# Contents

**Executive Summary** 3

1 Evaluation of the MPI Performance on the Arm-based cluster Dibona 4
   1.1 Sequana ARM Architecture - Dibona Cluster 4
   1.1.1 Compute nodes architecture 4
   1.1.2 Interconnection topology 5
   1.2 Software Environment Description 6
      1.2.1 Operating System 6
      1.2.2 MPI Library 6
      1.2.3 Compilers 6
   1.3 Experiment Description 6
      1.3.1 MPI Benchmark Suite 6
      1.3.2 MPI Task Mapping Strategy 7
   1.4 Experiment Results 7
      1.4.1 Point-to-point communication 7
      1.4.2 Collective communication 9
      1.4.3 One-sided communication 11
   1.5 Conclusion 13

2 Tasking support for MPI communications 14
   2.1 TinyMPI: a virtualized MPI implementation 14
   2.2 Dedicated communication thread for MPI-heavy tasks 14
Executive Summary

This document summarises part the work done on extensions to the message passing model within Work Package 7 of the Mont-Blanc 3 project.

The largest part of the document is taken up by a detailed experimental evaluation of the performance of MPI libraries on the project’s test platform Dibona. Particular focus is placed on the memory copy savings potential due to RDMA and direct access capabilities of the platforms hardware.

The second part of the documents is dedicated to a description of tasking support for MPI communication, though most of this effort has been reported elsewhere more comprehensively. In this document, we add in particular an approach to dedicate an OmpSs thread to do MPI communication and the problems that were found while doing the evaluation on the test platform Dibona. An experimental evaluation was done as part of Work Package 6 and reported in D6.9.
1 Evaluation of the MPI Performance on the Arm-based cluster Dibona

Following what was stated in Mont-Blanc 3 Deliverable D7.7 - Intermediate report on enhancements to message passing [1], this section includes the MPI experiments performed on the Dibona platform. The original plan for MPI evaluation and comparison was updated to reflect the current hardware and software environment.

1.1 Sequana ARM Architecture - Dibona Cluster

As described in the Deliverable D3.3 of the Mont-Blanc 3 project [2], the Arm-based cluster named Dibona delivered by Bull as part of its commitments to the Mont-Blanc 3 Project is based on Bull’s Sequana architecture. Sequana is a flexible system supporting a variety of interconnection and processor technologies. The Dibona cluster specifically, uses Infiniband EDR interconnect provided by Mellanox and the nodes are equipped with two Cavium ThunderX2 processors. The detailed characteristics of the components of this cluster are provided in the following subsections.

1.1.1 Compute nodes architecture

The logical block diagram in Figure 1 below shows the major functions of a Sequana Arm compute node. Each node has:

- 2 ThunderX2 (CN99xx) ARM v8.1 Cavium processor
  - 32 ARM v8.1 custom cores running at 2GHz
    - SMT 4 threads
    - 32KB L1 instruction cache
    - 32KB L1 data cache
    - 256KB L2 cache
  - 32MB distributed L3 cache (1MB per core)
  - 3 coherent ICI interfaces for the 2-socket configuration
- 16 DDR4 DIMM slots
  - 8 channels per socket
  - 1 16 GiB memory per channel, DDR4-2667MHz
  - 256 GiB total memory per node (8 x 2 x 16)
- 32x PCI Express lanes
  - routed to the mezzanine PCI Express
- Infiniband EDR
  - 1 interconnect channel up to 100 Gbps
  - to the interconnect L1 switch
  - internally connected PIC bus on CPU0
Currently, there are 27 dual-socket nodes available on the Dibona platform.

1.1.2 Interconnection topology

To provide a 45-node system, there is only one compute rack and one switch rack in the cell. The compute rack is made of 15 blades of 3 nodes, which are located only on front side of base compute cabinet.

In this configuration 2 L2 switches and 4 L1 switches are used in the switch rack. Figure 2 shows the topology of a 45-node Sequana cell. As seen in the figure it is fat-tree configuration. The Dibona cluster itself is not fully populated, it could accommodate 48 nodes in total. However it still implements a non-blocking fat-tree, which means that for each node connection to a L1 switch there is an uplink connection to a L2 switch.

Some items were intentionally omitted, to see the complete description please refer to the original document [2].

Figure 1: Logical block diagram of a Sequana Arm motherboard.

Figure 2: 45-node Sequana Arm cell IB EDR topology.
1.2 Software Environment Description

This section describes the software environment installed on Dibona, which includes compilers, operating system and kernel versions. It also describes the selected benchmarks that represent the three basics communication patterns.

1.2.1 Operating System

The nodes have the RedHat Enterprise Linux 7.5 for Aarch64 installed, that includes the Linux Kernel 4.14.0-49.2.2 patched to support perf uncore events and also to implement dynamic voltage and frequency scaling – DVFS. This kernel includes the Mellanox OFED module version 4.3-3.0.2.1.

1.2.2 MPI Library

The experiments shown in this report use two versions of the Open MPI library. The first is version Open MPI 2.0.2 compiled with multi-threading enabled and UCX support. The second version is Open MPI 3.1.2 also compiled with multi-threading enabled and UCX support.

The UCX framework, as described by the Unified Communication Framework Consortium[^2] is “a collaboration between industry, laboratories, and academia to create an open-source production grade communication framework for data centric and high-performance applications. UCX is performance oriented for low-overheads in communication path allowing near native-level performance while enabling cross platform unified API supporting various network Host Card Adaptors (HCAs) and processor technologies (x86, ARM and PowerPC).”

The libraries above were build using the compilers listed in the section below.

1.2.3 Compilers

Both the MPI library and the benchmark suite were build using the two compilers available on the Dibona platform: GNU and Arm compilers. The GNU compiler is GCC 8.3.0 and the Arm compiler is the Arm HPC Compiler 18.4.2 which is based on the LLVM 5.0.1 compiler.

1.3 Experiment Description

This section describes the coverage of the tests and also the MPI task binding for intra- and inter-node communication.

1.3.1 MPI Benchmark Suite

To evaluate the MPI communication performance several micro-benchmarks were run. The micro-benchmarks were selected based on the communication pattern and the MPI primitive it uses. The benchmark suites used is the Ohio State University MPI Benchmarks (commonly referred to as OSU benchmarks[^3]).

The selected micro-benchmarks evaluate latency and bandwidth of the basic communication patterns that were selected for analysis:

- Point-to-point communication (blocking and non-blocking)
- Collective communication (blocking and non-blocking)

[^2]: http://www.openucx.org
[^3]: http://mvapich.cse.ohio-state.edu/benchmarks/
• One-sided communication (standard and requested-based RMA operations)

1.3.2 MPI Task Mapping Strategy

In order to cover the different topology configurations several binding maps were used. As the target system is composed of 45 nodes with two 32-core processors each connected to a fat-tree fabric, the communication between any two of the 45 nodes will have the same communication characteristics such as latency and bandwidth. However, there is an intrinsic difference between the communication from CPU0 to CPU1. This difference exists because of the location of the Infiniband card, which is on the CPU0 PCI Express link.

Therefore, the following point-to-point communication possible scenarios we considered are:

• 2 MPI tasks each one running on a different core in the same CPU
• 2 MPI tasks running on 2 cores on different CPUs at the same node
• 2 MPI tasks running on a core in CPU0 at different nodes

1.4 Experiment Results

The OSU benchmarks were compiled using GNU compiler version stated above and also using the aforementioned Arm compiler. For both compilers the standard compilation flags were used, which are \
\[\text{-g -O2}\].

For the single node experiments, the different Open MPI transport layers were selected using the command line options below:

sm: --mca btl sm, self --mca btl_sm_use_knem 0 --mca btl_sm_use_cma 0
vader: --mca btl vader, self --mca btl_vader_single_copy_mechanism 3
vader-cma: --mca btl vader, self --mca btl_vader_single_copy_mechanism 1

Experiments were run using the three communication scenarios described before. UCX transport layer is only available for inter-node communication (across different nodes). For intra-node communication the MPI library uses a kernel module transport layer to avoid extra memory copies. For the dual-node experiments, the different Open MPI transport layers were selected using the command line options below:

openib: --mca btl openib, self
ucx: --mca pml ucx

1.4.1 Point-to-point communication

This section shows the latency and bandwidth performance for point-to-point MPI communication according to different compilers, MPI library version and transport layer.

As shown in figure 3 the compiler choice makes no difference on the communication performance. However, it is possible to notice that the communication between two MPI tasks running on the same processor have a slightly better performance when compared with two MPI tasks running on different processors in the same node. This performance difference is due to the fact that, even while using a 1-copy shared memory kernel module to improve intra-node communication, the data has to be copied through the 24x ICI from the memory banks
connected to processor in which the sender MPI task resides to the memory bank connected to the processor in which the receiver MPI task is running. Therefore, this operation add a small performance penalty to the MPI communication.

The **sm** transport layer is only available on Open MPI version 2.x. The **vader** transport layer has been introduced to Open MPI starting on version 1.8, and replaced completely the **sm** in the version 3.x. Therefore, on Open MPI 3.x the only intra-node transport layer that uses 1-copy acceleration is the **vader** transport layer. There are multiple options to run the **vader** transport layer. In figure 3 the performance of point-to-point MPI communication using **vader** configured with Open MPI’s implementation of shared memory. As we can see, in general, Open MPI 3.x performs better than Open MPI 2.x. Moreover, the same performance behaviour appointed in figure 3 regarding the difference between MPI tasks running on the same processor versus MPI tasks running on different processors is present when using the **vader** transport layer, as shown in figure 4.

There is a kernel module implementation to implement 1-copy intra-node communication that uses Cross Memory Attach[4] and is available in Open MPI 2.x and also in Open MPI 3.x named **vader-cma**. The results of experiment run using this transport layer are presented in figure 5. As shown in the figure, for small message sizes, the performance of the **vader-cma** is similar to the performance of **vader**. However, for message sizes above 4KiB, specially above 64KiB (the configured virtual memory page size) the **vader-cma** transport layer performs slightly better than the **vader** transport layer.

In with regards to inter-node communication, figure 6 depicts the MPI communication

![Figure 3: OSU point-to-point communication between two MPI tasks running in the same node using sm transport layer.](image)

![Figure 4: OSU point-to-point communication between two MPI tasks running in the same node using vader transport layer.](image)
performance between two MPI tasks running in different nodes using either openib and ucx transport layer. For small message sizes (under 16KiB), regardless the compiler, the version of the MPI library and transport layer used in the communication, the experienced performance is the same for all cases. However, for message sizes starting at 16KiB, the openib transport layer of Open MPI 2.x has a significant performance loss, this behaviour is not noticed in Open MPI 3.x as its implementation has been improved in the new version of the library. Strangely, for message sizes between 64KiB and 128KiB, while using the ucx transport layer in Open MPI 3.x has reduced the performance when compared to Open MPI 2.x. This performance loss is not expected and is under investigation.

1.4.2 Collective communication

This section shows the latency and bandwidth performance for collective MPI communication according to different compilers, MPI library version and transport layer.

In MPI, collective communication is a composition of point-to-point communication arranged in a specific order according to the collective protocol used. This collective protocol can be chose by the user or can be automatically selected by the MPI library. In general, the automatic collective protocol selection depends on the message size and on the number of MPI tasks involved in the communication. The experiments in this section where run using the standard MPI configuration, letting to the MPI library to select the best collective protocol to use.

As expected, and due to the nature of MPI collective communication, the performance
behaviour experienced on point-to-point communication is replicated on the collective MPI communication results:

- Figure 7 shows the slightly better performance for intra-node communication of MPI tasks running on the processor when compared to tasks running on different processors;

- Figure 8 shows that Open MPI 3.x performs better than Open MPI 2.x when using vader transport layer plus the difference in performance when the MPI tasks are running on the same processor or not;

- Figure 9 that for small message sizes, the performance of the vader-cma is similar to the performance of vader;

- Figure 10 shows that for small message sizes (under 16KiB), regardless the compiler, the version of the MPI library and transport layer used in the communication, the experienced performance is the same for all cases except for message sizes starting at 16KiB where the openib transport layer of Open MPI 2.x has a significant performance loss. Also, for messages sizes between 64KiB and 128KiB, while using the ucx transport layer in Open MPI 3.x has reduced the performance when compared to Open MPI 2.x.
1.4.3 One-sided communication

This section shows the latency and bandwidth performance for one-sided MPI communication according to different compilers, MPI library version and transport layer.

As expected, due to the nature of the one-sided communication, there is no difference in performance when the communication is restricted to the same node, regardless the Open MPI version or the compiler used. This similar behaviour for all the transport layers is shown in figures 11, 12 and 13.

Figure 9: OSU collective communication between two MPI tasks in the same node using vader-cma transport layer.

Figure 10: OSU collective communication between two MPI tasks running in different nodes using either openib and ucx transport layer.

Figure 11: OSU one-sided communication between two MPI tasks in the same node using sm transport layer.
Figure 12: OSU one-sided communication between two MPI tasks in the same node using vader transport layer.

Figure 13: OSU one-sided communication between two MPI tasks in the same node using vader-cma transport layer.

On the other hand, it is possible to notice a general improvement on the performance of the one-sided MPI communication from Open MPI 2.x to Open MPI 3.x. The only exception is the achievable bandwidth for small messages sizes (below 512KiB). This unexpected and undesirable behaviour is under investigation inside Bull and the network provider.

Figure 14: OSU one-sided communication between two MPI tasks running in different nodes using either openib and ucx transport layer.
1.5 Conclusion

Comparing the results presented in D7.5 to those in this document, shows that the performance of MPI communication on Arm platform has clearly improved over the course of the project.

Most of the performance improvement (an order of magnitude approximately) is due to the implementation of high performance transport layers such as vader and vader-cma and the availability of Linux kernel modules which allow these transport layer to work efficiently on the Arm platform. There is also improvement of the OpenIB drivers provided by the network vendor (Mellanox). While compilers did improve significantly from 2015 to now, the performance observed on the MPI library itself is marginal and cannot be noticed on the benchmark results presented in this deliverable. However, it is possible to notice a great performance improvement on large MPI applications presented in the deliverable D6.9. The replacement of accelerator drivers for Infiniband, specifically MXM by UCX, is also responsible for the performance improvement (7% approximately).

With the aforementioned, it is possible to say the MPI performance on Arm is similar to the MPI performance experienced on other architecture such as x86.
2 Tasking support for MPI communications

The project has been following three approaches related to tasking support for MPI communications:

- OmpSs-MPI interoperability library
- TinyMPI
- Dedicated communication thread for MPI-heavy tasks

The OmpSs-MPI interoperability library, together with its supporting features as external task execution synchronisation and polling services, have been described at length in D6.4 [7] and D7.7 [1]. Similarly, the virtualised MPI implementation, TinyMPI, has been described at length in [8] and is thus only briefly summarised in Sec. 2.1 of this document. The remainder of this section, i.e. Sec. 2.2, is dedicated to a description of employing a dedicated communication thread within the OmpSs runtime for MPI-heavy tasks.

2.1 TinyMPI: a virtualized MPI implementation

Computation–communication overlap and good load balance are two very important prerequisites for achieving high performance of parallel programs, even more so on large-scale machines with high core counts, such as the architecture envisaged by the Mont-Blanc project.

Unfortunately, those two traits are hard to achieve with MPI [9], the de facto standard interface for network communication in HPC software, because this requires the use of the nonblocking interface of MPI, which makes the code more complex and hard to maintain.

An alternative solution is to use a virtualized MPI implementation. Traditionally, at most one MPI rank is launched per CPU core, but a virtualized MPI would launch more than one rank per core and keep switching between them. When one MPI rank blocks in a communication call, the implementation can switch to a rank which is ready to compute, thus achieving overlap of computation and communication. This approach permits the user code to use the conceptually-simpler blocking interface of MPI. Load balancing can also be implemented by migrating MPI ranks between CPU cores and potentially even nodes—all done without any participation from the user code, which drives the complexity even further down.

TinyMPI is a virtualized MPI implementation developed under this project. Its internal details as well as results of a research effort which employed it are presented in Milestone 13 [8].

2.2 Dedicated communication thread for MPI-heavy tasks

In this section we describe an alternative approach to support MPI communication modes in task-based programming models such as OmpSs.

As outlined in D7.7 [1], in certain situations MPI communication can be subject to a priority inversion problem. In the simplest case, suppose an MPI task is on the critical path, but is not executed because all OmpSs worker threads happen to be executing long-running compute tasks. Since the OmpSs runtime does not interrupt running tasks, the high-priority MPI tasks may have to wait significant time until worker threads become available, thus increasing the length of the critical path. A conceptually simple approach to mitigate the problem is to dedicate a thread specifically for MPI-heavy communication tasks. See [10] for full details.
int main() {
    // ...

    #pragma omp task inout(buffer) comm_thread
    MPI_Recv(buffer, size, MPI_INT, source, MPI_COMM_WORLD);
    
    #pragma omp task in(buffer)
    // using buffer here

    // ...
}

Figure 15: Example of the usage of the comm_thread clause to mark a task as targeted for the communication thread.

Basic principle

To explore this approach, we have implemented a new scheduler for the OmpSs runtime system. In this scheduler, one thread is designated as communication thread; all other threads keep their usual role as worker threads. In contrast to worker threads, the communication thread will only pick up tasks that have been specifically marked by the programmer. Marking a task is done by a new clause, comm_thread, to the task construct as shown in Figure 15.

Similar to other threads, the communication thread manages its own task queue. Unlike other queues, only tasks marked for the communication thread are ever placed on this particular queue once all pending dependencies are matched. In a way, the communication thread is an execution resource onto which special, e.g. MPI-heavy, tasks can be off-loaded.

In general, MPI communication is expected to take less time than computation in an HPC application. Thus, at times there will be no tasks pending execution on the communication thread. In this situation, regular OmpSs worker threads will continue polling their tasks queue. In contrast, the communication thread may yield the CPU core to other threads and sleep until new tasks are placed into the communication queue. This warrants that there is little or no impact on regular worker threads doing CPU-heavy computations.

Typically, one would configure the OmpSs runtime system to use as many regular worker threads as there are cores in the CPU, and one additional thread for communication. Thus, OmpSs would oversubscribe the number of cores by one additional thread. Further, typically regular worker threads would be pinned to cores. However, the communication thread should remain un-pinned and be able to float across all present cores if necessary.

We have validated the communication thread approach with MB3 applications amongst others. A comparison of MPI-only, OmpSs without communication thread, and OmpSs with communication thread versions of the application LBC on the project’s test platform Dibona and a Cray XC40 is presented in D6.9 [6]. The experiments show, that the communication thread approach can be an efficient technique to overlap communication and computation, but is not generally superior to other approaches. One particular issue is detailed in the remainder of this section.

Susceptibility to thread starvation

When doing the performance evaluation on Dibona, we observed much lower performance for the communication thread version of LBC than expected from experiments on the Cray system.
Figure 16: Trace of LBC execution on the Cray system with oversubscription of threads. The figure shows timelines stacked on top of each other for 6 worker threads (top) and the communication thread (lowest). Time increases from left to right. The colours indicate the three tasks inner (pink), outer (purple), and exchange (red). The light blue colour on the communication threads indicates that no task is being executed. Task instances are separated by the green flags. As expected, the exchange tasks starts as soon as all inner tasks complete.

Figure 17: Trace of LBC execution on the Dibona system with oversubscription of threads. The figure shows timelines stacked on top of each other for 6 worker threads (top) and the communication thread (lowest). Time increases from left to right. The colours indicate the three tasks inner (blue), outer (red), and inner (yellow). The light blue colour on the communication threads indicates that no task is being executed. Task instances are separated by the green flags. There is a noticeable gap between completion of inner tasks and the execution of the exchange task on the communication thread.
More specifically, the duration of communication task was increasing dramatically when scaling the number of MPI ranks.

At the most abstract level, LBC is a generic stencil-code on a regular grid with nearest-neighbour communication only. For the purpose of this document, the basic element of the grid is called a *tile*. Only tiles on the surface of the grid need to participate in exchange of boundaries across MPI domains; we refer to these as *outer tiles*. Tiles which are not located on the surface of the grid, referred to as *inner tiles* do not participate in in MPI communication.

The OmpSs version of LBC defines three different tasks for boundary exchange (*exchange*), calculation on inner tiles (*inner*) and calculation on outer tiles (*outer*), respectively. There is only one instance of the *exchange* tasks, but many instances of *inner* and *outer* tasks. OmpSs dependencies are formulated such, that the tasks *exchange* can start as soon as all *inner* tasks complete; tasks *outer* can run in parallel to the task *exchange*. This behaviour is nicely seen in traces recorded on the Cray system and illustrated in Fig. 16. The red *exchange* task starts executing on the communication thread right after the last pink *inner* task finishes on the lowest worker thread. Note that as explained above, the communication thread is an additional thread on top a worker thread for each core.

The Dibona system behaves differently as illustrated in Fig. 17. Unlike on the Cray system, there is a noticeable gap between the completion of *inner* tasks on the regular worker threads and the start of the *exchange* tasks on the communication thread. Also the exchange tasks takes much longer to execute than expected from the volume of MPI transfers.

Closer inspection showed that the duration of tasks running on the communication thread varied significantly between task instances and could be as long as many times the duration of a full time-step. Similarly, the duration of the gap between the completion of inner tiles calculation and the communication task varied significantly. As a result, the execution time of LBC with communication thread on Dibona was much lower than without and did not scale with increasing number of MPI ranks.

In the end, we found that the operating system’s thread scheduler was not scheduling the communication thread sufficiently frequently. We observed, that even when the exchange tasks, which does several MPI calls, was being executed by the OmpSs runtime, the operating system would interrupt – *pre-empt the communication thread* – and give control to a regular worker thread. Moreover, even when the worker threads ran out of tasks and started busy-polling their (empty) queues, the operating system would still not schedule the communication thread. Obviously, the operating systems scheduler assigns a much higher priority to the worker
threads leading to \emph{thread starvation of the communication thread}. It is noted that on the Cray system, which was the primary test platform during development, the communication thread gets its fair share of core time.

We have briefly tried several approaches to mitigate the issue on Dibona. Firstly, we used an option of the OmpSs runtime that forced regular worker threads to yield the CPU after finishing each task. This gives the operating system’s scheduler the chance to re-schedule other threads. While this had a small effect, the communication thread was still not being scheduled frequently enough. Similarly, we changed the behaviour of the communication thread to mimic the worker threads and do busy-polling on its tasks queue rather than yielding the CPU and sleeping. This reduced the latency of scheduling the communication thread, but had a negative impact on the overall performance as threads were competing for system resources leading to frequent thread scheduling events.

Because of the short time, in the end we settled for another solution on Dibona. Rather than oversubscribing the number of cores with the additional communication thread, we reduced the number of worker threads by one and gave a core exclusively to the communication thread. Now the behaviour is much saner, as seen on Fig. [18] The gap between completion of \emph{inner} tasks and start of the \emph{exchange} task is significantly smaller. Further, the communication thread is no longer pre-empted by the operating system’s scheduler.

\textbf{Conclusion}

The work presented above show that the communication thread approach works in principle, but is susceptible to thread starvation. In practise, it requires that the operating system schedules threads with low priority, such as the communication thread, with sufficiently low latency. At the time of writing this document it is not clear yet why the operating system thread schedulers behave so differently. In any case, these experiments on Dibona showed that the communication thread approach is not as generally applicable as previously thought and will need more investigation in the future. In particular future study should critically evaluate the design of the communication thread and consider using thread scheduling policies such as \texttt{SCHED\_DEADLINE}, were the Linux operating system guarantees to schedule a thread before reaching a set deadline. Also, a critical evaluation of the alternative approach of using a polling service within the OmpSs runtime system should be done. Another research direction could address heterogenous processor designs such as Arm’s big.LITTLE architecture and place such non-compute threads on the LITTLE cluster.
References


