Machine Learning-Enhanced Interactive Multimedia Applications for a 64 Odroid Computing Cluster

Master Project Report

Reto Krummenacher
High Performance Computing Group
Department of Mathematics and Computer Science
University of Basel

November 27, 2023

Abstract

This report describes the development of interactive applications customized for public exhibitions of the μ-Cluster, a parallel computing system. The main goal is to demonstrate the potential of parallel computing and generate interest through interactive applications, including a game of Snake against an AI opponent, an application that matches user photos with celebrity images, and parallel computation of pi digits. The report discusses the integration of efficient communication between nodes, the implementation of user-friendly input methods, training experiments of the AI agent, and the workload distribution technique used for pi computation. Future work will include thorough testing on the μ-Cluster and investigating any limitations of the celebrity implementation.

Contents

1 Introduction

The μ-Cluster of the High Performance Computing Group at the University of Basel comprises 66 single board computers, 64 of it connected to LCD-Screens, sixteen of which are combined on each side of the cluster. Its primary objective is to showcase parallel computing for public exhibitions. At present, the cluster exhibits primarily videos. Applications that require interaction are not available, despite all the screens having touch screen capability.

1.1 Motivation

We aim to enhance the attractiveness of the μ-Cluster by developing new visually compelling applications with suitable levels of user engagement. Given the current hype around artificial intelligence, we intend to create programs that use machine learning. The μ-Cluster has four sides, which means the objective is to create 4 new applications. There were numerous ideas, including some natural language processing, where the cluster finishes sentences for visitors. In the end, we have decided in favor of:

- 'Play *Snake* against an AI Opponent': A user can play the game *Snake* against a machine-controlled AI instance trained with reinforcement learning. The game is controlled using a gamepad.
- 'Which Celebrity you look like?': The user may initiate the application by tapping on one of the touchscreens. The connected camera takes a photo which is compared to a database of celebrity images to determine which famous person a visitor resembles the most.
- 'How much Pi you like?': Demonstrating parallel computing through the calculation of a number of pi digits selected by the user.
- 'Displaying Information': Create new content to be displayed.

1.2 Contribution

This report details the implementation of 4 Python-based applications. It describes the communication methods employed among the computing nodes of the μ-Cluster, the integration of a gamepad for user pseudonym provision, the results of the AI agent training evaluation, and the parallel workload distribution algorithm for computing pi digits.

A significant limitation is that none of the applications could be tested on the μ-Cluster. This is because we were unable to create a container image that would work with the ARM architecture of the cluster nodes, nor were we successful in finding a way to update their operating systems. As a result, we performed all functionality testing on our own single machine to the extent possible.

1.3 Outline

The remainder of this report is structured as follows. The necessary background information can be found in section 2. The implementation of 'Play *Snake* against an AI Opponent' is explained in detail in section 3 followed by the work for 'Which Celebrity you look like?' presented in section 4. Some aspects of 'How much Pi you like?' are detailed out in section 5 before explaining 'Displaying Information' in section 6. Finally, section 7 concludes the paper.

2 Background

In this section, we provide background information on hardware and communication primitives, as well as a brief introduction to deep Q-learning. Readers who are already familiar with this material may wish to skip to Section 3.

2.1 MPI

The Message Passing Interface (MPI[1]) is one of the standard application program interfaces allowing information exchange between CPUs in distributed systems. The standard defines routines offered by libraries to be used in

[1] `https://www.mpi-forum.org/docs/` (accessed 2023-10-30)

traditional HPC languages such as C/C++ and FORTRAN.

When running a program with MPI, all the processes form a group, which is called a communicator. All the members, usually called ranks, are connected to each other. There are two major communication primitives: Point-to-Point and Collective. The former involves only two processes, while the latter involves all of them.

MPI for Python[2] offers bindings to access the MPI routines directly from within a Python script. The package development began more than 14 years ago by Lisandro Dalcin [1].

2.2 The Odroid

An Odroid is a compact single-board computer (SBC) with an integrated processor, memory, and storage, running a full operating system. Another example is the Raspberry Pi. In contrast to microcontroller boards like the Arduino, the SCBs offer more connectivity, featuring HDMI and USB ports. The μ-Cluster was built with the now discontinued Odroid C2, which comes with a 1.5 GHz quad-core ARM CPU, 2 GB RAM and even a small GPU.[3]

2.3 The μ-Cluster

The μ-Cluster[4] consists of 64 Odroid-C2 SBCs dedicated to computing, an Odroid for login and one for storage. In total, there are 66 Odroids interconnected in a star topology with 1 Gbit/s Ethernet switches. They form a network, where each Odroid has an IP-address and a corresponding speaking name, such as 'node1'.

The cluster comprises four layers, each of which has 16 Odroids connected to an LCD screen, providing touchscreen capabilities. A total of 16 of these screens make up what we call a panel. Notably, the Odroids within a given layer do not link to the same panel but rather to the same screen row within each panel. It remains unclear why this is so, but it may have been to facilitate simpler wiring, since only Ethernet cables are needed to connect the layers.

As a result, the arrangement of which Odroid controls which screen is unclear, as depicted in Figure 1. Each panel will display one application implemented in this project. Therefore, it is essential that each application run exactly on these specific nodes. The rank numbers of workers within an MPI communicator are assigned based on the order of appearance in the starting routine. Our implementation is based on the consecutive numbering of nodes, illustrated in Figure 1 with teal numbers. This numbering is crucial for features dependent on connected devices, such as the gamepad, which must be connected to node 47, identified as rank 9 within the application (Figure 1b).

2.4 Deep Q-Learning

Deep Q-learning is a type of active reinforcement learning (RL). Details about RL can be found in any standard machine learning textbook, for instance [2]. The main concept of Q-learning is that an agent determines the best actions to take in an environment by receiving rewards for its actions. The Q in Q-learning stands for quality value and represents the quality of an action. Every state of the environment is assigned a Q-value for each possible action in that state. In simple Q-learning, these Q-values are initially set to 0 and stored in a table.

The update rule is defined as follows:

$$Q(s,a) \leftarrow \overbrace{Q(s,a)}^{\text{current Q value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot [\overbrace{R(s,a,s')}^{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \overbrace{\max_{a'} Q(s',a')}^{\text{maximum expected future reward}} - Q(s,a)]. \tag{1}$$

Within the brackets is the temporal difference term, representing the disparity between the current Q-value and the highest future reward. This maximum expected reward is determined by the action a' taken in the current state s that leads to the maximum Q-value in the successor state s'. Additionally, the reward from taking action a in state s are added. During training, the hyperparameters of the learning rate and the discount factor are tuned. The first defines the pace of update of the Q-Values, while the second determines the relative importance of immediate rewards versus future rewards.

It is often infeasible to store the Q-Values in a table. One alternative solution is to approximate the Q-value with a weighted linear combination of features:

[2] `https://mpi4py.readthedocs.io/en/stable/` (accessed 2023-11-21)
[3] `https://www.hardkernel.com/shop/odroid-c2/` (accessed 2023-11-21)
[4] `https://hpc.dmi.unibas.ch/en/research/micro-cluster/` (accessed 2023-11-20)

node55, 0	node56, 1	node51, 2	node52, 3
node27, 4	node31, 5	node28, 6	node32, 7
node42, 8	node41, 9	node46, 10	node45, 11
node6, 12	node2, 13	node9, 14	node1, 15

(a) Panel to display 'How much Pi you like?'.

node54, 0	node50, 1	node53, 2	node49, 3
node23, 4	node24, 5	node19, 6	node20, 7
node43, 8	node47, 9	node44, 10	node48, 11
node10, 12	node5, 13	node14, 14	node13, 15

(b) Panel to display 'Play *Snake* against an AI Opponent'.

node58, 0	node57, 1	node62, 2	node61, 3
node22, 4	node18, 5	node21, 6	node17, 7
node39, 8	node35, 9	node40, 10	node36, 11
node11, 12	node15, 13	node12, 14	node16, 15

(c) Panel to display 'Which Celebrity you look like?'.

node29, 0	node8, 1	node25, 2	node63, 3
node64, 4	node4, 5	node3, 6	node34, 7
node37, 8	node38, 9	node26, 10	node30, 11
node7, 12	node33, 13	node59, 14	node60, 15

(d) Panel for 'Displaying Information'.

Figure 1: Arrangement of the 4 panels and which Odroid is controlling which LCD screen. The designated rank number is shown in teal. To operate the system, the gamepad should be connected to node 47 with rank number 9, marked by a gray box.

$$\hat{Q}(s,a) = \omega_0 + \omega_1 \cdot f_1(s,a) + \omega_2 \cdot f_2(s,a). \tag{2}$$

A basic illustration featuring 2 characteristics and 3 parameters. This equation permits the calculation of a gradient that indicates the direction of the highest rate of change of the equation at that point. Throughout the training stage, the parameters are upgraded based on the error between the current value and the target value:

$$\omega_i \leftarrow \omega_i + \alpha \cdot \underbrace{[\overbrace{R(s,a,s') + \gamma \cdot \max_{a'} \hat{Q}_\omega(s',a')}^{\text{target value}} - \overbrace{\hat{Q}_\omega(s,a)}^{\text{current value}}]}_{\text{error}} \underbrace{\frac{\partial \hat{Q}_\omega(s,a)}{\partial \omega_i}}_{\text{gradient}}. \tag{3}$$

If the concept is extended to approximate $Q(s,a)$ using neural networks, it is referred to as deep Q-learning. Such a model is trained by minimizing the loss, generally defined as the mean squared error between the target value and the current value.

In all active RL processes, it is necessary for the agent to first explore the environment by undertaking random moves. As the duration of training increases, the agent's actions ought to depend more on following the optimal path, which translates to selecting the action with the maximum Q-value. This dynamic is referred to as the tradeoff between exploration and exploitation. A typical approach to address this tradeoff is through the implementation of the decaying ϵ-greedy strategy, where ϵ denotes the probability of a random move. The precise values of these hyperparameters require additional tuning.

3 Play *Snake* against an AI Opponent

The aim of this application is to allow users to play the game 'Snake' on the μ-Cluster via a gamepad. To enhance the attraction for potential visitors during an exhibition, users are invited to play against an AI opponent referred to as the bot. The game has 3 finishing rules:

1. When the maximum play time of 2 minutes has elapsed and neither the user nor the bot are game over, the player with the highest number of consumed apples is the winner.
2. If the user is game over, the bot emerges victorious.
3. If the bot is game over, users can still continue to achieve victory in accordance with rule 1.

3.1 Process Flow and Code Structure

As a starting point, the process flow is shown in Figure 2. Once started, the application runs in an infinite loop until the μ-Cluster is shut down. The steps involved are:

- Welcome screen: Displays a welcome message, logos of Python libraries used in the implementation, and the list of current record holders.
- When the Start button on the gamepad is pressed, the virtual keyboard appears. The user enters their name or pseudonym under which they wish to play.
- The AI opponent is initialized, and the game is played until the end of the game according to the rules above.
- When the game is finished, the high score list is updated. The user can choose to play again or to return to the welcome screen.

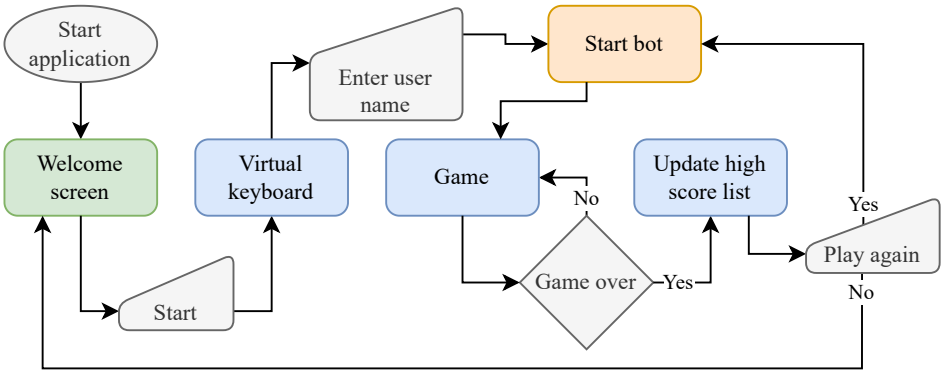

Figure 2: Flow diagram for the 'Play *Snake* against an AI Opponent' implementation. The colors correspond to distinct components of the project, as shown in Figure 7 in Appendix A. Green depicts the entry point, blue relates to the main components of the application, and orange signifies code dedicated to the AI agent. Trapezoids represent necessary user input.

To give an idea how the application looks like when running, screen captures are provided in Appendix B. This report is not intended to cover the details of the code. To get an overview of how the project is structured, a simplified relationship diagram can be found in Figure 7 in Appendix A. Nevertheless, we would like to present some special features of our implementation.

3.2 Efficient Message sending within the MPI Communicator

Since the Ethernet connection between the Odroids could be a performance bottleneck, we try to reduce the traffic from the beginning. The first step is to minimize the number of messages sent.

As mentioned in Section 2, all processors form a communicator to allow the transmission of messages between nodes. In our case, there are 16 ranks involved. The MPI standard offers point-to-point or collective communication. However, most of the messages need to be sent from one rank to a few others only. The best example is the transfer of the game state from the game node to the 4 display nodes. In short, the usual collective primitives are not very efficient due to an excessive amount of irrelevant messages.

To tackle this issue, we created our own kind of transmission, which involves a series of point-to-point transfers based on a list of recipients. The crucial part: The method must be called on the nodes taking part in the communication.

Another important aspect is the size of the message. The position of the snake is stored as a list of tuples (x, y), each representing the top-left pixel of a snake block. Since a snake block has a size of 20×20 pixels and the game canvas has dimension 1 800×1 000, $x = 0, 20, 40 \ldots, 1780$ and $y = 0, 20, 40 \ldots, 980$ respectively. To reduce the message size, the pixels are converted to positions, with $x = 0, 1, 2 \ldots, 89$ and $y = 0, 1, 2 \ldots, 49$. Values that can be stored in a single byte. Furthermore, the data is sent via *NumPy*[5] arrays, the fastest way according to the *mpi4py* documentation.[6]

3.3 Virtual Keyboard

A user can only enter text by either using the touchscreens or the gamepad. Both methods rely on a virtual keyboard displayed on a screen. Since the cluster's screens are rather delicate, we decided to implement a gamepad controlled virtual keyboard.

The starting point was the look and functionality of a standard touchscreen keyboard found in many messaging applications on a modern smartphone. In addition to keys for lowercase letters, there is the possibility to switch to uppercase letters or to special characters and numbers. The core of our implementation are the data structures that contain the characters to display. In Python, we use lists of lists. An example can be seen in Code 1. The activated character is tracked with the row and column index. For example, the letter 'F' is in row 1 column 3. The user changes these indices and thus the activated key with the directional pad of the game controller.

Code 1: Example of the keyboard layout as list of lists in Python.

```
__return_symbol = '\u23CE'
__shift_symbol = '\u21E7'
__backspace_symbol = '\u2190'
__layout_upper = [
    ['Q', 'W', 'E', 'R', 'T', 'Y', 'U', 'I', 'O', 'P'],
    ['A', 'S', 'D', 'F', 'G', 'H', 'J', 'K', 'L'],
    [__shift_symbol, 'Z', 'X', 'C', 'V', 'B', 'N', 'M', __backspace_symbol],
    ['123', ' ', __return_symbol]
]
```

Selecting the keys '123' or '__shift_symbol' will replace the current layout with the new one, and the new keyboard is displayed on the screen. The change from the lowercase letter layout to the number and special character layout when pressing the '123' key is shown in Figure 3.

3.4 The AI Opponent

Our AI agent implementation is based on the example by Patrick Loeber[7]. We used reinforcement learning, specifically deep Q-learning, to teach the agent how to play *Snake*. Various aspects of Q-learning have been discussed in Section 2. Here we state the important elements of our implementation.
The neural network consists of 3 layers:

1. Input layer: The input layer consists either of 11 or 15 units.
    - Input layer part 1: The game state is represented as a list of Boolean values. The first 3 elements indicate whether the snake is in danger of colliding with the walls or its tail while moving left, right, or straight ahead. The current direction of the snake (left, right, up, down) is also included. Another part is the position of the snake's head relative to the food (left, right, up, down).
    - Input layer part 2: During our initial training sessions, we observed the AI agent getting stuck by its tail. To prevent such situations, we added 4 new game state variables that measure the location of the snake's

[5] `https://numpy.org/` (accessed 2023-11-21)
[6] `https://mpi4py.readthedocs.io/en/stable/tutorial.html#point-to-point-communication` (accessed 2023-11-21)
[7] `https://github.com/patrickloeber/snake-ai-pytorch/tree/main` (accessed 2023-11-21)

(a) The virtual keybord with lowercase letters.

(b) The virtual keyboard with numbers and special characters.

Figure 3: Change of the keyboard layout from lowercase letters to numbers and special characters upon selecting the '123' key. Pictures taken while executing the 'Play *Snake* against an AI Opponent' application on our own individual machine, allowing up to 6 MPI ranks.

head relative to its body, represented by what we call the center of gravity. This metric is calculated by averaging the coordinates of the tail elements of the snake.

2. Hidden layer: The size of the hidden layer was set to 256, following Patrick Loeber's implementation.
3. Output layer: The output of the neural network is 3 Q-values.

The purpose of the neural network is to predict 3 Q-values from a given game state and to determine the action to take in that state. An action is encoded as a list with 3 Booleans: [turn left, go straight, turn right], where the index with the maximum Q-value is labeled 1 and the remaining indices are labeled 0. The model parameters are obtained by minimizing the mean squared error between the predicted Q-values and the target Q-values. At the beginning of each training, the agent needs to do some random moves. Our approach implements a decaying ϵ-greedy strategy to ensure sufficient exploration at the beginning of each training.

Training Experiments and Results

Training an agent with deep Q-learning requires numerous crucial choices. These decisions include the overall network architecture, as well as selecting appropriate values for key parameters such as the exploration-exploitation strategy and the learning rate. To gain insight into the effectiveness of particular choices, experiments were conducted on our own individual machine. The design of factorial experiments is shown in Table 1. Some points we like to emphasize:

- During each training session, the agent plays a continuous succession of games.
- For each game, the score, the number of eaten apples, is recorded.
- Each training run lasts at most 90 minutes. However, there are certain situations, where the agent is stuck. To account for such scenarios, we defined 2 abort rules. The first addresses the case, where the agent is unable to complete a game, whereas the second refers to periods where there is no improvement in the score.
- We used 2 different exploration strategies. Firstly, a greedy agent with a low starting ϵ and a small decay per game. Secondly, the curious agent, with a high starting ϵ but a larger decay. The reason for this decision: The agent plays a similar number of games until the ϵ is 0 no matter which strategy is applied.

The results of the experiments are presented in Table 2. The maximum score achieved was 93 for the configuration highlighted in green. Certain parameter combinations clearly perform worse in comparison to others, particularly a higher learning rate. In all cases, the larger value resulted in an abortion of the training run.

To evaluate the impact of extending the training duration, we used the best parameter configurations from the previous analysis and trained the agent for 7 hours. The evolution of the score over all 1 089 played games is displayed

Table 1: Design of factorial experiments to train the AI agent.

Factor	Value	Properties
Application	Deep Q-learning model	Input size= $\{11, 15\}$ Hidden size = 256 Output size = 3
Exploration strategy	Decaying ϵ-greedy	Curious agent: $\epsilon_{start} = 0.7$,$\epsilon_{decay} = 0.005$ Greedy agent: $\epsilon_{start} = 0.3$,$\epsilon_{decay} = 0.0025$
Optimization pace	Learning rate	$lr = \{0.001, 0.01\}$
Value of future reward	Discount factor	$\gamma = \{0.5, 0.9\}$
Metric	Performance of AI agent	Highest number of eaten apples in a game
Computing system	Acer Predator PH315-55	1 CPU Intel i7-12700H, 2.7 GHz; 14 cores 32 GB RAM, 24 MB cache
Execution control	Training time and abort rules	Maximal training time: 90 minutes Abort if game not finished after 5 minutes Abort if score was not improved after 20 minutes

in Figure 4. Even though the highest score of 104 was larger than before, no additional advancements in the agents' playing ability were observed. The average score is only slightly increasing, and the score per game depicted many spikes. Nonetheless, we employ this model for our artificial intelligence adversary. This way, there is a possibility for the human contender to win.

Table 2: The highest score of each training run with the corresponding parameter setting. The highest score was 93.

				Exploration strategy			
				Greedy		Curious	
				Input layer size		Input layer size	
				11	15	11	15
Learning rate	0.001	Discount factor	0.9	89	93	88	5
			0.1	83	19	7	5
	0.01	Discount factor	0.9	4	2	2	2
			0.1	4	2	2	2

4 Which Celebrity you look like?

The primary purpose of this application is for visitors to capture a photo of themselves using the connected camera, which is then compared to images in a database of celebrities to find the one that most resembles the visitor.

The process flow of our implementation is presented in Figure 5. Initially, the current camera input is displayed on 4 screens as a video stream. A user can initiate the matching process by tapping the 'start' screen. Moreover, a consent message is exhibited to respect user privacy by explaining the purpose of the captured image. The example of what this looks like can be found in Figure 14 in Appendix C.

After the user starts, a countdown appears to allow time for preparing to take the photo, as depicted in Figure 15. Subsequently, the taken picture gets saved, and then, face detection and matching with images of famous people are carried out. A message on the μ-Cluster screens indicates the ongoing process, as illustrated in Figure 16.

Finally, the taken photo, the image of the most similar celebrity and their name, the QR code for finding information about the celebrity database and a restart button are displayed. See Figure 17 for an example with certain components.

Details of the code are omitted. However, the simplified relationship diagram can be found in Figure 8 in Appendix

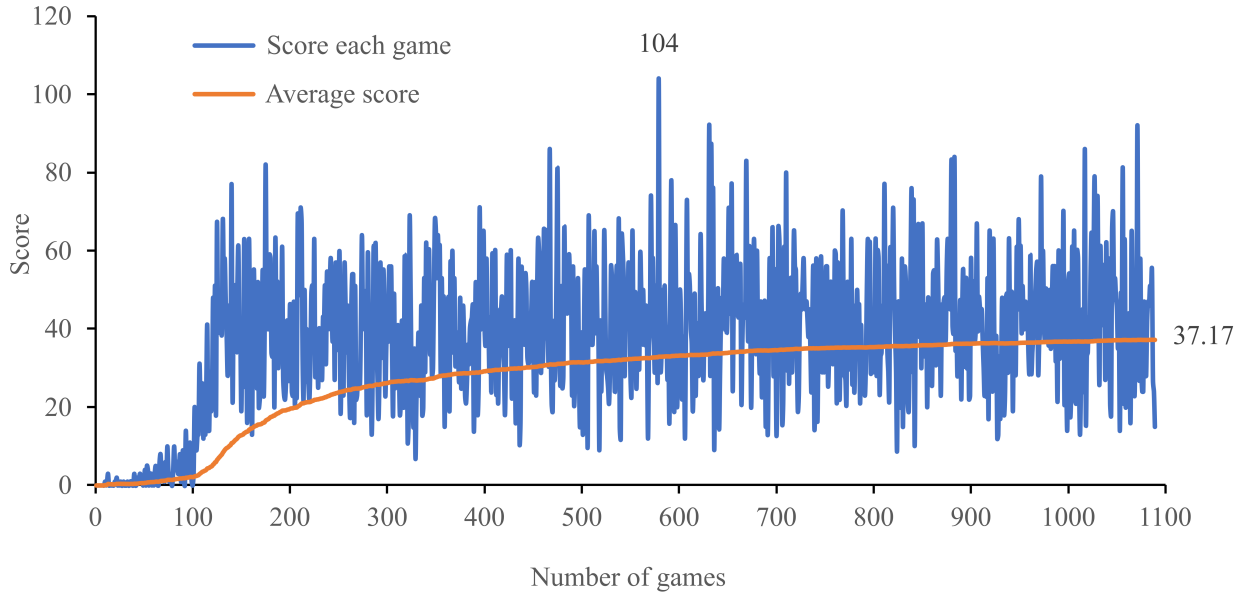

Figure 4: Evolution of the score during 7 hours of training and 1 089 played games. The neural network had an input layer size of 15. We used a greedy exploration strategy with $\epsilon_{start} = 0.3$ and $\epsilon_{decay} = 0.0025$. The learning rate was set to 0.001 and the discount factor was 0.9.

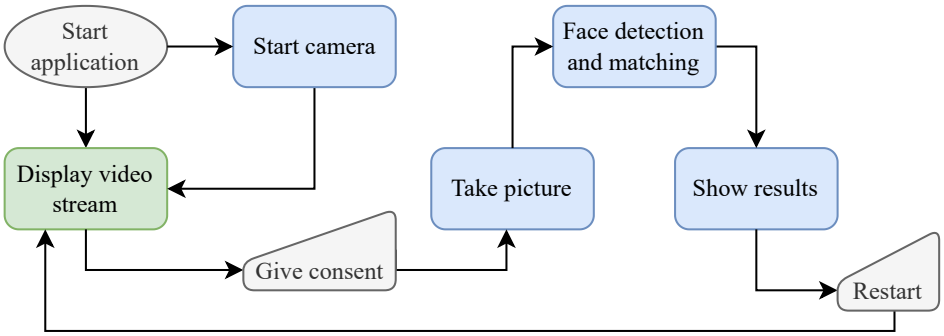

Figure 5: Flow diagram for the 'Which Celebrity you look like?' implementation. The colors correspond to distinct components of the project, as shown in Figure 8 in Appendix A. Green depicts the entry point and blue relates to the main components of the application. Trapezoids represent necessary user input.

A. Here are some of the key aspects of our implementation.

4.1 MPI Group Communication

Unlike the 'Play *Snake* against an AI Opponent' implementation, where we created our own communication methods, we used MPI groups[8] to allow communication between ranks. The purpose of MPI groups is to create additional group communicators with only a few ranks within an MPI world communicator. The MPI command to create them is *split* which is available in *mpi4py* as well.

Our contribution is to allow group communicators to be created based on world rank assignments to group identifiers stored in Python dictionaries. The implementation can be found in Code 2. Once the group communicator is in place, all MPI communication primitives can be used.

[8] `https://mpitutorial.com/tutorials/introduction-to-groups-and-communicators/` (accessed 2023-11-26)

Code 2: Python code to create MPI group communicators from a dictionary containing the group assigement of the world communicator ranks.

```
# Assignment of world ranks to groups.
groups = {1: [1,2,4],
          2: [0,3,5]}

# comm: The MPI world communicator.
for key, item in groups.items():
    if comm.rank in item:
        group_id = key

# New rank numbers are 0,...,size, with increasing order of comm.rank 1->0, 2->1, 4->2.
group_comm = comm.Split(group_id, comm.rank)
```

4.2 Celebrity Database: *CelebA*

The celebrity images are taken from *CelebA* [3]. The database is available online[9] and contains 202 599 images of no more than 10 177 celebrities. The images are stored with sequential numbers. Each picture is available in two versions. First, the original image and second, as an aligned and cropped image of just the face. There is a mapping from image names to a celebrity identification number. The names of the celebrities are available on request.

To facilitate the face matching during the demonstration, the face detection for the celebrity photos is done in advance. The resulting face information is stored in 16 files because we will perform face matching on 16 ranks of the cluster. It is not feasible to use the whole database for our purpose because it would take too long even if it is done in parallel. As a solution, we use only one image for each of the 10 177 celebrities. Still, each of the precomputed files has a size of about 70 MB.

Currently, the splitting process is suboptimal. It selects only the first image of each celebrity in the database, which may not be the most appropriate for the matching process. Furthermore, some of these images are unsuitable for display due to their sexually suggestive nature. At present, only the aligned and cropped images are used for display, which has the drawback of having few pixels. Thus, an improved splitting process needs to be implemented in the future.

4.3 Face Detection and Matching

For face detection, Python's *cv2* library from OpenCV[10] is used. Utilizing the Haar[11] cascade pretrained face detector returns the recognized face information.

Face matching is used to find the celebrity image that most closely resembles the user's face. We employ the structural similarity index measure (SSIM) provided by Python's *scikit image*[12] package. The SSIM produced falls within the range of [-1,1], with higher numbers indicating greater similarity. Face matching is performed across multiple ranks. After completing their individual computations, the highest SSIM score from each rank is gathered into a list to determine the overall best match.

Our current implementation yields results. However, the routine should be extensively tested in the future to evaluate its quality. We must match numerous photographs of different individuals against the celebrity database to determine if the algorithms function with any input. Additionally, we should conduct tests with pre-stored photos to establish if the same photo compares reliably to the same celebrity repeatedly.

[9] `https://mmlab.ie.cuhk.edu.hk/projects/CelebA.html` (accessed 2023-11-25)
[10] `https://opencv.org/` (accessed 2023-11-26)
[11] `https://docs.opencv.org/4.x/db/d28/tutorial_cascade_classifier.html` (accessed 2023-11-26)
[12] `https://scikit-image.org/docs/stable/auto_examples/transform/plot_ssim.html` (accessed 2023-11-26)

5 How much Pi you like?

The aim of this application is to show parallel computing in action. Computing the digits of pi (π) is one possible way. The number π is familiar to most people. Since the μ-Cluster is used during public exhibitions, we think it is more appropriate than other common demonstrations of parallel computing, such as visualizing the Mandelbrot set.

To offer an understanding of the steps involved, the process flow is shown in Figure 6. The idea is as follows: A user can choose how many digits of π they want to calculate. To select the desired number of digits, a user can tap on the screen with the corresponding number. To give an idea of what this looks like, a screen capture is shown in Figure 18 in Appendix D.

We then create groups of workers with 1, 2, 4 and 8 ranks. Each of these groups computes the chosen number of digits in a distributed manner. It is important to us that the user can see the progress of the individual groups. The participating ranks have distinct colors to indicate which group they belong to. In addition, each rank displays a message showing its identification number and its share of the workload. An example is shown in Figure 19.

The computational steps involved will be explained further below, but before now it is sufficient to mention that each step is indicated by a corresponding message. Examples are provided inFigure 20, Figure 21 as well as Figure 22 in Appendix D.

Finally, along with some summary messages, two graphs are displayed as a result. First, a pie chart showing how many digits have been computed compared to what is currently known and second, a bar chart showing the execution time of each worker group. The example can be found in Figure 23.

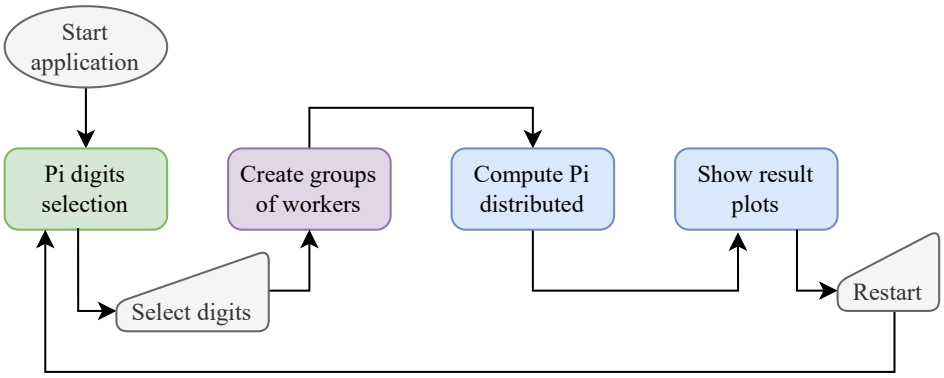

Figure 6: Flow diagram for the 'How much Pi you like?' implementation. The colors correspond to distinct components of the project, as shown in Figure 9 in Appendix A. Green depicts the entry point, blue relates to the main components of the application, and purple stands for commonly available functionality. Trapezoids represent necessary user input.

This report is not intended to cover the details of the code. To get an idea of how the application is structured, a simplified relationship diagram can be found in Figure 9 in Appendix A. Similar to the implementation of 'Which Celebrity you look like?', the communication between ranks is facilitated through the use of MPI groups. The calculation is the main contribution, which we will discuss next.

5.1 Pi Computation

In the past year, the known number of π digits has grown enormously. A project at the UAS Grison calculated a record 62.8 trillion in August 2021.[13] However, this was not valid for long, as Emma Haruka Iwao increased the number of known pi digits to 100 trillion by using Google infrastructure in 2022.[14]. A threefold increase compared to the 2019 record. All those record attempts used a software called γ-cruncher[15] which is based on a generalized hypergeometric series developed by the Chudnovsky brothers in 1988 [4].

[13] `https://www.fhgr.ch/en/themenschwerpunkte/applied-future-technologies/davis-centre/pi-challenge/` (accessed 2023-11-25)
[14] `https://cloud.google.com/blog/products/compute/calculating-100-trillion-digits-of-pi-on-google-cloud` (accessed 2023-11-25)
[15] `http://www.numberworld.org/y-cruncher/` (accessed 2023-11-25)

The Chudnovsky Series

The Chudnovsky series is defined as [5]:

$$\frac{1}{\pi} = 12 \sum_{k=0}^{\infty} \frac{(-1)^k (6k)!(13591409 + 545140134k)}{(3k)!(k!)^3 640320^{3k+3/2}} \tag{4}$$

The formula is highly convergent, adding on average 14.18 digits to π per iteration. Details regarding the computation of the convergence rate can be found in [6, p. 44]. We excluded the fractional power in the denominator from the infinite sum, as it is independent of k:

$$\begin{aligned} \frac{1}{\pi} = & \frac{12}{640320^{3/2}} \\ & \cdot \sum_{k=0}^{\infty} \frac{(-1)^k (6k)!(13591409 + 545140134k)}{(3k)!(k!)^3 640320^{3k}} \end{aligned} \tag{5}$$

To overcome the precision problems that arise from the fact that numbers with many digits are not accurately represented in a computer using the IEEE standard floating-point definition [7], we used Python's *decimal* module. [16]

Distributing the Workload across Ranks

The summation in Equation (5) can be distributed easily by assigning a share of the overall iterations to each rank in the worker group. However, preliminary tests have indicated that the execution time is all but linear in k, as presented in Table 3. While the initial 400 iterations take 3 seconds, the final 400 require more than 3 minutes, clearly a case of load imbalance.

Table 3: Execution time for the first 2 000 iterations of k of Chudnovsky summation in portions of 400.

Start iteration	End iteration	Execution time (MM:SS)
0	399	00:03
400	799	00:19
800	1199	00:47
1200	1599	01:43
1600	1999	03:03

To achieve greater acceleration in computing the Chudnovsky summation in parallel, it is necessary to allocate varying portions of the total number of iterations to each rank. Simply put, there should be uniform execution times across all ranks. Our solution is to calculate the number of iterations on each rank through an evenly distributed execution time. The Chudnovsky series is said to have a runtime complexity of $\mathcal{O}(n \cdot log(n)^3)$.[17] We assume to be able to compute the execution time with the runtime complexity as follows:

$$execution\ time = k \cdot log(k)^b, \tag{6}$$

with k being the number of iterations. Trying different values for b indicated that in our case $b = 14$. The reason for the much larger exponent in experiments compared to the runtime complexity might be the representation of π with Python's *decimal* module for arbitrary precision, instead of just computing the digits as done by γ-cruncher.

With Equation (6), we can compute the total execution time for the total number of iterations. The time is divided into equal portions given the number of ranks. Knowing the left-hand side of Equation (6) allows us to solve for k. Thus, the start and end of the Chudnovsky summation can be calculated on every rank. An example of applying this method is shown in Table 4. The execution time of each iteration block is not equal, but noticeably more balanced when compared to the usage of equal-sized portions. This approach is suitable for our case. Having a certain degree of imbalance when demonstrating the μ-Cluster could increase visitors' interest in parallel computing issues.

[16] `https://docs.python.org/3/library/decimal.html` (accessed 2023-11-25)
[17] `http://www.numberworld.org/y-cruncher/internals/formulas.html` (accessed 2023-11-26)

Table 4: Execution time for the first 2 000 iterations of k of Chudnovsky summation. The portions are computed by solving the function $t = k \cdot log(k)^{14}$ for k where t is the evenly distributed execution time.

Start iteration	End iteration	Execution time (MM:SS)
0	1149	01:00
1150	1454	00:59
1455	1672	01:02
1673	1848	01:03
1849	1999	01:07

6 Displaying Information

In the course of this project, we also updated the displayed information. This includes new QR code images as well as logos of Python packages used for our implementation. Furthermore, information about the μ-Cluster is now displayed directly. We created 8 new images, each showing some aspects of the cluster. For instance, details about the Odroids or the network connecting them. It is designed to be shown on 8 screens arranged in a 2×4 layout.

To give an idea of what this looks like, screen captures can be found in Appendix E. An example of showing some of the information is displayed in Figure 24, some of the QR codes are presented in Figure 25 and lastly, a view of the logos is depicted in Figure 26.

The implementation is rather simple, using MPI groups to communicate between the ranks. The simplified relationship diagram can be found in Figure 10 in Appendix A.

7 Conclusion and Future Work

In this project, we developed visually appealing applications for public demonstrations of the μ-Cluster. The visitors can now interact with the demonstrations utilizing both gamepad and touchscreen functionalities. The applications can run on a single machine, as demonstrated by the exemplary screen captures provided. However, the project's success cannot be evaluated until executed on the μ-Cluster.

To achieve this goal, we must update the Odroids' operating system. Once the cluster is operational, we will conduct extensive testing of our implementation. Additionally, we require further experimentation with the *CelebA* database splitting process. Finally, a visual inspection of the non-cropped, original images is unavoidable to ensure that they are suitable for display at exhibitions.

References

[1] L. Dalcin and Y.-L. L. Fang, "mpi4py: Status update after 12 years of development," *Computing in Science & Engineering*, vol. 23, no. 4, pp. 47–54, 2021. doi: 10.1109/MCSE.2021.3083216.

[2] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach, 4th Global Edition*. Pearson Deutschland, 2021.

[3] Z. Liu, P. Luo, X. Wang, and X. Tang, "Deep learning face attributes in the wild," in *Proceedings of International Conference on Computer Vision (ICCV)*, December 2015.

[4] R. P. Agarwal, H. Agarwal, and S. K. Sen, *Birth, growth and computation of pi to ten trillion digits (2013)*, pp. 363–423. Cham: Springer International Publishing, 2016. doi: 10.1007/978-3-319-32377-0_22.

[5] D. H. Bailey, P. B. Borwein, J. M. Borwein, and S. Plouffe, "The quest for pi," *The Mathematical Intelligencer*, vol. 19, pp. 50–56, 07 1997. doi: 10.1007/BF03024340.

[6] L. Milla, "An efficient determination of the coefficients in the Chudnovskys' series for $1/\pi$," *The Ramanujan Journal*, vol. 57, 02 2022. doi: 10.1007/s11139-020-00330-6.

[7] "IEEE Standard for Floating-Point Arithmetic," *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pp. 1–84, 2019. doi: 10.1109/IEEESTD.2019.8766229.

A Project Structure Diagrams

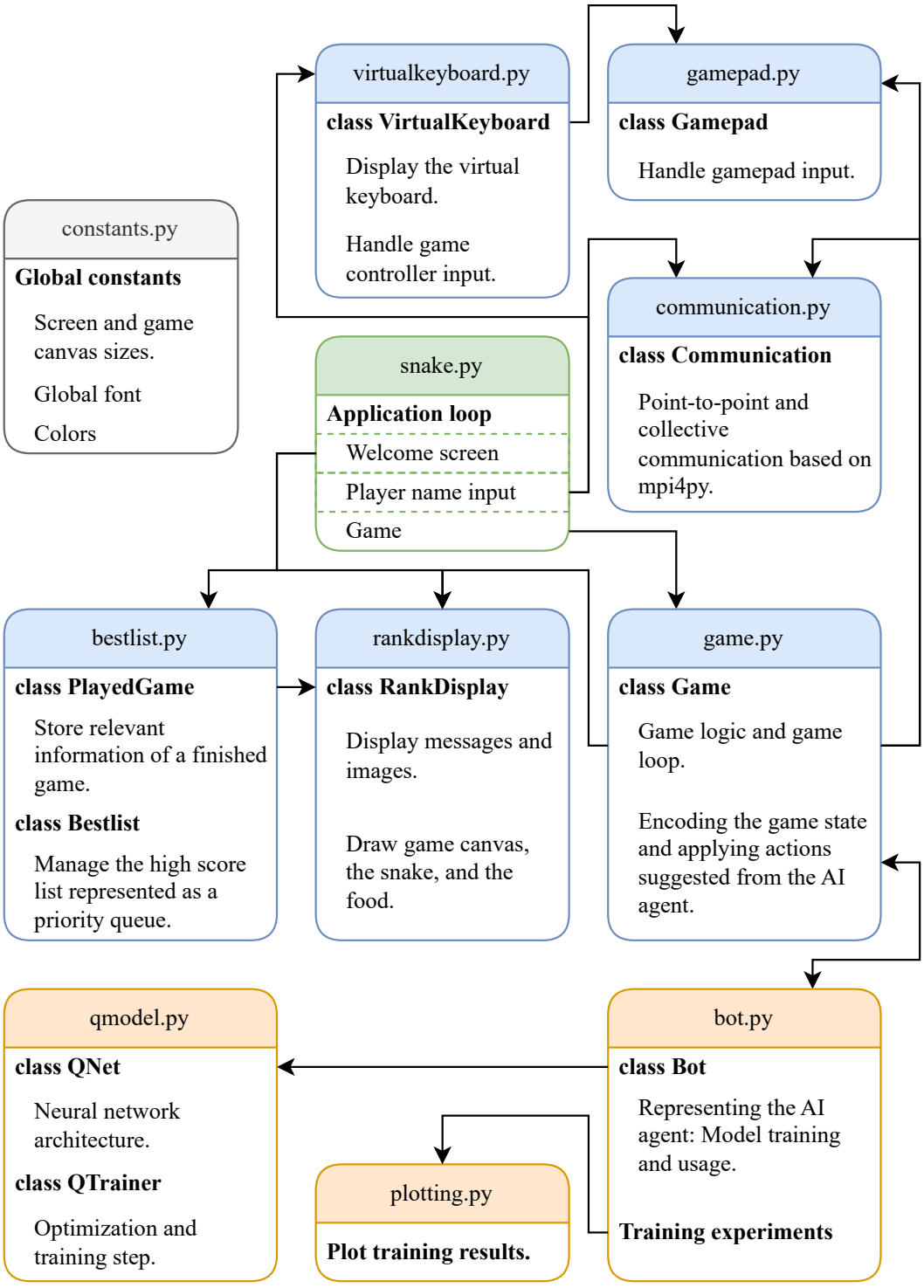

Figure 7: Simplified relationship diagram for the 'Play *Snake* against an AI Opponent' implementation. The colors correspond to distinct components of the project. Green depicts the entry point, blue relates to the main components of the application, orange signifies code dedicated to the AI agent, and gray indicates universal constants available.

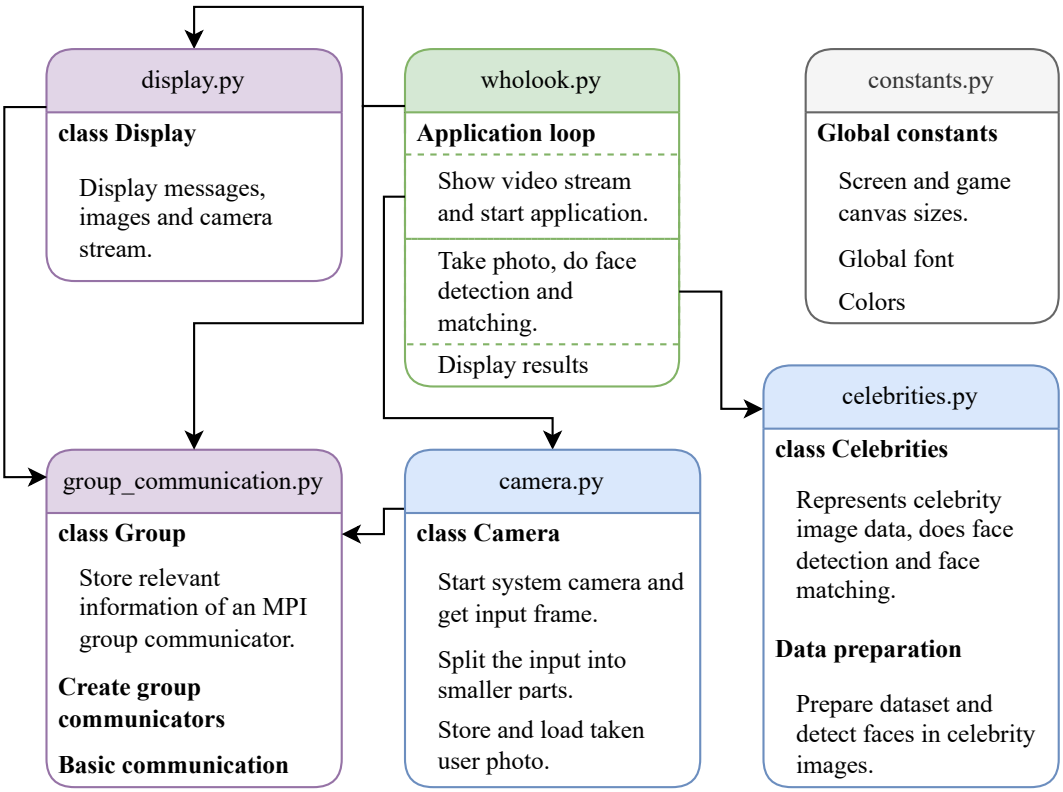

Figure 8: Simplified relationship diagram for the 'Which Celebrity you look like?' implementation. The colors correspond to distinct components of the project. Green depicts the entry point, blue relates to the main components of the application, purple stands for commonly available functionality, and gray indicates universal constants available.

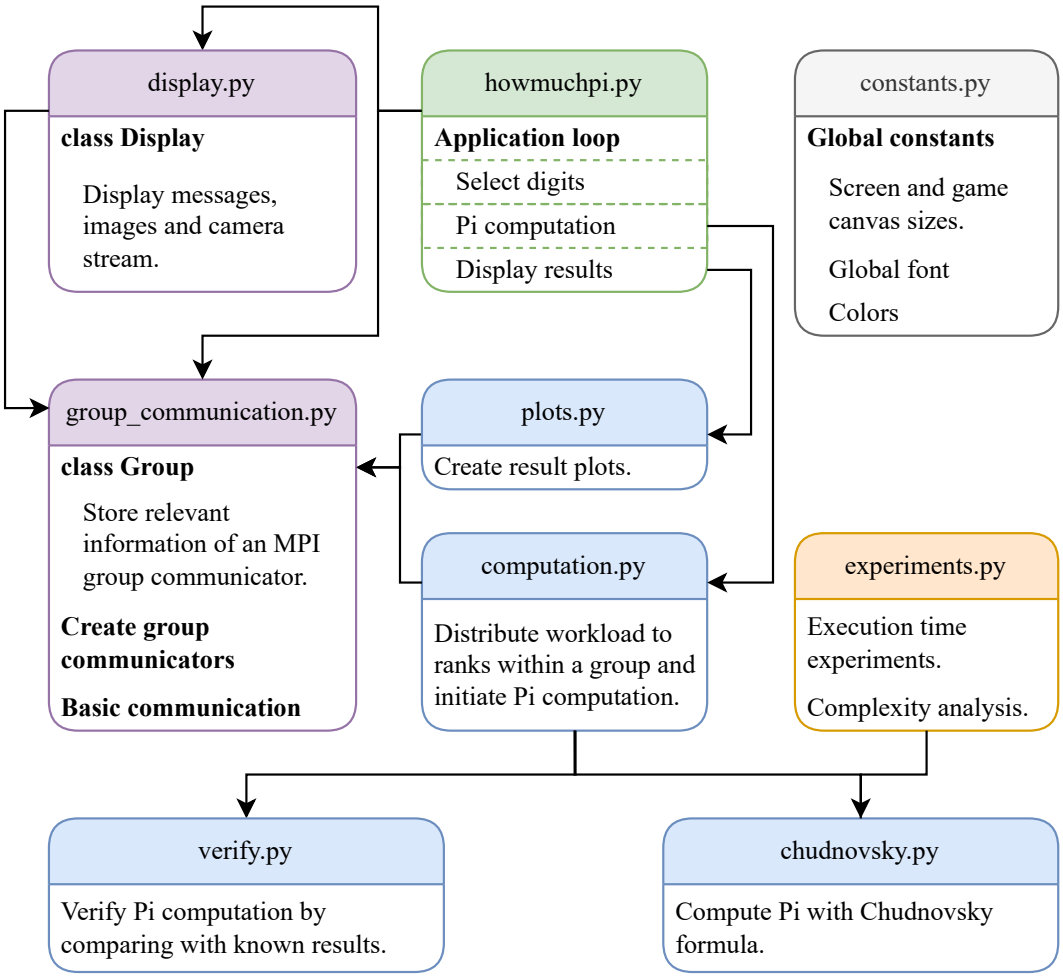

Figure 9: Simplified relationship diagram for the 'How much Pi you like?' implementation. The colors correspond to distinct components of the project. Green depicts the entry point, blue relates to the main components of the application, purple stands for commonly available functionality, orange signifies code dedicated to experiments, and gray indicates universal constants available.

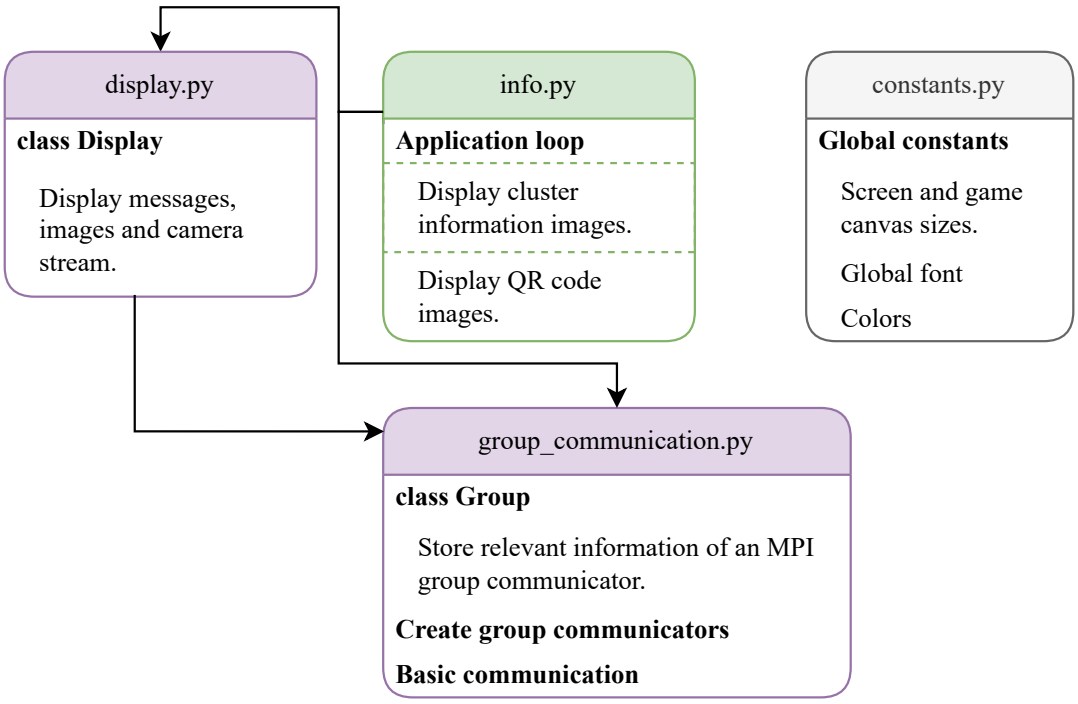

Figure 10: Simplified relationship diagram for the 'Display Information' implementation. The colors correspond to distinct components of the project. Green depicts the entry point, purple stands for commonly available functionality, and gray indicates universal constants available.

B Screen captures of 'Play *Snake* against an AI Opponent'

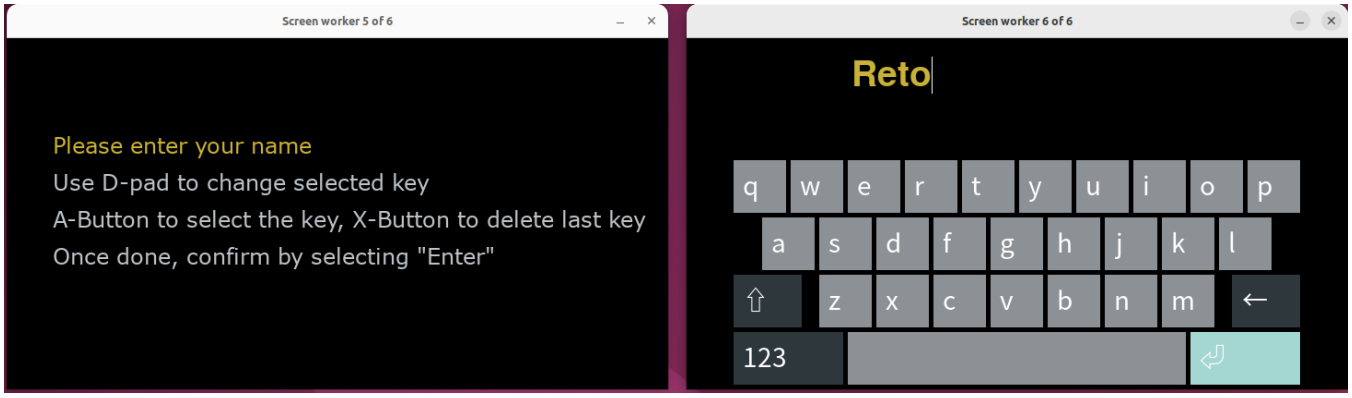

Figure 11: The two screens where the user is asked to provide their preferred pseudonym. One showing the instructions and the other the virtual keyboard controlled with the gamepad. Picture taken while executing the 'Play *Snake* against an AI Opponent' application on our own individual machine, allowing up to 6 MPI ranks.

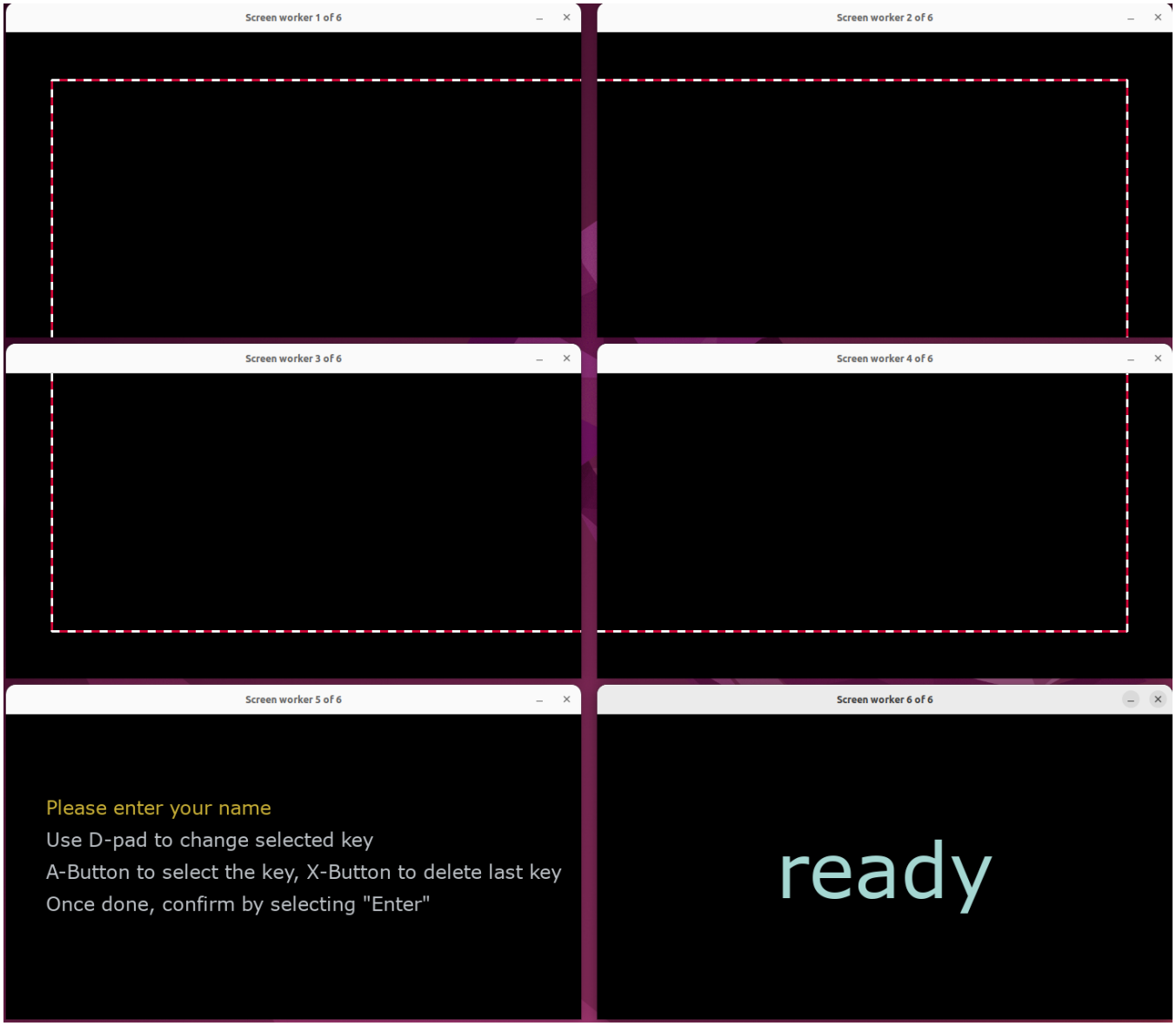

Figure 12: The situation before the game begins, displaying a countdown on the lower right screen for the user to be ready. Picture taken while executing the 'Play *Snake* against an AI Opponent' application on our own individual machine, allowing up to 6 MPI ranks.

Figure 13: The view while playing the game. The lower left screen shows the number of eaten apples. The lower right screen displays the remaining game time. Picture taken while executing the 'Play *Snake* against an AI Opponent' application on our own individual machine, allowing up to 6 MPI ranks.

C Screen captures of 'Which Celebrity you look like?'

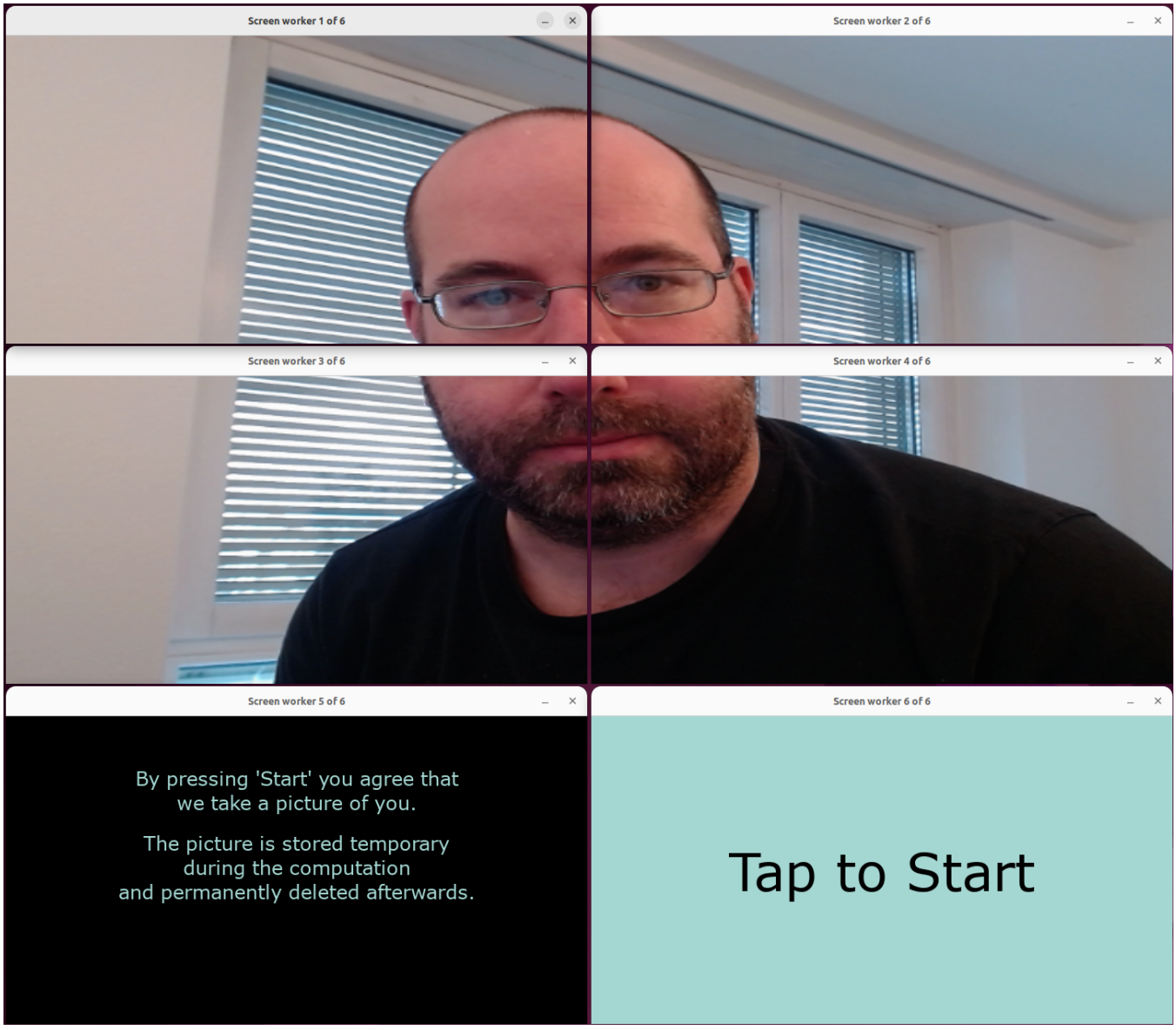

Figure 14: Show camera capture and allow user to start the application by tapping the screen. Picture taken while executing the 'Which Celebrity you look like?' application on our own individual machine, allowing up to 6 MPI ranks.

Figure 15: Display countdown on the lower right screen for the user to get ready for the photo. Picture taken while executing the 'Which Celebrity you look like?' application on our own individual machine, allowing up to 6 MPI ranks.

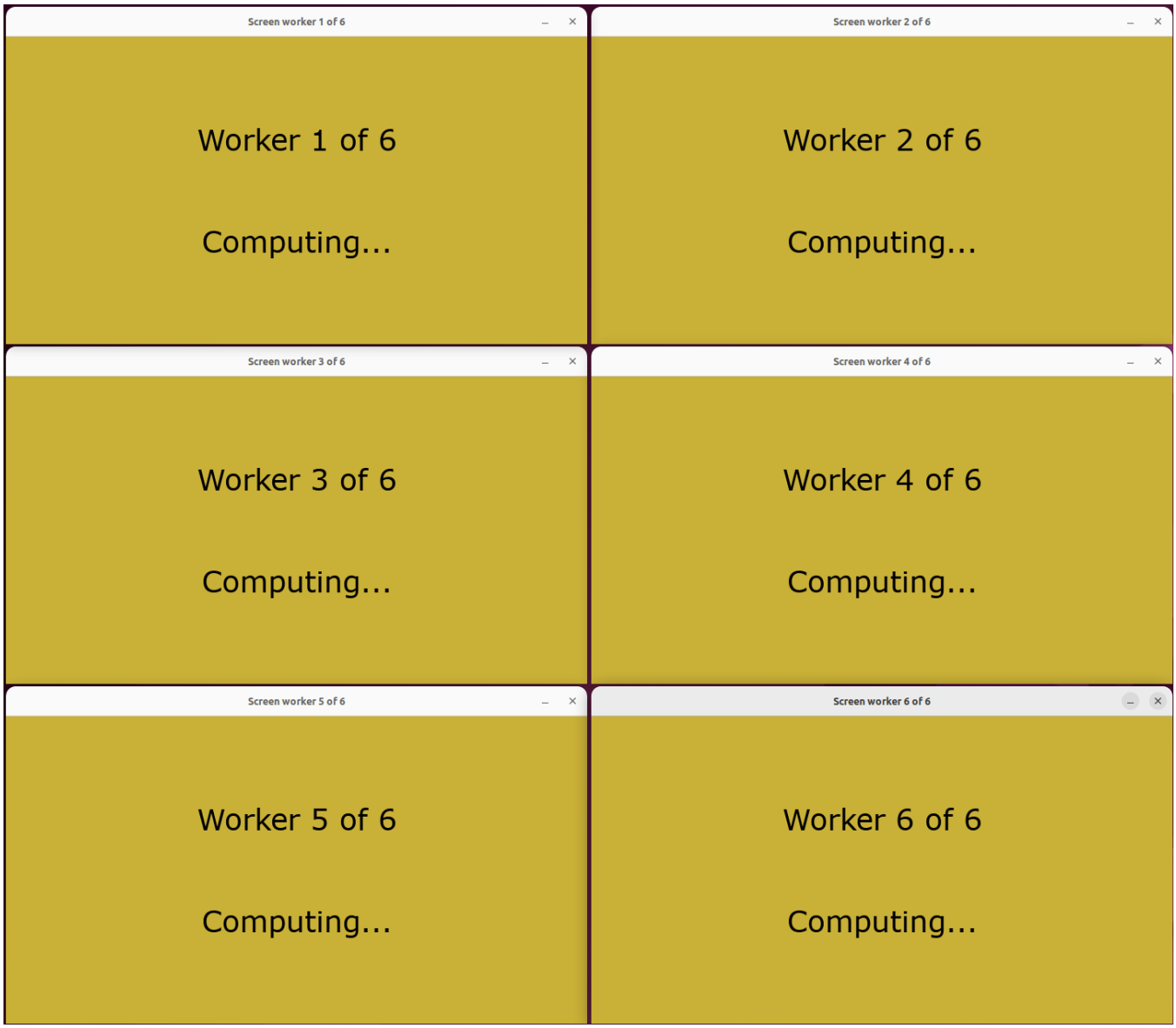

Figure 16: Show computing workers. Picture taken while executing the 'Which Celebrity you look like?' application on our own individual machine, allowing up to 6 MPI ranks.

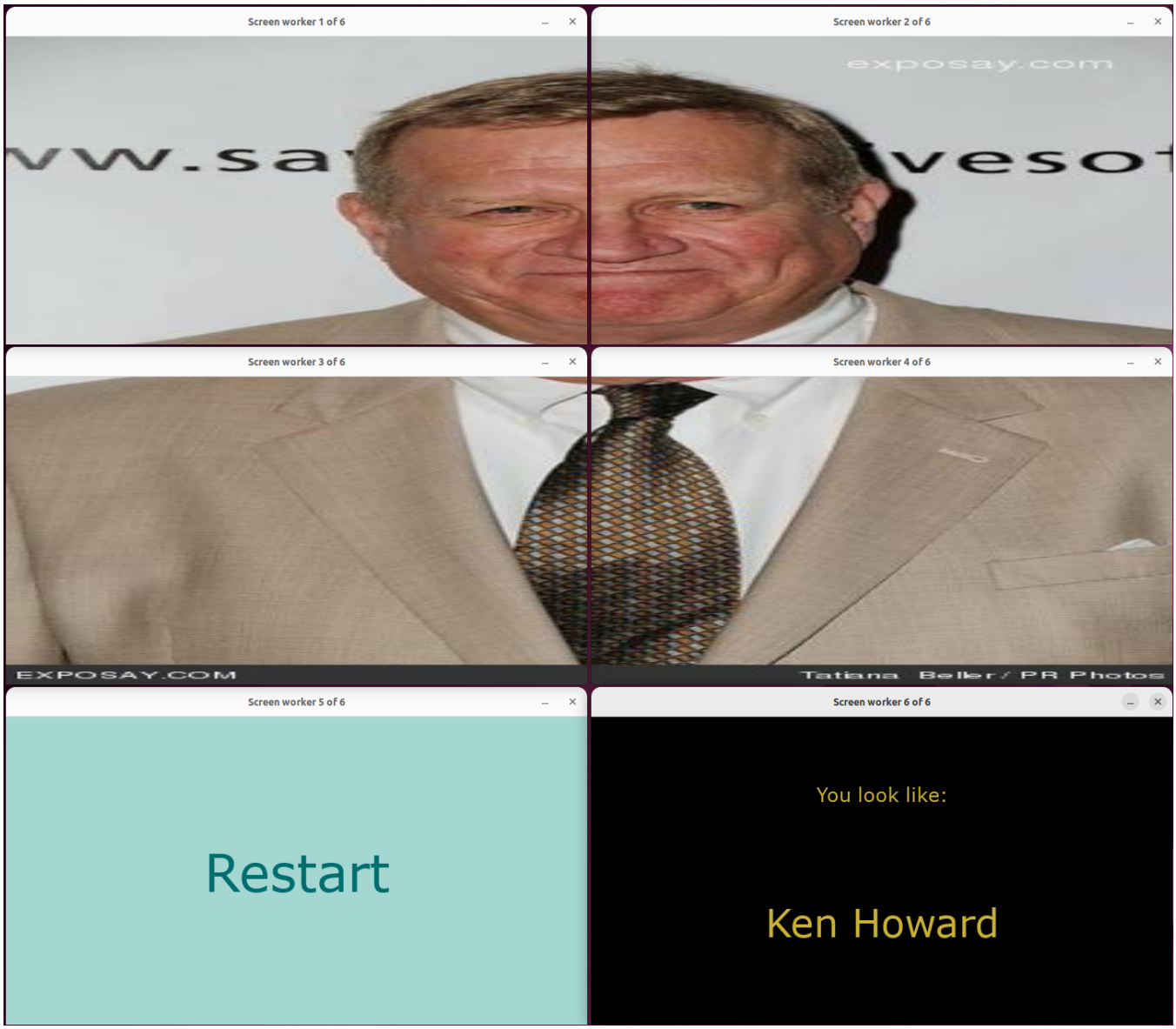

Figure 17: Display the results: The name and the photo of the celebrity the user resembles most, together with the restart button. Picture taken while executing the 'Which Celebrity you look like?' application on our own individual machine, allowing up to 6 MPI ranks.

D Screen captures of 'How much Pi you like?'

Figure 18: Welcome screen: By touching the appropriate screen, the user selects the number of digits of pi they wish to calculate. Picture taken while executing the 'How much Pi you like?' application on our own individual machine, allowing up to 6 MPI ranks.

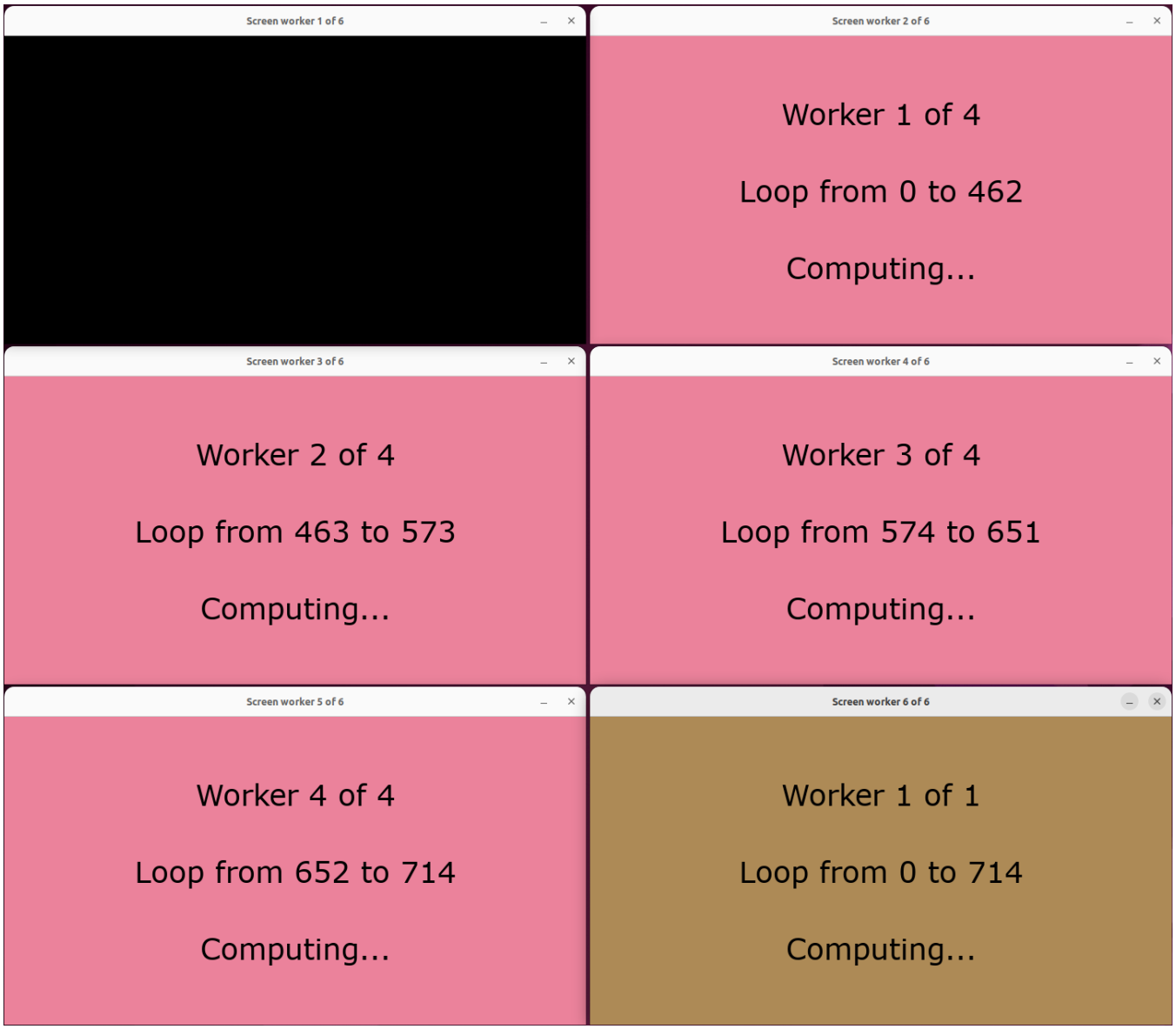

Figure 19: Chudnovsky series iteration with two groups. 4 ranks in red and 1 in gold. Each rank in red has a different workload. Picture taken while executing the 'How much Pi you like?' application on our own individual machine, allowing up to 6 MPI ranks.

Figure 20: Ranks within a group wait for all to finish, being able to combine the results. 3 ranks in the red group have already completed their workload. Picture taken while executing the 'How much Pi you like?' application on our own individual machine, allowing up to 6 MPI ranks.

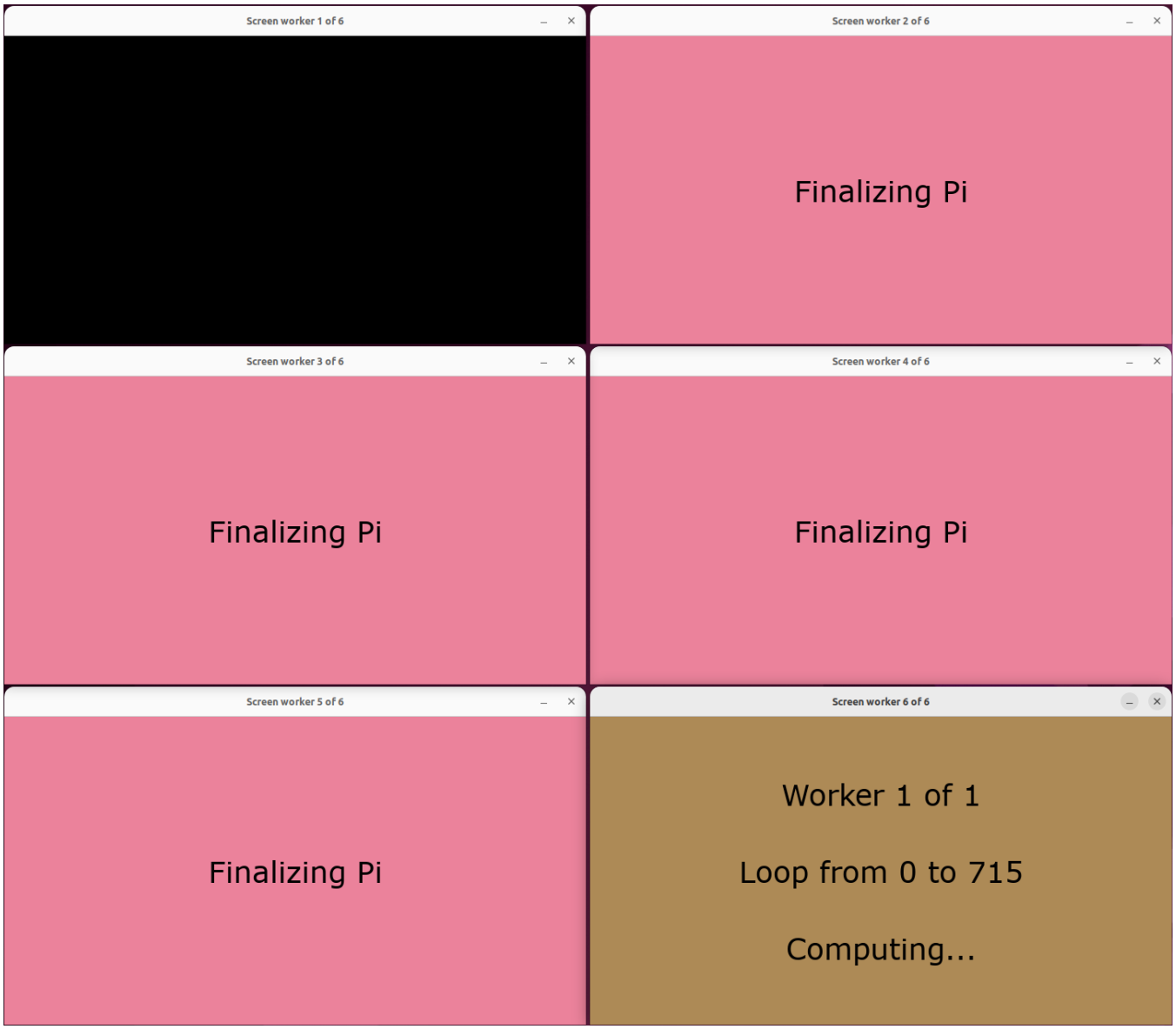

Figure 21: Ranks in the group red combined the individual results and are computing Pi. The single worker from gold is still occupied with iterating. Picture taken while executing the 'How much Pi you like?' application on our own individual machine, allowing up to 6 MPI ranks.

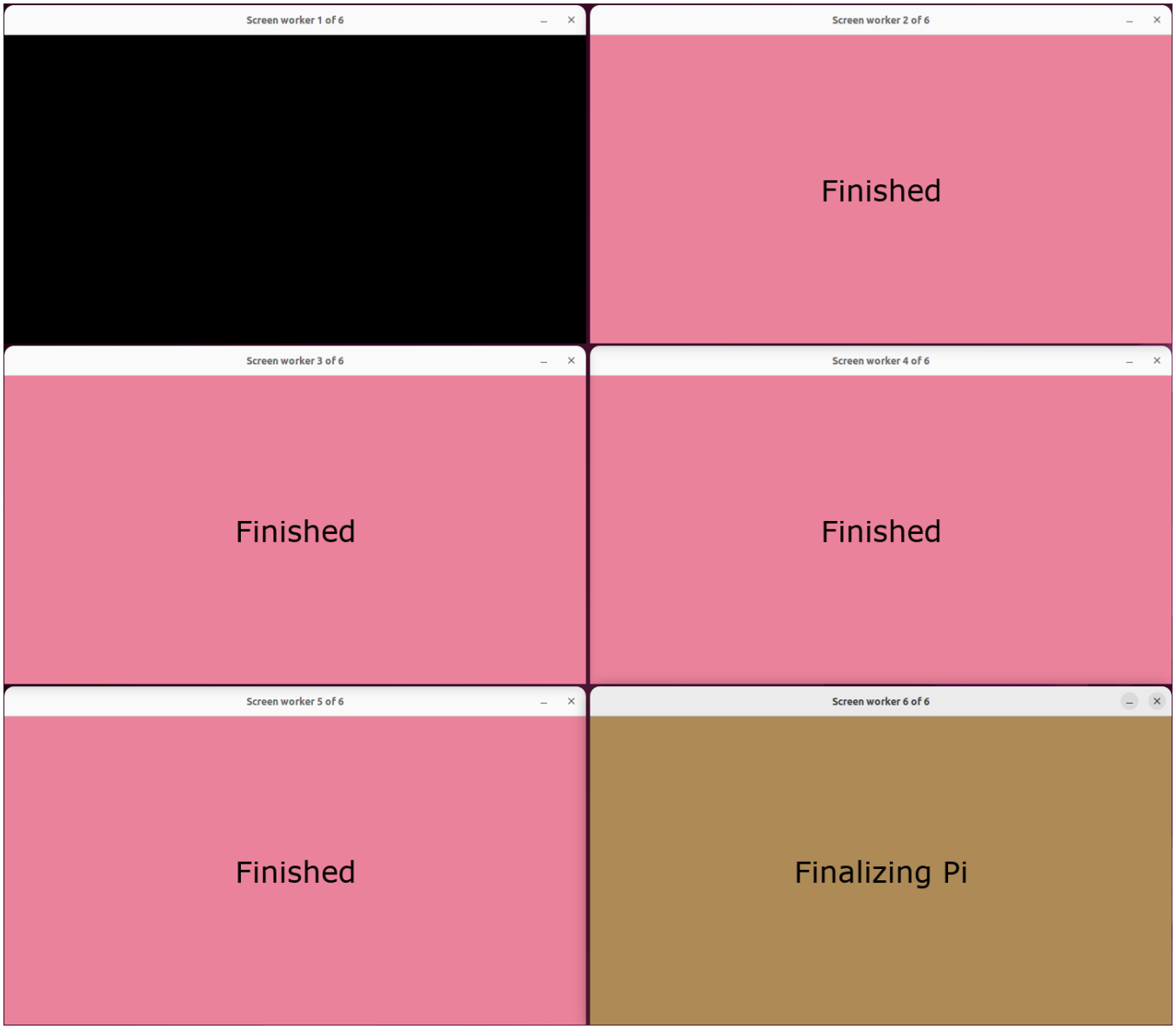

Figure 22: The red group has finished and waits for gold. Picture taken while executing the 'How much Pi you like?' application on our own individual machine, allowing up to 6 MPI ranks.

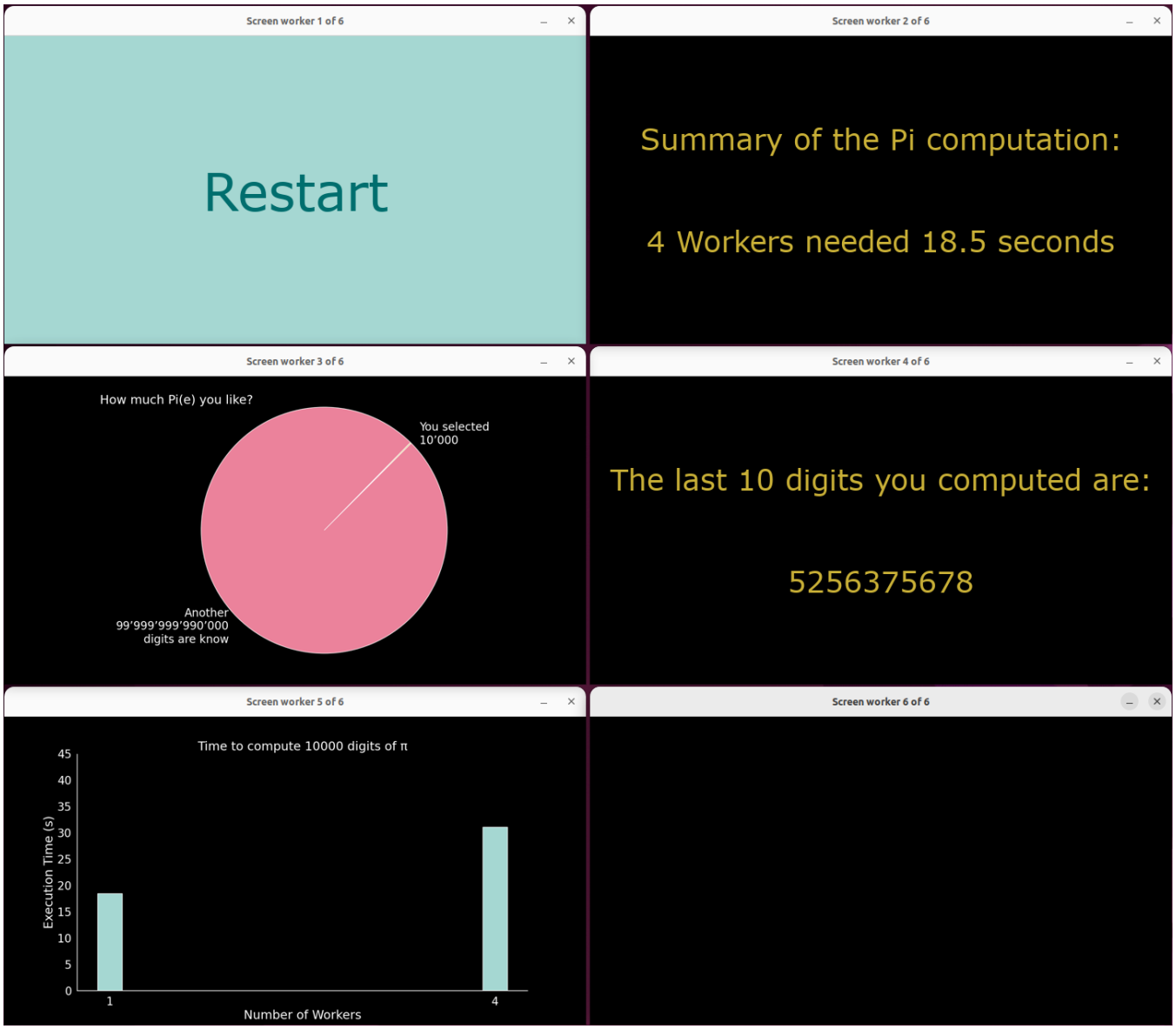

Figure 23: After all worker groups have completed their tasks, the execution time and some more information is displayed. Pushing 'restart' will restart the application. Picture taken while executing the 'How much Pi you like?' application on our own individual machine, allowing up to 6 MPI ranks.

E Screen captures of 'Displaying Information'

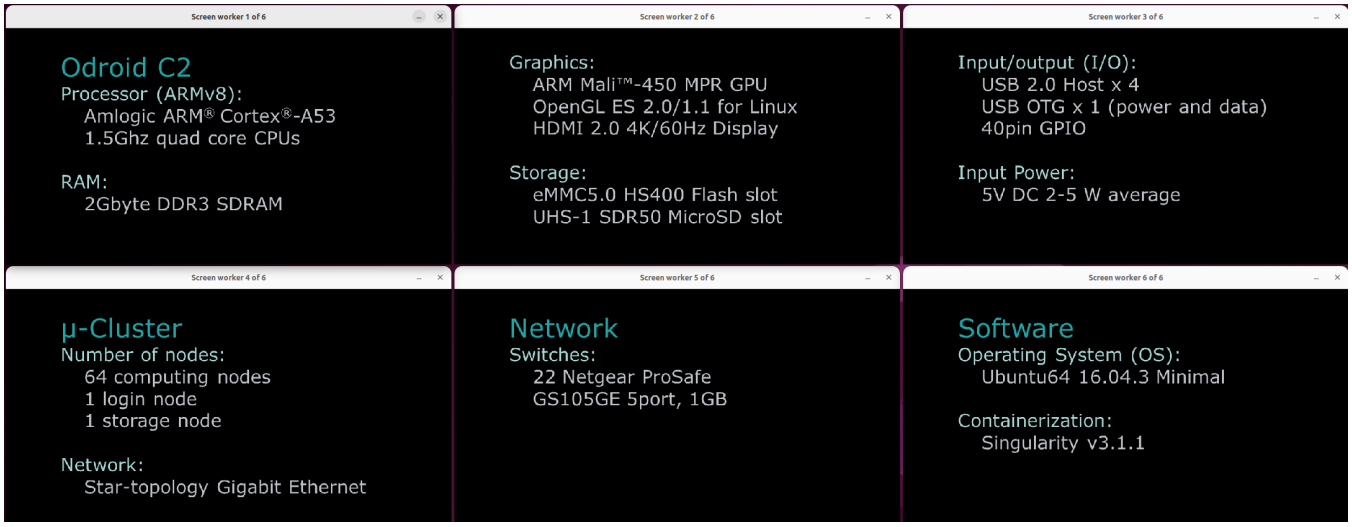

Figure 24: Show information about the cluster. Picture taken while executing the 'Displaying Information' application on our own individual machine, allowing up to 6 MPI ranks.

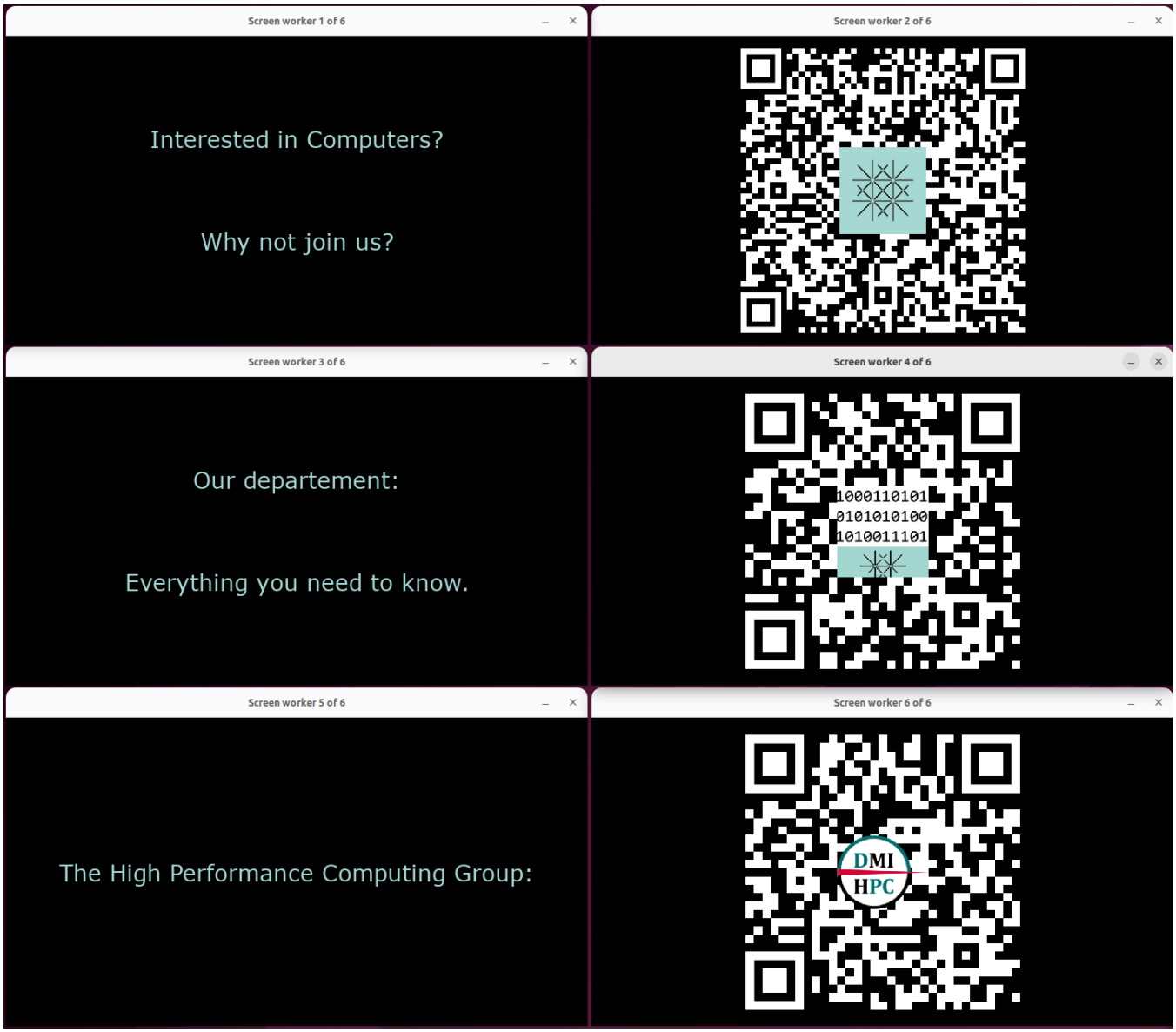

Figure 25: Show QR codes that lead to further content. Picture taken while executing the 'Displaying Information' application on our own individual machine, allowing up to 6 MPI ranks.

Figure 26: Show the logos of Python and used packages. Picture taken while executing the 'Displaying Information' application on our own individual machine, allowing up to 6 MPI ranks.