory: ( $8GB \times 0.75 = 6GB = 6 \times 1024^3 = 6{,}442{,}450{,}944\ bytes$)

Memory per array: Divide by 3 arrays: ( $6{,}442{,}450{,}944 \div 3 = 2{,}147{,}483{,}648$)

Elements per array: Divide by 8 (size of double): ( $2{,}147{,}483{,}648 \div 8 = 268{,}435{,}456$) elements per array.

As a result, we have DSTREAM_ARRAY_SIZE = 268,435,456 (elements per process). For 1 process per node the maximum DSTREAM_ARRAY_SIZE = 268,435,456 elements per node.

If it runs run multiple MPI processes per node then the STREAM_ARRAY_SIZE is divided by the number of processes per node to avoid exceeding available memory meaning DSTREAM_ARRAY_SIZE = ( $\frac{268{,}435{,}456}{4} = 67{,}108{,}864$)

The recommended Values are:
1 process per node: DSTREAM_ARRAY_SIZE=268,435,456
2 process per node: DSTREAM_ARRAY_SIZE= 134,217,728
3 process per node: DSTREAM_ARRAY_SIZE= 89,478,485
4 processes per node: DSTREAM_ARRAY_SIZE=67,108,864

To sum up:

Usable memory for STREAM: ( $8GB \times 0.75 = 6GB = 6 \times 1024^3 = 6{,}442{,}450{,}944\ bytes$)
Usable memory per core/process:

$$Per - core\ memory = \frac{Usable\ memory\ for\ STREAM}{Number\ of\ cores} = \frac{6{,}442{,}450{,}944\ bytes}{4} = \approx 1.5\ GB\ (1{,}610{,}612{,}736\ bytes)$$

As a result, for single process on entire RPi: use DSTREAM_ARRAY_SIZE=268,435,456, and for multiple processes (4 processes for 4 cores) in (1) RPi, use DSTREAM_ARRAY_SIZE=67,108,864.

To force execution on a single core the used command is:
(# 1 process on core 0)
taskset -c 0 mpiexec -np 1 ./stream_mpi.

To force execution on two cores the used command is:

(# 2 processes on cores 0 and 1)
taskset -c 0,1 mpiexec -np 2 ./stream_mpi.

To force execution on three cores the used command is:
(# 3 processes on cores 0, 1, and 2)
taskset -c 0,1 mpiexec -np 3 ./stream_mpi.

To force execution on four cores the used command is:
(# 4 processes on cores 0, 1, 2, and 3)
taskset -c 0,1,2,3 mpiexec -np 4 ./stream_mpi

The (-np) flag in mpiexec should match the number of processes you want to launch (1 for single-core, 2 for two-core, etc.). If you use (-np 1) for all commands, only one process will run, regardless of the specified cores.

Alternatively, the easiest method is to let MPI manage core binding automatically, as used in this study, where one MPI process is automatically assigned per core:

# 1 process
mpiexec --bind-to core -np 1 ./stream_mpi
# 2 processes
mpiexec --bind-to core -np 2 ./stream_mpi
# 3 processes
mpiexec --bind-to core -np 3 ./stream_mpi
# 4 processes
mpiexec --bind-to core -np 4 ./stream_mpi

In this study for $(np = 1)$ (1 process) the DSTREAM_ARRAY_SIZE=171,798,691 used ($\approx$ 48 % $RPi\ memory$ $(8GB)$ due to inability to compile for larger array sizes. The whole research focuses on using $(np = 2,3,4)$ with (2,3,4) processes respectively since there is value to compare efficiency by allocating the same percentage of usable physical memory.

## 3.1 Stream Variations and advanced Memory Performance in one RPi node.

The testing in one RPi node starts with the compilation of the stream.c package per used MPI process and corresponding DSTREAM_ARRAY_SIZE:

*# 1 process*
DSTREAM_ARRAY_SIZE $= 171{,}798{,}691 \approx$ 48 % $RPi\ memory$ $(8GB)$
Compilation:
$mpicc -O3 -fopenmp -DSTREAM_ARRAY_SIZE=171,798,691 stream.c -o stream_mpi
Command:
$ mpiexec --bind-to core -np 1 ./stream_mpi

*# 2 process*
DSTREAM_ARRAY_SIZE $= 134,217{,}728 \approx$ 75 % $RPi\ memory$ $(8GB)$
Compilation:
$mpicc -O3 -fopenmp -DSTREAM_ARRAY_SIZE=134,217,728 stream.c -o stream_mpi
Command:
$ mpiexec --bind-to core -np 2 ./stream_mpi

*# 3 process*
DSTREAM_ARRAY_SIZE $= 89{,}478{,}485 \approx$ 75 % $RPi\ memory$ $(8GB)$
Compilation:
$mpicc -O3 -fopenmp -DSTREAM_ARRAY_SIZE=89,478,485 stream.c -o stream_mpi
Command:
$ mpiexec --bind-to core -np 3 ./stream_mpi

*# 4 process*
DSTREAM_ARRAY_SIZE $= 67{,}108{,}864 \approx$ 75 % $RPi\ memory$ $(8GB)$
Compilation:
$mpicc -O3 -fopenmp -DSTREAM_ARRAY_SIZE=67,108,864 stream.c -o stream_mpi
Command:
$ mpiexec --bind-to core -np 4 ./stream_mpi

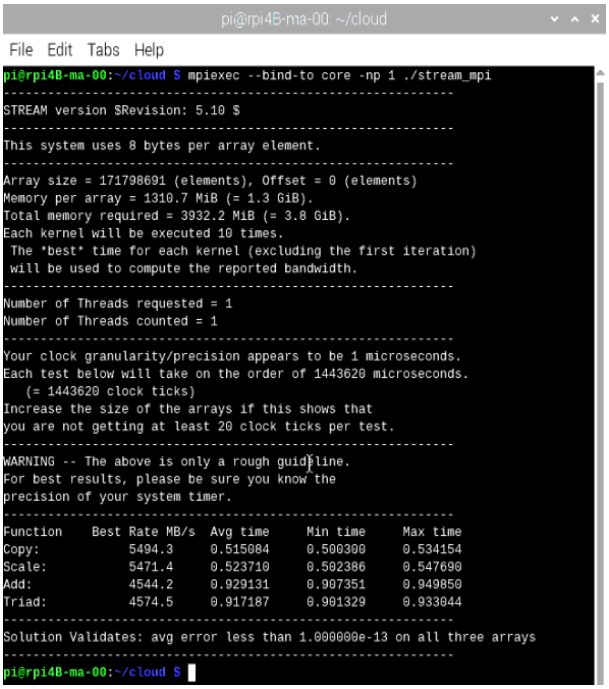


**Figure 6: Stream_mpi test for one RPi in one core with CPU (1) core usage**

Each MPI process acts independently and does not share data with the other. Since (--bind-to core) is used, MPI ensures that each process is pinned to a separate core, and the STREAM results show memory bandwidth per core, rather than total system bandwidth.

The STREAM benchmark tests were conducted using 1 to 4 MPI processes (cores) “Table 1”, “Figure 7”, “Figure 8”. The results reveal key insights about the performance characteristics of the system, including the memory bandwidth and average execution time of different operations. As an example, “Figure 6” presents the Stream_mpi test results in Command Line Interface (CLI) for one RPi in one core with one MPI process.

Below is an evaluation analysis of a STREAM benchmark test with varying numbers of MPI processes (np=1, 2, 3, 4) in one RPi, follows:

- *Performance Scaling:*
  From 1 to 2 MPI processes: Performance remains relatively stable, with a minor decrease in *Copy* and *Scale* Best Rate, while *Add* and *Triad* Best Rate remain close. This suggests

that the Raspberry Pi 4B efficiently utilizes two MPI processes without significant overhead.
From 2 to 3 MPI processes: Significant performance degradation is observed across all benchmarks. *Copy* Best Rate drops from 5096.2 MB/s to 1624.1 MB/s, and *Add* Best Rate drops from 4560.3 MB/s to 1278.7 MB/s. The steep decline suggests the system is struggling with memory contention or CPU resource distribution.
From 3 to 4 MPI processes: Performance continues to drop, but at a slightly slower rate compared to the transition from 2 to 3 processes. The *Copy* Best Rate further declines to 1039.2 MB/s, indicating that increasing MPI processes beyond two causes inefficient CPU resource sharing.

- *Time Metrics Trends:*
  *Copy*, *Scale*, *Add*, and *Triad* execution times all increase as MPI processes grow**.** The first transition (1 to 2 processes) nearly doubles the execution time in *Copy* and *Scale*, while the *Add* and *Triad* Avg Time show a significant increase as well.
  For (3 and 4) MPI processes introduce execution time instability, with *Scale* Avg Time exceeding 1 second. The execution time trends suggest that cache thrashing or memory bandwidth saturation is occurring.
  *Triad* Avg Time remains relatively high from (3 to 4) processes, showing a plateau effect, which may indicate that the memory system has hit a bottleneck and thus it is unable to accommodate further parallelism efficiently**.**

- *Bottleneck Identification:*
  Memory Bandwidth Saturation: The Raspberry Pi 4B has limited memory bandwidth, and with each additional MPI process, the available bandwidth per process decreases significantly.
  Cache Contention: Since the RPi 4B has a shared L2 cache (1MB) across all cores, the performance degradation between 2 and 3 MPI processes suggests excessive cache contention.
  Hyperthreading Absence: The RPi 4B does not support hyperthreading, meaning that each additional MPI process competes for physical CPU cores, resulting in diminishing returns.

- *Efficiency:*
  The (1) MPI process is the most efficient configuration, utilizing 48% of RAM with the best memory bandwidth performance.
  The (2) MPI processes still provide a reasonable tradeoff between performance and parallelization, but execution time increases significantly.
  For 3 and 4 MPI processes exhibit inefficiency, as performance drops despite the additional processes. This indicates that the system is unable to effectively distribute workload across all available cores.
  Practical Implication: Running more than two MPI processes on a single RPi does not yield any computational benefits and may even degrade performance.

As a conclusion, and based on the observations the best performance configuration is that when using 1 or 2 MPI processes per Raspberry Pi which provides the best balance between performance and efficiency.

## 3.2 Stream Variations and advanced Memory Performance in the whole Cluster.

The STREAM benchmark is a widely recognized tool for evaluating memory bandwidth and computational performance in high-performance computing (HPC) systems. In the context of a Beowulf cluster comprising 24 Raspberry Pi 4B nodes, STREAM variations offer a unique opportunity to analyze the interplay between memory performance and cluster-wide scalability. Each node in the cluster provides 8GB of RAM and utilizes the ARM Cortex-A72 processor, making it an ideal testbed for exploring memory bandwidth, latency, and the efficiency of data transfer across distributed nodes.

By leveraging STREAM variations, the cluster's advanced memory performance can be characterized through critical operations such as *Copy*, *Scale*, *Add*, and *Triad*. These operations simulate typical memory access patterns found in scientific and engineering workloads. Conducting STREAM on the entire cluster enables a comprehensive analysis of both local (intra-node) memory bandwidth and the effects of network communication on distributed memory access.

This study aims to highlight the cluster's ability to handle memory-intensive applications, focusing on scalability, memory bandwidth utilization, and on overall performance. Such insights are essential for optimizing workloads in distributed computing environments and understanding the limitations of small-scale, low-cost clusters in HPC scenarios.

The Methodology is very simple and explained below:

- *Distribute the stream.c source code:*
  scp stream.c pi@nodeX:/path/to/destination/

- *Compile with MPI:*

  mpicc -O3 -fopenmp -
  DSTREAM_ARRAY_SIZE=67108864 stream.c -o
  stream_mpi

The above command refers to 75% physical memory for each RPi involving 4 processes per node.

- *Copy the compiled stream_mpi binary to all worker nodes.*
  pi@rpi4B-ma-00:~/cloud $ scp stream_mpi pi@192.168.X.YYY:/home/pi/cloud

- *Define a machine file:* where this file contains the worker RPis for the testing.

- *Run STREAM across the cluster:* To ensure one process per core across all nodes it's needed to modify the "*machinefile*" in such a way so as the MPI to assign one process per core per involved RPi.

Taking into account from the STREAM Variation testing in (1) RPi there was an observation revealing that for both memory bandwidth and execution time, the optimal configuration is ($np = 2$) where this setup achieves the highest bandwidth while maintaining manageable execution times. Despite this observation the whole Beowulf cluster tested for (2, 3, 4 processes) per RPi to evaluate the performance.

As a result, this setup used to STREAM test the Beowulf cluster with the below recommended Value such as:

2 processes per node: DSTREAM_ARRAY_SIZE= 134, 217,728.

Compile: $mpicc -O3 -fopenmp -DSTREAM_ARRAY_SIZE=134,217,728 stream.c -o stream_mpi.

3 processes per node: DSTREAM_ARRAY_SIZE= 89, 478,485.

Compile: $mpicc -O3 -fopenmp -DSTREAM_ARRAY_SIZE=134,217,728 stream.c -o stream_mpi.

4 processes per node: DSTREAM_ARRAY_SIZE=67,108,864

Compile: $mpicc -O3 -fopenmp -DSTREAM_ARRAY_SIZE=67,108,864 stream.c -o stream_mpi.
It's very critical to secure the specific number of MPI processes run in each RPi during the test cases. The above critical condition is achieved by using this command: $ export OMP_NUM_THREADS=YY; mpiexec -f machinefile -np YY --bind-to core ./stream_mpi, where (YY) represents the particular MPI processes related to the number of RPi involved in the test defined in machinefile.

The results of the STREAM benchmark in the whole Beowulf cluster for 2 processes per RPi are addressed in "Table 2" and depicted in a graph in "Figure 9", "Figure 10".

The results of the STREAM benchmark in the whole Beowulf cluster for 3 processes per RPi are addressed in "Table 3" and depicted in a graph in "Figure 11", "Figure 12".

The results of the STREAM benchmark in the whole Beowulf cluster for 4 processes per RPi are addressed in "Table 4" and depicted in a graph in "Figure 13", "Figure 14".

**Table 1. STREAM Benchmark results in one RPi (1 to 4 MPI processes)**

| STREAM Benchmark to 1 RPi (1-4 MPI processes) | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **Cores Used** (MPI Processes) | **Copy:** Best Rate (MB/s) | **Copy:** Avg Time (s) | **Scale:** Best Rate (MB/s) | **Scale:** Avg Time (s) | **Add:** Best Rate (MB/s) | **Add:** Avg Time (s) | **Triad:** Best Rate (MB/s) | **Triad:** Avg Time (s) |
| 1 (48% RAM usage) | 5494.3 | 0.515084 | 5471.4 | 0.52371 | 4544.2 | 0.929131 | 4574.5 | 0.917187 |
| 2 (75% RAM usage) | 5096.2 | 1.04129 | 5351.1 | 0.958763 | 4560.3 | 1.539043 | 4552.9 | 1.513349 |
| 3 (75% RAM usage) | 1624.1 | 1.017769 | 1408.4 | 1.125475 | 1278.7 | 1.803441 | 1872.4 | 1.728948 |
| 4 (75% RAM usage) | 1039.2 | 1.088948 | 1050.5 | 1.07822 | 964.3 | 1.715477 | 989.9 | 1.722957 |

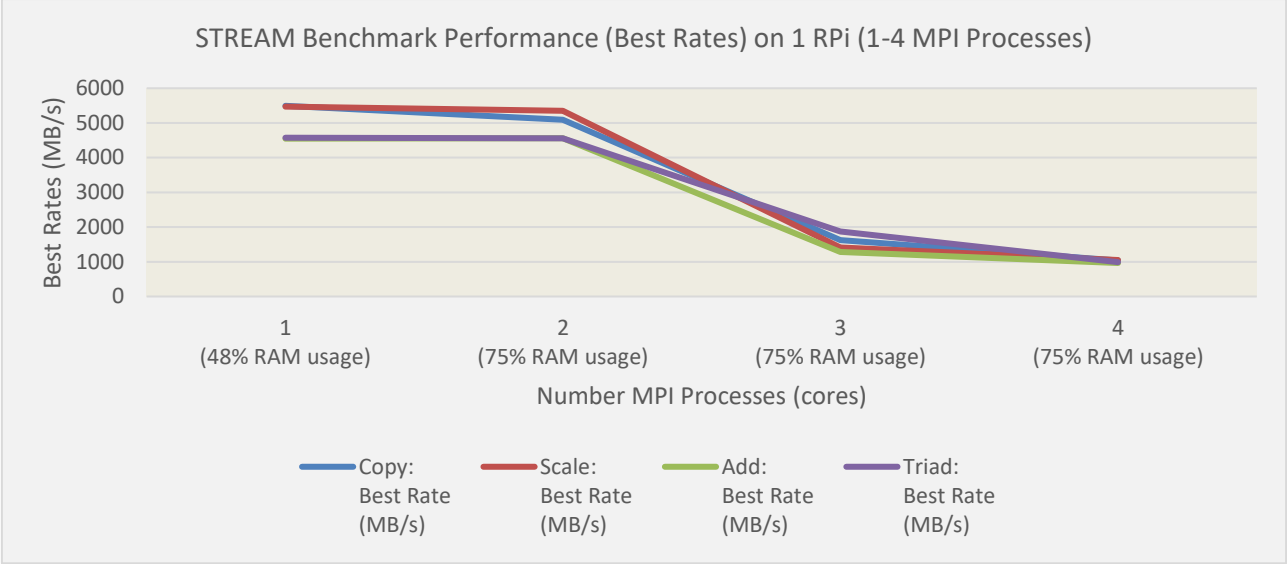


**Figure 7: STREAM Benchmark Performance (Best Rates) on 1 RPi (1 to 4 MPI processes)**

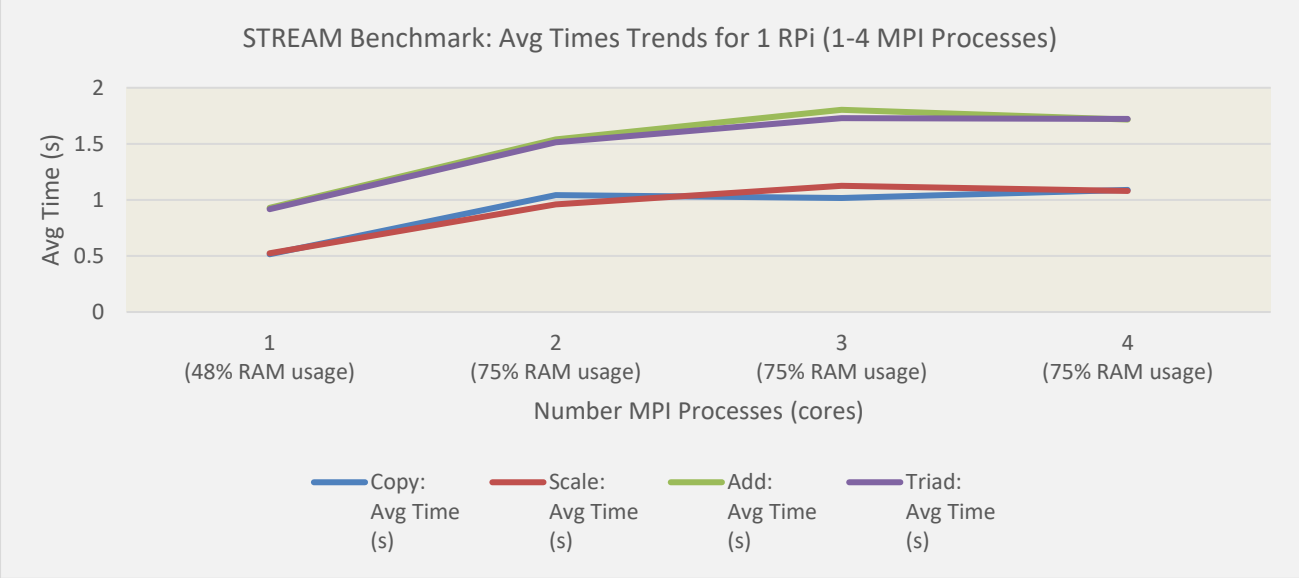


**Figure 8: STREAM Benchmark: Average Times Trends from 1 to 4 MPI processes (cores) in one RPi**

**Table 2. STREAM Benchmark results: Beowulf Cluster, from 1-24 RPi (2 to 48 MPI processes) (2 MPI processes per RPi)**

| **STREAM Benchmark results: Beowulf Cluster, from 1-24 RPi (2 to 48 MPI processes) _2 MPI processes per RPi 75% RAM usage** | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **RPi's** | **Cores Used** (MPI Processes) | **Copy:** Best Rate (MB/s) | **Copy:** Avg Time (s) | **Scale:** Best Rate (MB/s) | **Scale:** Avg Time (s) | **Add:** Best Rate (MB/s) | **Add:** Avg Time (s) | **Triad:** Best Rate (MB/s) | **Triad:** Avg Time (s) |
| 1 | 2 | 5096.2 | 1.04129 | 5351.1 | 0.958763 | 4560.3 | 1.539043 | 4552.9 | 1.513349 |
| 4 | 8 | 5740 | 0.959117 | 5504.2 | 0.87531 | 4894.6 | 1.452127 | 4917.6 | 1.40689 |
| 8 | 16 | 5707.6 | 0.927779 | 5541.2 | 0.895493 | 4919.7 | 1.438768 | 4913.4 | 1.400841 |
| 12 | 24 | 5768.3 | 0.898212 | 4148.3 | 0.981586 | 4221.9 | 1.559666 | 4611.8 | 1.471469 |
| 16 | 32 | 5795.7 | 0.887485 | 5534.6 | 0.872828 | 4883.5 | 1.457799 | 4920.1 | 1.40687 |
| 20 | 40 | 5782.6 | 0.902209 | 3924.1 | 1.010822 | 4121.4 | 1.569748 | 4554.9 | 1.482986 |
| 24 | 48 | 5710.5 | 0.918919 | 4154.4 | 0.975173 | 4221.4 | 1.551108 | 4640.8 | 1.465694 |

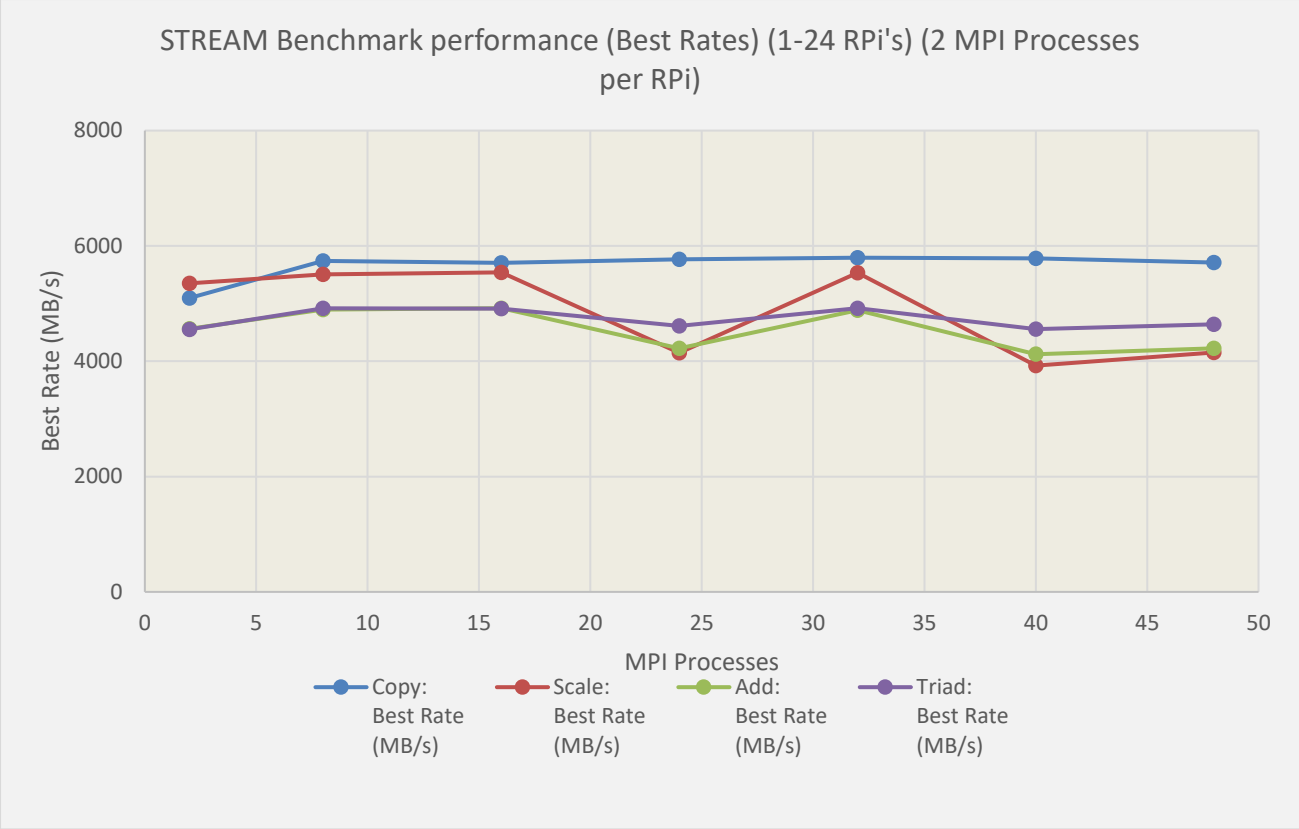


**Figure 9: STREAM Benchmark: Beowulf Cluster (Best Rates) from 2 to 48 MPI processes (cores) (2 MPI processes per RPi)**

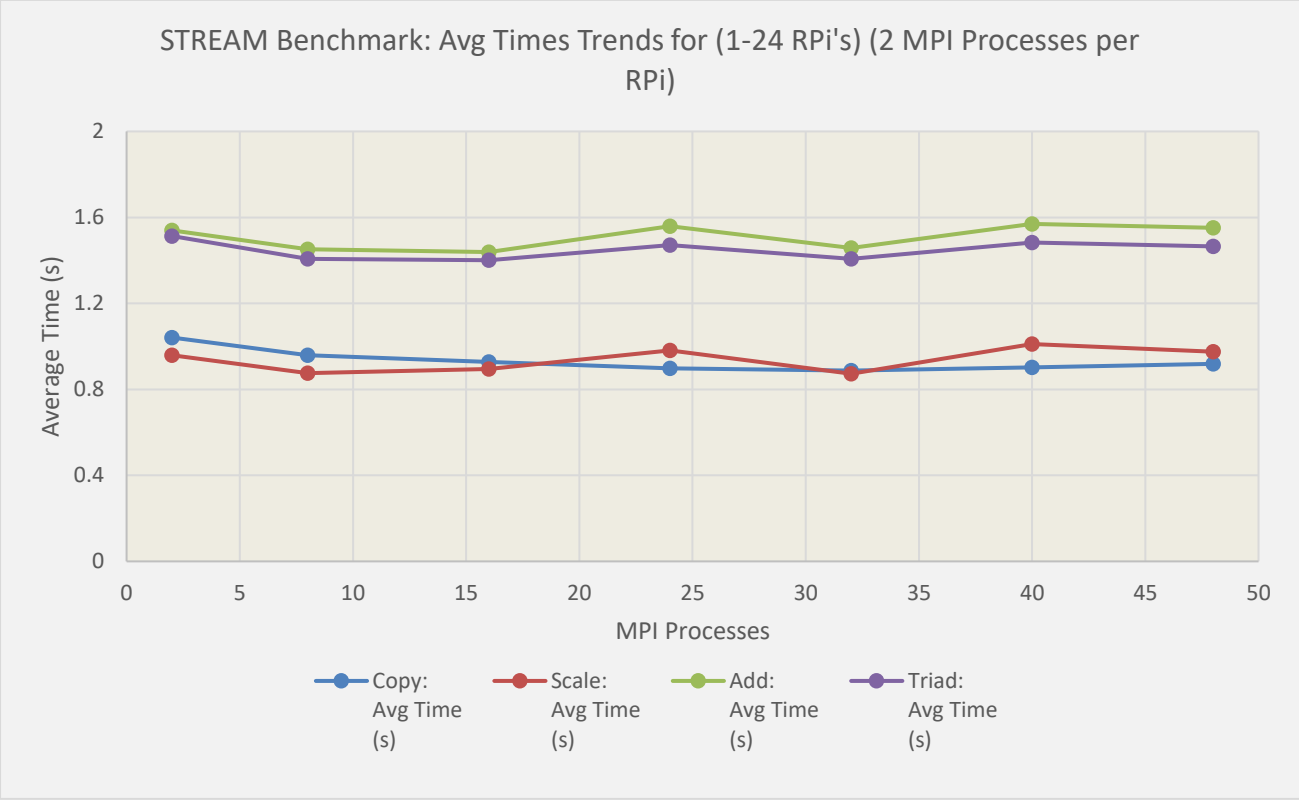


**Figure 10: STREAM Benchmark: Beowulf Cluster Avg Times Trends from 2 to 48 MPI processes (cores) (2 MPI processes per RPi)**

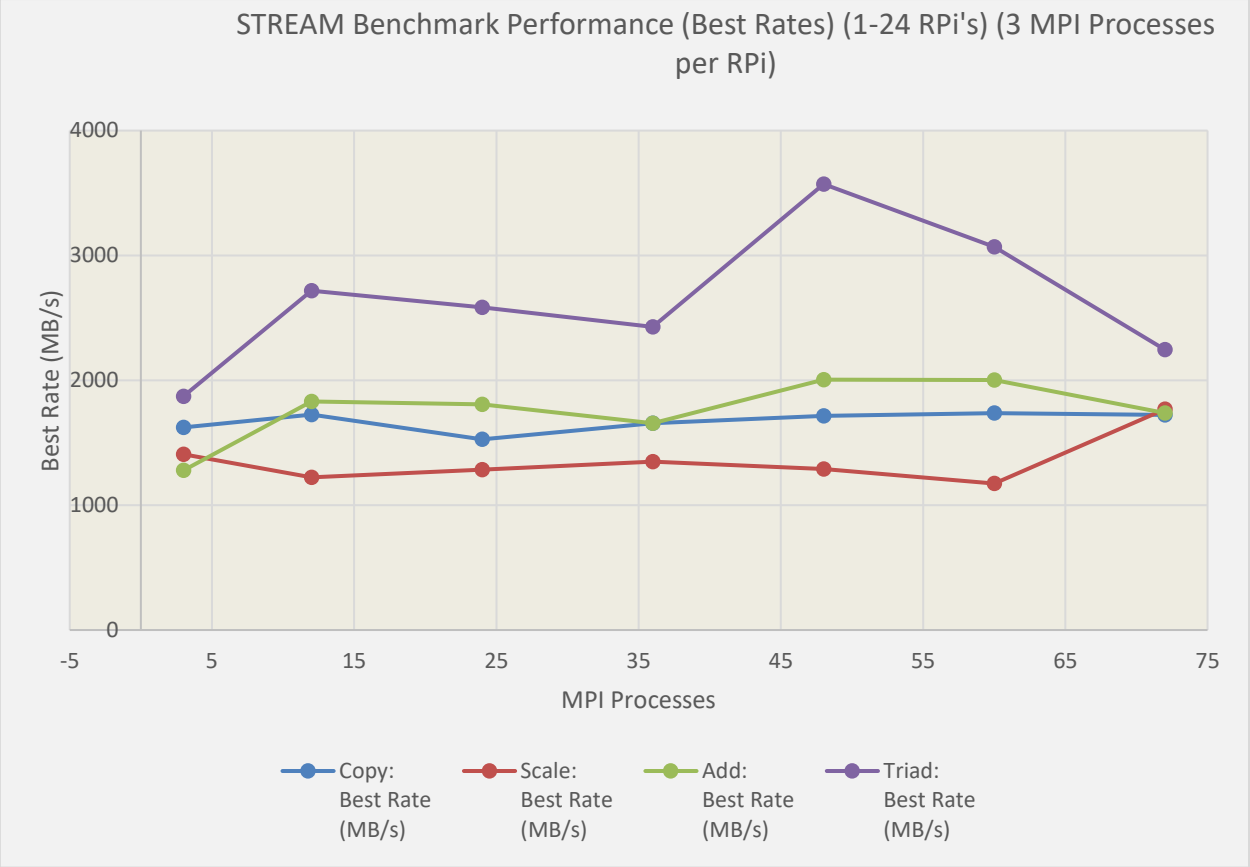


**Figure 11: STREAM Benchmark: Beowulf Cluster (Best Rates) from 3 to 72 MPI processes (cores) (3 MPI processes per RPi)**

**Table 3. STREAM Benchmark results: Beowulf Cluster, from 1-24 RPi (3 to 72 MPI processes) (3 MPI processes per RPi)**

| **STREAM Benchmark results: Beowulf Cluster, from 1-24 RPi (3 to 72 MPI processes) (3 MPI processes per RPi) 75% RAM usage** | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **RPi's** | **Cores Used** (MPI Processes) | **Copy:** Best Rate (MB/s) | **Copy:** Avg Time (s) | **Scale:** Best Rate (MB/s) | **Scale:** Avg Time (s) | **Add:** Best Rate (MB/s) | **Add:** Avg Time (s) | **Triad:** Best Rate (MB/s) | **Triad:** Avg Time (s) |
| 1 | 3 | 1624.1 | 1.017769 | 1408.4 | 1.125475 | 1278.7 | 1.803441 | 1872.4 | 1.728948 |
| 4 | 12 | 1726.5 | 0.973419 | 1222.7 | 1.209682 | 1830.7 | 1.681236 | 2716.4 | 1.941561 |
| 8 | 24 | 1527.3 | 1.018542 | 1285.4 | 1.206808 | 1806.7 | 1.700405 | 2583.2 | 2.005628 |
| 12 | 36 | 1655.6 | 0.983558 | 1349.8 | 1.100449 | 1655.2 | 1.632339 | 2427.2 | 1.668023 |
| 16 | 48 | 1714.5 | 0.992582 | 1289.5 | 1.143349 | 2005.4 | 1.634786 | 3571.4 | 1.744102 |
| 20 | 60 | 1737.1 | 0.971126 | 1173.4 | 1.251281 | 2002.5 | 1.701607 | 3068.7 | 2.054771 |
| 24 | 72 | 1723.8 | 0.988274 | 1767.8 | 1.133976 | 1737.7 | 1.682659 | 2246.6 | 1.887046 |

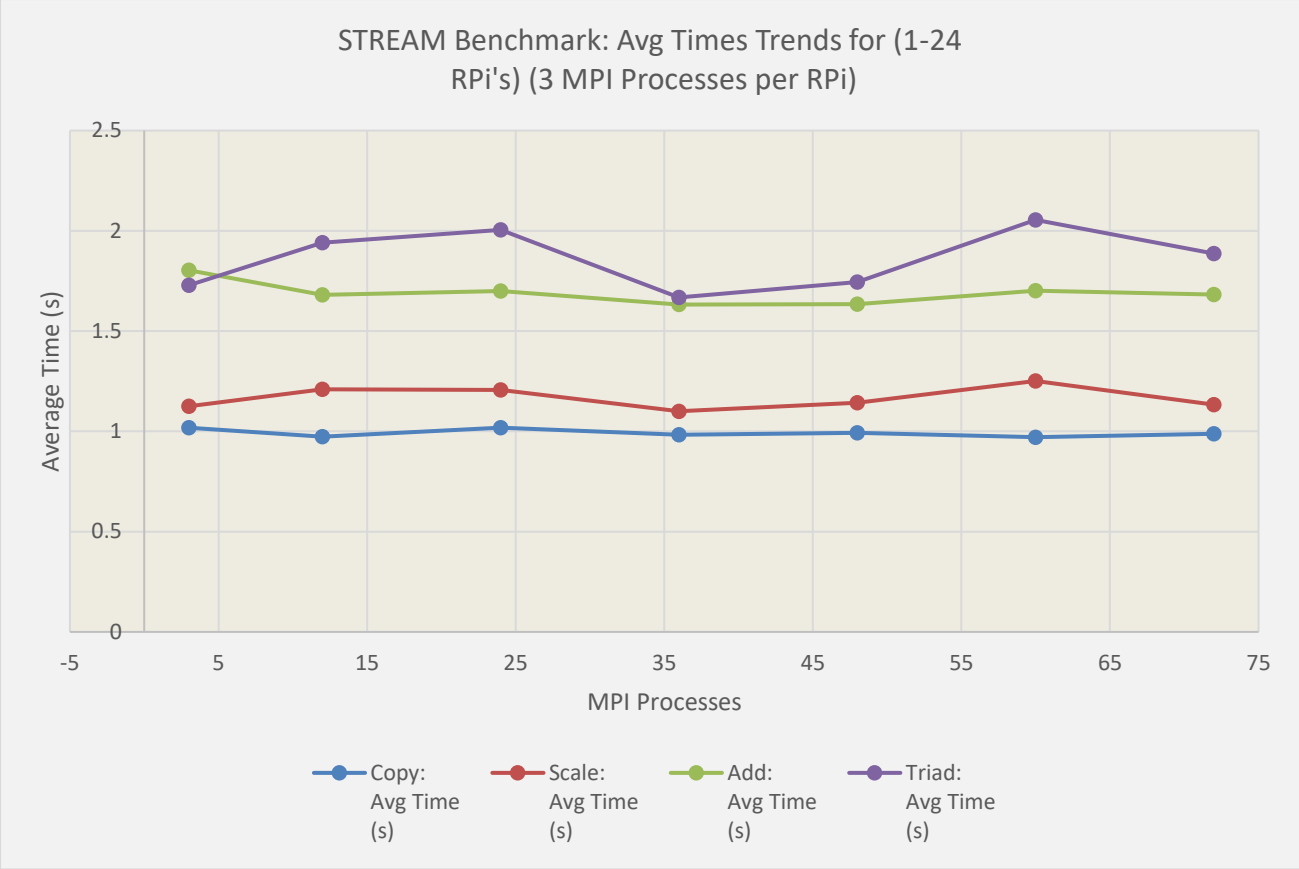


**Figure 12: STREAM Benchmark: Beowulf Cluster Avg Times Trends from 3 to 72 MPI processes (cores) (3 MPI processes per RPi)**

**Table 4. STREAM Benchmark results: Beowulf Cluster, from 1-24 RPi (4 to 96 MPI processes) (4 MPI processes per RPi)**

| **STREAM Benchmark results: Beowulf Cluster, from 1-24 RPi (4 to 96 MPI processes) _4 MPI processes per RPi 75% usage** | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **RPi's** | **Cores Used** (MPI Processes) | **Copy:** Best Rate (MB/s) | **Copy:** Avg Time (s) | **Scale:** Best Rate (MB/s) | **Scale:** Avg Time (s) | **Add:** Best Rate (MB/s) | **Add:** Avg Time (s) | **Triad:** Best Rate (MB/s) | **Triad:** Avg Time (s) |
| 1 | 4 | 1039.2 | 1.088948 | 1050.5 | 1.07822 | 964.3 | 1.715477 | 989.9 | 1.722957 |
| 4 | 16 | 1070.5 | 1.065174 | 1207.7 | 0.953312 | 1029.9 | 1.654204 | 1185 | 1.65954 |
| 8 | 32 | 1119.7 | 1.040162 | 1427.1 | 0.933345 | 1416.1 | 1.69616 | 2069.5 | 1.721273 |
| 12 | 48 | 1131.8 | 1.041562 | 1259.4 | 1.0033 | 1140.7 | 1.798188 | 1711.2 | 1.772907 |
| 16 | 64 | 1296.3 | 1.026871 | 1350 | 0.958608 | 1873.3 | 1.721798 | 3515.8 | 1.655572 |
| 20 | 80 | 1231.1 | 1.048694 | 1176.2 | 1.042699 | 1450.6 | 1.810133 | 2538.3 | 1.758598 |
| 24 | 96 | 1119 | 1.081926 | 1149.9 | 1.020317 | 1113.7 | 1.765204 | 1479.9 | 1.784741 |

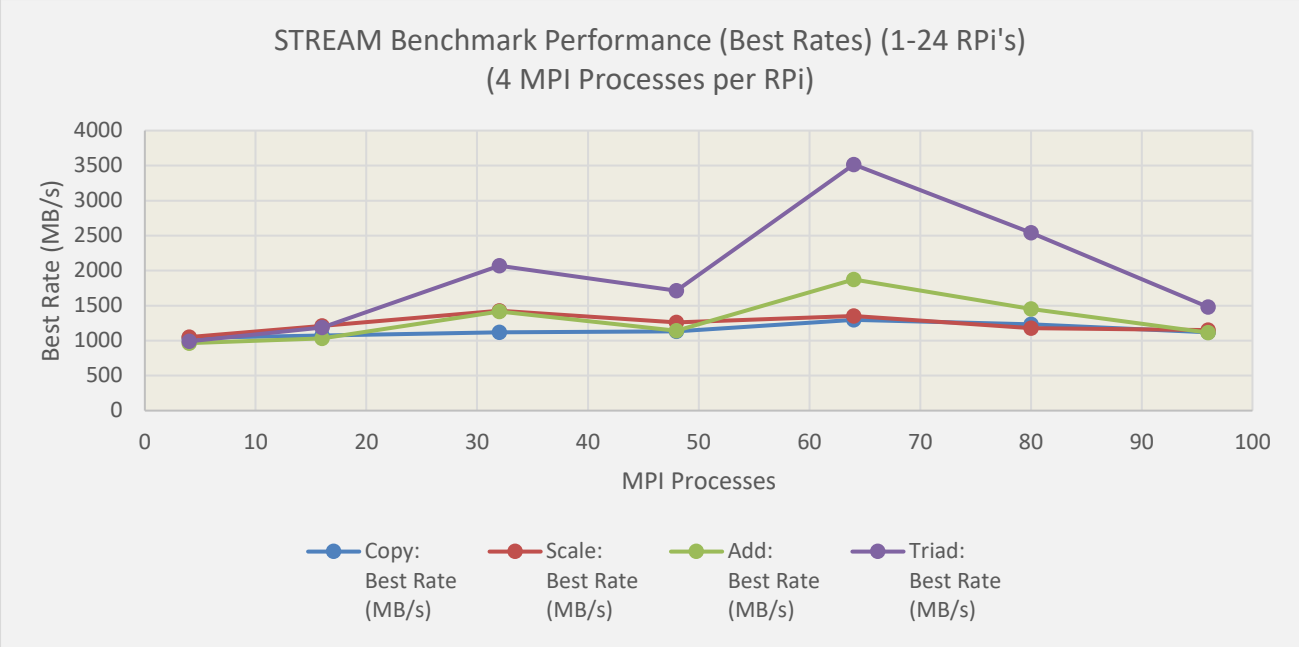


**Figure 13: STREAM Benchmark: Beowulf Cluster (Best Rates) from 4 to 92 MPI processes (cores) (4 MPI processes per RPi)**

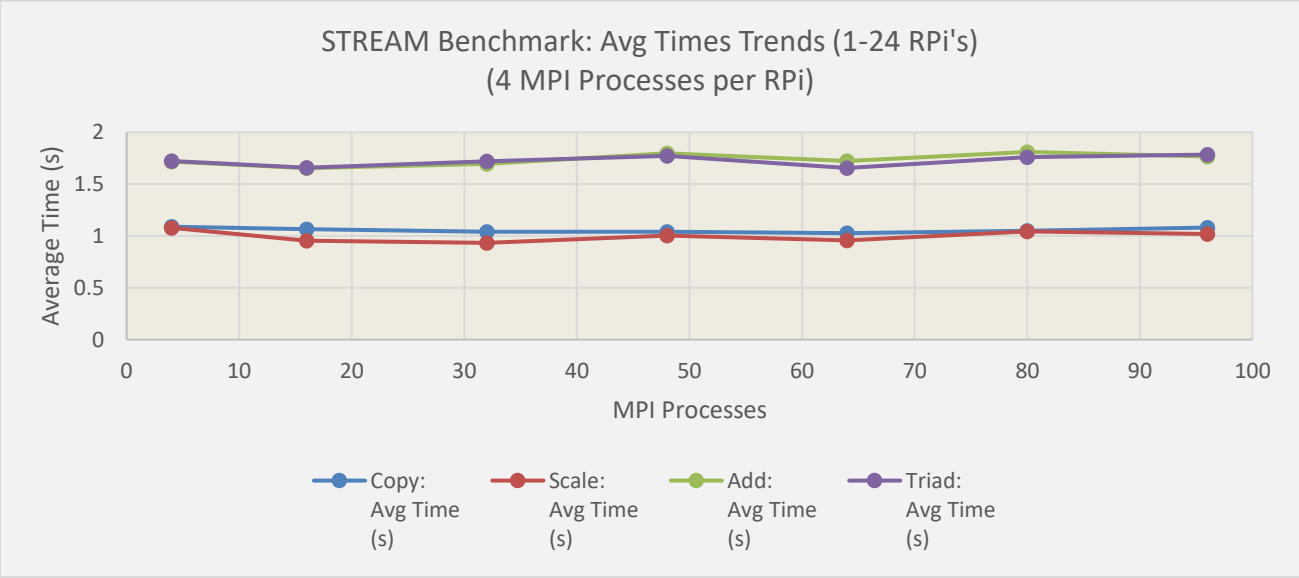


**Figure 14: STREAM Benchmark: Beowulf Cluster (Avg Times Trends) from 4 to 92 MPI processes (cores) (4 MPI processes per RPi)**

## 3.3 Stream Variations Analysis in the whole Beowulf Cluster

STREAM benchmark analysis with 2 MPI processes per RPi in whole Beowulf Cluster, “Table 2”, “Figure 9", “Figure 10”:

- *Performance Scaling:*
  The performance scaling observed in the datasets when 2 MPI processes per RPi in whole cluster follows a trend where memory bandwidth improves as the number of nodes increases. The *Copy* (Best Rate) starts at 5096.2 MB/s for 2 MPI processes (1 RPi) and peaks at 5795.7 MB/s at 32 MPI processes (16 RPi). Similarly, the *Scale*, *Add*, and *Triad* (Best Rates) generally increase as more nodes contribute resources. However, performance stabilizes beyond 16 RPi (32 MPI processes), suggesting diminishing returns due to network communication overhead or memory access limitations.
  At 48 MPI processes (24 RPi), the performance for *Copy* (Best Rate) (5710.5 MB/s) and *Triad* (Best Rate) (4640.8 MB/s) remains close to peak values, but Scale (4154.4 MB/s) and *Add* (4221.4 MB/s) show minor regression. This implies that for some computational tasks, inter-node communication or system contention might be affecting performance at full system utilization.

- *Bottleneck Identification:*
  Inter-Node Communication Overhead: As the number of MPI processes increases, inter-node communication grows

significantly. At 24 RPi (48 MPI processes), communication latency and network congestion start outweighing computational benefits, leading to slower scaling of the bandwidth gains.

Memory Bandwidth Utilization: The results suggest that memory bandwidth saturation starts beyond 16 RPi, particularly for the Scale and Add operations. This is likely due to the single LPDDR4-3200 memory channel in Raspberry Pi 4B devices, which becomes a limiting factor in sustaining performance across high MPI counts.

Process Scheduling Inefficiencies: The Linux scheduler may struggle to optimally allocate CPU resources when many MPI processes run concurrently, resulting in process migrations or unnecessary context switching
.

- *Efficiency:*
  Memory Bandwidth Efficiency: The best bandwidth efficiency is seen around 16-32 MPI processes, (8-16 RPis) where memory utilization is high, and performance scaling is still benefiting from increased parallelism.

  Network Efficiency: Since a Gigabit Ethernet switch is used, interconnect efficiency may degrade as more nodes communicate, leading to increased latency in memory access for remote processes.

  Computational Efficiency: Efficiency starts declining at 48 MPI processes, as seen from the slower scaling of *Copy* and *Scale* operations. This suggests that the cluster may not optimally utilize additional computational resources beyond a certain point due to bottlenecks in interconnects and memory access.

As a conclusion, the STREAM benchmarking analysis with the setup of using 2 MPI processes per RPi shows that scaling is effective up to 16-20 RPi (32-40 MPI processes), after which performance stagnation begins. Memory contention, inter-node communication, and cache limitations emerge as primary bottlenecks when running at full capacity. To optimize efficiency, load balancing, and MPI communication strategies (such as message aggregation), and cache-aware computation techniques should be explored.

STREAM benchmark analysis with 3 MPI processes per RPi in whole Beowulf Cluster, “Table 3”, “Figure 11", “Figure 12”:

- *Performance Scaling:*
  The performance scaling when using 3 MPI processes per RPi shows a non-linear trend across different benchmarks as the number of MPI processes increases. Initially, there is a slight improvement in performance, but after a certain threshold, efficiency drops due to increasing communication overhead and memory contention.
  Copy Operation: The best rate starts at 1624.1 MB/s (1 RPi, 3 MPI processes) and gradually increases up to 1737.1 MB/s (20 RPis, 60 MPI processes) before slightly declining to 1723.8 MB/s at 24 RPis, 72 MPI processes. The performance scaling shows some improvement but is limited by memory bandwidth saturation and inter-node communication overhead.

  Scale Operation: The best rate fluctuates slightly, showing an increasing trend up to 20 RPis, 60 MPI processes (1173.4 MB/s) before dropping at 24 RPis, 72 MPI processes (1767.8 MB/s). This suggests that data movement efficiency is hindered by increased thread synchronization requirements.

  Add Operation: The best rate follows a similar trend, peaking at 20 RPis, 60 MPI processes (2002.5 MB/s) before dropping to 1737.7 MB/s at 24 RPis, 72 MPI processes. This suggests increased overhead from excessive memory access operations.

  Triad Operation: The best rate increases steadily, reaching 3571.4 MB/s at 16 RPis, 48 MPI processes, before declining to 2246.6 MB/s at 24 RPis, 72 MPI processes. This suggests a major bottleneck in memory access latency and cache contention.

- *Bottleneck Identification:*
  Memory Bandwidth Saturation: The Raspberry Pi 4B is limited by its LPDDR4-3200 memory, which cannot sustain high-performance demands as the number of parallel processes increases. The *Copy* and *Scale* best rates plateau around 1700 MB/s, indicating that memory throughput has reached its hardware-imposed limit.

  Inter-Node Communication Overhead: With a Gigabit Ethernet interconnect, higher MPI processes (above 48-60 MPI) create excessive data exchange between nodes. *Add* and *Triad* operations demonstrate variations beyond 60 MPI processes.

  Process Synchronization Delays: As more MPI processes are introduced per node, synchronization overhead increases, leading to reduced performance scaling. Increase in avg times (s) for Scale and Add operations confirms synchronization inefficiencies

- *Efficiency:*
  Best Efficiency Observed at 16-20 RPis (48-60 MPI Processes): The Copy and Scale operations maintain consistent best rates (~1700 MB/s), suggesting this is the most optimal configuration before performance degradation. The *Add* and *Triad* operations peak at 16 RPis (48 MPI processes), indicating that this is the best balance of compute and memory access efficiency.
  Efficiency Declines Beyond 72 MPI Processes: The increasing average execution times suggest that adding more MPI processes does not translate into proportional speedups. The best rate declines after 60 MPI processes which suggests that network latency, memory contention, and process scheduling inefficiencies limit further scalability.
  .

As a conclusion the performance results when using 3 MPI processes per RPi, highlight the challenges of scaling beyond a certain threshold in a Raspberry Pi-based Beowulf cluster. While there is an initial increase in memory bandwidth up to 16 RPis (48 MPI processes), the overall efficiency deteriorates beyond this point due to cache contention, memory bandwidth saturation, and MPI communication overhead. The *Triad* and *Add* benchmarks show peak performance at intermediate scales but suffer from inter-process synchronization inefficiencies at higher process counts. These findings emphasize the importance of optimizing workload distribution and memory access patterns to enhance the scalability of Raspberry Pi clusters in high-performance computing applications.

STREAM benchmark analysis with 4 MPI processes per RPi in whole Beowulf Cluster, “Table 4”, “Figure 13", “Figure 14”:

- *Performance Scaling:*
  The performance trend when using 4 MPI processes per RPi shows diminishing returns as the process count increases. Up to 16 RPis (64 MPI processes), there is a moderate improvement in bandwidth for *Copy*, *Scale*, *Add*, and *Triad* operations. However, beyond 16 RPis, performance fluctuates and slightly declines in some cases, particularly in *Scale* and *Triad* benchmarks. The *Copy* (Best Rate) peaks at 1296.3 MB/s at 16 RPis but then drops slightly, showing inefficiencies in memory handling. Triad performance fluctuates significantly, with a peak at 16 RPis (3515.8 MB/s), but declines to 1479.9 MB/s at 24 RPis.

- *Bottleneck Identification:*
  Several bottlenecks impact when using 4 MPI processes per RPi performance based on the observations such as:
  Memory Bandwidth Saturation: The LPDDR4-3200 memory channel limits further gain in performance despite increased MPI processes.
  Interconnect Latency: As the number of MPI processes grows, communication overhead due to inter-node messaging increases, introducing additional latency.
  Process Scheduling Overhead: The Linux scheduler distributes workloads sub-optimally at high MPI counts, further affecting efficiency.

- *Efficiency:*
  The efficiency when using 4 MPI processes per RPi configuration declines significantly beyond 16 RPis (64 MPI processes). The increasing execution time in Scale and Add operations suggests higher contention in memory access. Furthermore, Triad and Copy bandwidth reduction at 24 RPis (96 MPI processes) indicates that system resources are overloaded with excessive parallelism, leading to diminishing computational returns.

When using 4 MPI processes per RPi, this setup demonstrates moderate scaling improvements up to 16 RPis (64 MPI processes), performance beyond this point stagnates or declines due to memory bandwidth limitations, interconnect overhead, and cache contention. The findings suggest that increasing MPI processes beyond an optimal threshold does not provide additional performance benefits in a Raspberry Pi-based cluster. Optimizing process affinity, communication patterns, and workload distribution could help mitigate these scaling issues and improve efficiency.

# 4. CONCLUSION

The final STREAM Benchmark results for the entire Beowulf cluster synthesize the insights obtained from the single-node (RPi-1) and multi-node reflecting how the Beowulf cluster scales with increasing MPI processes and nodes. The key focus is on Performance *Scaling*, *Bottleneck Identification*, and *Efficiency* across different configurations.

With a single RPi, the best bandwidth performance occurs at 1-2 MPI processes, with performance degradation beyond 2 MPI processes. Severe performance drops at 3 and 4 MPI processes due to resource contention

In summary, for the cluster performance, utilizing 2 MPI processes per RPi, 3 MPI processes per RPi, and 4 MPI processes per RPi in the whole cluster we have:

2 MPI processes per RPi in cluster: based on observations this setup achieves the best balance between performance and resource utilization.

3 MPI processes per RPi in cluster: based on observations this setup shows diminishing returns, with stagnation in key benchmarks due to memory and interconnect limitations.

4 MPI processes per RPi in cluster: This setup exhibits performance regression, indicating that increasing MPI processes beyond 48 does not yield further gain.

In terms of bottleneck identification based on observations a Memory Bandwidth Saturation appears **e**xceeding LPDDR4-3200 limits efficient scaling. Cache Contention appears when more processes cause increased cache misses and memory stalls. It may be an Interconnect Overhead when growing MPI communication costs reduce efficiency and maybe a process scheduling issues shows up interpreting that Linux struggles with optimal resource allocation at higher process counts.

# 5. FUTURE WORK

For future work, a comprehensive investigation into the *HPCG benchmark* performance on the Beowulf cluster could provide deeper insights into its computational efficiency and memory bandwidth utilization under realistic workloads. This research should focus on evaluating the interplay between communication overhead and computational intensity across increasing MPI processes, particularly for sparse matrix operations. Additionally, profiling the energy consumption and thermal behavior during HPCG tests can help optimize the cluster for power-efficient high-performance computing. Comparing HPCG results with STREAM metrics would also offer a clearer understanding of bandwidth-bound versus compute-bound performance characteristics in the cluster.

# 6. ACKNOWLEDGMENTS

My sincere gratitude to Assistance Professor Ioannis S. Barbounakis for the precious guidelines, knowledge and contribution for the completion of this research.

# 7. REFERENCES

[1] Z. Xu, W. Zhang, and A. Y. Zomaya, "A heterogeneous platform with GPU and FPGA for power-efficient high-performance computing," *2014 IEEE International Symposium on Integrated Circuits (ISIC)*, 2014, pp. 1-4, doi: 10.1109/ISIC.2014.7029447.

[2] C. Pilato, H. Patel, and J. Teich, "Heterogeneous computing utilizing FPGAs," *Journal of Signal Processing Systems*, vol. 90, no. 3, pp. 471-482, 2018, doi: 10.1007/s11265-018-1382-7.

[3] Raspberry Pi 4 Model B. [Online]. Available: raspberrypi.com/products/raspberry-pi-4-model-b/.

[4] Raspberry Pi 4 Model B specifications. [Online]. Available: https://magpi.raspberrypi.com/articles/raspberry-pi-4-specs-benchmarks

[5] McCalpin, J. D. (1995). Memory bandwidth and machine balance in current high-performance computers. *IEEE Technical Committee on Computer Architecture (TCCA) Newsletter*. Retrieved from https://www.cs.virginia.edu/stream/ref.html

[6] Henning, S., & Hasselbring, W. (2023). Benchmarking Distributed Stream Data Processing Systems. *arXiv*

*preprint arXiv:2303.11088*. Retrieved from *https://arxiv.org/pdf/1802.08496*

[7] Gupta, N., Brandt, S. R., Wagle, B., Nanmiao, Kheirkhahan,A., Diehl, P., Kaiser, H., & Baumann, F. W. (2020). *Deploying a Task-based Runtime System on Raspberry Pi Clusters.* arXiv preprint arXiv:2010.04106

[8] Fridman, Y., Desai, S. M., Singh, N., Willhalm, T., & Oren, G. (2023). *CXL Memory as Persistent Memory for Disaggregated HPC: A Practical Approach.* arXiv preprint arXiv:2308.10714.

[9] Dimitrios Papakyriakou, Ioannis S. Barbounakis. High Performance Linpack (HPL) Benchmark on Raspberry Pi 4B (8GB) Beowulf Cluster. International Journal of Computer Applications. 185, 25 (Jul 2023), 11-19. DOI=10.5120/ijca2023923005