TRANSPARENT APPLICATION DEPLOYMENT IN A SECURE, ACCELERATED AND COGNITIVE CLOUD CONTINUUM

Grant Agreement no. 101017168

Deliverable D4.2
Performance Maximization under Maximum Affordable Error for the HW and SW IPs

Programme:	H2020-ICT-2020-2
Project number:	101017168
Project acronym:	SERRANO
Start/End date:	01/01/2021 – 31/12/2023

Deliverable type:	Report
Related WP:	WP4
Responsible Editor:	USTUTT
Due date:	31/03/2022
Actual submission date:	31/03/2022

Dissemination level:	Public
Revision:	FINAL

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 101017168

Revision History

Date	Editor	Status	Version	Changes
05.01.22	USTUTT	Draft	0.1	Initial ToC
07.01.22	USTUTT	Draft	0.2	Add Introduction
09.01.22	USTUTT	Draft	0.21	Add description of HPC Workflow
11.01.22	USTUTT	Draft	0.22	Add description about Kalman
12.01.22	USTUTT	Draft	0.23	Add performance in Kalman part
13.01.22	USTUTT	Draft	0.23	Add description about Framework
14.01.22	USTUTT	Draft	0.3	Add description about FFT
15.01.22	USTUTT	Draft	0.4	Add description about Dev. Interface
17.01.22	USTUTT	Draft	0.41	Improve description about Kalman
20.01.22	USTUTT	Draft	0.42	Improve performance for Kalman
20.01.22	USTUTT	Draft	0.43	Add performance to FFT
20.01.22	USTUTT	Draft	0.44	Improve Introduction
01.03.22	AUTH	Draft	0.5	Initial part about FPGA
04.03.22	AUTH	Draft	0.51	Add description about FPGA
05.03.22	USTUTT	Draft	0.6	Changes due to new FPGA part
06.03.22	USTUTT	Draft	0.7	Add text to HPC Interface part
07.03.22	USTUTT	Draft	0.8	Changes in Conclusion
08.03.22	USTUTT	Int. R.	0.9	Ready for internal review
25.03.22	USTUTT	Int. R.	0.91	Addressing comments
27.03.22	USTUTT	Final	1.0	Font and formatting optimization

Author List

Organization	Author
USTUTT	Javad Fadaie Ghotbi, Dmitry Khabi
AUTH	Dimosthenis Masouros, Dimitris Danopoulos, Aggelos Ferikoglou, Ioannis Oroutzoglou, Argyris Kokkinis, Kostas Siozios, Spyridon Nikolaidis
ICCS	Aristotelis Kretsis, Panagiotis Kokkinos, Polyzois Soumplis, Emmanouel Varvarigos

Internal Reviewers

Chocolate Cloud ApS

INNOV-ACTS Limited

Abstract: This deliverable (D4.2) presents the outcomes of Task 4.2 „*Performance Maximization under Maximum Affordable Error in Cloud and Edge*“, Work Package 4 „*Cloud and Edge Acceleration*“ of the SERRANO project, during the first iteration of the incremental implementation plan. The deliverable presents the initial specifications of the SERRANO HPC Services.

Keywords:: HPC Services, FFT Filter, Kalman Filter, Performance

Contents

List of Figures

List of Tables

Abbreviations

FPGA Field Programmable Gate Array; Field Programmable Gate Arrays is semiconductor device that can be configured by a customer or a designer after manufacturing.

FLOPS Floating Point Operations Per Second; Number of floating point operations per second.

FFT Fast Fourier Transformation

DFT Discrete Fourier Transformation

HPC High Performance Computing; Hochleistungsrechnen

HLRS germ.: Höchstleistungsrechenzentrum in Stuttgart; engl. The High Performance Computing Centre Stuttgart

MAEP Mean Absolute Error Percentage

HLS High-Level Synthesis

CU Compute Unit

DSP Digital Signal Processing

LUT Look-Up Table

FF Flip-Flop

MPI Message Passing Interface; MPI is a standard for message passing in parallel computations on the distributed memory.

VVUQ Verification, Validation, and Uncertainty Quantification.

NUMA Non-Uniform Memory Access.

OpenMP Open Multi-Processing; OpenMP is a programming interface for shared memory programming.

SSH The Secure Shell Protocol (SSH) is a network protocol for operating network services

securely over an network. The most notable applications are remote command-line execution and trasporting the data between the remote computers.

1 Executive Summary

Section 2 serves introduction and briefly describes the purpose of this deliverable, and addresses its correlation with SERRANO objectives. It includes information regarding the HPC service that was implemented and the structure of this deliverable.

Section 3 presents the data workflow, the main structure, and component of the HPC service. It describes the interaction between the user and the HPC service.

Section 4 introduces the Kalman filter and FFT filter and parallelization strategy that applied in these filters.

Section 5 presents the performance analysis of the HPC service and its component in different data sizes.

Section 6 presents Kalman filter acceleration using the approximation technique of FPGA.

Section **??** presents the performance approximation approach for HPC service.

Section 7 presents the interface of the HPC service and introduces the parameters that the user should provide to execute the use case task.

2 Introduction

SERRANO platform provides the capabilities for the user to compute certain tasks on various computational platforms such as High Performance Computing (HPC) systems, Cloud and Field Programmable Gate Array (FPGA). The HPC service that has been developed at High Performance Computing Center Stuttgart (HLRS) is a signal processing software package that can be applied for large data volume to accomplish the objectives of Task 2 Work Package 4 (Task 4.2) in the SERRANO project.

The current version of the HPC service contains two kernel filters such as Kalman Filter and FFT (Fast Fourier Transformation) filter and will be closely integrated with the next incremental integration step into the SERRANO platform. The motivation to select Kalman and FFT filters was, that these algorithms are an important part of the SERRANO project considered use cases [2]. This software package involves also the tools for the performance prediction and estimation of accuracy under different conditions.

This whole framework has the separate pre-processing step of data (signals) which is suited to prepare the data in the correct format for the HPC service. The main part of the software contains three steps. The first step is reading input data. The second step is the filtration process of the data. The last step is writing the output data. This software has been implemented in OpenMP/MPI and are appropriate to use in HPC systems [11],[5].

2.1 Document structure

The data workflow and main structure of this HPC service are introduced in section 3. The Kalman filter is the first filter developed in our framework. In section 4 we describe this filter and its parallelization by using both shared and distributed memory paradigms. The performance analysis of the Kalman filter and HPC service will be investigated in different data sizes in section 5, and we introduce the nonlinear approximation method to approximate the performance of the HPC service in section **??**.

Section 6 describes the acceleration of the Kalman filter in the FPGAs that has been developed by AUTH (Aristotle University of Thessaloniki).

The FFT (Fast Fourier Transformation) filter is the second filter that is placed in our HPC service, in section 4.2 we introduced this filter and explore the execution time behavior in different data sizes.

The description of this HPC service is the last section that is discussed in this deliverable, particularly two bash scripts were described which provide a possibility to the users to execute the implemented algorithms to solve their use case tasks.

2.2 Audience

This document is open to the public and should be useful to anyone looking for a preliminary explanation of the SERRANO components, as well as a definition of the overall SERRANO architecture and interfaces. Furthermore, this document could be beneficial to the general public in gaining a better grasp of the SERRANO project's structure and scope.

3 SERRANO HPC Workflow

We briefly demonstrate the workflow that is applied in the HPC service in the SERRANO platform. The main purpose of this HPC service is signal processing and applying filters on noisy signals. The kernel filter is supposed to process signals, and signals usually come from sensors that measure some experiment such as displacement, motion, or noises. Therefore signals are the input data for our software package, and to be used in the HPC service it has to be provided in binary format. This software has a separate pre-processing step that could read the input signals from any format, casting them in different data types, and then store them in binary format.

The principle and workflow of HPC service are demonstrated in Figure 3.1. The function f describes the use case task defined by the user, and the function F is the HPC service that solves the use case task f. In this scenario, the input data is provided locally by the user in workspace WS, therefore the HPC instance of F will act on it.

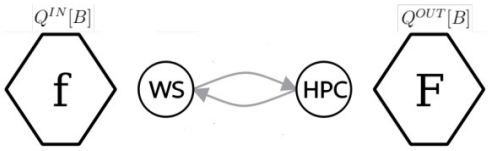


Figure 3.1: Workflow of HPC service

The function f has the following properties.

- Algorithm f: the certain filtration method is applied, for instance Kalman filter
- Input data size $Q^{IN}[B]$:
  - $Q^{IN}[B] = N^{IN} * k^{IN}$
  - N^{IN} : number of elements for input data
  - k^{IN} : data type (precision) of input data

The HPC service F characterizes by the following parameters:

- HPC service F: the parallel filtering algorithm applied, for instance parallel Kalman filter
- Output data size $Q^{OUT}[B]$: indicates the size of the output data, which will be produced by HPC instance and transferred to WS

- $Q^{OUT}[B] = N^{OUT} * k^{OUT}$
- N^{OUT} : number of elements for output data
- k^{OUT} : data type (precision) of output data

The data transfer in the HPC service could be described in Figure 3.1 the producer will provide the input data and place it in the workspace WS, and determine the use case task. When the HPC service is called, it has access to the input data (signals) and starts processing them and once the job is finished by the HPC service, the output data (signal) is generated as a binary file in the workspace WS.

The main structure of the HPC service could be described in Figure 3.2, the first step with the help of the MPI-IO-Read method [7] the input data is read and loaded into a local buffer in parallel in each MPI-rank, especially we are uniformly distributing the input data among the MPI-rank, second step the data (local buffer) is passed into filtration process (Kalman filter, FFT filter) to process the data, and in the third step the output data is written by MPI-IO-Write method [7] in parallel as a binary file in workspace WS.

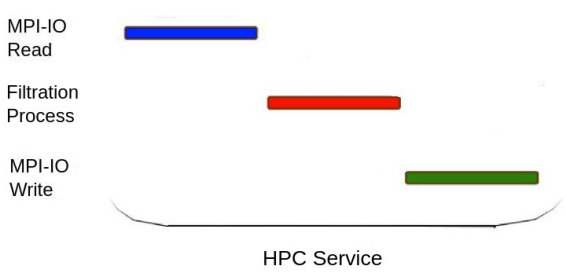


Figure 3.2: HPC service structure

Implementation of HPC Service in SERRANO : It described before this HPC service has been implemented in parallel distributed memory in OpenMP/MPI, additionally, the data structure of the implementation is completely templated. We have introduced two different data types for input and output data, which provide this capability for the user to choose various data types for input/output data in correlation with transprecision techniques and Verification, Validation, and Uncertainty Quantification (VVUQ), objectives of Task 4.2 .

4 Filtration Process

Filteration is the main part of our HPC service in the SERRANO platform. When the input data is ready, it is passed into the filters. We provide two filters 1) Kalman filter, 2) FFT filter (Fast Fourier transformation), both of these filters have had hybrid parallel shared and distributed memory implementation in OpenMP/MPI. We present the parallelization of these filters, especially in the Kalman filter. The strong scalability property of the Kalman filter and FFT filter will be investigated, which means that we explore how the execution time would decrease by increasing the number of processes by fixing data size.

An independent prototype of the parallel Kalman filter will be introduced into our framework, this prototype computes the Kalman filter with multiple parameters in parallel.

4.1 Kalman Filter

Kalman Filter is also called linear quadratic estimation used to estimate unknown variables with a series of measurements such as statistical noise and uncertainties [16]. Kalman Filter has a wide range of applications in guidance, navigation, and finance. Every electrical device in the real world like cell phones and airplanes has sensor devices and its purpose is to measure values like position velocity or acceleration. However, the sensors will not always give us the true measurement because they have some noise. Consider when an airplane is at high speed, and it has some turbulence and the plane shakes which leads the sensor in the airplane also shakes, therefore the sensors will not give us the correct position measurement. This is the place that the Kalman filter comes into play by providing optimal sensor reading from noisy measurement.

Kalman filter is based on a linear dynamical system discretized in the time domain. We have a process model 4.1 that represents the transformation of the state value x_{k-1} to x_k at time step $k-1$ and k respectively [6].

$$x_k = Fx_{k-1} + Bu_{k-1} + w_{k-1} \tag{4.1}$$

In formula 4.1 the state transition matrix is represented by F, the control input matrix defined by B, and w_{k-1} is the process noise term that added into the model and is supposed to be the zero-mean Gaussian with covariance Q (i.e $w_{k-1} \approx N(0, Q)$) [12].

the measurement model 4.2 demonstrates the relation between the state value x_k and

measurement value y_k at time step k, where H is the measurement matrix, and v_k is measurement noise that is defined by zero-mean Gaussian with covariance R (i.e $v_k \approx N(0, R)$).

$$y_k = Hx_k + v_k \tag{4.2}$$

The purpose of the Kalman Filter is to estimate the state value x_k, and it is represented by $\hat{x_k}$. To describe how the Kalman filter works, we need to define some parameters.

- P represents the error covariance between x_k and $\hat{x_k}$ ($Cov(x, \hat{x}) = P$).
- K represents Kalman gain.
- We will assume H and F are constant terms in linear filters.

The Kalman filter algorithm is described in (4.1.0), the algorithm receives the input data x_k at time step k, and return the approximation of it by $\hat{x_k}$, this current value $\hat{x_k}$ also used to estimate the next input data x_{k+1}. The Kalman gain K and error covariance P will also be updated in each iteration of Kalman filter.

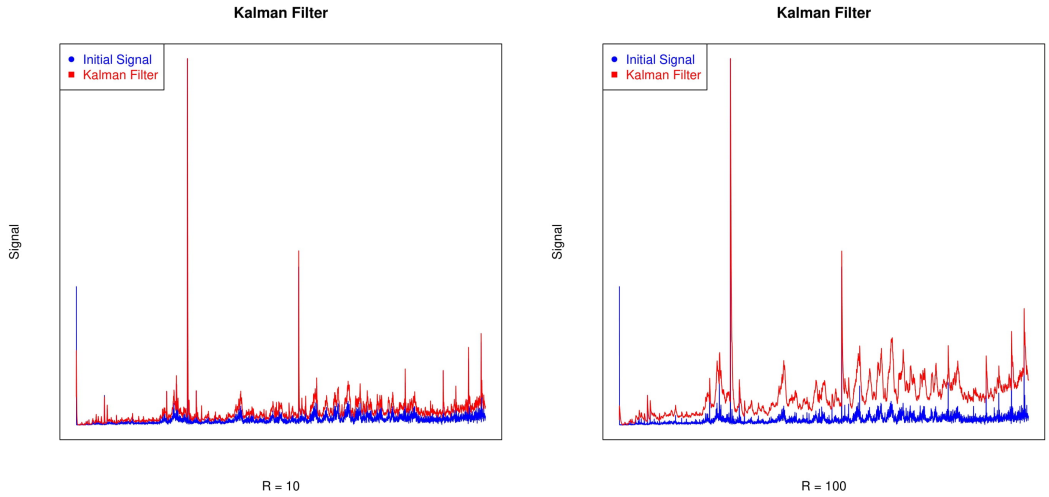
Algorithm 4.1.0 Kalman Filter (x_k, P, K)

1:	$K = P.H/(H.P.H + R)$	(update the Kalman gain)
2:	$\hat{x_k} = \hat{x_k} + K.(x_k - H.\hat{x_k})$	(x_k approximation)
3:	$P = (1 - K.H).P + Q$	(update error estimation)
4:	$(P, K, \hat{x_k})$	(updated values for the next approximation of x_{k+1})

In the parameter setting the bigger value designated to the R, the more noises removed from the input data, and consequently the fewer noises removed from input data by choosing the smaller value of R. On the other hand, the bigger value of K (Kalman gain) we have, the slower the filter will converge. The idea is to make a filter fast but also reduce noise as much as possible. Consequently, we have to find a balance between speed and noise removal. The effect of different values of R in the removing noises from the input data is shown in Figure 4.1. By choosing $R = 100$ more noises removed from the input data (signal) and therefore the difference between the input input data (signal) and filtered data is higher Figure 4.1b, consequently by setting $R = 10$ fewer noises removed from the input data Figure 4.1a.

4.1.1 Parallelization of Kalman Filter

In this section, we will explain the parallelization of the Kalman filter. In this approach, the Kalman filter algorithm would call concurrently on the input data. The serial algorithm of



(a) Kalman filter with $R = 10$ (b) Kalman filter with $R = 100$

Figure 4.1: Kalman filter with different value of R

Kalman filter described in the previous section, we have observed in each approximation of Kalman filter, recent updates of parameters are required, this property makes the Kalman filter algorithm inherently sequential, and then parallelization of such algorithm is not trivial.

Our interest is to provide parallelization of Kalman filter where the Kalman filter could execute simultaneously on part of data and produce the approximation that comparable to sequential Kalman filter. Parallelization would have been accomplished at two-level: 1) parallel shared memory 2) parallel distributed memory.

Parallel shared-memory implementation of Kalman filter : In parallel computing, all processors may have access to shared memory to exchange information between processors, and parallel shared memory implementation of Kalman filter means multi-threaded implementation of Kalman filter algorithm. We use OpenMP which is an implementation of multi-threading in C/C++, in this framework, a specific number of threads would spawn, and tasks will be divided among the threads. Consequently, the thread pool run and work in part of data concurrently, moreover each thread is pinned to a different core.

First, the input data is partitioned among the threads uniformly (we assumed that total number of threads is m), therefore a portion of data that belongs to thread p represented by $(x_k)_{tid=p}$, then each thread independently applies the sequential Kalman filter algorithm to its part of data. Kalman Filter algorithm approximate $\hat{x_k}$, and updates parameters P, K. After each thread computes their approximation and parameters updates, the barrier called to the thread region [1], and thread $tid = p$ $(p > 0)$ accessed to the last update

[1]The barrier is a synchronization technique for a group of threads, which indicates that any thread must stop at this point until all threads have crossed the barrier.

of x_k, P, K in thread $tid = p-1$, finally the sequential Kalman filter algorithm is applied again in thread $tid = p$ $(p > 0)$ with the new parameters and re-approximate x_k just in a few iterations (in our case we set to the first seventy iterations).

Algorithm 4.1.0 Parallel shared memory algorithm of Kalman filter $((x_k)_{tid=p}, P, K)$

1: Apply Kalman Filter($(x_k)_{tid=p}, P, K$))
2: Save the last update of (x_k, P, K) in each thread
3: Putting barrier in thread region
4: Apply again Kalman Filter($(x_{k<70}, P, K)$) with new parameter from thread $p-1$

Parallel distributed-memory implementation of Kalman filter : In distributed computing, each processor has access to private memory (distributed memory), therefore to access the data from other processes we require some protocol. Message Passing Interface (MPI) is a standard communication library that provides point to point and collective communication, in addition in distributed computing like parallel computing the data is uniformly distributed along with the processes, so each process works on its part, and data will be exchanged via Message Passing Interface (MPI) through processes.

We introduce parallel distributed memory of Kalman filter, where the Kalman filter parallelized in multi-process in distributed memory (we assumed that the total number of processes is n). In this case, the input data will be distributed among the processes uniformly, therefore $(x_k)_{proc=i}$ is a portion of the input data that belongs to the process i. Each process could apply the Kalman filter on its data $(x_k)_{proc=i}$. After each process, i finishes its approximation, with the help of MPI-send method the last updates of (x_k, P, K), are sent to the next neighbor process $i+1$ $(i < n)$, once the process i $(i > 0)$ receive the data, then re-approximate the x_{k+1} just in few first iterations (again in our case we fix the re-approximation to the first seventy iterations).

Algorithm 4.1.0 Parallel distributed memory algorithm of Kalman filter $((x_k)_{proc=i}, P, K)$

1: Apply Kalman Filter($(x_k)_{proc=i}, P, K$))
2: Send the last update of (x_k, P, K) to the process $i+1$ if $i < n$
3: Receive the last update of (x_k, P, K) from the process $i-1$ if $i > 0$
4: Apply again Kalman Filter($(x_{k<70}, P, K)$) with new parameter received from process $i-1$

4.1.2 Scalability Property of Kalman Filter

We have introduced the hybrid parallel Kalman filter implementation, which provides the capability that Kalman filter algorithm executed in multi-process and multi-thread, therefore it is interesting to observe the benefit of hybrid parallel implementation of Kalman

filter. To address this question, the strong scalability property of the parallel Kalman filter algorithm will be investigated. To verify the strong scalability property, we consider big input data size and measure the execution time by increasing the number of processes, if the execution time is dropping by increasing the number of processes, then we could conclude that the parallel Kalman filter algorithm has strong scalability property. In this case, 1GB of data is considered as input data, and measure the execution time in $\{1, 2, 4, 8, .., 64, 128\}$ processes (hardware architecture are explained in section 5.1) and observed how execution time behaves by adding more processes (Table 4.1).

We define a new parameter `speed up in num-proc k`:

$$\texttt{Speed up in num-proc k} = \frac{\texttt{Execution time in num-proc 1}}{\texttt{Execution time in num-proc k}} \quad (4.3)$$

Number of proc	Execution time [s]	Speed up
1	1.53833	-
2	0.790223	1.94
4	0.422934	3.63
8	0.240806	6.38
16	0.14528	10.58
32	0.0729897	21.07
64	0.0376237	40.88
128	0.0186645	82.42

Table 4.1: Execution time of Kalman Filter in different processes in 1GB of data

It is observed in Table 4.1 that the speed up parameter is increasing (the speed up is not linear because of some communication overhead that exists in parallelization of the Kalman filter), by increasing the number of processes, that implies the execution time would drop by increasing the number of processes, therefore we could conclude that the parallel Kalman filter has strong scalability property.

4.1.3 Parameter Analysis of Kalman Filter

A prototype of the Kalman filter has been implemented that provides us this means that Kalman filter could process input data (signals) with multiple parameters in parallel. In this scenario instead of all the processes communicating to each other, we have partitioned the total number of processes into several group communicators (Figure 4.2), and the processes that involved in each group communicator can communicate to each other. The group communicator are constructed with (`MPI-Comm`) method in MPI.

It has been mentioned in section 3 one of the objectives of the SERRANO project is developing an appropriate framework for Verification, Validation, and Uncertainty Quantification VVUQ (see task 4.2), therefore introducing this prototype is also in direction of the

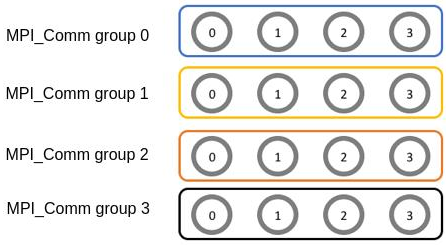


Figure 4.2: MPI group communicators

VVUQ that could help us to compute the Kalman filter with several parameters in parallel and then verify the best parameters setting.

In this experiment, we have investigated the error behavior in the Kalman filter. Kalman Filter computes $\hat{x_k}$ the approximation of input data x_k, this approximation naturally produces the error (the difference between the approximated value $\hat{x_k}$ and the input data x_k), which is called the absolute error.

$$e_k = |\hat{x_k} - x_k|$$

We could also measure the error for all the input data k, by computing the L_2, and max norms 4.4.

$$||e||_{L2} = (\sum_k e_k^2)^{1/2} \qquad ||e||_{max} = max_k\{e_k\} \tag{4.4}$$

The relative error, η_k defined by absolute error divided by input value

$$\eta_k = \frac{e_k}{|x_k|}$$

This error could be measured for all the input data k by computing the L_2, and max norms.

Error in Kalman filter could be controlled by parameters R (noise covariance), we have observed in the previous section R parameter would have a substantial effect on the input data (signals). Besides, that error will be varied by the parameter R, the higher quantity of the R parameter will remove more portion of noises from the input data, therefore the error norm will be bigger, and as result, the error norm will grow by the increasing parameter R.

In this experiment, we will investigate the error behavior for $R = [0, 100)$, with 16 MPI ranks and 4 MPI communicator groups, and each communicator group took an interval of R. For instance the group communicator 0 takes the interval $[0, 25)$, group communicator 1 takes an interval $[25, 50)$, group communicator 2 takes an interval $[50, 75)$ and group communicator 3 takes an interval $[75, 100)$. Then each communicator group computes the Kalman filter and error for several values of R in its interval, and all these processes have been implemented in parallel. It is demonstrated in Figure 4.3 the max and L_2 error norm for different values of R that is computed by this prototype of Kalman filter.

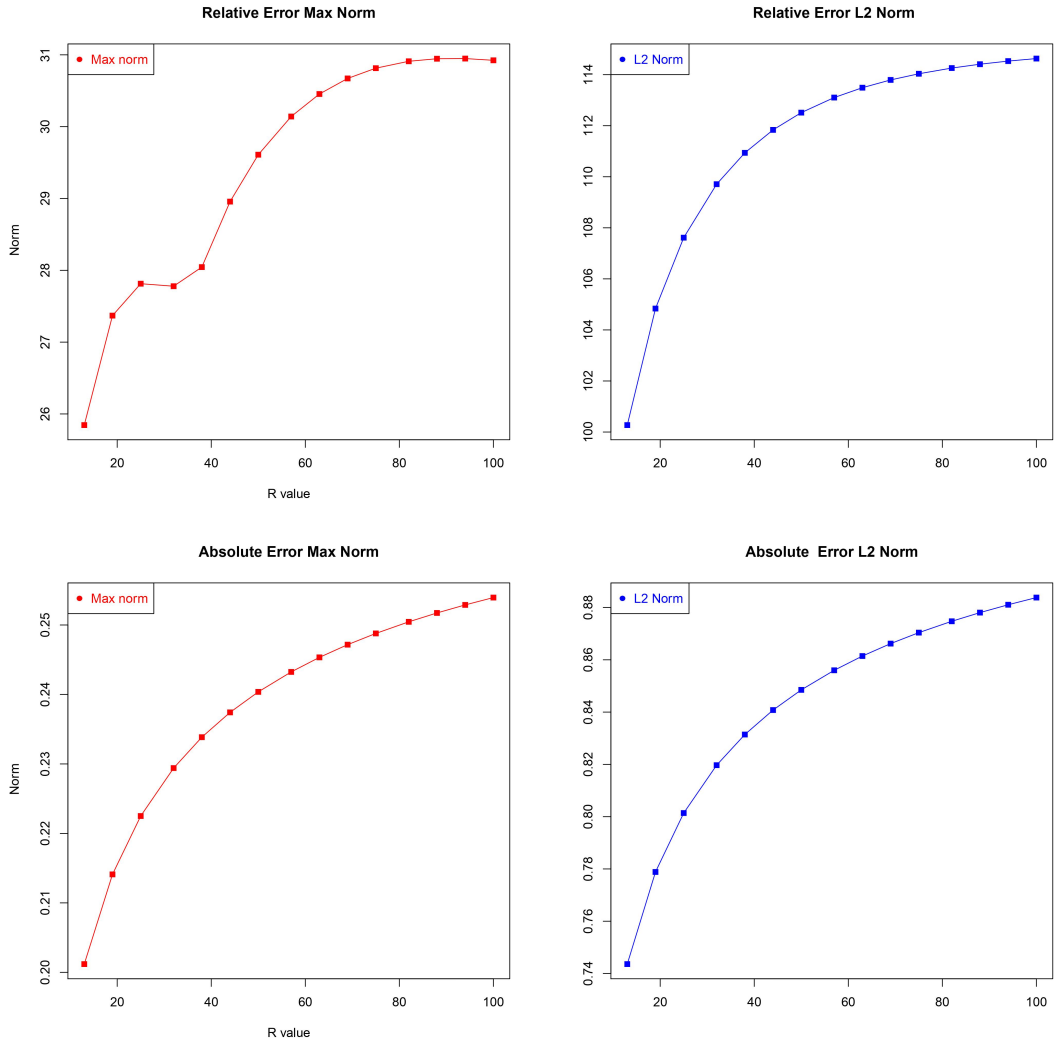


Figure 4.3: The absolute and relative error behavior with multiple value of R

4.2 Fast Fourier Transformation

Fast Fourier Transformation is a method that converts the signals from the original space (time-space) into the frequency domain [10]. To introduce this process consider the function $f : \mathscr{T} -> \mathscr{C}$. The Fourier series [15] defined with

$$\sum_{k=-\infty}^{+\infty} c_k e^{2\pi k i \theta} \tag{4.5}$$

c_k coefficients in Fourier series are defined by

$$c_k = \int_{\mathscr{T}} e^{-2\pi k i \theta} d\theta. \tag{4.6}$$

Fourier series (4.5) converge to function f if the function $f \in C^1(\mathcal{T})$ [2], we could also express that $\{e^{2\pi k i\theta}\}_k$ functions are the base functions for the space $C^1(\mathcal{T})$, and it required to compute the coefficients c_k, to find the approximation of each function $f \in C^1(\mathcal{T}))$.

We could also introduce DFT (Discrete Fourier Transformation) [13] that is just compute the coefficient of c_k when the values of function f are defined on the finite number of points $\{0, 1/N, 2/N, ..., N-1\}$ which represent by $f(n/N) = x_n$, then we have

$$X_k = \frac{1}{N} \sum_{n=0}^{N-1} x_n e^{-ik\frac{2\pi n}{N}} \tag{4.7}$$

In order to show the process of DFT, consider the continues time signals in 4.8, we represent the signal in first one second by discretizing the time interval in $[0, 1]$ in Figure 4.4a. This time interval has been partitioned into 256 parts.

$$f(t) = 0.5sin(2.\pi.10.t + \pi/6) \qquad t = 1/256 \times i \quad i \in [0, 255] \tag{4.8}$$

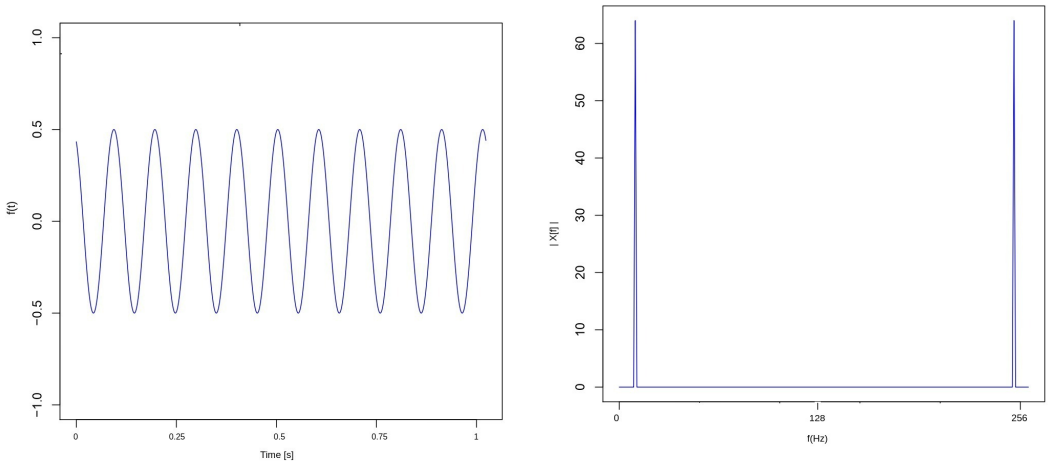
Computing the values of function f in each of the time step t, will result the following discrete vector of signals,

$$f = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ . \\ . \\ . \\ x_{255} \end{bmatrix} \tag{4.9}$$

Vector signal f is in the time domain, and the Fourier transformation or in this case Discrete Fourier Transformation (DFT) transforming the signals into frequency domain, therefore the result of DFT $(X[f])$ would be the sequence of complex number.

$$X[f] = \begin{bmatrix} X_{0re} + jX_{0im} \\ X_{1re} + jX_{1im} \\ X_{2re} + jX_{2im} \\ . \\ . \\ . \\ X_{255re} + jX_{255im} \end{bmatrix}$$

[2] continuous functions that have continuous first derivative



(a) Discrerete signal f is time domain

(b) Discrerete signal f in the frequency domain (magnitude of the spectrum of DFT)

Figure 4.4: Discrete signal f in time and frequency domain

We compute the magnitude of spectrum to present DFT in 4.10, and in Figure 4.4b it has been shown the magnitude of spectrum of $X[f]$.

$$|X[f]| = \begin{bmatrix} \sqrt{X_{0\,re}^2 + X_{0\,im}^2} \\ \sqrt{X_{1\,re}^2 + X_{1\,im}^2} \\ \sqrt{X_{2\,re}^2 + X_{2\,im}^2} \\ \cdot \\ \cdot \\ \cdot \\ \sqrt{X_{255\,re}^2 + X_{255\,im}^2} \end{bmatrix} \tag{4.10}$$

The computational cost of DFT is high, if we want to compute the DFT using the formula 4.7 the computational cost is $O(N^2)$. The FFT algorithm (Fast Fourier Transformation) [14] provides us a fast way to compute the DFT, and computational cost is just $O(Nlog(N))$, which is more cheaper than DFT.

We are using the `FFTW3` library [4] to compute the coefficient X_k in Discrete Fourier Transformation (DFT) using FFT algorithm. The `FFTW3` library also provides us the parallel version FFT algorithm using MPI, therefore we can distribute the input signal uniformly across the processes and then `FFTW3` plan compute the DFT.

In order to use the parallel FFT in `FFTW3` library, it requires to include `<fftw3-mpi>`, and initialize the MPI setting.

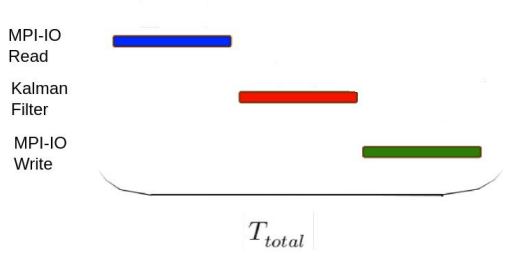
```
#include <fftw3-mpi.h>
int main()
{

 MPI_Init(&argc, &argv);
 fftw_mpi_init();

         fftw_plan plan;
         fftw_complex *input_data;
         fftw_complex *output_data;

         plan =
         fftw_mpi_plan_dft_1d
            ( N
            , input_data
            , output_data
            , MPI_COMM_WORLD
            , FFTW_FORWARD
            , FFTW_ESTIMATE);

        fftw_execute(plan);
        fftw_destroy_plan(plan);

 MPI_Finalize();
 return 0;
}
```

`fftw-plan` is a plan that is created to compute FFT and then the parameters are passed to this plan. The total number of signals is determined by `N`, the `input-data` is buffer that contains the input data (signal), and FFT of input signals will be stored in the `output-data`. With the help of `fftw-execute`, we execute the plan, and after that, we destroy it.

5 Performance Analysis of HPC Service

We have described the fundamentals of HPC service in previous sections, therefore we are willing to measure the Performace and execution time of the HPC service and its components. As explained before the main structure of the HPC service contains three-step. The First MPI-IO-Read method reads the input data in parallel from workspace WS, in the second step the Kalman filter is used for filtering the input data, and in the last step the output data is written into workspace WS by MPI-IO-Write method. We could show in Figure 5.1 the execution time of each of these components. The total execution time 5.1 is shown with $\mathtt{T}_T$, which is equal to the summing up the execution time of MPI-IO-Read, parallel Kalman filter, and MPI-IO-Write that presented by $\mathtt{T}_R$, $\mathtt{T}_K$, $\mathtt{T}_W$ respectively.

Figure 5.1: The main structure of HPC service



$$\mathtt{T}_T = \mathtt{T}_R + \mathtt{T}_K + \mathtt{T}_W \tag{5.1}$$

First, it will be described how the performance FLOPS of the Kalman filter is computed. FLOPS is criteria to measure the performance, in order to measure FLOPS we should consider the code, then sum up all floating-point operations, and divide by the execution time of the code [9]. In this case, all the ordinary operations like subtractions, multiplication read, write should be counted. Consider the Kalman filter algorithm in each iteration we have these floating-point operations.

Floating point operation	Number of operation
Multiplication	7
Division	1
Summation	4
Subtraction	1
Read bytes	1.sizeof(input-data)
Write bytes	1.sizeof(output-data)

Table 5.1: Floating point operation in Kalman Filter

Therefore the FLOPS of Kalman filter that presented by $\texttt{FLOPS}_K$ is computed by the following formula

$$\texttt{FLOPS}_K = \frac{\sum \texttt{Number of Operation}}{\texttt{T}_K}$$

Bandwidth is another important criterion in analyzing the performance of the framework. We introduce three types of Bandwidth, Read-Bandwidth, Write-Bandwidth, and Total-Bandwidth,

$$\texttt{Read-Bandwidth}_K\texttt{[B/s]} = \frac{\texttt{data to read [B]}}{\texttt{T}_K}$$

$$\texttt{Write-Bandwidth}_K\texttt{[B/s]} = \frac{\texttt{data to write [B]}}{\texttt{T}_K}$$

$$\texttt{Total-Bandwidth}_K\texttt{[B/s]} = \frac{\texttt{data to read [B] + data to write [B]}}{\texttt{T}_K}$$

We also describe the floating-point operation that exists in the MPI-IO-Read method. It is obvious that in the MPI-IO-Read method we just have one primary operation for Read-bytes, therefore we can show in the table 5.2.

Floating point operation	Number of operation
Multiplication	0
Division	0
Summation	0
Subtraction	0
Read-bytes	1.sizeof(input-data)
Write-bytes	0

Table 5.2: Floating point operation in MPI-IO-Read method.

Based on the table 5.2 that shows the primary floating-point operation, we could conclude that FLOPS and Bandwidth have the same interpretation in the MPI-IO-Read method.

$$\texttt{Read-Bandwidth}_R\texttt{[B/s]} = \texttt{FLOPS}_R$$

The floating-point operation in the MPI-IO-Write method is similar to the MPI-IO-Read method, however, in this case, we have a Write-bytes operation.

We also define the total FLOPS of the HPC service with,

$$\texttt{FLOPS}_T = \frac{\sum \texttt{Number of Operation in Kalman Filter}}{\texttt{T}_T}$$

5.1 Hardware Architecture

We briefly describe the hardware architecture that applied in our experiment and how we distribute and assign the jobs (pinning pattern) in parallel in hardware.

Compute node in Hawk supercomputer has 8 NUMA nodes, and each NUMA node has 16 cores (for technical details refer to section 5 in [3] and [8]), therefore the total number of cores in one compute node is 128.

```
total-num-cores   =   num-numa x num-cores

total-num-cores   =   8 x 16
```

The MPI jobs will be submitted to this architecture in a symmetric pattern. Consider the case that we want to execute the job with 8 MPI-rank to this compute node, then the distance of rank is defined by

```
dist-rank =  total-num-core / MPI-size
```

`dist-rank` is a parameter that determines the distance of MPI-rank's from each other and specifies the pinning pattern of cores in compute node. In this case, the `dist-rank` is 16, which means that MPI-rank will be pinned in the following pattern:

```
MPI-rank=0    pinned to core 0
MPI-rank=1    pinned to core 16
MPI-rank=2    pinned to core 32
MPI-rank=3    pinned to core 48
MPI-rank=4    pinned to core 64
MPI-rank=5    pinned to core 80
MPI-rank=6    pinned to core 96
MPI-rank=7    pinned to core 112
```

Moreover, if the job also runs in multithread, then each thread should be pinned to a specific core. We also define a bound to the number of threads that are submitted, it should be smaller or equal to the `dist-rank` parameter.

```
num-threads  <=  dist-rank
```

Consider the previous case, that job submitted with 8 MPI-rank and dist-rank was 16, then the number of threads should be smaller or equal to 16. If the job is executed with 16 threads then the maximum capacity of the compute node has been used.

5.2 Numerical Experiment

We have explained the computation of the Performance (FLOPS) of the HPC service and demonstrated the hardware architecture and pinning pattern of parallel jobs in compute node in the Hawk. To address the challenges regarding optimal utilization of many processors platforms (Task 4.2). Therefore we plan to benchmark the Performace and execution time of the HPC service and its component in different data sizes $\{1GB, 5GB, 10GB\}$ [1]. In this experiment, the HPC service will be executed in different numbers of processes (num proc) $\{1, 2, 4, 8, 16, 32, 64, 128\}$. The performance (FLOPS), execution time, and speed-up (4.3) will be computed in each process, and then we find the maximum performance (FLOPS), minimum execution time, and maximum speed-up which is the main objective of Task 4.2. Therefore we define these parameters:

- Core * : Number of processes that maximum performance achieved.
- Max FLOPS : Maximum performance (FLOPS).
- Min exe : Minimum execution time [s].
- Max speed-up : Maximum speed-up.

In these experiments, the compiler and MPI versions have been fixed with this configuration

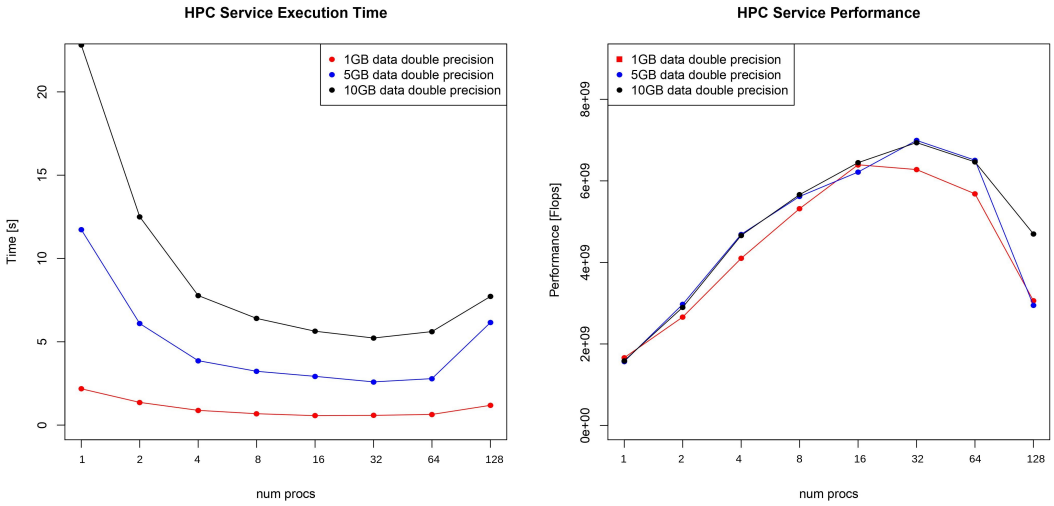
```
compiler = gcc/9.2.0     MPI = openmpi/4.0.4
```

Total Performance of HPC service : We have computed the execution time and performance of the HPC service for three data sizes. In Figure 5.2a it has been observed the execution time would drop at the same rate by increasing the number of processes for all data set until (32,64) processes, and by moving to the next higher number of processes the execution time would increase and lose the scalability property. The performance (FLOPS) of HPC service (Figure 5.2b) would increase by increasing the number of processes and pick performance reaches between 16 and 32 processes. Table 5.3 presents maximum performance, the number of cores (MPI-rank) that maximum performance of the HPC service achieved, and the minimum execution time of this experiment (5.2).

Data size	Core *	Max FLOPS	Min exe [s]	Max speed-up
1GB	32	6.19507e+09	0.54	3.74
5GB	64	6.79655e+09	2.55	4.25
10GB	64	6.79655e+09	5.14	4.35

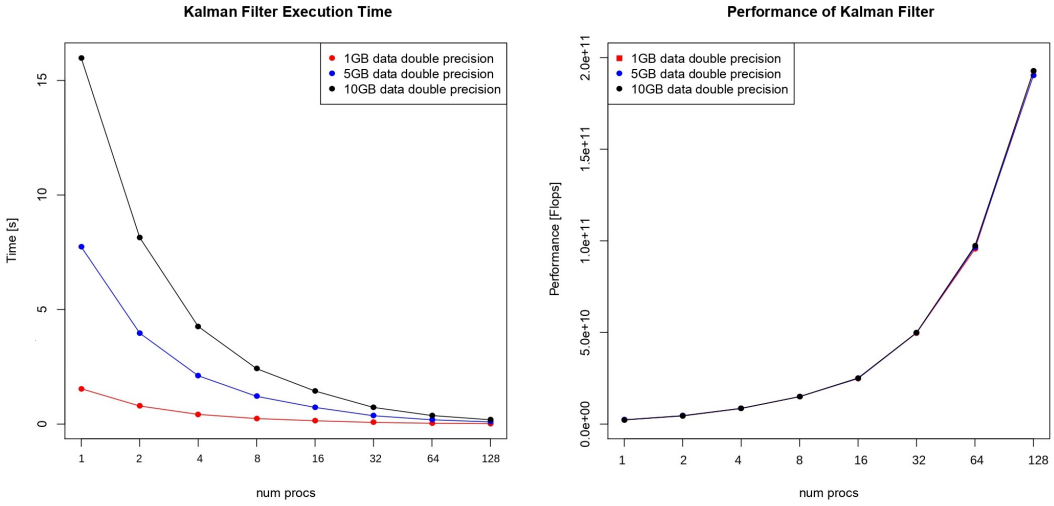
Table 5.3: Maximum performance (FLOPS) and minimum execution time of HPC services

[1] $1GB = 1.000.000.000$ bytes



(a) Execution time of the HPC service (b) Performance of the HPC service

Figure 5.2: Execution of the HPC service in different data sizes



(a) Execution time of the Kalman filter (b) Performance of the Kalman filter

Figure 5.3: Execution of the Kalman filter in different data sizes

Performance of Kalman Filter : The performance and execution time of the Kalman filter are computed and presented for different data sizes in Figure 5.3a. It shows that Kalman filter execution time would decrease approximately by a factor of two by increasing the number of processes, which also proves the strong scalability property of parallel Kalman filter. The performance (FLOPS) of the Kalman filter incremented by increasing the number of processes (Figure 5.3b). It has been presented in Table 5.4 the maximum performance and maximum speed-up of the Kalman filter for different data sizes.

Performance of MPI-IO-Read/Write : It mentioned before the performance (FLOPS) and bandwidth have the same meaning in MPI-IO-Read/Write methods, and by measur-

Data size	Core *	Max FLOPS	Min exe [s]	Max speed-up
1GB	128	5.24044e+14	0.01	82.42
5GB	128	2.28193e+13	0.09	81.04
10GB	128	4.32897e+13	0.18	84.97

Table 5.4: Maximum performance (FLOPS) and minimum execution time of the Kalman filter

ing the performance (FLOPS) we compute the bandwidth. It has been presented in Figures 5.4a, 5.4c the execution time in MPI-IO-Read and MPI-IO-Write methods will decrease at the beginning, however the scalability property breaks at a certain number of processes, and consequently, bandwidth would start to decrease (Figures 5.4b, 5.4d). The maximum bandwidth for MPI-IO-Read and MPI-IO-Write methods have been presented in Tables 5.5 and 5.6 respectively, both of these methods suffer from low speed-up, and this phenomenon would have affected the total Performance and execution time of the HPC service.

Data size	Core *	Max FLOPS	Min exe [s]	Max speed-up
1GB	64	1.8431e+10	0.05	3.16
5GB	64	1.98345e+10	0.22	5.35
10GB	64	1.83819e+10	0.42	2.87

Table 5.5: Maximum bandwidth and minimum execution time of the MPI-IO-Read method

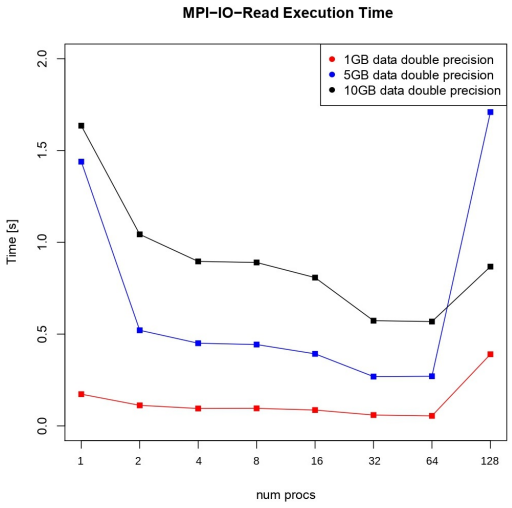
Data size	Core *	Max FLOPS	Min exe [s]	Max speed-up
1GB	4	9.71499e+10	0.36	1.38
5GB	4	8.57689e+09	1.29	1.97
10GB	4	5.76323e+09	2.43	1.89

Table 5.6: Maximum bandwidth and minimum execution time of the MPI-IO-Write method

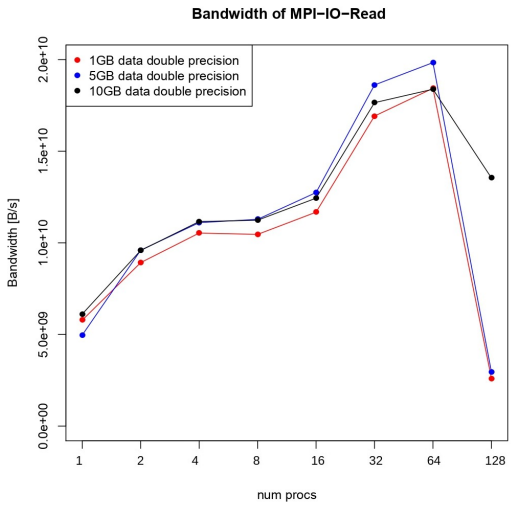
5.2.1 Using Different Data Type in HPC service

The data precision that has been used in experiments in section 5.2 was fixed to double-precision, however, as mentioned before the data structure of the HPC service is templated and could support different data precision. Choosing a suitable data type is transpression and approximation techniques (see Task 4.2) that could improve the execution time of the HPC service. Therefore it is demonstrated in Figure 5.5a for 1GB of data changing the data type from double-precision to single-precision (float) could improve the execution time of HPC service. Table 5.7 presents the minimum execution time, and by using single-precision minimum execution time improved by 46%.

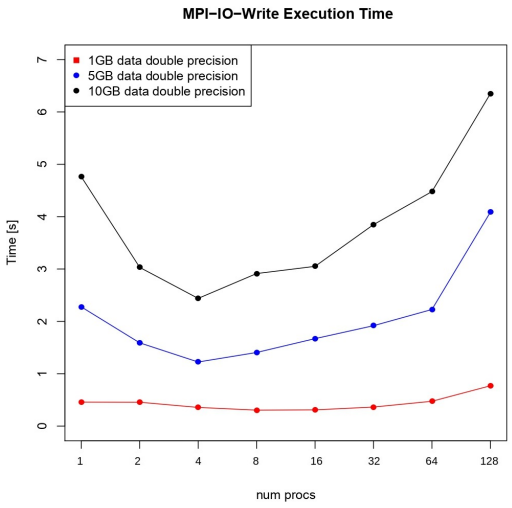
Additionally, the data structure of the HPC service supports different data types for input and output data, which means that the input and output data should not necessarily have the same data precision. For instance, the input data could be in double-precision and the output data will be single-precision, in Figure 5.5b and Table 5.8 present the effect of this



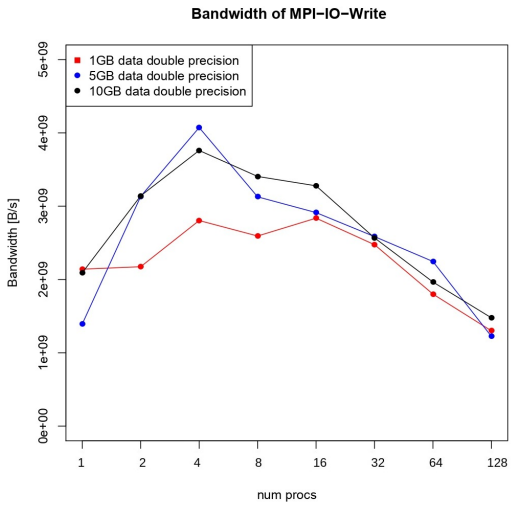
(a) Execution time of the MPI-IO-Read method



(b) Performance of the MPI-IO-Read method

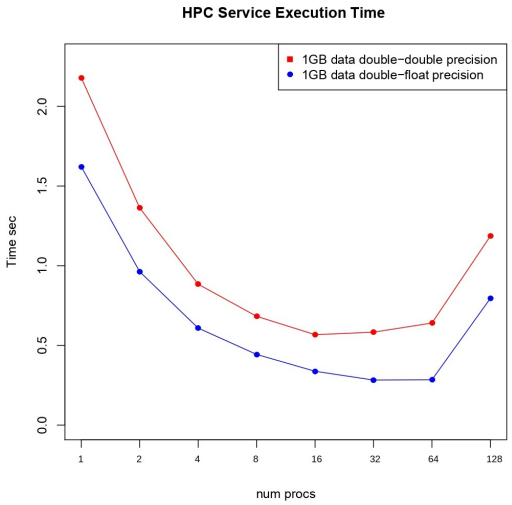


(c) Execution time of the MPI-IO-Write method

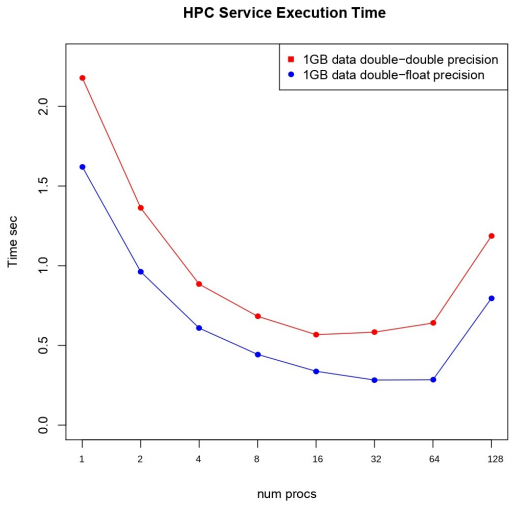


(d) Performance of the MPI-IO-Write method

Figure 5.4: Execution of the MPI-IO-Read/Write in different data sizes



(a) Execution time of the HPC service with single-precision for input and output data



(b) Execution time of the HPC service with double-precision for input data and single-precision for output data

Figure 5.5: Execution of the HPC service in different data precision

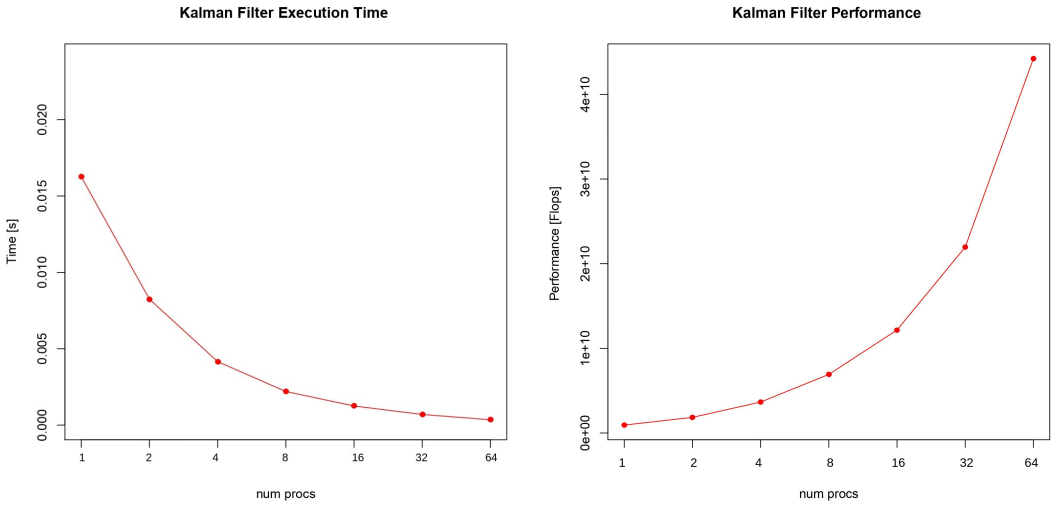
Input data precision	Output data precision	Min exe [s]	Execution time improvment
double-precision	double-precision	0.54	-
single-precision	single-precision	0.25	46%

Table 5.7: Execution time improvment by changing a data precision

scenario in the execution time of the HPC service compares to the case that both input and output data were fixed to double-precision.

Input data precision	Output data precision	Min exe [s]	Execution time improvment
double-precision	double-precision	0.54	-
single-precision	single-precision	0.27	50%

Table 5.8: Execution time improvment by changing a data precision for output data



(a) Execution time of the Kalman filter (b) Performance of the Kalman filter

Figure 5.6: Execution of the Kalman filter in the Exescc cluster

5.2.2 Energy Consumption of Kalman Filter

We also compute the Kalman filter with small data sizes. In this case, we have used input data that has $\{524288\}$ elements ($4MG$[2]), and provided by project partner AUTH and they used this input data for their experiment that will be presented in section 6. In the first step the execution time and performance of the Kalman filter with this input data time will be presented in Table 5.9 and Figure 5.6a. The Performace and execution behave the same as previous cases in big data sizes. We observe that execution time decreased by almost a factor of two by using more cores, and the performance incremented respectively (Figure 5.6b).

Number of proc	Execution time [s]	FLOPS	Speed up
1	0.0162542	9.35411e+08	-
2	0.00824142	1.84487e+09	1.95
4	0.00415742	3.65834e+09	3.90
8	0.00219816	6.92049e+09	7.61
16	0.00125168	1.21688e+10	12.8
32	0.000694357	2.19643e+10	23.18
64	0.000347838	4.42292e+10	47.05

Table 5.9: Execution time and performance of the Kalman Filter

Computing the energy consumption of kernels and finding the parameters to minimize the energy consumption is one of the objectives of task 4.2. We also compute the energy consumption of the Kalman filter in this experiment, the energy measurement has been done in the Exescc cluster [3] at HLRS. We have applied four different frequencies in energy

[2] 4.000.000 bytes

measurement (for technical details refer to section 5 in [3]), and it has been observed in Figure 5.7 by using more processes less energy will be consumed in the execution of the parallel Kalman filter. In Table 5.10 we present the power and energy consumption of the Kalman filter in Turbo frequency mode.

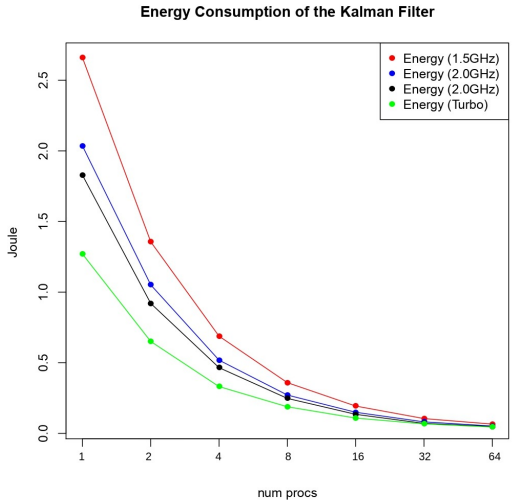


Figure 5.7: Energy consumption of the Kalman filter

Number of proc	Power (Watt)	Energy (Joule)
1	228.97	1.27
2	231.97	0.65
4	234.97	0.33
8	237.97	0.18
16	240.97	0.10
32	243.97	0.06
64	246.97	0.04

Table 5.10: Power and energy consumption in Turbo frequency in Kalman filter

5.3 Performance Analysis of FFT Filter

We investigate the execution time of the FFT filter exclusively in this section, and in the next step, we try to benchmark both the FFT filter and Kalman filter in the HPC service. It is essential to mention that the parallel FFT filter works if the number of input signal be 2^N. Therefore if we receive some input signal first we need to check that the number of signals satisfies this condition, otherwise, it is required to cut the signal to the lowest nearest 2^N.

We measured the execution time of the FFT filter in `FFTW3` library in different data sizes. The data set that we choose in this experiment has the following size.

$$2^{30}\texttt{sizeof(double)} = \texttt{8.6GB}$$

$$2^{29}\texttt{sizeof(double)} = \texttt{4.3GB}$$

$$2^{21}\texttt{sizeof(double)} = \texttt{16MB}$$

The execution time of the FFT filter has been measured in two big data sets $\{4.3GB, 8.6GB\}$ in the Figure 5.8 it has been presented the execution time decreases by increasing the number of processes, however, the rate of reductions are not the same in all processes. The Table 5.11 provides the information regarding the parameters that introduced 5.2 minimum execution time and maximum speed-up.

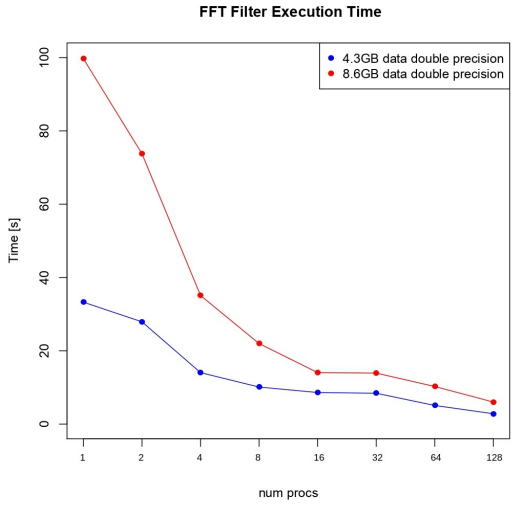


Figure 5.8: The execution time of FFT filter for $\{4.3GB, 8.6GB\}$ data sizes

The execution time of the FFT filter has been measured for small data size ($\{16MB\}$) in Figure 5.9, it has been observed that the execution time would start to increase untill process 4, and by increasing the number of the processes the execution time would increase. We are also interested to investigate the effect of changing the data type from

Data size	Core *	Min exe [s]	Max speed-up
4.3GB	128	2.78	11.52
8.6GB	128	5.95	14.15

Table 5.11: Minimum execution time and Maximum speed-up of the FFT filter method

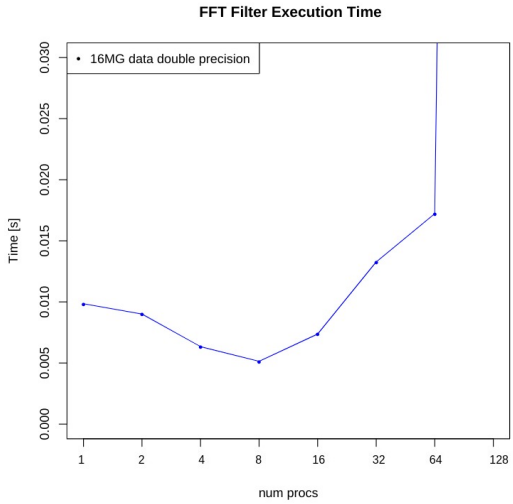


Figure 5.9: The execution time of FFT filter for $16MG$ data size

double-precision to single-precision in the execution time of the FFT filter. In Table 5.12 it has been shown the input data once considered as single-precision the execution time improve 46%. It is transpression and approximate computing techniques (see Task 4.2) that changing the data types could improve the execution time of the FFT filter.

Input data precision	Output data precision	Min exe [s]	Execution time improvment
double-precision	double-precision	2.78	-
single-precision	single-precision	1.22	43%

Table 5.12: Execution time improvment by changing a data precision

6 Hardware Approximation Techniques

Approximate computing methods are used as an alternative to exact computing in error-tolerant applications in order to reduce the power consumption and the occupied on-chip area. Application examples such as, data mining and deep neural networks can tolerate errors and hence approximate techniques are used for improving their performance and energy-efficiency. There are numerous proposed approximate computing methods that are applied on hardware platforms in the effort of accommodating the ultra low-power and small area demands of critical embedded applications.

- **Precision Scaling** is a common technique that aims to reduce the precision of the operands. This approximation methodology is performed by cutting the least significant bits of the inputs or of the intermediate results. The purpose of precision scaling is to minimize the hardware resources that are used for the implementation of multipliers and adders on the hardware platform and therefore decrease both the chip's power consumption and the occupied area.

- **Approximate Memoization** is another widely used approximate technique that targets both GPUs and FPGAs. The goal of approximate memoization is to improve the application's performance by reducing the latency of the computationally intensive operations. This method is typically used for decreasing the number of clock cycles that are required for performing operations that are either difficult to implement on hardware, such as logarithmic or require many clock cycles.

- **Loop Perforation** is a method that induces error on the application's operation but may also significantly improve its performance. Its operating principle is to to skip the execution of loop iterations that cause an important latency overhead.

Besides the described approximation techniques that can be easily integrated on hardware designs trading-off quality for high-performance and low-power, there are application specific approximations that are based on the algorithmic characteristics of the tested application. Applying application specific approximation is not always possible but may lead to energy-efficient designs with improved performance without compromising the algorithm's quality.

6.1 Kalman Filter Acceleration using Approximation Techniques on FPGAs

Batch Processing Approximation : As it was previously mentioned, the Kalman filter is a sequential algorithm due to the inherent dependency of each estimated value from its previous one. In general, to accelerate a sequential application the original code has to be restructured in a way that parts of the execution flow can be parallelized. In the Kalman filter's case, batching is performed to the original signal, enabling the simultaneous filtering of each sub-signal. Although this strategy breaks the dependency, it is evident that the original algorithm's output is approximated.

To identify the way this approximation method affects the Kalman filter's output, we conduct an experiment where we execute the modified algorithm for different batch sizes and measure the Mean Absolute Error Percentage (MAEP). The batch size is the most important parameter in this approach as the more batches we use, the higher parallelization we can achieve. To highlight the batch size effect, the largest available input signal, which is composed of around 100 assets (6 MB), is used. Figure 6.1 depicts the results of our experiment.

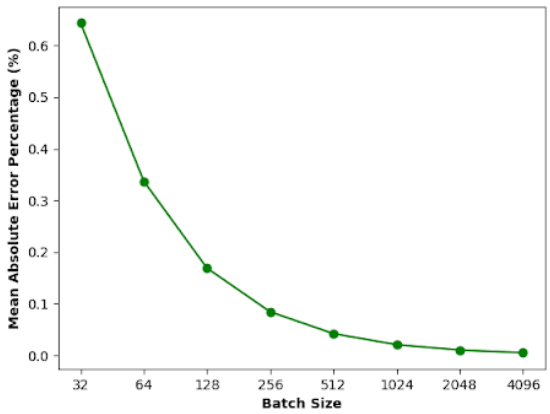


Figure 6.1: Batched Kalman filter's MAEP for different batch sizes

Figure 6.1 shows that as the batch size increases, and hence less batches are used, the algorithm's MAEP decreases, showcasing that the output becomes similar to the accurate. For a batch size equal to 32 i.e., 65536 batches, the batched Kalman filter's MAEP is equal to 0.64% while for a batch size equal to 4096 i.e., 512 batches, the MAEP is almost equal to 0%. The error resilient nature of the Kalman filter allows us to significantly increase the algorithm's parallelism without losing the output signal's quality.

Precision Scaling : Although batch processing approximation enables the increase of parallelism in the Kalman filter, the mapping to a Field Programmable Gate Array (FPGA) is not trivial, mainly due to the amount of resources. The usage of floating point arith-

metic leads to designs that are not able to fit in a device, limiting the parallelization we can achieve. Vitis High Level Synthesis (HLS) provides integer [1] and fixed-point[2] arbitrary precision data types in order to overcome this issue. By using arbitrary precision data types we can define the exact number of bits we want to use for the data representation, creating efficient designs in terms of resources, power and performance.

Arbitrary precision data types also enable a simple implementation of the precision scaling approximation technique. As it is already mentioned, we can decrease the number of bits used for the decimal part, to trade-off accuracy for resources utilization, power consumption and performance. To showcase the impact of precision scaling, we decreased the number of decimal bits used for the kernel's data representation and performed simulations to calculate the MAEP, the latency and the consumed resources. The resources that are mentioned throughout this section are referred to the reprogrammable memory and compute blocks that are available on the FPGA's programmable logic region.

	MAPE	Latency	BRAM	DSP	FF	LUT
FP32	-	52.8 msec	11 %	~0	~0	1 %
16	-	89.5 msec	11 %	~0	1 %	5 %
8	0.196 %	61.5 msec	11 %	~0	1 %	4 %
4	0.371 %	49.3 msec	11 %	~0	1 %	4 %
2	**1.088 %**	**42.3 msec**	**7 %**	**~0**	**1 %**	**3 %**
1	51.36 %	37.1 msec	7 %	~0	1 %	3 %

Figure 6.2: Mean average percentage error, latency and resources utilization

Table 6.2 [3] [4] [5], shows the aforementioned metrics when floating point as well as arbitrary precision data types with different decimal bits are used. It is observed that for arbitrary precision data types with 16 decimal bits the kernel's output is identical to the accurate. As the decimal bits decrease, the MAPE increases reaching 51.38% for 1 bit. Nevertheless, for 8, 4 and 2 bits the MAPE is below 1.1% showcasing that using precision scaling we can create efficient designs without losing accuracy. In addition, decreasing the decimal bits also leads to lower latency and less consumed resources. For our final Kalman filter design we used 2 decimal bits for the data representation as it leads to a MAPE of 1.088%. It should be noted that by simply decreasing the decimal bits we are able to achieve a x1.25 speedup.

[1] Arbitrary precision integer data types (https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Overview-of-Arbitrary-Precision-Integer-Data-Types)
[2] Arbitrary precision fixed point data types (https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Overview-of-Arbitrary-Precision-Fixed-Point-Data-Types)
[3] DSP : Digital Signal Processing
[4] LUT : Look-Up Table
[5] FF: Flip-Flop

The parallelization opportunities that arise using approximation, allow an easy mapping to a Field Programmable Gate Array (FPGA). Our approach was to create as many as possible identical compute units (CU) on the device, split the original signal into equal parts and assign them to different CUs. Each CU performs batching to the sub-signal it is assigned. Leveraging the architectural characteristics of FPGAs, we are able to perform Kalman filtering to each batch in parallel. In the final Kalman filter design, 4 compute units are created with each of them operating in 32 batches simultaneously. More information concerning the proposed parallelization strategy can be found in section 5.4 of the deliverable D4.1.

6.1.1 Evaluation

To evaluate the approximate version of the Kalman filter, we executed the non-approximate as well as the batched version of the filter on the Alveo U50 FPGA [6] device that is available in the SERRANO platform. This experiment aims to highlight the impact of our approximation strategy in terms of performance, power, and consumed resources.

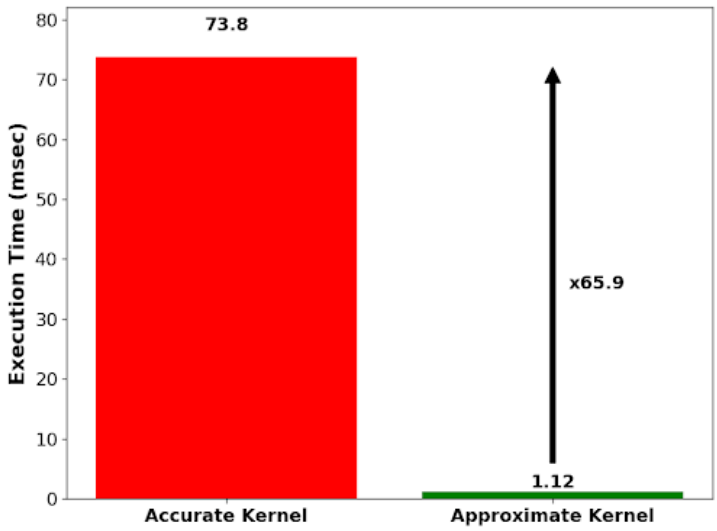


Figure 6.3: Accurate vs approximate kernel execution time

Figure 6.3, shows the kernel execution time for the accurate and approximate version of the Kalman filter kernel. Through approximation we are able to achieve a x65.9 speedup compared to the accurate kernel version. It should also be noted that without approximation we would not be able to accelerate this application as the CPU version of the Kalman filter requires only 15.6 msec to execute in the reference x86 CPU baseline, and in the table 5.4 we roughly achieved this number not only for 6MG of data but for different data sizes.

[6] Alveo U50 acceleration card https://www.xilinx.com/support/documentation/data$_sheets/ds965-u50.pdf$

	BRAM	DSP	FF	LUT	Power
Accurate	*77.3 %*	*0.12 %*	*~ 0*	*0.77 %*	*5.6 W*
Approximate	*49.7 %*	*4.32 %*	*~ 0*	*20.5 %*	*7.4 W*

Figure 6.4: Accurate vs approximate kernel resources utilization and power consumption

Table 6.4 presents the resources utilization as well as the power consumption for the accurate and the approximate versions of the kernel. The consumed resources of the approximate kernel concern the overall design, meaning that each CU consumes 12.4% BRAMs, 1.1% DSPs, 0 FFs and 5.1% LUTs. It is evident that the approximation leads to a significant decrease in the resources utilization, allowing us to further leverage the device capabilities. Finally, we observe an x1.32 increase in the power consumption. Nevertheless, considering the execution time of the two designs, we understand that the overall energy consumption of the approximate design is lower than the energy consumption of the accurate implementation.

In the next version of deliverable D4.2, AUTH will apply the aforementioned approximation techniques to other kernels provided by the SERRANO's Use Case (UC) providers. We will also investigate the benefits of other approximation techniques (e.g., loop perforation, approximate memoization).

7 SERRANO Interface for HPC Services

As mentioned before in this document and the D5.2 and D4.1, the HPC services are being developed on the Hawk supercomputer and additionally tested on the SERRANO testbed, namely the Excess cluster [3]. We described in this document the initial components of the HPC services and now we are interested to explain how that HPC services will be executed from the SERRANO orchestrator on an HPC system, which is controlled by a PBS manager, such as PBS-Pro or SLURM (see D2.1 [1] for details).

The SERRANO HPC Interface is being developed, which connects the orchestrator and the HPC system via Secure Shell Protocol (SSH). Figure 7.1 shows the schema of it. As it is described in D4.1, the SERRANO HPC interface doesn't require any extensions in the administration of the HPC system and its operation mode. This allows the use of the interface on most of the HPC systems.

The SERRANO HPC Gateway is the main component of the interface and is being implemented in the python language, as also the main part of the orchestrator. The database component servers log their activities.

In the following PBS, scripts are described, which can be put directly or via SSH protocol on the login node into a PBS queue for its further execution on the compute node. These scripts are created by the gateway according to the requirements of the users and their tasks. Conditions for this are that an HPC service exists for this task and the orchestrator decided that the HPC system solves this task most efficiently among all available platforms.

As telemetry data the consumed energy of the already executed jobs and the current global situation about the filling of the HPC system will be transmitted to the orchestrator[1].

Since in section 3 we define f as a use case task that is predefined by the user and is implemented and installed on the HPC system, at the first step the interface requires some information from the user regarding the input data (Listing 7.1). The user should insert the information about the required functionality, input data size, input data type (precision). Optionally, the desired performance or energy consumption can be specified. In this document, we will not consider the energy issues in detail, since this is just the subject of our work in Task 5.2 (see D5.2 for details [3]).

[1] If the energy mode is selected, it may be more efficient to start the job on a not fully utilized HPC system than on other hardware, even if this hardware, such as FPGA, consumed less power during the execution. The reason for this is the high static power that an HPC system has. Experience has shown that the HPC systems at Universities are not always fully utilized, especially during the summer holidays.

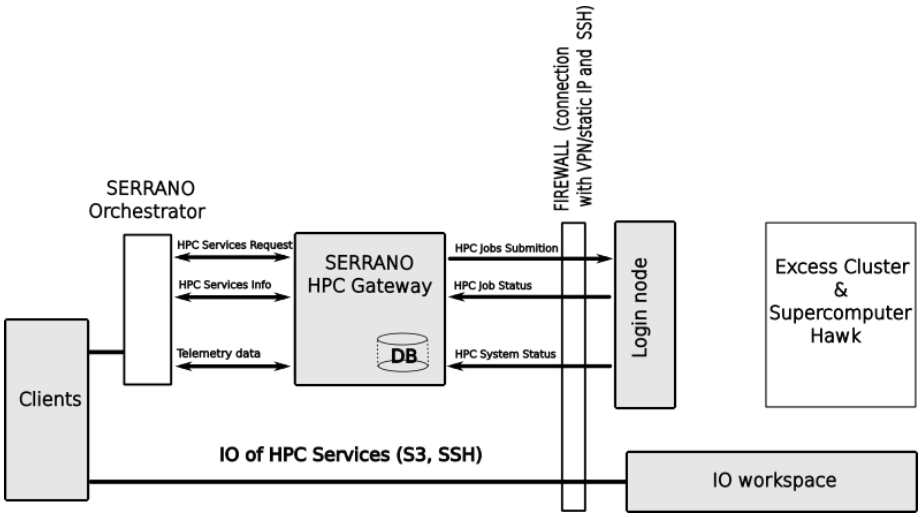


Figure 7.1: Workflow of HPC service

Therefore based on benchmarking of the HPC service that has been performed, and approximation of performance for various precisions described in section 5, the orchestrator will get a clue about the maximum possible performance, energy consumption, and the appropriate parallelization parameters for the specified input data.

Listing 7.1: bash script for parallel configuration

```
#!/bin/bash
data-Size = 0.7GB
data-Precision = "float"
demanded-Performance = 6e+9
```

For instance, in (Listing 7.1), the user provides the information about the input data (signal), 700 megabytes of data which in "float" precision, demanded-Performance is $6e+9$. After pre-processing that information and executing the related bash script on the login node the orchestrator will receive information, regarding estimated performance, the number of required compute nodes, and the number of MPI-rank to achieve maximum performance for the specified input data. The approximation formula for the performance already has been described in section **??**. Based on this formula the interface can also specify the configuration parameters for further similar tasks without executing the script on the login node. The interface will be also expanded with the energy consumption approximation to enable maximum performance as well as maximum energy-efficient modes.

bash The Maximum Performance is 6.99307e+09 Flops, achieved by 32 MPI rank. This performance 6e+09 was achieved by 15 MPI rank. Estimate formula Y = -1.78864e+08 + 2.75304e+09.(MPI-Rank)$^{(}0.296)$.

Since it is described in section 3 that we have WS workspace, and the user should place the input data (signal) in the appropriated format, for example, binary with single precision, in WS. Therefore it is required to be directed to the WS workspace if the workspace

doesn't exist it will be generated with the command 7.1, where the value for the `serrano-ws-id` will new generated or reused. The workspace can be reused, if the input data is already on the WS available, for example from the previous run of one of the HPC services. The WS is automatically deleted after 31 days.

```
ws-allocate serrano-ws-id 31
```
(7.1)

In the next step, the Input-Data/Output-Data folders should be created where the input data (signal) will be placed in the Input-Data folder, and respectively the output data (filtered data) will be generated in the Output-Data folder.

After the orchestrator has figured out the optimal number of parallelization parameters (MPI-rank) and placed the input data in the Input-Data folder in workspace WS, then the HPC service can be submitted into the PBS queue for further execution on the free computed nodes. For this purpose, the interface has to define a PBS job script (Listing 7.2) where the interface has to define the number of the needed compute node.

Listing 7.2: bash script for execution of HPC Service

```
#!/bin/bash
#PBS -N job_name
#PBS -l select=1:node_type=rome:mpiprocs=128
#PBS -l walltime=00:20:00

BIN=/opt/serrano/hpc-services/kalman/bin/app_kalman
Input = "/workspace/sensor001"
Output = "/workspace/sensor001"
R    = 10
ranks = 4
numas = 8
cores-numa = 16
. . .
mpirun -n $ranks $BIN $R $numas $cores-numa $Input . . .
. . .
```

In this bash script, the user will provide additional parameters for the execution of the HPC service: 1) the name of the input data. 2) the R parameter for Kalman filter that removes the noises from input data explained in section 4. 3) the number of MPI-rank that has been obtained recently. 4 and 5) the mapping parameters regarding the hardware architecture (see section 5.1). By executing this bash script the HPC service will be executed on the specified number of the compute nodes and compute in parallel the Kalman filter for the input data. The result will be generated in the WS workspace, which is specified with the bash variable Output.

The interface has also to monitor the job submission, its execution and triggers the event by the orchestrator, when the job, namely, the HPC service, is executed. The further de-

tails will be described in the following deliverables after the first experiments on the Excess cluster and the Hawk supercomputer.

8 Conclusion and Outlook

In this deliverable, we have described the first implementation of the HPC service that we have developed at HLRS to achieve the objectives of task 2 work package 4 of the SERRANO project. Signal processing for the large volume of data was the main purpose of this HPC service. We have presented the parallelization of the Kalman, in HPC and FPGA hardwares, and execution time and power consumption have been investigated in both cases. Usually, HPC systems handle big data sizes and support high precision compare to FPGA devices.

Mainly The Performance of the HPC service has been considered for multiple data sizes and data precision in this deliverable, and we have sought the parameters to obtain maximum performance.

For the next steps we could explore these issues:

Since we have discovered the MPI-IO-Write method could worsen the total Performace of the HPC service, we are interested to explore a new approach that prevents us to write out all the output data. This requires further approximation techniques and must be discussed with the use case provider in detail so that the computed result remains suitable for the purposes of the use cases.

Integrating the HPC service where more Kernel could be accelerated, and providing the possibility that FFT filter and Kalman filter could work simultaneously.

Since we have introduced a prototype of Kalman filter in section 4, it is possible to integrate a framework that can process the stream of input data with different data types and parameters in parallel in the correlation of VVUQ, to verify the best parameters configuration.

In our recent experiment, we have performed our experiments in one compute node, testing the performance of the HPC service by using more compute nodes and researching a sophisticated method to approximate the performance and energy consumption of the HPC service will be explored in the future.

References

[1] Serrano Consortium. *Deliverable D2.1 SERRANO Architecture*. 2021.

[2] Serrano Consortium. *Deliverable D4.1 HW/SW IPs for workload acceleration in disaggregated DCs*. 2022.

[3] Serrano Consortium. *Deliverable D5.2 Algorithmic Framework, Performance and Power Models*. 2022.

[4] *FFTW library*. URL: http://www.fftw.org.

[5] MPI Forum. *Official website: MPI Forum*. This website contains information about the activities of the MPI Forum, which is the standardization forum for the Message Passing Interface (MPI). Feb. 2022. URL: https://www.mpi-forum.org.

[6] Felix Govaers. *Introduction and Implementations of the Kalman Filter*. Rijeka: IntechOpen, 2019. URL: https://doi.org/10.5772/intechopen.75731.

[7] Torsten Hoefler et al. *Using Advanced MPI: Modern Features of the Message-Passing Interface*. The MIT Press, 2014. URL: https://dl.acm.org/doi/10.5555/2717108.

[8] *HPE Hawk*. Accessed: 2021-12. URL: https://kb.hlrs.de/platforms/index.php/HPE_Hawk.

[9] Jim Jeffers, James Reinders, and Avinash Sodani. *Intel Xeon Phi Processor High Performance Programming*. Morgan Kaufman, 2016.

[10] Yitzhak Katznelson. *An Introduction to Harmonic Analysis*. Cambridge University Press, 2004. URL: https://doi.org/10.1017/CBO9781139165372.

[11] OpenMP. *The OpenMP API specification for parallel programming*. OpenMP 5.2 Released with Improvements and Refinements. Feb. 2022. URL: https://www.openmp.org.

[12] *Wikipedia: Covariance*. Accessed: 2022-02-12. URL: https://en.wikipedia.org/wiki/Covariance.

[13] *Wikipedia: Discrete Fourier Transform*. Accessed: 2022-02-25. URL: http://en.wikipedia.org/wiki/Discrete_Fourier_transform.

[14] *Wikipedia: Fast Fourier transform*. Accessed: 2022-02-25. URL: https://en.wikipedia.org/wiki/Fast_Fourier_transform.

[15] *Wikipedia: Fourier series*. Accessed: 2022-02-24. URL: https://en.wikipedia.org/wiki/Fourier_series.

[16] *Wikipedia: Kalman filter*. Accessed: 2022-02-22. URL: https://en.wikipedia.org/wiki/Kalman_filter.