Visualising Task-Mapping in Many-Core Systems

Project Preparation Report for BEng Individual Project
ELE00090H

Jacqueline Walker

Supervisors
Dr. Gianluca Tempesti, Dr. Yuriy Zakharov

Vanbrugh College
University of York
January 22, 2024

Contents

Abstract

This report details the preparatory material necessary for undertaking the final year project Visualising Task Mapping in Many-Core Systems. The aim of the project is to develop a program that helps a user to create an initial mapping of an application process graph onto a 2D mesh many-core system.

This Project Preparation Report starts with a brief overview of the network-on-chip architecture and why it is useful, what many-core systems are, application process graphs and why they are needed, and task-mapping. Then the background on how to design a functional and visually pleasing UI based on fundamental principles on how a user interacts with an application is explained. Next, a literature survey is undergone, describing the sources used for the background, what information they gave and how they related to other sources.

Following this, the aims and objectives of the project are clarified, detailing the preliminary aims of the project along with a brief criteria of how to evaluate the success of the project.

Finally, a preliminary technical approach is given, where analysis of project planning methodologies, programming languages and UI libraries is underwent. These analyses will lay the groundwork for the approach in the following Initial Report.

Statement of Ethics

After considering the project and the Royal Academy of Engineering's guide to Ethics[1], there are no ethical issues that have been identified.

1 Background & Literature Survey

1.1 Background on Many-Core Systems

As core counts and power consumption increase, it became apparent that a new architecture needed to be found[2]. Bus-based architectures suffer from a number of issues, namely the lack of scalability and limited bandwidth[3] which cause increasingly complex designs and limit performance. Therefore a new paradigm was sought out.

The Network-on-Chip (NoC) architecture is an alternative to previous designs. Instead of directly connecting each component via a bus, the components are connected to a networking interface (NI) that handles communication between components[2]. These NIs are then connected to a networking switch. Communication is then done in a similar way to off-chip networks, either reserving a path for data to travel or via splitting the data into small packets to be routed to their destination component[4]. Using a network allows for a more scaleable architecture due to different components communicating using shorter local wiring rather than long global wires[5]. It also leverages the pre-existing understanding and research of networks, to allow for a better base knowledge of how the architecture works. Like off-chip networks, bandwidth scales with the network size and is not limited by the bus width[2].

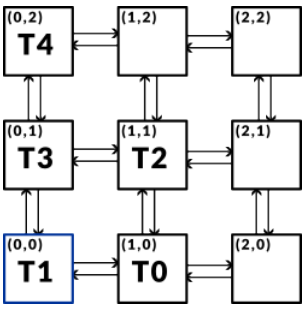

Figure 1.1: *A visualisation of a 3×3 2D mesh topology for a many-core system.*

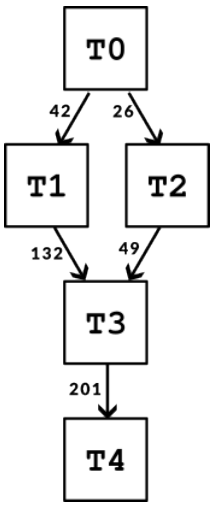

Figure 1.2: *An example Application Process Graph.*

The NoC structure serves as the backbone of many-core systems. Many-core systems consists of hundreds or thousands of relatively simple cores. These cores can then be used to run hundreds or thousands of simple tasks (referred to as processes)in parallel[5]. Many-core systems can be made in many different topologies, depending on use case. One such topology is the 2D mesh, which consists of a $N \times N$ grid and is one of the more popular topologies[2]. This allows for every switch, except for those on the edge, to be connected to 4 other switches[4]. Other topologies exist that try to solve different problems, such as number of routers or congestion between nodes, however they are beyond the scope of the project. The 2D mesh topology can be easily visualised as a 2D grid of nodes, as seen in Figure 1.1.

In order to utilize many-core systems, applications that need to be run must be split into simple tasks, referred to as processes. Processes only depend on the result from the previous process and, for our purposes, must not by cyclic. Processes are best visualized using an Application Process Graph (APG)[6]. An APG is a directed, acyclic graph that represents the application broken down into processes that depend only on the previous process(es). The weight of the edges are the bandwidth requirement for the communication from the previous process to the current process[4].

From the APG, we need to then map a given process onto the best node possible for that given process. This process is called task mapping. It is usually not feasible to iterate through all mappings and compare due to the number of potential mappings exponentially increasing with the number of nodes[6]. Ideally, the task-mapping must aim to keep the total communication cost and bandwidth of the application as low as possible[4]. In real systems, task-mapping is a continuous process done from the start and end of a system running. However, for our project we only consider an initial mapping of a many-core system created by the user.

1.2 Background on UI Design

UI design is built upon fundamental principles derived from how users tend to interact with applications in general. From an understanding of these principles we can then understand how to design and organize an application. These principles tend to vary between resources, so a compiled version has been created for Figure 1.3. Please note that these principles may not always be applicable in every context[7].

Safe Exploration The user must be able to try options out and be able to easily revert back without penalty. Without an easy way back users may become discouraged.

Instant Notification The user should be given immediate results after using an operation. If given no notice or change, users may become frustrated with the speed or usability of the application.

Familiarity Users tend to interact with an application in ways that they have learnt from previous applications. Therefore the UI should keep to well-understood conventions, like using `CTRL-S`, `CTRL-C`, `CTRL-V`.

Consistency Users expect that operations will work the same even in different contexts. Users also expect the UI to not change positioning or grouping in different contexts.

Skimming Users tend to skim-read the UI rather than fully reading everything. Therefore the UI should be kept as simple as possible to understand with fewer options and obvious grouping.

Figure 1.3: *A list of principles and their meanings. Derived from [7]–[9]*

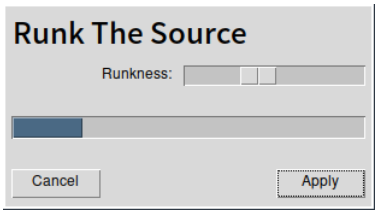

Figure 1.4: *An example of a program giving instant notification to the user that an action is being done via a progress bar.*

From these principles, we can then derive how to design a UI for maximum 'usability'. For example, in order to apply the principle of *Instant Notification*, for each action the user does, a notification or progress bar should be shown to the user to show that the operation is being worked on. If no notification or progress was shown, the user may believe the action wasn't done and may re-do the action multiple times in frustration. An example of this is shown in Figure 1.4.

Figure 1.5: *An example of the Familiarity principle. At first glance, which button looks more like it would be responsible for 'saving'?*

The principle of *familiarity* also dictates how we would design our UI. As we are designing for a desktop application, it would be best to follow the conventions for typical desktop applications[9]. For example, it would be best to use a floppy disk icon for saving as the user instinctively knows that a floppy disk icon means saving. Along with this, the save/load buttons should be placed in the top-left of the application to conform to the muscle-memory of previous applications[7]. Other conventions should be stuck to, such as using `CTRL-C`, `CTRL-X`, `CTRL-V` for copying/cutting and pasting respectively.

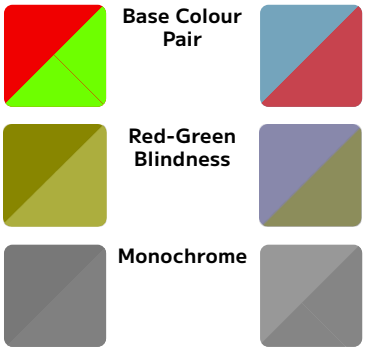

Figure 1.6: *An example of two colour pairs. One pair has poor contrast, is hard to distinguish for colourblind people and increases eyestrain. The other pair changes both hue and saturation to help distinguish the pair.*

The principle of *Skimming* is one of the most influential principles and has the most design consideratins because of this. One way to create an easy-to-skim UI is to keep objects that do similar functions visually close to each other, so that the user can instinctively tell they are grouped[9]. If objects are poorly-spaced, the user will have to spend more time trying to understand the UI, slowing down productivity. Another way is to ensure that any information displayed to the user is structured into labelled sections or bullet-points, helping to aid the user in picking up key information. Along with this, colours need to be chosen to aid readability. Since the eyes are adapted more to contrast than brightness, colours used in text should be different in both hue and saturation to keep them distinct[9]. Colours must also be chosen to aid users with colour blindness, i.e. by avoiding red & green colours near each other[8]. Examples of these are given in Figures 1.6 and 1.7.

Due to how the eye has evolved, the majority of the 'resolution' of the eye is at the centre of our sight. Because of this, users may struggle to notice changes around where they are looking[9]. This can be an issue when designing error pop-ups and notifications. In order to capture the eye's focus a pop-up needs to have a distinctive colour reserved only for errors i.e. bright red or yellow. Other considerations may be to dim the application while a pop-up is present[9].

From these principles and general considerations, there should now be an understanding of how to design a UI that is both good-looking and functions in an intuitive manner to the user. These principles will be described when doing the preliminary design work in Chapter 4.

There are 1024 cores in the hardware topology, resulting in a 32x32 2D mesh size. Additionally there are 218 processes needing to be mapped.

Core count: 1024
Process count: 218
Mesh size: 32x32

Figure 1.7: *A comparison of unstructured vs structured text.*

1.3 Preliminary Literature Survey

For the background of a many-core system we needed to understand what a many-core system is, how it's architecture worked and what task-mapping is. Therefore we decided to focus our research on sources that helped to explain the basics of many-core systems and NoCs. Some textbooks exist that aim to collate the ongoing research into many-core systems into one single location, which were pointed out by the project supervisor. One such book is *Designing 2D and 3D Network-on-Chip Architectures*[2]. This book helped to lay out the understanding of why NoCs came to be and how they worked, however certain parts are extremely verbose at times and needed other sources to understand. *On-Chip Networks*[5] is a shorter read than other sources and served more as a brief overview of the topic as a whole. To this end, the book achieved its goal quite well. *Network-on-Chip*[4] helped mainly to further clarify how the 2D mesh topology was laid out and how data moved around the system. Colin Bonney's PHD thesis[6] also helped to fill in any gaps left during research. Since the research done for the background was mainly for fundamentals on how a many-core system operates, there was little differences in opinion and each source tended to corroborate each other.

For the background on UI design, we wanted to understand the fundamentals on how to make a UI that both looked good and was easy to use. For this, a good starting point was the book *Human-Computer Interaction*[8] which is a well-known textbook for HCI and, in part, UI design. This helped to clarify certain fundamental principles on how a user interacts with systems and a few points on designing with accessibility in mind. From this, a source would be needed that helped to understand more about using these fundamentals to design a cohesive UI. The book *Designing with the Mind in Mind*[9] took the idea of deriving principles based on how the mind parses information, rather than how a user interacts with a system. This source showed how best to structure information for easy parsing along with understanding how a user tends to move around the interface. Finally, *Designing Interfaces* aimed to be a comprehensive guide on how to design a good-looking UI, split into sections on each component. This helped to collate the principles of how a user interacts with a system, as many sources tended to use slightly different principles for similar user tendencies. It is my belief that these sources answered the original question posited before starting research.

2 Aims & Objectives

2.1 Aims of Project

The aim of this project is to create a program that can visualise the process of creating an initial task-mapping of a many-core system from a pre-existing APG. Since the task-mapping is to be done by a user, the many-core system should not support a mesh size greater than 6×6 nodes. From this description we can then extrapolate a preliminary set of objectives, as seen in Figure 2.1. In later reports, such as the Initial Report, a more precise set of objectives will be given for a minimum, reasonable and ambitious implementation.

1. The program should allow the user to create a 2D mesh of any size from 3×3 to 6×6 inclusive.
2. The program should allow the user to import a pre-defined Application Process Graph for use with the 2D mesh.
3. The program should allow the mapping of tasks from the APG to nodes on the 2D mesh via some graphical means. At the very least, a context menu[1] should be supported.
4. A node in the 2D mesh should show what co-ordinate it is at and what task is mapped to it through text on the node.
5. The program should allow the user to export the task-mapping as an as-yet-undefined file format.

Figure 2.1: *A preliminary set of objectives for the project. These objectives describe basic functionality needed for the program.*

2.2 Criteria for Evaluation

Below lists a preliminary set of criteria for evaluating the outcomes of the project. These criteria aim to expand on the aim defined in Figure 2.1 with further considerations on the quality and robustness of the code for the project, along with considerations for the UI. These criteria are derived in part from [10].

- Are all the preliminary objectives met?
- Is the user interface easy to understand?
- Is the visualization of the 2D mesh clear to the user?
- Is the visualization of the APG clear to the user?
- Is the program easily compiled (if applicable)?
- Is the program easily runnable?
- Is documentation given to the user in some form?

[1] A context menu is the menu that usually appears when the mouse is right-clicked.

- Are exceptions properly handled? Is the user only prompted for relevant exceptions?
- Does the codebase keep a consistent coding style?
- Is the code easy to maintain if more features are needed to be added?
- Is the codebase fully testable via unit testing[1]?

[1]Unit testing is a testing ideology that aims to test each component of a system as it is developed, reducing the need for full testing at the end of development. Definition from [11].

3 Approach

As this is the Project Preparation Report, precise plans will be omitted until the Initial Report. Instead, we will be analysing different parts of the groundwork needed for the project. These include:

- Project Management Methodology
- Programming Language of Choice
- UI Library of Choice
- Preliminary UI Designs

By investigating these sections we should be able to start the design of the project in the Initial Report with all the groundwork finished.

3.1 Project Management

In order to create a plan for the initial project, a investigation into different project management methodologies would be useful. Since our project is entirely software-development based, our chosen methodology should be designed to work with software projects and prioritise proper testing and coding standards. Along with this, our methodology chosen should also be able to adapt to being for an individual rather than a team. In Table 3.1 a table is given listing various project management methodologies is given, along with a description of how they work, their benefits and drawbacks. From this, a potential project management methodology is suggested for use with the project.

Methodology	Description	Advantages	Disadvantages
Scrum (Agile)	Each stage of the project is done in 30-day long 'sprints', i.e. design takes 30 days, development takes Nx30 days, etc.	Ensures that most important functionality is implemented first; quick prototyping	Testing is only done at the end of a sprint; Issues fitting in with traditional planning graphs
eXtreme Programming (Agile)	Project is split into week long stages of 'iteration', where small parts of the project are designed, tested, developed and released each 'iteration'.	Quicker prototyping; focus on small teams/individuals; constant testing throughout	Hard to give guarantees of time; Issues fitting in with traditional planning graphs
Waterfall	Each stage of the project is completed sequentially until the end of the project, i.e. design ->development ->unit testing ->system testing ->release	Easier guarantee of time; each stage is fully complete before moving to next stage	Very in-flexible if issues occur; hard to go back to previous stages and add functionality
V-Model (Waterfall)	Same as Waterfall but testing is done at the end of each stage.	Easier guarantee of time; Ensures each component is fully tested and finished before moving to next stage	Hard to go back to previous stages and add functionality; hard to create prototypes early in the process

Table 3.1: *A table listing different project management methodologies, their advantages and disadvantages. Information taken from [12]*

From this table, we can see that an Agile methodology would most likely be best for our project. SCRUM works best with a larger team, needing a dedicated 'SCRUM Master' to keep the team motivated[12]. XP works best with smaller teams, where team member's roles can be easily re-defined. With some modification it is my opinion that eXtreme Programming would be a good methodology to use, due to it's quick prototyping and insistence on continuous testing[12].

In order to ensure that this methodology can be easily tracked, deadlines should be enforced to ensure that certain functionality are to be implemented by a specific date. Other modifications should also be made to ensure that an individual is able to follow the methodology.

3.2 Programming Language

From Figure 2.1, it can be seen that our chosen programming language will need to be able to support these objectives. Ideally, the language should be able to be run and compile on University machines and on both Unversity OSes. It should also be able to have the ability to package needed libraries with the program when running. Along with this, the language should also support object-orientation and event-driven programming as most graphics libraries use objects and events inherently. From this, we can derive a table to see which programming language would be best to choose.

Language	Support for Linux	Support for Windows	Object-Orientated	Supports Events	Can Bundle Libraries
C	Y	P	N	Y	Y
C++	Y	P	Y	Y	Y
Java	Y	Y	Y	Y	P
Python	Y	Y	Y	Y	P

Table 3.2: *Evaluation of Programming Languages based on necessary features. Y means Yes, P means Partial Support, N means No.*

From Table 3.2, we can see that the best programming languages in terms of meeting our criteria are Python, Java and C++. C suffers from lack of support for object-orientation and a lack of high-level UI toolkits. C++ is a good choice, but suffers due to compilation of C++ on Windows being more complicated than on Linux, along with needing parts of the code to be rewritten depending on which OS the program is compiled on[1]. Both Python and Java have good support on both Windows and Linux due to them being either interpreted or running in a VM. Libraries can be bundled with Java during the compilation, but would then make support for compilation on University machines difficult. Python also has these issues, but steps are being made to pre-package libraries with an executable via PyInstaller[13]. Finally, I have previous experience in both Java and Python, therefore choosing either of these languages would help during development. Because of these factors, further research will be done into UI libraries for both Java and Python to determine which language will be best for the project.

3.3 GUI Library

As seen from Figure 2.1, our GUI library should support various functionality. The GUI libraries must:

- Support drawing primitive objects onto the screen (squares, lines, etc.)
- Support drawing text onto the screen
- Support modifying drawn object properties

[1]This is due to Windows not following the POSIX standard, and therefore missing out many useful functions, along with parts of the standard libraries. This issue also affects C.

- Support basic UI elements (buttons, sliders, etc.)
- Allow customization of styles of elements
- Be simple to code with
- Allow function/method calls based on interaction with elements

As mentioned in Section 3.2, we will only be comparing GUI libraries that support either Java or Python. Below list a variety of libraries available that support both Linux and Windows.

UI Library	Language	Support for Primitive Shapes	Support for Modifying Elements	Support for UI Elements	Style Customization	Function Calls from UI Elements	Availability of Tutorials	Ease of Coding
Tkinter	Python	Y	Y	Y	Y	Y	Y	4/5
Kivy	Python	Y	Y	Y	Y	Y	Y	3/5
PyQT	Python	Y	Y	Y	Y	Y	P	3/5
PySimpleGUI	Python	Y	N	Y	P	P	Y	2/5
Swing	Java	P	Y	Y	Y	Y	P	2/5
JavaFX	Java	Y	Y	Y	Y	Y	Y	3/5
AWT	Java	Y	P	P	P	Y	N	1/5

Table 3.3: *A comparison of different Python and Java UI libraries. Y means yes, P means Partial Support, N means No.*

As seen from Table 3.3, either Tkinter or Kivy would be a good choice if using Python, or JavaFX if Java is used. Other libraries are either lacking functionality (PySimpleGUI lacks modifying the UI once drawn) or are too hard to code with (PyQT is too complex a UI library for a simple project). In order to test whether these chosen libraries are fit for use in the project, simple programs were written in order to test basic functionality needed for the project (i.e. draw a simple 2D grid of nodes, have a button that changes the canvas in some form, etc.). These example programs can be seen in Appendix A.

3.3.1 Tkinter with Python

Tkinter, while quite old, is still a good choice for a GUI library. Documentation and tutorials are numerous due to its age and popularity. However, the look is dated at times and can suffer from issues when doing fine tweaking of margins around components.

3.3.2 Kivy with Python

Kivy is a relatively modern UI library with great support for different layouts, akin to the flexibility given with CSS. However, creating a canvas is obtuse, due to each component 'being' a canvas, along with operations only changing pixels on the canvas. Along with this, features such as tagging objects requires you to use their custom `.kv` layout file.

3.3.3 JavaFX with Java

JavaFX has some novel features, such as being able to customize elements through CSS and being able to use Swing or AWT components with little difficulty. However due to JavaFX no longer being packaged with Java anymore, issues arise with packaging JavaFX with the program. Going ahead with JavaFX would mean needing to learn and utilise a build system to automate packaging JavaFX with the application. Along with this, JavaFX's canvas is similar to Kivy's, where the canvas must be drawn over to edit any object.

Overall it is my belief that Tkinter would be the best choice for the GUI library due to its vast amount of resources due to its maturity, along with its object-based approach to drawing to a canvas. Because of this decision we will be using Python in our project. Python includes a great mix of support for object-orientation along with simpler functional programming support.

3.4 Preliminary UI Design

The UI should serve the function of drawing the eye to the 2D mesh and the APG. Because of this, both graphs should stay central and large to show their importance to the user. Along with this, a menubar should be shown at the top of the program, where the options for saving, loading and creating mappings can be stored. This should keep the muscle memory of previous applications intact, fulfilling the principle of *familiarity*. The text given to the user should also stay as structured as possible, using separators between sections if necessary. There should be as few options as necessary to keep the UI easy to skim-read at a glance. Below shows a potential design that fulfills these criteria.

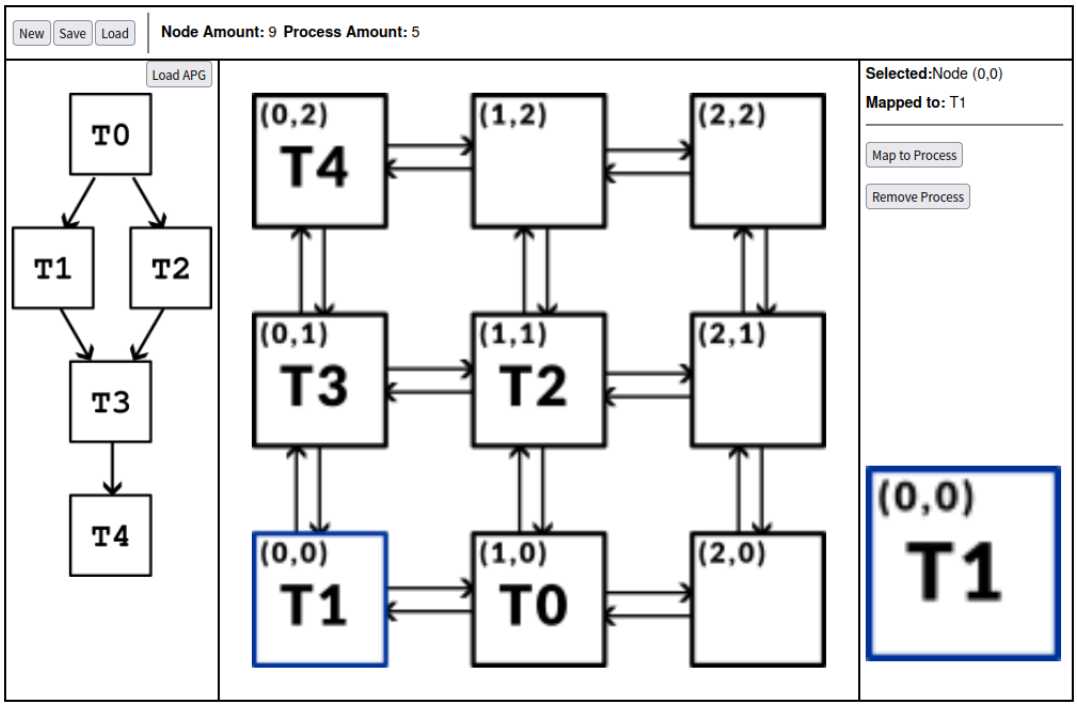

Figure 3.1: *A potential layout for the application, keeping focus towards the 2D mesh and the APG. The APG has 5 processes and the 2D mesh is 3x3. The UI is split into 3 columns; one for the APG, one for the 2D mesh, and one for modifying the selected node.*

Work must be made to ensure pop-ups are easy to parse. By following previous conventions on how pop-ups should look (i.e. exit in top right, OK button in bottom left and cancel button in bottom left), we should keep the pop-ups looking familiar to the user. Ideally, we would also like each pop-up to look similar to previous pop-ups to keep consistency. Along with this, each pop-up should have a brief description of what the action will do, in case the user is trying out the interface for the first time. Below shows an example pop-up for creating a new 2D mesh for mapping to.

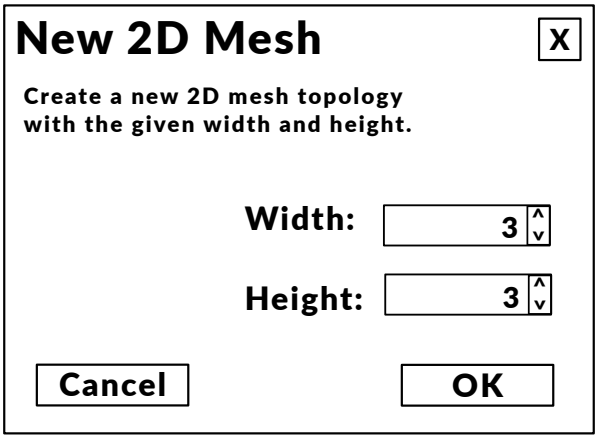

Figure 3.2: *An example design of a pop-up for creating a new 2D mesh. Includes a small description of what the pop-up does, keeps options visually close together to ensure the user can tell they are grouped. The options are also next to the OK button to show the next action the user should do.*

References

[1] R. A. of Engineering. "Engineering ethics in practice: A guide for engineers." (), [Online]. Available: https://raeng.org.uk/media/cz5du0gl/engineering_ethics_in_practice_full.pdf (visited on 12/14/2023).

[2] K. Tatas, K. Siozios, D. Soudris, and A. Jantsch, *Designing 2D and 3D network-on-chip architectures*. Springer, Aug. 2016.

[3] H. G. Lee, N. Chang, U. Y. Ogras, and R. Marculescu, "On-chip communication architecture exploration: A quantitative evaluation of point-to-point, bus, and network-on-chip approaches," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 12, no. 3, May 2008, ISSN: 1084-4309. [Online]. Available: https://doi.org/10.1145/1255456.1255460.

[4] S. Kundu and S. Chattopadhyay, *Network-on-Chip: The Next Generation of System-on-Chip Integration*. CRC Press, 2015.

[5] L.-S. Peh and N. E. Jerger, *On-Chip Networks*. Morgan and Claypool, 2009.

[6] C. A. Bonney, "Fault tolerant task mapping in many-core systems," Ph.D. dissertation, Department of Electronic Engineering, University of York, May 2016.

[7] J. Tidwell, *Designing Interfaces*. O'Reilly Media, 2011.

[8] A. Dix, J. Finlay, G. Abowd, and R. Beale, *Human-Computer Interaction*. Pearson Education Ltd., 2004.

[9] J. Johnson, *Designing with the Mind in Mind: Simple Guide to Understanding User Interface Design Guidelines*. Morgan Kaufmann, 2021.

[10] M. Jackson, S. Crouch, and R. Baxter. "Software evaluation: Criteria-based assessment." (), [Online]. Available: https://www.software.ac.uk/sites/default/files/SSI-SoftwareEvaluationCriteria.pdf.

[11] G. F. Geeks. "Unit testing - software testing." (), [Online]. Available: https://www.geeksforgeeks.org/unit-testing-software-testing/.

[12] J. Charvat, *Project Management Methodologies: Selecting, Implementing and Supporting Methodologies and Processes for Projects*. John Wiley and Sons, 2003.

[13] R. M. et al. "Pyinstaller manual." (), [Online]. Available: https://pyinstaller.org.

A Testing UI Libraries

Below lists the code used to test certain graphics libraries that fit the criteria shown in Chapter 4.

A.1 Tkinter (Python)

```
01 from tkinter import *
02 from tkinter import ttk
03
04 class GridCanvas( Canvas ):
05     # draws a load of squares in a 2D grid; extends tk.Canvas functionality
06     NODE_SZ = 125
07     PADDING_SZ = 25
08
09     def __createNode( self, x, y ):
10         nid = self.create_rectangle( x, y,
11                                      x + self.NODE_SZ, y + self.NODE_SZ )
12
13         return nid
14
15     def __drawGrid( self, node_amount ):
16         for width in range( 0, node_amount ):
17             for height in range( 0, node_amount ):
18                 x = self.PADDING_SZ + ( self.NODE_SZ + self.PADDING_SZ )
19                     * width
20                 y = self.PADDING_SZ + ( self.NODE_SZ + self.PADDING_SZ )
21                     * height
22                 self.__createNode( x, y )
23
24
25     def __init__( self, parent, node_amount ):
26         super().__init__( parent, width = 500, height = 500,
27                           background = '#fff' )
28         self.__drawGrid( node_amount );
29
30
31 class GUIWindow( Tk ):
32
33     # example callback func, modifies canvas to be blue when called
34     def callback( self ):
35         print( "Hello!" )
36         self._canvas.configure( background = "blue" )
37
38     def __init__( self, *args, **kwargs ):
39         super().__init__( *args, **kwargs )
40
41         # create frame to hold canvas and sidebar
42         self._frame = ttk.Frame( self, padding = 6 ).grid( row = 0,
43                                                            column = 0 )
```

```
44          self.columnconfigure( 0, weight = 1 )
45          self.rowconfigure( 0, weight = 1 )
46
47          # create canvas
48          self._canvas = GridCanvas( self._frame, 3 )
49          self._canvas.grid( row = 0, column = 0 )
50
51          # create sidebar with button inside
52          self._sidebar = ttk.Labelframe( self._frame, text="Options",
53                                          padding = 5 )
54          self._sidebar.grid( row = 0, column = 1, sticky = ( N, S ) )
55
56          ttk.Button( self._sidebar, text = "Test!",
57                      command = self.callback ).grid( row = 0, column = 0,
58                                                      sticky = N )
59
60  gui = GUIWindow();
61  gui.mainloop();
```

A.2 Kivy with Python

```
01 from kivy.app import App
02 from kivy.uix.gridlayout import GridLayout
03 from kivy.uix.widget import Widget
04 from kivy.uix.button import Button
05 from kivy.graphics import Color, Rectangle
06
07 class Canvas( Widget ):
08     NODE_SZ = 125
09     PADDING_SZ = 25
10
11     def __createNode( self, x, y, highlighted=False ):
12         with self.canvas:
13             Color( 1, 0, 0 )
14             if highlighted:
15                 Color ( 0, 0, 1 )
16             Rectangle( pos = ( x, y ), size = ( self.NODE_SZ,
17                                                 self.PADDING_SZ ) )
18
19     def __drawGrid( self, node_amount ):
20         for width in range( 0, node_amount ):
21             for height in range( 0, node_amount ):
22                 x = self.PADDING_SZ + ( self.NODE_SZ + self.PADDING_SZ )
23                     * width
24                 y = self.PADDING_SZ + ( self.NODE_SZ + self.PADDING_SZ )
25                     * height
26                 self.__createNode( x, y )
27
28     def __init__( self, *args, **kwargs ):
29         super().__init__( *args, **kwargs );
30
31         self.__drawGrid( 3 );
32
33     def updateRect( self ):
34         self.__createNode( self.PADDING_SZ, self.PADDING_SZ, True )
35
37 class UIContainer( GridLayout ):
38
39     def btnCallback( self, instance ):
40         # on callback, tell canvas to do func
41         self.canv.updateRect();
42
43     def __init__( self, *args, **kwargs ):
44         super().__init__( *args, **kwargs )
45
46         # 2 columns on layout
47         self.cols = 2
48
49         # add canvas on col 1
50         self.canv = Canvas( width = 500, height = 500 )
51         self.add_widget( self.canv )
```

```
52
53          # add button on col 2 that calls btnCallback() on pressed
54          btn = Button( text = "Click me!", width = 250 )
55          btn.bind (on_press = self.btnCallback )
56          self.add_widget( btn )
57
58 class TestApp( App ):
59
60      def build( self ):
61          return UIContainer()
62
63 TestApp().run();
```

A.3 JavaFX with Java

```
01 import javafx.application.Application;
02 import javafx.scene.Scene;
03 import javafx.scene.layout.HBox;
04 import javafx.stage.Stage;
05 import javafx.scene.control.Button;
06 import javafx.event.ActionEvent;
07 import javafx.event.EventHandler;
08 import javafx.scene.canvas.*;
09 import javafx.scene.paint.Color;
10
11 public class Main extends Application {
12     private static final int NODE_SZ = 125;
13     private static final int PADDING_SZ = 25;
14
15     private GraphicsContext gc = null;      //how to draw on canvas
16
17     private void drawNode( int x, int y )
18     {
19         this.gc.strokeRect( x, y, NODE_SZ, NODE_SZ );
20     }
21
22     private void drawGrid( int node_amount )
23     {
24         for ( int w = 0; w < node_amount; w++ )
25         {
26             for ( int h = 0; h < node_amount; h++ )
27             {
28                 int x = PADDING_SZ + ( NODE_SZ + PADDING_SZ ) * w;
29                 int y = PADDING_SZ + ( NODE_SZ + PADDING_SZ ) * h;
30
31                 drawNode( x, y );
32             }
33         }
34     }
35
36     @Override
37     public void start(Stage stage) {
38         //create hbox to store elements in
39         HBox hbox = new HBox();
40
41         //create canvas to draw on
42         Canvas canvas  = new Canvas( 500, 500 );
43         this.gc = canvas.getGraphicsContext2D();
44
45         // draw grid
46         drawGrid( 3 );
47
48         //create button that redraws over grid
49         EventHandler<ActionEvent> ev = new EventHandler<ActionEvent> () {
50             @Override public void handle( ActionEvent e )
```

```
51              {
52                  //since cant edit pixels, must draw over grid
53                  gc.setStroke( Color.BLUE );
54                  gc.strokeRect( PADDING_SZ, PADDING_SZ, NODE_SZ, NODE_SZ );
55              }
56          };
57          Button button = new Button("Click me!");
58          button.setOnAction( ev );
59
60          // add elements next to each other horizontally
61          hbox.getChildren().add( canvas );
62          hbox.getChildren().add( button );
63
64          //create scene to store hbox and display
65          Scene sc = new Scene( hbox, 750, 500 );
66          stage.setScene( sc );
67          stage.show();
68      }
69
70      public static void main(String[] args) {
71          launch();
72      }
73 }
```