Towards a Parallel Benchmark for Space Applications: Distributing OBPMark's Image Processing

Mahmoud M. Elbarrawy*, Carlos Gonzalez Cortes*, Andreas Lund†, Daniel Lüdtke*

*Institute for Software Technology
German Aerospace Center (DLR)
Braunschweig, Germany
†Institute for Software Technology
German Aerospace Center (DLR)
Weßling, Germany

Abstract*—Modern space applications require high computing power and high reliability from on-board processors. To meet these requirements, the German Aerospace Center (DLR) is developing a Scalable On-board Computer for Space Avionics (ScOSA) system with a distributed non-shard memory architecture. As performance is an important criterion in the selection of hardware for space missions, the European Space Agency has published an open source benchmark suite called OBPMark. It is a set of benchmarks based on typical space applications and designed to measure system-level performance. However, there is currently no standard tool for evaluating the performance of distributed on-board computers. In this paper, we propose a parallelization strategy for running the OBPMark image processing benchmark on a distributed on-board computer. We used a split-map-reduce model to integrate the #1.1 image calibration and correction benchmark of OBPMark into the ScOSA system. We evaluated the developed distributed benchmark on the existing ScOSA High Performance Nodes (HPNs) consisting of 5 Xilinx Zynq 7020 SoCs. The results show a significant reduction of the benchmark execution time from 9.0 to 2.8 seconds using 5 nodes. In the case of dual-core with 4 nodes, the execution time was reduced to 2.5 seconds. We conclude that OBPMark is a valuable tool for evaluating the performance of distributed on-board computers with non-shared memory architectures and contributes to the standardisation of performance evaluation in the space domain.

***Index Terms*—parallel computing, image processing, space application, data processing, benchmarking, OBPMark, ScOSA**

I. Introduction

Today's space missions generate large amounts of data. Due to limited downlink capacity and contact opportunities, on-board computers (OBCs) must be powerful enough to process this data directly on board. Furthermore, more and more space applications include artificial intelligence, data analysis or edge computing approaches, which are usually computationally intensive [1], [2]. One class of state-of-the-art on-board computer architectures are heterogeneous distributed systems consisting of reliable low-performance nodes and high-performance COTS nodes [3]. The heterogeneous architecture distributes the workload and provides redundancy in case of failures due to the harsh space environment. The German Aerospace Center (DLR) is developing an on-board heterogeneous distributed computer through the Scalable On-Board Computer for Space Avionics (ScOSA) project [4], [5].

In space missions system engineers often chose performance as a criterion to select the OBC that fulfills the mission requirements. Benchmarks are a standard way to evaluate a system based on certain qualities such as reliability, performance, and security [6].

In the space domain, there have been attempts to evaluate space-grade hardware based on different benchmarks. For example, NASA developed the NAS Parallel Benchmarks (NPB) to measure the performance of parallel supercomputers [7]. The EDN Embedded Microprocessor Benchmark Consortium (EEMBC) has developed benchmarks for industrial automotive, networking, and telecommunications applications. EEMBC has also been used to evaluate the performance of space-qualified single- and multi-core LEON processors [8], [9]. The Graphics Processing Unit for Space Application (GPU4S Bench) project is a benchmarking suite for space GPU on-board devices [10], [11].

Finally, the On-Board Processing Benchmarks (OBPMark) is a benchmark suite that targets on-board devices regardless of the type of processing unit. OBPMark is being developed by the European Space Agency (ESA). It includes a set of sub-benchmarks inspired by common space applications: image processing, data compression, standard encryption, processing building blocks, and machine learning inference [12]. OBPMark has the potential to become a standard for benchmarking state-of-the-art and future OBCs.

However, OBPMark does not yet take into account distributed on-board computers with non-shared memory. Benchmarking distributed systems can be challenging because of the need to develop an appropriate workload distribution strategy and account for data transfer and synchronization

Published Paper:
M. Elbarrawy et al., "Towards a Parallel Benchmark for Space Applications: Distributing OBPMark's Image Processing," 2023 European Data Handling & Data Processing Conference (EDHPC), Juan Les Pins, France, 2023, pp. 1-8, doi: 10.23919/EDHPC59100.2023.10396256.

delays. Therefore, this work describes a strategy to extend OBPMark's #1.1 Image Calibration and Correction pipeline to support distributed on-board benchmarking using the ScSOA framework.

This paper explains the selected OBPMark benchmark workload, then describes the proposed parallelization and distribution strategy, followed by an explanation of the evaluation setup and scalability test results using the available ScOSA on-board computer development model.

II. Related Work

Assessing the performance of distributed systems is not a new idea. Benchmarking real-time distributed systems (RTDS) has been presented by Kamenoff et. al in [13]. The RTDS can be thought of as a group of tasks (processes) that run on different nodes and send messages to each other. Both the tasks and the messages are time-constrained, and if the deadline is passed, then the benchmark fails or gets a low score for that task. Jin et. al in [14] are modeling the performance of the distributed system by two factors: the available data and the transmission paths available for it. Then the algorithm estimates the time for each path and give the system the freedom to select the shortest path. Some other frameworks have been developed for evaluating the distributed cluster of computers like the PEEL framework [15] and the IPACS-Project (Integrated Performance Analysis of Computer Systems) [16]. Still, non is expanding benchmarking for embedded distributed real-time systems in space.

Benchmarking on-board computers has been done before. For example, Lovelly et. al in [17] and Kosmidis et. al in [11] have been doing benchmarking for space applications in their experiments. The closest benchmark to distributed systems is the NAS parallel benchmark [7]. However, it is used for supercomputer evaluation and not for embedded on-board distributed systems.

These attempts to assess the performance of distributed systems can serve as a basis for developing distributed benchmarks for space applications. However, there is a gap regarding benchmarking space targeted distributed on-board computers with non-shared memory. We consider standardization and the evaluation of system-level performance as key requirements, so we decided to adapt the OBPMark benchmark suite to the ScOSA system.

III. ScOSA

DLR is developing the Scalable On-board Computing for Space Avionics (ScOSA) as a hybrid distributed on-board computer [5]. One of the fundamental goals of ScOSA is to provide the spacecraft with access to both reliability and performance by combining reliable space-grade computing nodes with high-performance commercial off-the-shelf (COTS) nodes.

ScOSA consists of both hardware and software components. From a hardware perspective, the ScOSA on-board computer is a heterogeneous distributed system consisting of Reliable Computing Nodes (RCNs) and High Performance Computing Nodes (HPNs) connected via SpaceWire or Ethernet. The RCN nodes are radiation-tolerant LEON3 processors, while the HPN nodes are Xilinx Zynq 7020 system-on-chip (SoC) combining dual-core ARM A9 CPUs and an FPGA in one package, DDR3 RAM, and two NAND flash memories [5].

The ScOSA middleware consists of three main components: a communication protocol called *SpaceWireIPC*, a task distribution framework called *Distributed Tasking Framework*, and a set of *System Management Services* to provide FDIR techniques. It currently supports GNU/Linux and RTEMS real-time operating systems [5]. This work uses the task distribution capabilities of the ScOSA middleware to implement a benchmark application capable of utilizing all available computing power and measuring system-level performance.

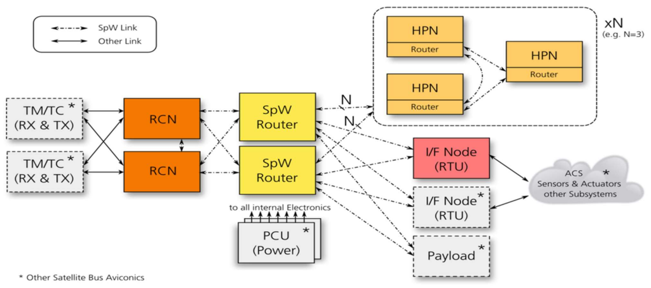

Fig. 1. ScOSA hardware architecture example [3]. In this configuration, 2 RCNs and as many HPNs as needed are connected via SpaceWire. The system supports task distribution to remote nodes, and in case of failures, it is able to reconfigure moving tasks from a failed node to any working node.

Figure 1 shows a possible ScOSA hardware architecture configuration. In this configuration, the system consists of two RCNs and N HPNs connected by the SpaceWire router. Final configurations are planned with 1 or 2 RCNs and up to 8 HPNs; however, the current development model consists of 1 RCN and 5 HPNs. For this work we used 5 HPNs running Yocto Linux, each of the inter-connected by Ethernet.

IV. The OBPMark image processing benchmarks

The OBPMark is a new benchmark suite, currently in public beta version v0.3.1 [18]. The image processing benchmark is the first in this suite and consists of two sub-benchmarks: #1.1 Image Calibration and Correction and #1.2 Radar Image Processing. In this paper, we have selected the #1.1 Image Calibration and Correction benchmark because it is an excellent workload example for space applications, as many satellites today have a camera on board. This benchmark is based on deep-space telescopes with long exposure times, so multiple frames are captured and then stacked to form a final image [19]. Therefore, the workload is computationally and data intensive.

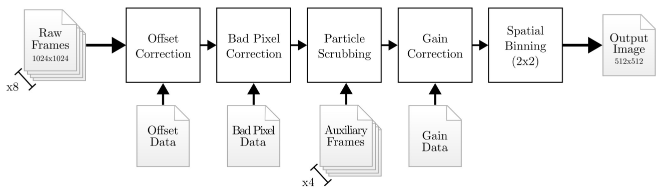

Fig. 2. OBPMark #1.1 Image Calibration and Correction benchmark pipeline. Image adapted from OBPMark repository documentation [18].

Figure 2 shows the pipeline implementation of the image processing benchmark for Image Calibration and Correction. The input is a set of frames (8 frames) and the output is a final processed image output. The benchmark also provides auxiliary data to process every input frame. The pipeline process explanation is as follow, where:

I : is the current input pixel.
J : is the output pixel.
F : is the frame.
x, y : are the position of the pixel inside the frame.

1) Image Offset Correction: In the image offset correction step, a constant offset value $C_{x,y}$ is set to each input pixel $I_{x,y}$ of each input frame, and the output $J_{x,y}$ is the difference between the input pixel value and a constant offset value $C_{x,y}$. However, if the input $I_{x,y}$ is less than $C_{x,y}$, then the input is equal to the output. These offset values are predefined by the benchmark as auxiliary data.

$$J_{x,y} = I_{x,y} - C_{x,y} \tag{1}$$

2) Bad Pixel Correction: In the bad pixel correction step, the look-up table $M_{x,y}$ provided by the benchmark determines whether the pixel is corrupted or not. Based on the $M_{x,y}$ value, the output is either the same as the input or the output pixel is a function of f. The function f is the average value of good neighboring pixels with a mask size of **3x3** pixels. To handle corners and edges, the mask size changes to **2x2** for corners, **3x2** for top/bottom edges, and **2x3** for left/right edges.

$$M_{x,y} \begin{cases} 1 & , \ J_{x,y} = f(I_{x,y}, x, y) \\ 0 & , \ J_{x,y} = I_{x,y} \end{cases} \tag{2}$$

3) Radiation Scrubbing: In radiation scrubbing, the process is not a one-to-one operation as in the previous steps, where the operations are performed on the same frame for each pixel and auxiliary data predefined by the benchmark. It depends on the temporal order in which the frames are captured. After determining that a pixel is affected by a radiation disturbance, a pixel scrubbing process starts, where the calculation of the new pixel value is the average of the values of this pixel from the previous and future two frames. For a sequence of frames from t = 0 to t = 7, four additional auxiliary frames are provided (t = -2, t = -1, t = 8, t = 9).

$$J_{x,y} = \frac{I_{(x,y),t-2} + I_{(x,y),t-1} + I_{(x,y),t+1} + I_{(x,y),t+2}}{4} \tag{3}$$

4) Gain Correction: This step consists of simply multiplying by a gain value from the auxiliary data. The gain value ranges from 0.950 to 1.050.

$$J_{x,y} = I_{x,y}.G_{x,y} \tag{4}$$

5) Spatial Binning: In the spatial binning process, each **2x2** pixels block within each input frame is added together to form only one pixel. Thus, the resulting frame is one-fourth the size of the input frame.

$$J_{x,y} = I_{x,y} + I_{x+1,y} + I_{x,y+1} + I_{x+1,y+1} \tag{5}$$

6) Temporal Binning: In this step, the sum for each x,y location, pixel by pixel, of all eight frames is used to construct an output frame.

$$\sum_{t=0}^{7} F_{i,t} \tag{6}$$

V. Parallelization and Distribution Strategy

We faced several challenges in order to evaluate the performance of a distributed system using the #1.1 Image Calibration and Correction benchmark. First, we need to take into account the possible data dependencies in the distributed image processing algorithm. For example, in the radiation scrubbing equation (3), there is a dependency between the previous and future two frames. Similarly, in the bad pixel correction (2) and spatial binning (5) equations, we can see that the operations depend on the information of neighboring pixels. In addition, the spatial binning step will shrink the size of the input frames, which means that we cannot perform this operation until all the previous steps have been completed [20].

After reviewing the #1.1 Image Calibration and Correction algorithm implementation of OBPMark, we propose to keep the pipeline implementation as it is, but instead of using the entire frames as input, we split them and feed the pipeline with as many sub-frames as CPUs are available. And at the end, we get partially processed sub-frame outputs that are merged to form the final output. This is similar to a split-map-reduce approach in data processing.

The reason for keeping the pipeline as it is, without splitting its steps between the processing units, is the dependency between the steps of the pipeline. In addition, transferring all eight frames from one processing node step to another would increase the data transfer latency in the system compared to transferring only a portion of the input data. In the implementation, the pixel data is stored in a binary format file with a one-dimensional representation. Thus, the input frames are divided into horizontal sub-frames of nearly equal size for a fair distribution.

However, there is a challenge with the spatial dependencies of the bad pixels and spatial binning steps in this approach.

For example, if the frames are simply divided equally without taking into account the newly constructed edges and corners, the bad pixel function f will treat the corners and edges as if it were a full image, and the pipeline output will not be the same. Also, spatial binning requires that the height of the input frame be an even value so that it can be shrinked by half, which should be taken into account in the newly constructed subframes.

To overcome such challenges, we consider an overlapping area when splitting the sub-frames to preserve information about neighboring pixels.

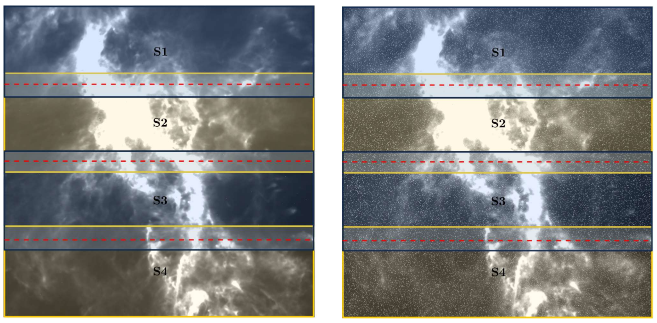

Fig. 3. Overlapping Splitting of a frames into 4 sub-frames. The red dashed lines represent the fair split without overlapping. The algorithm then creates an overlapping region for each sub-frame S_1 to S_4. As shown, the sub-frames S_2 and S_3 contain an overlapping area of the previous and following sub-frames, while sub-frames S_1 and S_4 contain an overlapping area of the following or previous sub-frame respectively. Backgrounds images are the frames 1 (left) and 2 (right) of the OBPMark #1.1 Image Calibration and Correction benchmark input data [18].

Figure 3 shows a simplified illustrative example of how a frame can be split among four CPUs; however, the splitting is done for all eight input frames and auxiliary data in the same way, and the algorithm can handle more than four nodes with the constraint that the number of nodes is less than the height of the input frame and that the input frames are of even height value in size, which is necessary for the spatial binning equation (5) to work properly.

Algorithm 1 Splitting

Input: frame (F), frame height (H), divisions ($N_{divisions}$)

Output: list of sub-frames (SubFrame_i)

```
1:  $S \leftarrow H/N_{divisions}, \quad R \leftarrow H - (S * N_{divisions})$
2:  for $i < N_{divisions}$ do
3:      $k_{up} \leftarrow 2, \quad k_{down} \leftarrow 2$
4:      if $i == 0$ then
5:          $k_{up} \leftarrow 0$
6:      else if $i == N_{divisions} - 1$ then
7:          $k_{down} \leftarrow R$
8:      end if
9:      $\text{HeightSlice}_i \leftarrow [S * i - k_{up} : S * (i+1) + k_{down}]$
10:     $\text{SubFrame}_i \leftarrow F_{[\text{HeightSlice}_i, \text{width}]}$
11: end for
```

In the splitting algorithm Alg. 1, the S value is the height of the sub-frame size without including the overlap. $N_{divisions}$ represents the number of sub-frams for the available nodes. R is the remainder in case the division has a remainder the last sub-frame will take this reminder. The overlap area is represented as $K_{up/down}$, where one row is added to hold pixel information for the bad pixel correction step, and another row is added to hold pixel data for the spatial binning step. Thus, we will end up with overlap areas at the top and bottom of the sub-frames; however, for the first and last sub-frames, the overlap area is only at the top or bottom.

With this approach, we will end up with a processed row of pixels as redundant values in each overlapping area, which we can ignore in the merge process to get the same output result as the original benchmark.

VI. Parallelization and distribution with ScOSA

We used the Distributed Tasking Framework (part of the ScOSA Middleware) to parallelize and distribute the benchmark. The Distributed Tasking Framework is based on the Tasking Framework, which is an event-driven, multi-threaded execution platform for real-time on-board software systems [21]. The Tasking Framework is developed by DLR and is published as open source [22]. The framework allows the user to run multiple functions in parallel without worrying too much about the complexities going on underneath. Thus, a Distributed Tasking Framework application will use the following building blocks:

1) *Tasks* are processing units that implement the application's main functionalities. *Tasks* can be mapped to any node in the distributed system and can be connected via *Channels* to form an execution graph. *Tasks* execution is triggered by *Events* or data arriving on the connected *Channels*.
2) *Channels* are data generators or containers that separate data acquisition from data processing and transfer it between *Tasks* as the application requires.
3) *Events* are a special type of *Channel* that can be used to trigger *Tasks* periodically or once, depending on the application implementation.
4) *Writers/Readers* are a special type of *Task* that allow nodes to communicate and send data to each other.

Thus, the distribution strategy is basically a split-map-reduce scheme (see Section V) using the Distributed Tasking Framework. Figure 4 shows the tasking diagram of the benchmark application that guided the implementation. It consists of the following blocks:

1) *SplitTask*: In the *SplitTask* the data is loaded from the file system and then divided into N sub-frames (where N is the number of processing nodes in the ScOSA system) according to the split algorithm 1. The sub-frames are then packed into a `msg` structure (see Listing 1) and pushed to the *SubFrames* channels. The *SplitTask* runs on the first node (coordinator node). For benchmarking, we include the time spent in the split algorithm and the time spent sending the sub-frames to the remote nodes.

2) *OBPMarkTask*: The *OBPMarkTask* is the *map* step. Here, the sub-frame data is received and processed using the original #1.1 Image Calibration and Correction benchmark code. The partial results are packed into a `msg_output` structure (see Listing 1) and pushed to the *ProcFrames* channels. The *OBPMarkTask* tasks run on all nodes in parallel (worker and coordinator nodes). In our performance measurements, we consider the time required to receive the sub-frame data, apply the original benchmark process, and send the partial result sub-frames back to the coordinator node. In addition, we measure the time spent in the original benchmark code alone (elapsed time) so that we can compare it to the total execution time and determine the added overhead.

3) *MergeTask*: The *MergeTask* is the *reduce* step where all partial sub-frame results are received and merged to form the output image. This task is executed on the coordinator node. The execution time takes into account the time needed to receive the partial results and create the output image. Similar to the original code, we do not consider the time needed to store the output image in the file system.

Finally, after the system has been properly configured by the ScOSA middleware, an *Event* triggers the execution of the benchmark application. Due to the event-driven nature of the Distributed Tasking Framework, all processing steps are synchronized using *Channels* and are only executed when all necessary data have arrived [23].

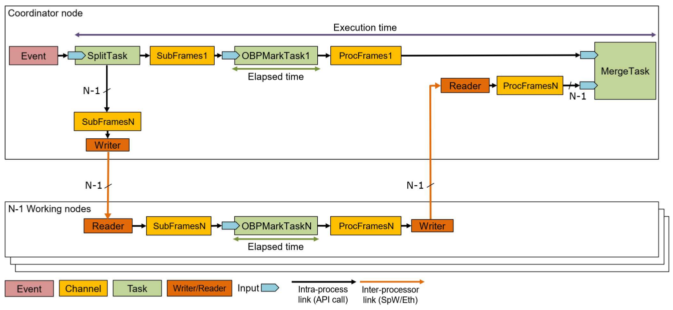

Fig. 4. Tasking diagram of the benchmark application to implement a split-map-reduce scheme. The coordinator node splits the input frame (*SplitTask*) and sends N sub-frames to the remote node's map tasks (*OBPMarkTaskN*). The coordinator node also process one sub-frame (*OBPMarkTask1*). The partial results are sent back to the coordinator node to merge them (*MergeTask*) and generate the output image. An *Event* triggers the application execution.

Listing 1. Structures used for data serialization.

```c
#define NNODES 3  ///< Nnumber of nodes (or CPUs)
#define MSG_SIZE ((int)(1024/NNODES)+1)*1024 //H*W
#define MSG_SIZE_HALF (((int)(1024/NNODES)+1)/2)
  *1024 //H*W
#define NUM_FRSME 12 // 8 Frames + 4 Auxiliary

/** Struct to serialize a frame */
typedef struct
{
  uint16_t frame[MSG_SIZE] = {0};
} frame_msg;

/** Struct to input data */
struct image_data_msg
{
  unsigned int num_frames = {0};
  frame_msg frames[NUM_FRSME];
  uint16_t offsets[MSG_SIZE]    = {0};
  uint16_t gains[MSG_SIZE]      = {0};
  uint8_t bad_pixels[MSG_SIZE] = {0};
  uint8_t scrub_mask[MSG_SIZE] = {0};
  uint32_t binned_frame[MSG_SIZE_HALF] = {0};
  uint32_t image_output[MSG_SIZE_HALF] = {0};
};

/** Struct to send sub-frame data. */
struct msg
{
  image_data_msg image_msg;
  unsigned int w_size                 = {0};
  unsigned int h_size                 = {0};
  unsigned int h_overlap_array[NNODES] = {0};
  unsigned int h_output_array[NNODES]  = {0};
  struct timeval starttime;
};

/** Struct to send processed partial results. */
struct msg_output
{
  uint32_t image_output[MSG_SIZE_HALF] = {0};
  unsigned int w_size                  = {0};
  unsigned int h_size                  = {0};
  unsigned int h_output_array[NNODES]  = {0};
  struct timeval starttime;
};
```

VII. Evaluation setup

The selected system to test our approach is a development model of the ScOSA on-board computer. It consists of 5 Xilinx Zynq-7020 SoC high-performance nodes, each with a dual-core ARM Cortex-A9 processor @ 886 MHz and 1 GB RAM. The nodes are connected via Gigabit Ethernet in an isolated local network.

Scalability tests were performed by running the distributed benchmark workload on up to 5 nodes in single-core mode and up to 4 nodes in dual-core mode (8 CPUs). For each number of nodes, the following steps were performed

- Build and deploy the benchmark application for the selected number of nodes.
- Clean the output data in the coordinator node.
- Run the distributed benchmark application on the selected number of nodes.
- Compare the output with the validation data in the coordinator node.
- Repeat the experiment X times.

We choose $X = 10$ times to ensure that the results are consistent, reliable, and reproducible. We checked that the results of the distributed benchmark application were valid. We then averaged the execution time to account for the OS and network usage disturbances on the data. The code was built using the Xilinx Yocto 3.0 SDK with the following optimization flags `-O3` and `-mcpu=cortex-a9`. We used ScOSA framework version 1.2.0.We consider the one-node benchmark results as a basis for fair comparison. The one-node version is not the original benchmark application, but the distributed version running on a single CPU, including the initialization, data transfer, and merging steps described in previous sections.

VIII. Results

The benchmark results for the single-core test case are shown in Table I and Figures 5 and 6. It is observed that it is possible to distribute the benchmark workload to speed up the execution time. We observed a speedup of 3.2x using 5 nodes with a total execution time reduction from 9.0 to 2.8 seconds. This means that 5 ScOSA HPN nodes can process 0.37 Mpixels per second. Figure 5 shows the difference between elapsed time time and execution time. The elapsed time refers to the average time spent by each node in the original OBPMark #1.1 Image Calibration and Correction benchmark code, while the execution time refers to the total time required to execute the benchmark application including the split, distribute, and merge steps. We can observe an overhead added by the distribution strategy and the ScOSA framework itself. This overhead represents 23% of the time in the two-node case and up to 53% of the time in the five-node case. In our tests, we found that most of this overhead is caused by the data transfer protocol. In total, about 36 MB of data is transferred from the split task to the remote nodes and 2 MB of data is transferred back to the merge task.

TABLE I
Benchmark scalability results. Single core.

Nodes	Execution time (ms)	Mpixels/s	Speedup
1	9054	0.12	1.0
2	5105	0.21	1.8
3	3957	0.26	2.3
4	3265	0.32	2.8
5	2828	0.37	3.2

Image size 1024x1024 pixels

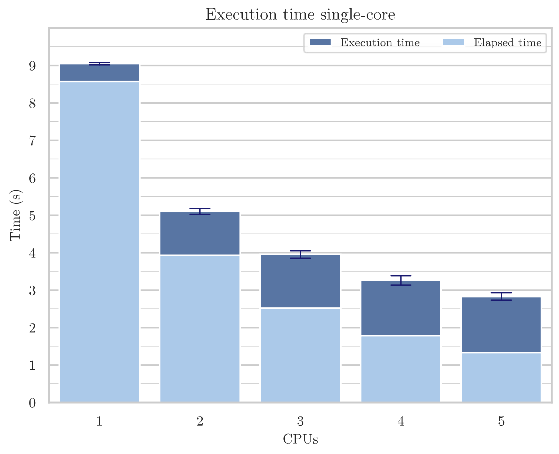

Fig. 5. Benchmark execution time results for the single core test case.

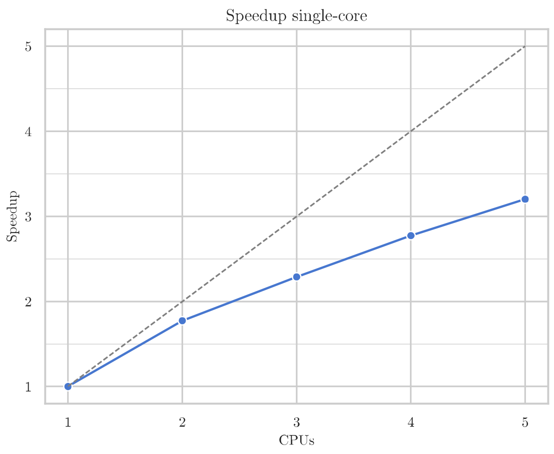

Fig. 6. Benchmark speedup results for the single core test case.

The results of the multi-core test cases are shown in Table II and Figures 7 and 8. The benchmark was distributed over up to 8 CPUs using 2 cores per node. We observed a speedup of 3.7x using 8 CPUs with a total execution time reduction from 9.0 to 2.5 seconds. Thus, 4 dual core ScOSA HPN nodes can process 0.47 Mpixels per second. Similar to the single core case, Figure 8 shows an overhead of up to 66% when using 8 CPUs (4 nodes). In this case, the overhead may worsen because CPU cores are used by communication protocol and task execution threads. This is especially noticeable when moving from 3 to 4 CPUs. In the first case, only one of the two cores per node is used by the task execution threads, while in the second case, both cores are used by the task execution threads, but also by other parts of the framework.

TABLE II
Benchmark scalability results. Dual-core.

Nodes	CPUs	Execution time (ms)	Mpixels/s	Speedup
1	1	9056	0.12	1.0
	2	5675	0.18	1.6
2	3	3923	0.27	2.3
	4	3680	0.28	2.5
3	5	3209	0.33	2.8
	6	2987	0.35	3.0
4	7	2711	0.39	3.3
	8	2459	0.43	3.7

Image size 1024x1024 pixels.

IX. Discussion

The results show the ability of the ScOSA framework to distribute heavy workloads and to make use of all the available computing resources in a distributed on-board processing

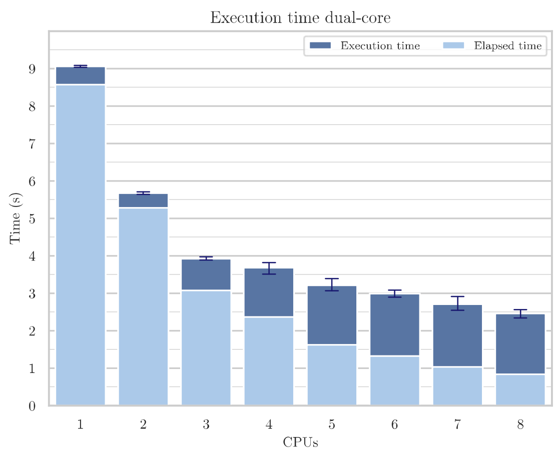

Fig. 7. Benchmark execution time results for the dual-core test case.

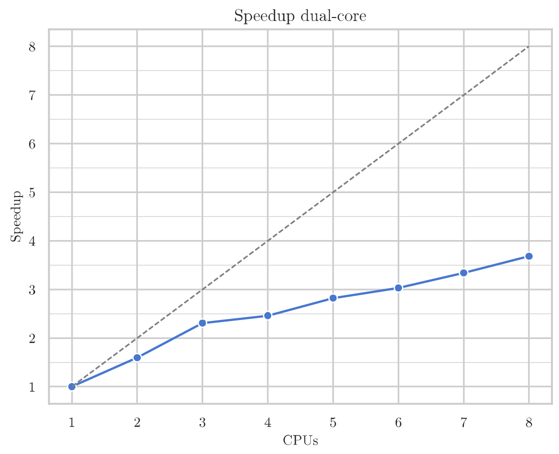

Fig. 8. Benchmark speedup results for the dual-core test case.

platform. The ability to scale the application in different configurations also shows that ScOSA applications are flexible and reconfigurable with little effort.

However, the results also show an overhead in execution time compared to the time actually spent in the benchmark code. In fact, this overhead worsens as more nodes are added to the system or as more parallel tasks are added to a node. There are several reasons for this. First, the parallelization strategy takes into account an overlap region when splitting the input data, so the more parallel tasks we add, the more redundant data is transferred and processed. Also, when more nodes are added, the coordinator node has to handle more concurrent connections to send and receive partial data and results, which consumes CPU time. Finally, there is an overhead due to the communication protocol implemented in the ScOSA framework. ScOSA uses the custom protocol SpaceWireIPC to send management and data messages to remote nodes. SpaceWireIPC supports the UDP and SpaceWire protocols as underlying data transfer layers. It implements reliable and unreliable communication mechanisms. During these tests, we used only UDP over Ethernet in a local network to interconnect the computing nodes, but we still used SpaceWireIPC reliable messages to send the input and partial results data. These effects were particularly noticeable in the dual-core tests, where CPU time is shared among worker threads, system management threads, and communication protocol threads.

Nevertheless, during this work the distributed version of the OBPMark #1.1 Image Calibration and Correction benchmark was used to polish parts of the ScOSA framework codebase and to measure the impact of the changes on performance. Therefore, we expect further improvement of the results based on this application.

During this work, we did not focus on performance comparisons with other platforms, but rather on the scalability characteristics of the ScOSA framework. Thus, the baseline measurements consist of exactly the same distributed application running on a single node. As shown in the results, these base execution times also include some overhead due to the internal task communication and synchronization methods. It would be possible to obtain better results for the single node (single or dual core) case by using the original benchmark code with different parallelization strategies. However, such tests were beyond the scope of this work.

X. Conclusions

The paper presented a strategy for benchmarking a novel distributed on-board computer architecture. We chose the OBPMark #1.1 Image Calibration and Correction benchmark because it is based on typical space application workloads and provides realistic system-level performance measurements. Therefore, we developed a parallelization and distribution strategy for the benchmark using the ScOSA framework. We designed a split-map-reduce ScOSA application and mapped the processing tasks to remote nodes to effectively distribute the workload across the available processors. By scaling the application from 1 to 5 nodes, we showed that the ScOSA framework can indeed utilize all the available computing resources and can be adapted to different hardware configurations with little effort. The OBPMark #1.1 Image Calibration and Correction parallelization strategy consisted of horizontally splitting the input frame, taking into account overlapping regions, and processing each section independently and in parallel to finally merge the results. This shows that the OBPMark benchmark suite is portable and can be adapted to different use cases.

We were able to reduce the execution time from 9.1 to 2.8 seconds in the 5 nodes single-core configuration and to 2.5 seconds in the 4 nodes dual-core configuration, which is a speedup of 3.2x and 3.7x respectively. In fact, we observed that the benchmark was able to stress the processing and networking parts of the system. The results showed an execution time overhead caused by the additional data generated by the overlapping split strategy, the transmission of input data and results to and from remote nodes, and the communication protocol, all of which limited the scalability of the solution. Thus, the selected benchmark resulted in a valuable tool that guided improvements to the ScOSA framework code base.

We expect to continue to use the OBPMark benchmarks as a standard tool for measuring the performance of on-board processing platforms. In particular, an in-orbit demonstration of the ScOSA distributed on-board computer on a 12U CubeSat is planned for launch in late 2024. On this mission, a similar benchmarking application will be run periodically during the mission lifetime to study the processing capabilities of the platform under realistic conditions. We would also like to extend the work by integrating the rest of the OBPMark suite as a distributed version with the ScOSA system for future work.

XI. Acknowledgment

The authors would like to thank all current and former project members of the ScOSA Flight Experiment and its predecessor activities. We also want to thank the OBPMark Team for providing early access and supporting the project.

References

[1] G. Giuffrida, L. Diana, F. de Gioia, G. Benelli, G. Meoni, M. Donati, and L. Fanucci, "Cloudscout: A deep neural network for on-board cloud detection on hyperspectral images," *Remote Sensing*, vol. 12, no. 14, 2020.

[2] T. Tzanetos, M. Aung, J. Balaram, H. F. Grip, J. T. Karras, T. K. Canham, G. Kubiak, J. Anderson, G. Merewether, M. Starch, M. Pauken, S. Cappucci, M. Chase, M. Golombek, O. Toupet, M. C. Smart, S. Dawson, E. B. Ramirez, J. Lam, R. Stern, N. Chahat, J. Ravich, R. Hogg, B. Pipenberg, M. Keennon, and K. H. Williford, "Ingenuity Mars Helicopter: From Technology Demonstration to Extraterrestrial Scout," in *2022 IEEE Aerospace Conference (AERO)*, March 2022, pp. 01–19.

[3] A. Lund, Z. A. H. Hammadeh, P. Kenny, V. Bensal, A. Kovalov, H. Watolla, A. Gerndt, and D. Lüdtke, "ScOSA system software: The reliable and scalable middleware for a heterogeneous and distributed on-board computer architecture," *CEAS Space Journal*, Mai 2021.

[4] C. J. Treudler, H. Benninghoff, K. Borchers, B. Brunner, J. Cremer, M. Dumke, T. Gärtner, K. J. Höflinger, D. Lüdtke, T. Peng *et al.*, "ScOSA-scalable on-board computing for space avionics," in *Proceedings of the International Astronautical Congress, IAC*, 2018.

[5] D. Lüdtke, T. Firchau, C. Gonzalez, A. Lund, A. Nepal, M. Elbarrawy, Z. Hammadeh, J.-G. Meß, P. Kenny, F. Brömer, M. Mirzaagha, G. Saleip, H. Kirstein, C. Kirchhefer, and A. Gerndt, "ScOSA on the way to orbit: Reconfigurable high-performance computing for spacecraft," in *2023 IEEE Space Computing Conference (SCC)*. Los Alamitos, CA, USA: IEEE Computer Society, jul 2023, pp. 34–44.

[6] J. v. Kistowski, J. A. Arnold, K. Huppler, K.-D. Lange, J. L. Henning, and P. Cao, "How to Build a Benchmark," in *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '15. New York, NY, USA: Association for Computing Machinery, Jan. 2015, pp. 333–336.

[7] R. VanderWijngaart and B. A. Biegel, "NAS parallel benchmarks," in *Supercomputing 2002*, 2002.

[8] J. Jalle, M. Hjorth, J. Andersson, R. Weigand, and L. Fossati, "DSP Benchmark Results of the GR740 Rad-Hard Quad-Core LEON4FT," in *3rd ESA workshop on Digital Signal Processing for Space*. Gothenburg, Sweden: ESA, Jun. 2016, pp. 60–62.

[9] C. Gaisler, "GR740 Technical Note on Benchmarking and Validation," https://www.gaisler.com/doc/gr740/GR740-VALT-0010.pdf, Cobham Gaisler, AB, Tech. Rep. Doc. No GR740-VALT-0010, Issue 3.4, 2019.

[10] L. Kosmidis, J. Lachaize, J. Abella, O. Notebaert, F. J. Cazorla, and D. Steenari, "GPU4S: Embedded GPUs in Space," in *2019 22nd Euromicro Conference on Digital System Design (DSD)*. Kallithea, Greece: IEEE, Aug. 2019, pp. 399–405.

[11] L. Kosmidis, I. Rodriguez, A. Jover-Alvarez, S. Alcaide, J. Lachaize, O. Notebaert, A. Certain, and D. Steenari, "GPU4S: Major Project Outcomes, Lessons Learnt and Way Forward," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. Grenoble, France: IEEE, Feb. 2021, pp. 1314–1319.

[12] D. Steenari, L. Kosmidis, I. Rodriguez-Ferrandez, A. Jover-Alvarez, and K. Förster, "OBPMark (on-board processing benchmarks) – open source computational performance benchmarks for space applications," in *2nd European Workshop on On-Board Data Processing (OBDP2021)*, Jun. 2021, publisher: Zenodo Version Number: 1.0.

[13] N. Kamenoff, "One approach for generalization of real-time distributed systems benchmarking," in *Proceedings of the 4th International Workshop on Parallel and Distributed Real-Time Systems*, Apr. 1996, pp. 202–203.

[14] H. Jin, Y. Li, Z. Han, H. Wu, and W. Qiang, "Aeneas: real-time performance evaluation approach for distributed programs with reliability-constrains," *Cluster Computing*, vol. 10, no. 2, pp. 175–186, 2007.

[15] C. Boden, A. Alexandrov, A. Kunft, T. Rabl, and V. Markl, "PEEL: A Framework for Benchmarking Distributed Systems and Algorithms," in *Performance Evaluation and Benchmarking for the Analytics Era*, ser. Lecture Notes in Computer Science, R. Nambiar and M. Poess, Eds. Cham: Springer International Publishing, 2018, pp. 9–24.

[16] G. Falcone, H. Kredel, S. Kreuter, M. Krietemeyer, D. Merten, M. Meuer, M. Merz, F.-J. Pfreundt, D. Reinig, and H. Ristau, *IPACS-benchmark: integrated performance analysis of computer systems (IPACS); benchmarks for distributed computer systems*. Berlin: Logos-Verl., 2006.

[17] T. M. Lovelly, D. Bryan, K. Cheng, R. Kreynin, A. D. George, A. Gordon-Ross, and G. Mounce, "A framework to analyze processor architectures for next-generation on-board space computing," in *2014 IEEE Aerospace Conference*, Mar. 2014, pp. 1–10, iSSN: 1095-323X.

[18] European Space Agency (ESA). Maintained by: David Steenari., "Obpmark (on-board processing benchmarks) repository," https://github.com/OBPMark/OBPMark.

[19] D. Steenari, "(on-board processing benchmarks)," European Space Agency, Tech. Rep. V0.3-DRAFT, August 2022.

[20] M. Elbarrawy, "Performance evaluation for a distributed on-board computer," Master's thesis, Deggendorf Institute of Technology, January 2023.

[21] Z. A. H. Hammadeh, T. Franz, O. Maibaum, A. Gerndt, and D. Lüdtke, "Event-driven multithreading execution platform for real-time on-board software systems," in *15th Workshop on Operating Systems Platforms for Embedded Real-Time applications (OSPERT), Stuttgart, Germany, July 9, 2019*, A. Lackorzynski and D. Lohmann, Eds., 2019, pp. 29–34.

[22] Deutsches Zentrum für Luft- und Raumfahrt (DLR), "Tasking-framework," https://github.com/DLR-SC/tasking-framework/wiki.

[23] Z. A. Haj Hammadeh and O. Maibaum, "Tasking framework: An open-source software development library for on-board software systems," in *Embedded Systems Week, Sep. 20-25, 2020, Virtual Conference, Tutorials*, September 2020.