Visualizing Task-Mapping in Many-Core Systems

Initial Report

ELE00080H

Jacqueline Walker

Supervisors

Dr. Gianluca Tempesti, Dr. Yuriy Zacharov

Vanbrugh College

University of York

March 6, 2024

Contents

Abstract

The project aims to build a graphical program that can create an initial task mapping for a given many-core system. A many-core system is a computer architecture that consists of hundreds or thousands of simple cores connected into a network. These cores can then run simple 'tasks' that only depend on results from previous tasks. The process of task mapping is the assignment of tasks to given cores, which the user is responsible for.

This report aims to detail the research and planning undertook before the development of the project. It describes the background reading necessary for understanding many-core systems and task mapping, the breakdown of the project into the aims of the project and features the program should have, and preliminary planning needed before starting development.

Statement of Ethics

After consulting the project and the Royal Academy of Engineering's guide to Ethics[1], there are no ethical considerations to be made.

1 Introduction

As multi-core systems became more commonplace and power consumption increase, it became more apparent that a new architecture needed to be designed[2]. Bus-based architectures suffer from a number of issues, such as a lack of scalability and limited bandwidth[3], which cause increasingly complex designs and limited performance. The Network-on-Chip (NoC) architecture is an alternative to previous bus-based designs. Instead of directly connecting each component via a bus, the components are instead connected to a networking interface (NI) and switch that handles communication between components[2]. Communication is then done in a similar way to off-chip networks, either by reserving a path to send the data or splitting the data into 'packets' to be routed to the destination[4]. Using a network architecture allows the designers to leverage pre-existing understanding and research into off-chip networks, meaning that a designer would have a better base-knowledge of how the architecture works. Like off-chip networks, bandwidth scales with network size unlike with bus-based networks[2].

Many-core systems consist of hundreds or thousands of relatively simple processor cores. These cores can then run a large number of simple tasks (referred to as processes) in parallel[5]. Many-core systems would be hard to implement without the use of the NoC architecture, mainly due to the high amount of cores. The layout (or topology) of a many-core system can vary depending on the use case. One such topology is the 2D mesh, which consists of a $M \times N$ grid. Using a grid layout allows every node to connect to 4 other nodes, except at the edges[4]. A visualization of the 2D mesh topology is given in Figure 1.1. Other topologies exist to solve different parameters, such as number of routers needed, traffic congestion around the network, etc.

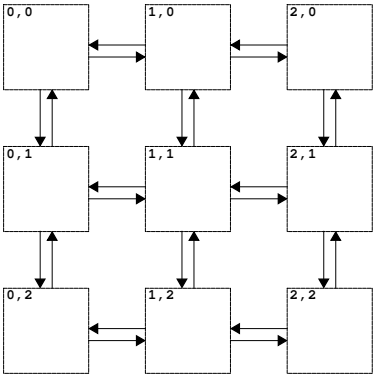


Figure 1.1: A visualization of a 3x3 2D mesh topology for a many-core system.

In order to utilize many-core systems effectively, applications needing to be run should be split down into processes. These processes depend only on the result from its previous process(es), and have a given computational load. Communication between processes is also given a communication load of how much data needs to be sent[4]. These processes are best visualized using an Application Process Graph (APG)[6], also referred to as a task-graph. An example of one is given in 1.2.

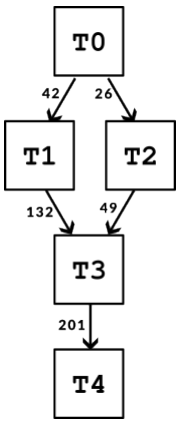


Figure 1.2: A visualization of an example Application Process Graph.

From the APG, we then need to figure out how to place each task onto a core. This process is called task mapping. It is usually not feasible to iterate through all mappings and compare parameters due to the number of mappings increasing exponentially with mesh size[6]. Ideally, the task mapping must aim to keep the total communication load as low as possible[4]. In real systems, task mapping is a continuous process from the start and end of a system running.

1.1 Aims of Projects

The aim of the project is to create a graphical user interface (GUI) application that allows the user to create an initial task mapping[1] of a many-core system. The initial mapping should be done manually by allowing the user to map tasks onto specific cores. The GUI should allow the user to:

A1. Import a task graph

A2. Create a 2D mesh between 3×3 and 6×6 inclusive

A3. Map given tasks onto specific cores

A4. Show the task mapping and communication load graphically

A5. Export the task mapping

This project should also aim to be useful as a visual aid for reports or for demonstrating the concept of task mapping to someone studying many-core systems or NoCs. In order to achieve this aim, the GUI should be visually appealing and intuitive to new users. Along with this, the project should keep code clean and readable whenever possible. The objectives are then as follows:

O1. Design a visually-appealing graphical user interface that is appropriate for the project.

O2. Research into various graph file formats for importing/exporting.

O3. Research various methods for visualizing task graphs and many-core systems.

O4. Ensure that the final software adheres to a well-documented style guide and is comprehensively tested.

[1] i.e. the task mapping when the many-core system is first started.

2 Background

2.1 Designing UI

A good UI design is both visually appealing and one that doesn't hinder productivity. In order to understand how to design a good UI we must first know the fundamentals of how users interact with UI's. These principles are collated from multiple sources, namely [7]–[9], as many books list differing fundamentals. Please note that these may not be applicable in every context.

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.

Applying these principles will lead to an understanding in how to create a well-structured UI. For the principle of *Instant Notification*, it can be seen that each action a user does, visual feedback should be given, such as a notification or progress bar should be shown. If the user doesn't see any change, they may re-do the operation multiple times in frustration. An example of this is shown in Figure 2.1.

The principle of *Skimming* is one of the more influential principles and therefore has the most design considerations due to this. In order to create a UI that is easily skimmable, objects that have similar functions should be visually grouped or close to each other, so the user can instinctively tell they are related[8]. If objects are poorly-spaced, the user will need to spend more time to parse the UI, slowing down their productivity. Another consideration is how to structure information to help users pick up key information. By keeping text brief and by emboldening key information, users need less time to pick up the meaning of the text. Other ways of giving focus to important information are by giving a contrasting background colour or by making the text visually bigger[7]. An example of this is given in Figure 2.4.

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[8]. 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 on a menu bar to conform to the muscle-memory of previous applications[7]. An example of this is given on Figure 2.2.

When choosing colours for an application, care must be taken to ensure 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[8]. Along with this, care must be taken to choose colours that are visible for people with colour-blindness as certain pairings are hard to distinguish[9]. An example of this is given in Figure 2.3. Certain colours also can be 'reserved' for certain situations, i.e. yellow for warnings, or red for errors. These colours are usually chosen as they have high saturation and to capture the eye's focus.

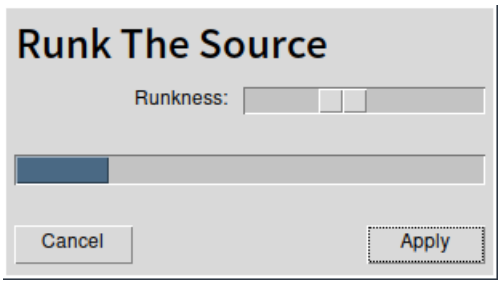


Figure 2.1: An example of a program giving instant notification to the user via a ongoing progress bar.

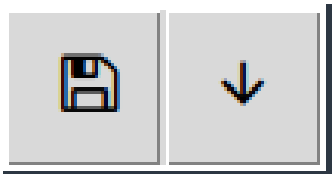


Figure 2.2: An example of the Familiarity principle. At first glance, which button looks more like a 'save' button?

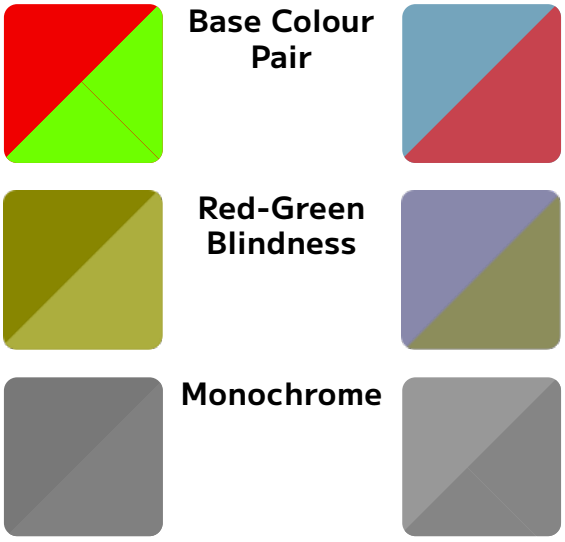


Figure 2.3: An example of two colour pairs. One pair has poor contrast and saturation, along with being hard to distinguish for people with colour-blindness. The other pair changes both hue and saturation to fix these issues.

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

Core count: 1024
Mapped cores: 218
Mesh size: 32x32

Figure 2.4: A comparison of structured and unstructured text.

2.2 Routing Algorithms

A routing algorithm is used to determine the path that communication will take from node to node[5]. Communication is split into 'packets' which can then be transported along each node, referred to as a 'hop' on the path until reaching its destination. Routing algorithms may give multiple valid paths. Routing may be 'deterministic', where the path chosen between nodes is fixed, or 'adaptive', where the path changes depending on network conditions[2].

Since our project focuses on the initial task mapping, all routing algorithms looked at will be deterministic.

One routing algorithm is X-Y routing. This algorithm works via taking the horizontal direction until it matches the destination's first, then the vertical direction until it matches the destination's last[2]. Other routing algorithms exist that prioritise moving a certain direction either first or last, such as North-last or West-first. Example paths from a selection of routing algorithms are given in Figure 2.5.

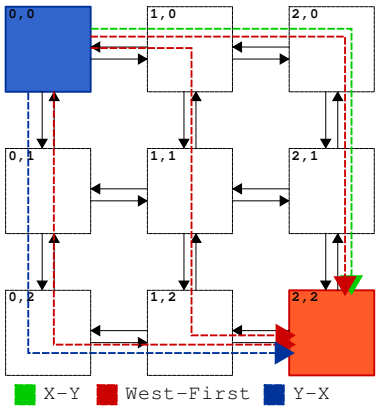


Figure 2.5: An example of different paths determined by different routing algorithms.

2.3 Task Graph Formats

DOT[10] is a language used for specifying graphs, nodes, and edges connecting nodes. It allows the user to add attributes to be displayed with or to modify a node or edge, allowing versatility in how graphs may be displayed. Many graph generation programs already allow exporting to DOT natively, and many libraries exist to allow interfacing with DOT files, such as pydot[11].

```
digraph example {
    rankdir="LR";
    a -> b;
    b -> c;
    b -> d;
    c -> d;
    d -> e;
    a -> e;
}
```

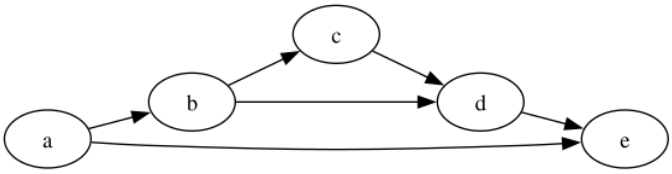


Figure 2.6: A simple directed graph written in DOT and a resultant graph rendered by xdot[12].

Other task graph file formats exist, such as the STG file format[13] from Waseda University. This file format is designed specifically for task graphs for use with many-core systems of varying sizes. Each file is split into two parts: a task graph part, and an information part. The task graph part stores each task, its processing time, its predecessor, and its communication cost. The information part is denoted with a '#' symbol and stores other information about the task graph. Many formats are based off an existing file format, such as GEXF and GraphML using XML, or GDF being similar to database tables. The file formats may be useful to revisit later in development.

3 Approach

3.1 Feature Breakdown

As seen in the Introduction, our project needs to allow the user to create an initial mapping of a many-core system. Specific features need to be given in order to meet the aims specified. Since the project may face many setbacks or may be easier than expected, features have been split into three categories: minimal, reasonable and extra. Minimal features are those core to the program's aims and objectives, reasonable features are those that should be able to be completed in the time given and will give the most benefit to the program, and extra features are those that should be done if time permits but are not necessary. These features are non-exhaustive but show what the project should be capable of at the end of development.

Minimal features:

MF1. Allow the user to import a task graph in DOT file format

MF2. Allow the user to create a 2D mesh of size 3×3 to 6×6 inclusively

MF3. Draw the task graph on part of the UI

MF4. Draw the 2D mesh on part of the UI

- (a) Each node should be annotated with any assigned task, the co-ordinate in the mesh and the computational load

MF5. Show an overlay of the task graph onto the 2D mesh for showing communication load

MF6. Allow the user to right-click onto a node and select a task to assign

MF7. Export the mapping as a CSV file

Reasonable Features:

RF1. Allow the user to pan and zoom around the 2D mesh and task graph visualization

RF2. Allow the user to export in a custom XML format[1]

RF3. Allow the user to edit the 2D mesh in an edit menu

- (a) Allow the user to modify values from the task graph on the node
- (b) Allow the user to set the 'status' of a node
- (c) Allow the user to switch whether the node is zero-indexed or one-indexed
- (d) Allow the user to change the origin of the node from top-left to bottom-left

RF4. Allow the user to export the mapping as an image

RF5. Allow the user the select from two routing algorithms to display which path the communication between nodes will take

[1]The custom XML format is being developed mainly for a PhD project in the same field of study, but interoperability between programs would be a good feature.

Extra Features:

EF1. Allow the user to import different file formats of task graphs

EF2. Include accessibility options for the program

(a) High-constrast theming for the GUI

(b) Modification of the font size

EF3. Allow user to specify channel bandwidth and check if communication load exceeds.

EF4. Allow the user to add arbitrary text to a node

EF5. Allow the user to export and import the project settings for a given task graph and 2D mesh

EF6. Display border nodes for I/O that tasks may be assigned to

3.2 Challenges Facing The Project

Due to this project being a more-complex program being undertaken in the course of three months, there are potential challenges that may arise throughout the project. Below lists some challenges that may occur and how to mitigate these issues via planning:

Challenge: The project ends up too complex to complete fully.

Mitigation: The inclusion of a minimum set of features allows for the scaling down of the project to a simple-yet-sufficient program. The project will also use an Agile-based project planning methodology to account for variations to the proposed timeline.

Challenge: The software may end up having too many bugs to fix at the end of development.

Mitigation: By utilising an Agile-based project planning methodology that emphasizes constant testing and prototyping, bugs should be caught throughout the development process, rather than at the end.

Challenge: an unexpected loss of data occurs.

Mitigation: By utilising a versioning system, such as Git, the loss of significant amount of data is mitigated.

3.3 Project Planning Methodology

As mentioned in the previous section, the project planning methodology to be used is a modified version of Extreme Programming[1], an Agile-based methodology. It has been modified to work for an individual, mainly by removing the requirement for pair programming and the requirement for continuous feature generation. Instead, more preliminary planning will be undertook and a strict cap on the end of development has been set.

3.4 Structure Breakdown

As described in the Project Preparation Report, the programming language will be Python[14] using Tkinter[15] for the graphics. These were chosen due to the large amount of resources available, their maturity and their ease of coding in comparison with other languages and libraries. Because of this, a large chunk of the final program will be event-driven due to its nature as a graphical program and therefore certain functions may need to handle being called before the object they refer to has been initialized. For example, a user may try to export the task mapping before any task graph has been imported.

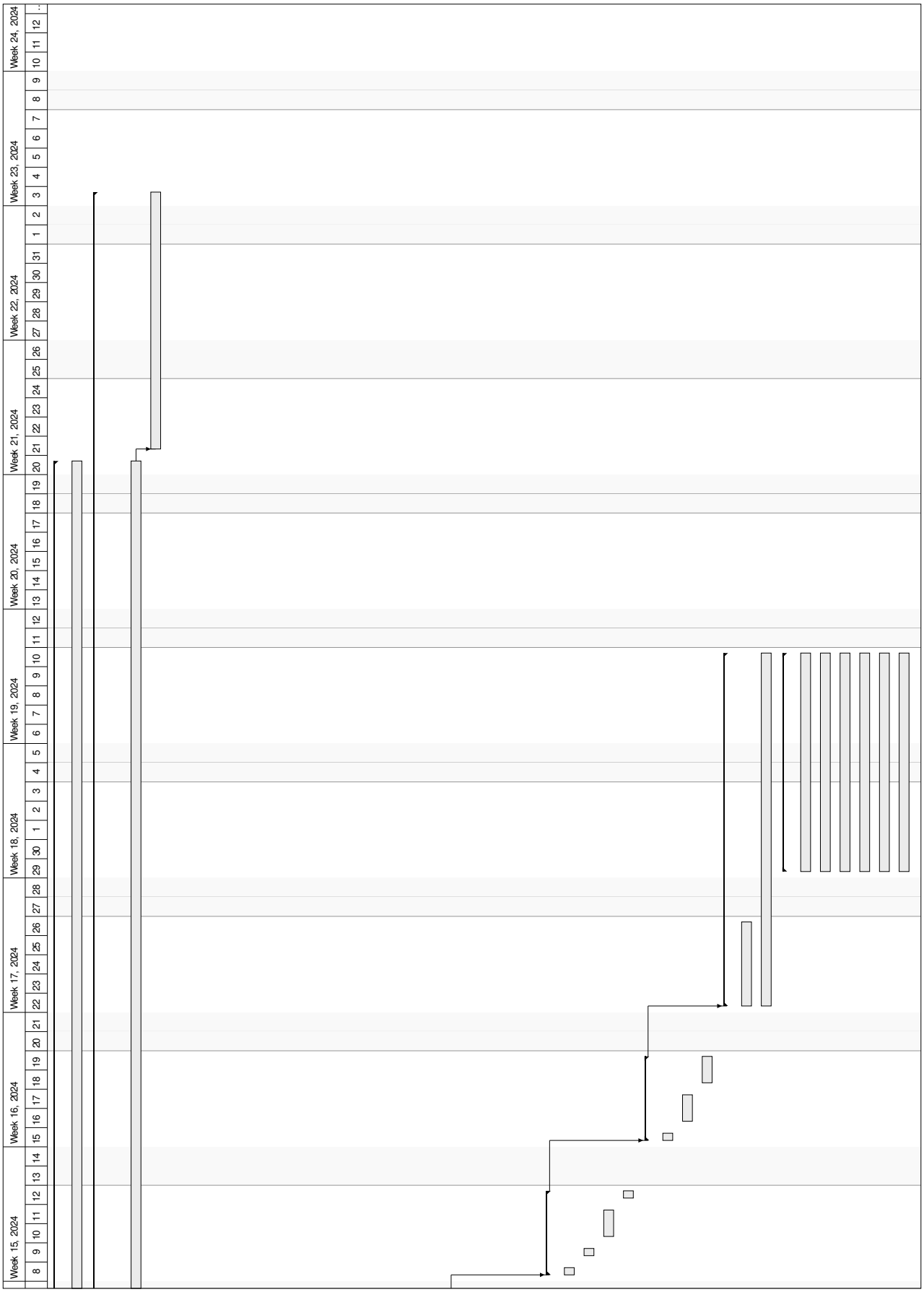
The program should be split into three main components: the GUI, the task graph, and the 2D mesh. The GUI will be responsible for the visual style of the program, the main loop and set up/tear down, and calling other components on an event happening. The task graph will be responsible for the import of task graphs and the visualization of the task graph. The 2D mesh will be responsible for the visualization of the 2D mesh, the logic necessary for assigning tasks and computing the communication cost, and the exporting of the 2D mesh to a given file format. Ideally, each of the features needed should be able to fit into a given main component.

3.5 Project Timetable

The GANTT chart for this project is given in Figure 3.1. As can be seen from the GANTT chart, the project will consist of week long 'sprints', where a feature is developed from start to finish, including testing and final planning for that feature. While each sub-task in a sprint has been given a day on which it should be completed by, I deem this to be flexible as long as the sub-task is completed within the sprint.

The chart has been designed to meet certain sets of features by specific dates. For example, there is a demonstration day for the 29th of April, therefore all the reasonable set of features have been planned to be finished by that date. Time has also been allotted near the end of the deadline so that the report can be fully written in time.

[1]This was chosen in the Project Preparation Report after an analysis of multiple project planning methodologies. Extreme Programming can be found at [16].


Week 15, 2024
Week 16, 2024
Week 17, 2024
Week 18, 2024
Week 19, 2024
Week 20, 2024
Week 21, 2024
Week 22, 2024
Week 23, 2024
Week 24, 2024

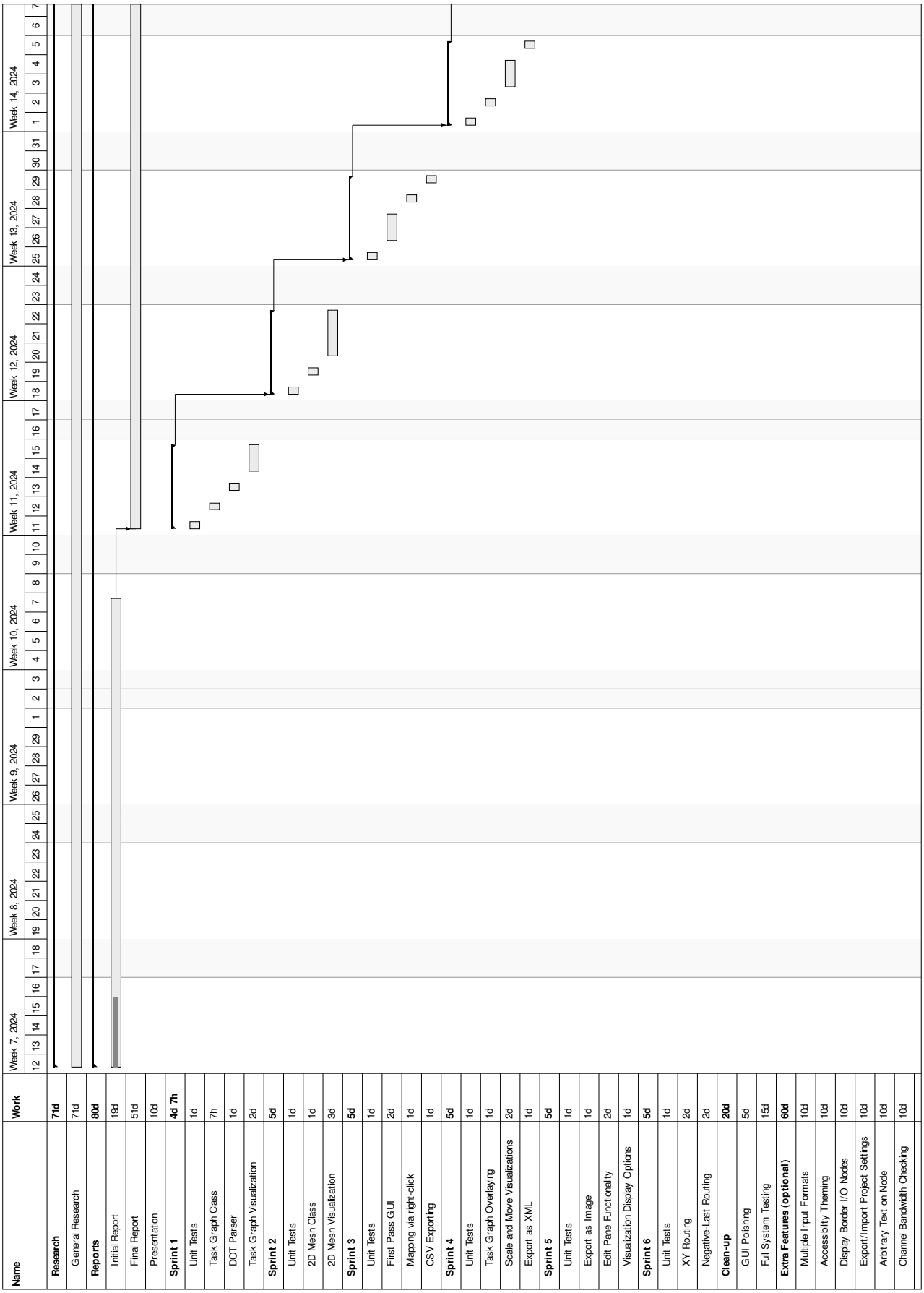


Figure 3.1: The GANTT chart for the project. Generated from GNOME Project.

3.6 GUI Designs

Since the GUI may need multiple colours for graphics and visualisations, a small palette of complementary colours would be good to choose. A good palette found is Sweetie 16[17], a simple palette that covers the entire colour range. These colours could then also influence the styling of the program as a whole.

Figure 3.2: Sweetie 16 colour palette to be used to keep colours distinct in visualizations.

The UI should draw attention to the main focus of the program, the task-graph and the 2D mesh. Because of this, both graphs should stay central and large to the main view of the program. As described in RF3, there should be a menu for editing display options of the graphs and editing of nodes. For this, a side menu should be available for storing these options. The text given to the user should also stay as structured as possible to allow the user to parse key information as quick as possible. There should be as few options as possible to keep the UI as easy-to-parse as possible. A menu bar should be available for importing and exporting of task mappings.

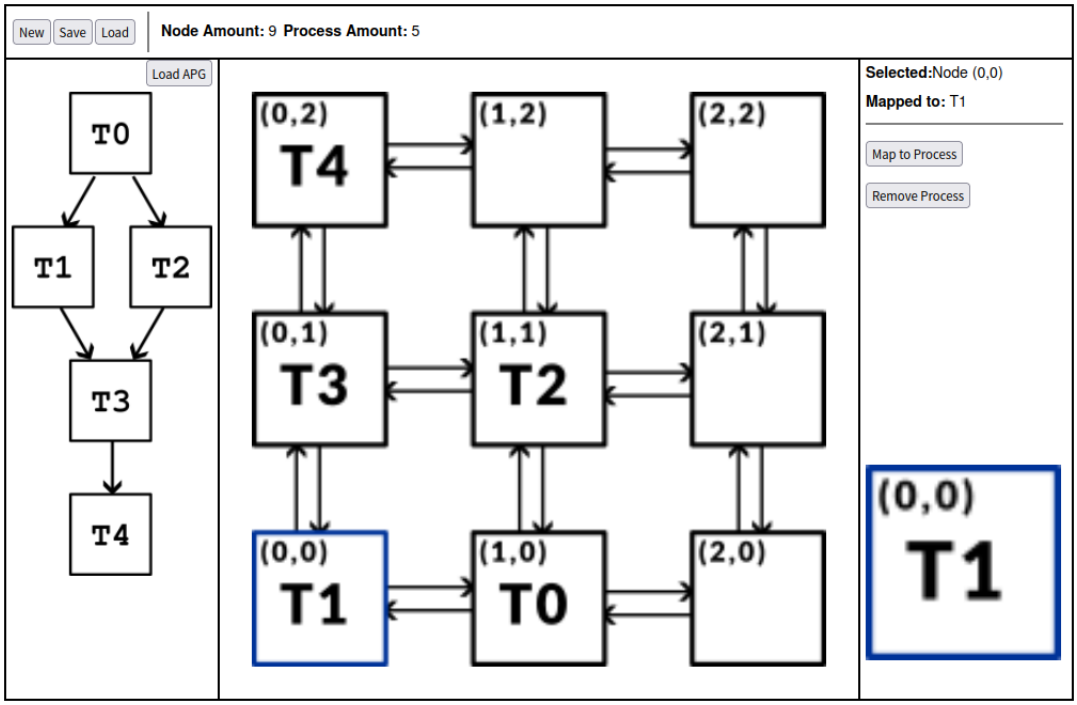


Figure 3.3: A potential layout for the application, keeping focus towards the 2D mesh and the APG. The APG has 5 processes and the 2D mesh is 3x3. The UI is split into 3 columns; one for the APG, one for the 2D mesh, and one for modifying the selected node.

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

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

[10] Graphviz. "Dot language." (), [Online]. Available: https://graphviz.org/doc/info/lang.html (visited on 03/06/2024).

[11] E. Carrera. "Pydot." (), [Online]. Available: https://pypi.org/project/pydot/ (visited on 03/06/2024).

[12] J. Fonseca. "Xdot.py." (), [Online]. Available: https://github.com/jrfonseca/xdot.py (visited on 03/06/2024).

[13] W. University. "Stg format." (), [Online]. Available: https://www.kasahara.cs.waseda.ac.jp/schedule/format_e.html (visited on 03/06/2024).

[14] P. S. Foundation. "Python." (), [Online]. Available: https://www.python.org/ (visited on 03/06/2024).

[15] P. S. Foundation. "Tkinter - python interface to tcl/tk." (), [Online]. Available: https://docs.python.org/3/library/tkinter.html (visited on 03/06/2024).

[16] D. Wells. "Extreme programming: A gentle introduction." (), [Online]. Available: https://www.extremeprogramming.org (visited on 03/01/2024).

[17] J. Em. "Sweetie 16 palette." (), [Online]. Available: https://lospec.com/palette-list/sweetie-16 (visited on 03/06/2024).