VISUALISING TASK-MAPPING IN MANY-CORE SYSTEMS

BEng Project Final Report

23/24

ELE00080H

UNIVERSITY of York

School of Physics, Engineering and Technology

Jacqueline Walker

Vanbrugh College

Supervisors

Dr. Gianluca Tempesti, Dr. Yuriy Zacharov

Document compiled May 24, 2024

Abstract

The project aims to create a program that allows a user to create an initial task mapping for a given many-core system via a graphical user interface (GUI). A many-core system is a computer system that consists of thousands of simple cores capable of running many tasks in parallel. An initial task mapping is the process of assigning tasks to given cores in the system, preferably to keep communication between cores as low as possible.
This report details the entire project from the initial research and clarification of of the aim of the project to the final product, a Python-based program capable of meeting the project aims. The report follows the usual design process as it happens, where background into many-core systems and GUI design principles are researched, to initial development planning and feature clarification. Then follows a detailed breakdown of the implementation of each part of the program, grouped roughly into each major section of the program. Finally, an evaluation of the final product is done on both how it fits the features given and on how the final product is.

Acknowledgements

I would like to express my deep gratitudes to Dr. Gianluca Tempesti for his continual support throughout this project, along with my thanks to Dr. Yuriy Zacharov for feedback on my previous work.

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.

Contents

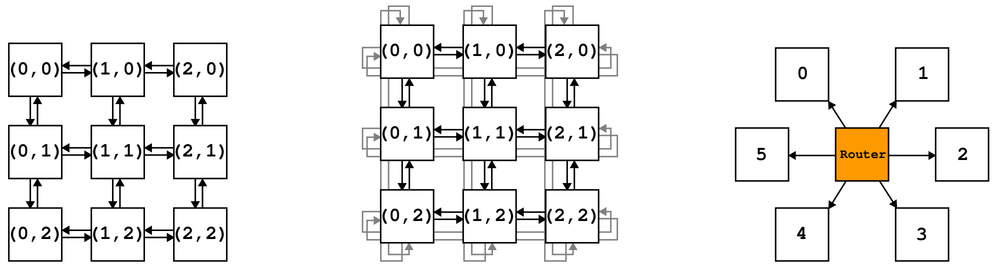
1 | Introduction

1.1 | Preliminary Background

As multi-core processors have become more commonplace and increased in core counts, it has become clear that there is a limit to its performance. A single core can only do so much work before it consumes too much power to sufficiently cool[2]. A new paradigm was seen as necessary to keep performance increasing at a steady rate. Previous architectures relied on a bus, which all components in the processor connect to. However, when more components are connected, the bandwidth of the bus cannot increase, and power draw becomes greater[3].

The Network-on-Chip (NoC) architecture is an alternative to bus-based architectures. Instead of directly connecting each component to a bus, they are instead treated as nodes in a network, where each node has a Networking Interface (NI) that handles communication[4]. This communication can take a variety of routes between components. Bandwidth then can scale with network size. Using an architecture based on networks allows the designers to leverage pre-existing research and familiarity of off-chip networking[4].

A many-core (or manycore) processor is a subcategory of multi-core processors that consist of hundreds or thousands of cores, where each core is relatively simple. This allows an engineer to run a large number of tasks in parallel[5], which is useful for programs that require a high amount of computation. Many-core systems would be hard to implement without the use of the NoC architecture's high degree of parallelism and scalablility[2]. They can be designed in a variety of layouts (known as topologies) depending on use case. One such example is the 2D mesh, where nodes are connected in a $M \times N$ grid[4]. Every node in the mesh has four neighbours, except



(a) *A 2D mesh of size* 3×3. **(b)** *A 2D torus of size* 3×3. **(c)** *A star topology.*

Figure 1.1: *A selection of many-core topologies.*

for nodes at the edge. This topology may be extended into a 2D torus via connecting opposite edges together, ensuring every node has four neighbours. Other topologies exist, based on pre-existing off-chip network topologies, such as star or tree[4]. Example topology layouts are given in Figure 1.1.

In order to run a program on a many-core system, the program must be broken down into simple tasks that can run on a core. An easy way to visualise this is by utilizing an Application Process Graph (APG)[6]. An APG is a graph of all the tasks in a program, their communication between tasks, and the amount of computation needed for a task. It is visualized as a directed graph with no cycles. Each edge on the graph is labelled with the communication between the two tasks[7]. Each node on the graph is given a task ID and the computation load. An example of an APG is given in Figure 1.2.

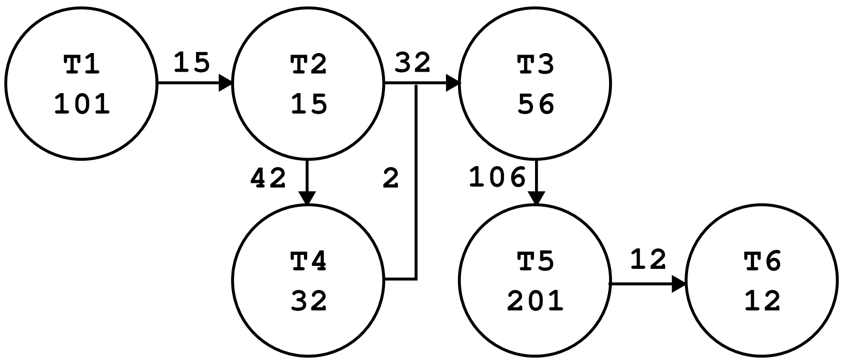


Figure 1.2: *An example APG with labelled communication and computational loads.*

Task mapping is the process of allocating given tasks to given cores. The mapping should aim to keep the total communication cost around the many-core system as low as possible[7]. It is not feasible to brute force through all possible mappings as the number of potential mappings increases exponentially with mesh size[6]. For real systems, task mapping is done initially and may be continuously re-mapped depending on network and core conditions.

1.2 | Aims of Project

The aims of the project is to create an application that allows the user to specify an initial task mapping for a given many-core system. The application should be able to do this via some form of graphical user interface. This project also aims to have use for demonstrating the concept of task-mapping to someone studying many-core systems, along with some use for generating visual aids for figures in reports.

From this, we can derive a set of general aims of the application:

- Allow the user to import a task graph.
- Allow the user to specify a 2D mesh (with sizes small enough to be able to assign to manually)[1].

- Allow the user to assign and deassign a specific task from the task graph to a given core in the mesh.
- Allow the user to see how communication between cores will happen based on the specified mapping.
- Export the task mapping and parameters of the 2D mesh.

[1]If the size is too large, it will take too long for a user to assign to and would look too complicated as a diagram.

2 | NoC Background

2.1 | Background

In a real system, each node in the network may be a CPU core, a memory element or a block of intellectual property (i.e. an MP4 decoder)[4]. Because of this, certain components may consume different amount of power, even when idle. Along with this, the reduction in size has caused the power density to increase to the point where some chips are not able to be cooled if every core was active at the same time. To battle this, certain components would be turned off to keep heat at an acceptable level[6], known as ’dark silicon’. Along with this, certain cores need to be kept free in case of a sudden failure on a running core, so that a task may be migrated, known as ’fault tolerance’[6].

Communication between nodes is done via bi-directional channels, each with a directional link in opposite directions[4]. Each given link has a maximum amount of communication that may go through it, known as the ’bandwidth’. Routing algorithms dictate the path that communication will take from a given ’source’ node to a given ’destination’ node. A good routing algorithm should distribute paths around the mesh to avoid filling up the bandwidth on one given link[5]. For specific routes, there may be more than one valid path between nodes. For a ’deterministic’ algorithm, the route chosen will be the same every time, whereas for an ’adaptive’ algorithm, the route may change dependant on current traffic and other network conditions[4]. Care must be taken to avoid cycles being generated in the paths of multiple routes. This is called ’deadlock’ and, in a real system, can be caused when two nodes are stuck waiting for the other node to communicate[5].

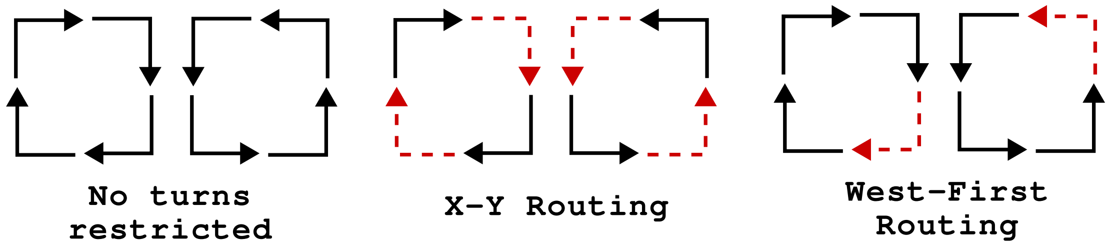


Figure 2.1: *Examples of turn models for certain routing algorithms.*

One of the simplest routing algorithms is X-Y routing, where the route taken is the difference in width between the two nodes, then the difference in y between the two nodes[4]. Other deadlock-free algorithms exist, based on the idea of restricting turning in at-least two directions to prevent cycles from forming[8]. One such turn-restriction is the west-first algorithm, that restricts turning from a direction that is not west to west. Examples of the turn restrictions are given in Figure 2.1.

2.2 | Objectives

From this further background on many-core systems, it can be seen that certain features should be added to our project:

- Ability to set the state of a given core to a spare or 'dark' state.
- Noting communication done between nodes in terms of links.
- Ability to set a maximum bandwidth for all links in a system
- Ability to include deadlock-free routing algorithms to calculate communication load between nodes.

3 | GUI Background

3.1 | Background

A good user interface (UI) should be designed to make things as intuitive as possible for the reason you use the software[9]. For example, if a user loads up `google.com`, they are intending to search something. Therefore the ability to search something should be prioritised in the UI. As seen in Figure 3.1, the search box is prominent, big and central with white-space surrounding the search box to separate it from other elements[10].

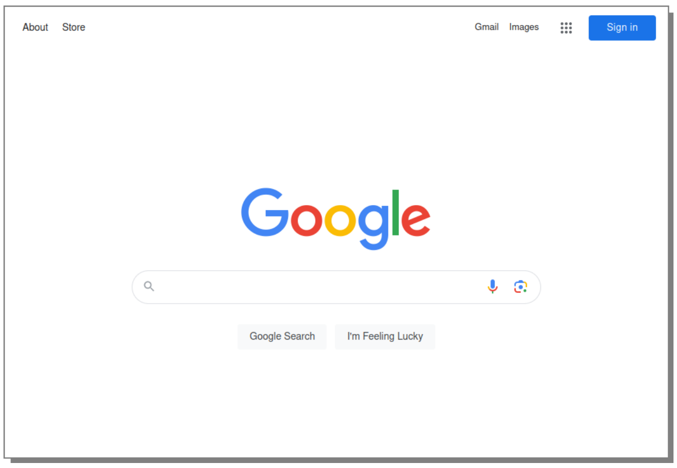


Figure 3.1: *The homepage for* `google.com`. *Credit: Google LLC.* [1]

From this, it can be seen that certain design patterns can help to prioritise important functions or sections of the UI. Users tend to interact with UIs in the same way no mateer the program. These interactions can then be used to derive a few fundamental design principles.

Instant Gratification The user should be given some form of immediate results from taking an action[9].

Safe Exploration The user should be able to easily undo any action they take without consequence[9].

[1]This figure falls under fair use as it is included for non-commercial research purposes.

Skim Reading The user does not tend to take in the UI all at once, rather looks at the general structure and grouping of the UI[11].

Familiarity The user tends to interact with the UI in ways they have been able to with previous programs or systems[10].

Consistency The user remembers how to do an action based on where it is positioned, not what it was called. Therefore actions should stay in the same position for different contexts.

These principles should dictate how the UI is designed. For example, the principle of *Skim Reading* dictates that parts of the UI that do similar actions should be grouped close together, as the human eyes instinctively assume they are related. Along with this, text given to the user should be well-structured and brief to allow for the user to pick up key information easier[11]. Another way to give focus to text is to put a contrasting background behind it or to make it bigger than its surrounding elements[9]. An example of this is given in Figure 3.2.

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 3.2: *An example of unstructured vs. structured text passages.*

Another key principle is the principle of *Familiarity*. It dictates that if we have a similar function to a previous program, our layout should mimic that program. For example, most programs put their save/load buttons in the top-left corner of their program so the user instinctively expects them to be there[9]. The principle of *Instant Notification* also dictates that we should give clear visual aid for when an action has started, even if the action takes a long time to complete. This may be done via a progress bar or via a pop-up box. An example of this is given in Figure 3.3.

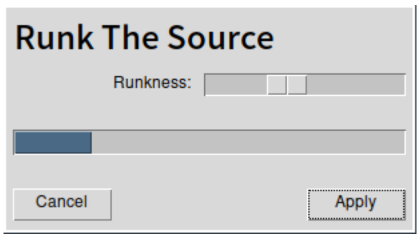


Figure 3.3: *An example showing a progress bar for an ongoing action.*

Care must also be taken to design the interface in a way that isn't hard for someone who is visually impaired to use. For example, certain colour combinations (those with little difference in saturation and contrast) are harder for someone with colour-blindness to separate. Both hue, contrast and saturation should ideally be modified to ensure a colour combination can be used[11].

4 | Development

4.1 | Project Constraints

Certain features of a real many-core system are out-of-score for our project. Along with this, our project is at a Bachelor-level with a fixed time constraint, so certain topics are considered too complex to research and implement. Below lists some of the constraints of our project:

- Task graphs with cycles within are not considered.
- Many-core topologies other than 2D mesh are not considered.
- Only the initial mapping of the system will be covered.
- Routing is simplified to only consider the communication on a given link and assume that all links are instantaneous and are error-free.

4.2 | Initial Project Plan

This section describes all the potential features of the project based on the background of the project from previous chapters. Each feature has been categorised into one of three categories: minimal (for those core to the project), reasonable (for those that should give a well-rounded final product within the time allotted), and extra (for those that would be good if time permits). The features given are not all of the features that may be added, but reflect what the project should be capable of at the end of development.

Minimal[1]**:**

1. Allow the user to import a task graph in some file format
2. Allow the user to create a 2D mesh of size 3×3 to 6×6 inclusively
3. Draw the task graph on part of the UI
4. 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

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

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

7. Export the mapping as a CSV file

Reasonable:

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

9. Allow the user to export in a custom XML format[2]

10. 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

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

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

Extra:

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

14. Include accessibility options for the program

    (a) High-constrast theming for the GUI

    (b) Modification of the font size

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

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

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

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

[1]The following sets of features have been included from the Initial Report[12].
[2]The custom XML format is being developed mainly for a MEng and PhD project with similar aims.

4.3 | Project Management

As specified in the Initial Report[12], this project management methodology to be used was a modified version of Extreme Programming, an Agile-based management framework[13]. It is based on the idea of fast prototyping and constant refactoring to improve code quality and add extra features as they become necessary. It has been modified to work for an individual, as the original specification required pair programming and no strict end-date for development. Extreme Programming also specifies that unit tests must be the first section done before development of a feature.

Feature	**W1**	**W2**	**W3**	**W4**	**W5**	**W6**	**W7**	**W8**	**W9**
Task Graph Data Structure	■								
Task Graph Visualisation	■								
Mesh Data Structure		■							
Mesh Visualisation		■							
GUI and Mesh Mapping			■						
CSV Exporting			■						
XML Exporting				■					
Overlay Task Graph on Mesh				■					
Image Exporting					■				
Edit Pane					■				
Routing Algorithms						■			
Clean-Up							■	■	
Extra Feature Implementation								■	■

Figure 4.1: *Rough timeline for project*

From this methodology, the idea was to create a new fully-tested prototype with a new feature or set of features roughly by the end of each week, to align with the weekly reports we are required to send out. A rough timeline is given in Figure 4.1.

4.4 | Project Risks

Risk: Unexpected data loss occurs.

Mitigation: Changes to the program are committed via git[14], and synced multiple times per day to GitHub. Along with this, milestone snapshots are stored on my personal computer.

Risk: The project ends up too complex to complete in time.

Mitigation: Multiple sets of features have been given, so if the project cannot be completed in full, partial completion is available.

Risk: There is too much/too little work to do within a week

Mitigation: By using a more flexible project management scheme, such as an Agile-based one with flexible deadlines, this is able to be mitigated by reducing/increasing the work in one week, or by planning the timeline effectively with catch-up time.

4.5 | Criteria For Success

Below lists the criteria for success. These criteria are chosen to focus more on the quality of the program as a whole, instead of on whether a feature has been implemented or not. Each criteria has been written in a way to keep answers as objective as possible.

- Does the user interface conform to some clear structure?
- Are the visualisations of the task graph and mesh topology distinct from the rest of the user interface?
- Does the user interface give clear indication of ongoing or completed actions?
- Is the user interface able to be used by someone with visual impairments?
- Is the program able to be easily packaged for distribution?
- Does the program need no extra programs to run itself?
- Is documentation given to the user about how to use the program?
- Is documentation available for developers to extend the program?
- Is the codebase structured and documented in a clear and well-defined format?
- Is the codebase able to be fully automatically tested?

4.6 | Pre-existing Solutions

As far as can be seen, there is no pre-existing software that can be used to fit the features for this project. Most software designed in this field focus on accurate simulation of a many-core system itself. One such simulator is HORNET[15], which focuses on accurate network and core simulation. Other simulators exist which do similar jobs, such as McSimA+[16], and gem5[17]. These simulators shown are too powerful for our purposes and include a large set of features that are not relevant for our project.

There seems to be no software currently available that focuses only on the task mapping of a program to a many-core system, let alone one that is focused on students in this field.

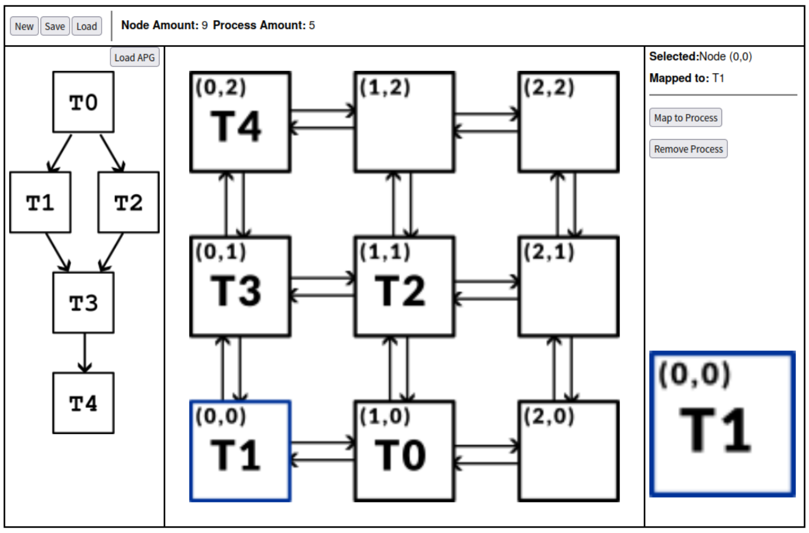


Figure 4.2: *The initial design for the main UI*

4.7 | Initial UI Designs

Figure 4.2 shows the initial design for the main UI. It is designed in accordance with the UI rules specified in Chapter 3. The UI should draw attention to the main focus of the program, the task-graph and the 2D mesh. Both graphs should stay central and large to keep focus on them. 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 the storing these options. Any text given to the user should stay structured to allow the user to parse key information as quick as possible. There should be as few options as possible to keep the UI simple as this is meant to be a learning tool for students. A menu bar should be available for importing and exporting of task mappings.

5 | Implementation

5.1 | Code Structure

As described in the Initial Report[12], the programming language we will be using is Python[18], mainly due to the high amount of documentation and libraries available for itself. In order to best utilise Python's features and to easily allow for further modification, we should aim to make separate parts of the program using object-orientation. This allows us to split our program into self-contained 'classes', where each class stores variables and associated functions, known as 'methods'. These variables and methods may be able to be accessed outside of the class ('public') or may only be accessible within the class itself ('private'). All classes usually start with a method that is called upon creation, known as an 'initialization' method, which usually defines the public and private variable for that class. It is usually standard practise to allow public access to private variables via a 'getter' and 'setter' method. This allows controlled access to these private variables without the risk of the user overwriting with an invalid type.

Also mentioned in the Initial Report[12], we will be using Tkinter[19] for our UI and graphics. Tkinter is in-built into Python and was chosen due to its maturity and ease of use in comparison to other libraries. Tkinter is based on the idea of 'widgets', which are components of the UI from buttons to text to windows. Each widget has a parent which dictates which part of the UI it is contained by, along with a grid-based positioning system for choosing exactly where a widget may be. One widget of note is the `Canvas` widget, which allows the drawing of shapes onto itself via x-y coordinates. In contrast to other UI libraries looked at, the canvas allows modification and re-ordering of these shapes after drawing as each shape has a custom ID to refer to itself with. This `Canvas` widget will be very useful as it will be responsible for the visualisation of the task graph and the mesh.

The code written for our project should aim to be consistent and well-documented. This is done by adhering to a given coding style. In the case of Python, the generally accepted coding style is PEP-8[20], which specifies documentation style, line length and indentation, along with best practices. We will also extend this to include type-hinting on all public methods. This will allow a developer to see which type an argument should be and what type the method returns.

Since our project is built using a UI, there are many cases where methods need to be called upon a certain action being done, such as a button being pressed or a mouse being clicked. This is done via a 'callback' method, where a function that takes an event from the UI is called every time that event happens and is then responsible for calling whatever methods are necessary. Due to the nature of our project, we must ensure that any callbacks are well-documented with what action calls them.

Our codebase should be structured into three main 'libraries':

Main UI Responsible for the UI and main loop of the program. Also responsible for I/O.

Task Graph Responsible for storing the task graph, along with visualisation of itself.

Mesh Responsible for storing the mesh, along with visualisation of itself and of calculation of task-mapped routes.

Each of these libraries should mainly communicate via getter/setter methods and callback functions. Each library should also have robust error handling since actions may occur where, for example, a task graph is asked for tasks before one has been imported.

5.2 | Task Graph Data Structure

TaskGraph	
nodes	List of all nodes in TaskGraph
edges	List of all edges in TaskGraph
selected_node	Currently selected node

Table 5.1: *Attributes for a TaskGraph object*

TGNode	
tid	Task ID
comp_load	Computational load of task
in	Edges where node is a destination
out	Edges where a node is a source

Table 5.2: *Attributes for a TGNode object*

TGEdge	
parent	TGNode which is source
child	TGNode which is destination
comm_load	Communication between the two nodes

Table 5.3: *Attributes for a TGEdge object.*

The task graph is stored as a directed graph, where each task is a node on the graph that stores a computational load and a task ID (for identification). A list of edges were stored, which stored the source node, the destination node and a communication load. Each node also stored any edges where it was a parent (allowing a user to traverse down the graph from any given node). The task graph itself stored all the nodes and edges in two separate lists for easy access.

An algorithm was sought out that could figure out the optimal placements for each node on the graph to maximize readability. Originally, it was thought that a pre-existing library would be available for re-use, such as by embedding a Matplotlib[1] graph, or by utilizing an existing GraphViz[21] drawing library. However, these methods would not allow for on-the-fly modification of the drawing or for integration with Tkinter. For these reasons, a custom graph drawing implementation was designed.

The task graph drawing algorithm is based on the Sugiyama method[23]. A full overview of the implementation can be found in Appendix B, however the basics of the algorithm is to group nodes into 'layers' such that each edge points downwards. Then, 'dummy nodes' are added to ensure that an edge only spans one layer at most. Layer assignment then allows us to assign each node a y-coordinate based on the layer it resides in, along with an x-coordinate based on the number of nodes in a given layer. An issue with this method is that dummy nodes and edges to and from them need to be filtered out when accessed from outside the task graph.

The task graph data structure is also responsible for the importing of task graphs from disk. The format to be used was the DOT file structure, as it is a standard graph file format that allows for many versatile attributes to modify the graph, and has a variety of different programs that can generate files in that format. The library used for this task is `pydot` [24], as the other libraries (`graphviz` [21] and `graphviz-python` [25]) did not allow easy extraction of attributes and were instead meant for graph displaying and modification. In contrast, `pydot` gives easy access to all attributes of a node or edge via inbuilt Python dictionaries.

Further additions to the task graph included being able to select a task from the task graph via clicking on it. This was done by registering a callback function, named `click_callback()` that checks, upon the mouse being clicked within the task graph, whether the position of the mouse is within a task on the task graph. If so, the node is highlighted and set as the currently selected node. This function also runs a callback supplied by the main UI, so that the main UI knows that the selected node had changed.

5.3 | Mesh Data Structure

Initially, the mesh data structure was designed to consist of a pair of 2D arrays: one for the state of each core, and another for the task assigned to each core. The size of the mesh would then dictate the size of the 2D arrays. This approach worked fine for initial prototypes used to test assignment of tasks to cores and initial export

[1]MatPlotLib[22] is a Python library for generating graphs.

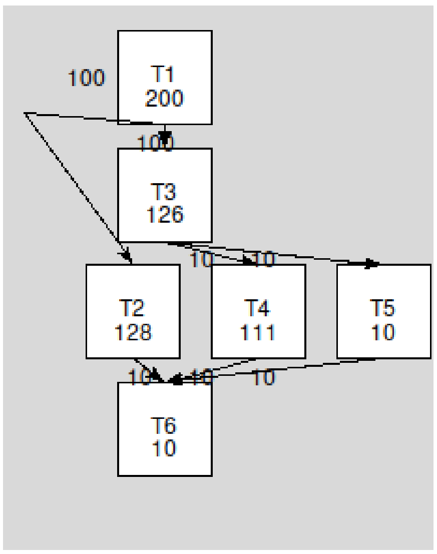


Figure 5.1: *Visualisation of TaskGraph from the program.*

Mesh	
cores	List of all cores in Mesh
rows, cols	Size of Mesh
routing_algo	Currently used routing algorithm
routing_funcs	Callback function for each routing algorithm loaded

Table 5.4: *Attributes for a Mesh object*

MeshCore	
task	Current task assigned to core
state	Current state assigned to core
x, y	Used for drawing onto the canvas
north, south, east, west	Communication load for outgoing link in each direction

Table 5.5: *Attributes for a MeshCore object*

functionality, however when starting development on visualising the mesh it was seen that more had to be stored for each core, such as where to draw it. Therefore, in the final library each core has its own `MeshCore` object that stores information about each core on the mesh. The `Mesh` object then stores a list of all the `MeshCore` objects in use.

The user needs some method to assign a task to a given core. Along with this, the program needs to know which tasks are available to assign. This functionality has been rolled into one object, `ContextMenu`. This object stores an inbuilt Tkinter `Menu` that is bound show itself when the user right-clicks within the Mesh visualisation. Along with this, it stores a list of all tasks in the currently loaded task graph that gets updated often and shown within the context menu. When the user right-clicks somewhere within the Mesh, the `ContextMenu` callback checks whether the click is within one of the cores. If it is, the menu is shown with a list of tasks which are not assigned yet. If the core that is clicked on has a task assigned to it, it adds an extra option that allows the user to de-assign that task. Clicking on one of the options then calls a method within the `MeshCore` that handles cleanly assigning that task.

Another feature was decided to be added to allow scaling of the visualisation, so that a user could zoom in and out. Originally, this was done via the Tkinter `Canvas`' inbuilt method for scaling already drawn objects. However, this method often went out of sync with the method for checking which core was clicked on. Therefore, it was decided that instead scaling would be controlled manually and that the visualisation would be called to redraw at a given scale. This was done by binding a callback to the scroll wheel and checking whether is has scrolled up or down to increment or decrement the scale value.

The mesh data structure is also responsible for the generation of routes to/from each mapped core. The mesh is given a list of routing algorithm functions loaded in from the main UI, and depending on the chosen routing algorithm will run that function. Each routing algorithm is handed a source core, a destination core, and the corresponding task graph edge, where the routing algorithm will then populate the necessary communicational loads for any core in the route. More information on the routing algorithm feature is given in Subsection 5.5.1.

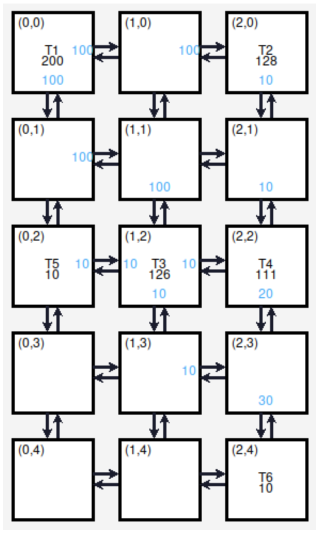


Figure 5.2: *Visualisation of the mesh with mapped tasks.*

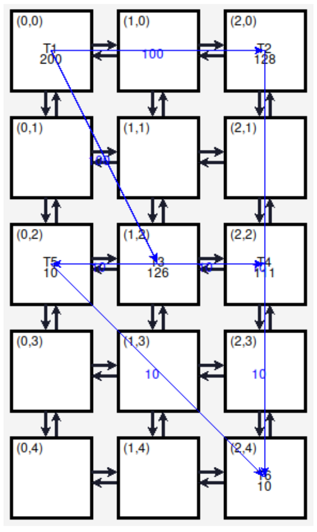


Figure 5.3: *Visualisation of the mesh with mapped tasks and optional overlay.*

The mesh visualisation is designed to show all the relevant information about the mesh in a concise format. Each core has a co-ordinate in the top-left corner and outgoing communication load in all cardinal directions. If a task is mapped to that core, the task ID and computational load is given in the centre of the core. There are a number of options to modify for the visualisation, such as the ability to change to origin of the co-ordinates from top-left to bottom-left, or the optional communication overlay. The assigned state may also be modified, showing as a different fill colour (yellow for spare, dark grey for broken). The visualisations can be seen in Figures 5.2 and 5.3.

5.4 | Export Formats

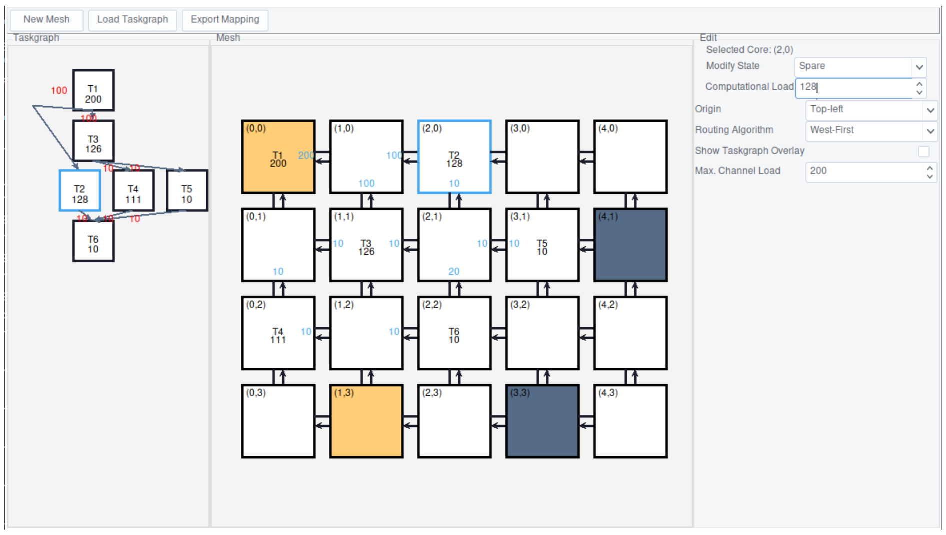
The original export format specified was a simple CSV-based one used to mainly test that data could be exported correctly. The format is made of four sections: task mapping of cores, states of cores, task data, edge data. Each section is separated by a line with only one value `0`. The task mapping of cores and states of cores is exported where one row of the mesh is one row of the CSV. The size of the mesh can then be inferred from the number of values in each row and how many lines there are in the section. The task ID is given for a core that has a task assigned to it, else it defaults to the value -1. The task data is given for each task in the format `task ID, comp load`. The edge data is given in the format `source task ID, dest task ID, comm load`. An example export file is given in Figure 5.4.

```
1,2,3
-1,4,-1
6,-1,5
0
ACTIVE,ACTIVE,ACTIVE
ACTIVE,ACTIVE,ACTIVE
ACTIVE,ACTIVE,ACTIVE
0
1,200
2,128
3,126
4,111
5,10
6,10
0
1,3,100
3,4,10
3,5,10
2,6,10
4,6,10
5,6,10
1,2,100
```

Figure 5.4: *Example CSV export.*

A later export format was added, based off a version used by another project in the same field. It uses a custom XML specification to store a task graph and associated mesh, along with further information such as core ages and router status. Most of the attributes specified in the XML format are not able to be given with the current scope of the project, so they are omitted from the exported result and the specification modified. When a task mapping is exported to XML, the program checks the generated XML export against an XML validator to ensure that the file has no missing data.

The last export format added was the ability to export an image of the mesh visualisation. This is done via the use of `ImageGrab`, a module contained within PIL[26], an image processing library. Care was taken to ensure that the export dialog was closed before taking the image, as it would have been visible in the resulting image. This was done by enforcing a wait time of 1 second between choosing the export option and taking the screenshot.



Selected Core: (2,0)
Modify State: Spare
Computational Load: 128
Origin: Top-left
Routing Algorithm: West-First
Show Taskgraph Overlay: ☐
Max. Channel Load: 200

Figure 5.5: *The main UI of the program.*

5.5 | GUI Components

5.5.1 | Main UI

As can be seen in Figure 5.5, the structure of the main UI has stayed very similar to the initial designs. The UI is still split into 3 columns, each one focusing only on one part of the program. The edit pane on the right hand side dynamically updates based on selected tasks and cores. For example, if a core is selected, it will update to show the currently selected core and give dropdown menus for choosing the state of the core, and likewise for a task on the task graph. It also houses the options that call for modifying the mesh visualisation.

The main UI is also responsible for the dynamic loading of routing algorithms from disk. An additional feature of the program is that custom routing algorithms can be given and applied to the mesh. Upon the main UI being initialised, it checks the `routing-algos/` folder for any python files and tries to load them as a module. Each module has a function that is called to calculate the communication load for a given route on the mesh. Further information on this process is given in Appendix A.

5.5.2 | Dialog Boxes

Our program needs two custom-built dialog boxes: one for setting the size of the mesh data structure, and one for choosing what format to export in. The only other dialog box is for selecting an import file from disk, and is provided by Tkinter's inbuilt FileDialog[27] class. These custom dialog boxes were made by extending Tkinter's SimpleDialog[27] class, which allows a developer to build custom dialog boxes via overriding an inbuilt `body()` method.

Error message boxes generated by the program also use a Tkinter-provided class known as `messagebox`. They are made by calling the method `showwarning()` with a provided title and description.

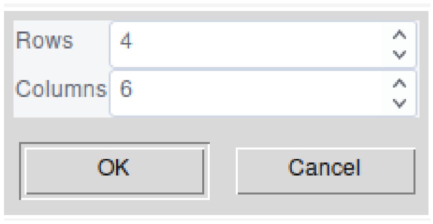


Figure 5.6: *Custom dialog box for mesh size prompt.*

5.6 | Testing Framework

As described in Section 4.3, Extreme Programming specifies that unit tests should be written for a feature before implementation of that feature starts. Testing must also be done for the entire program as a whole, known as integration testing. This section describes how the testing methodology was undertaken throughout the project.

5.6.1 | Automated Testing

Testing is automated with the use of the inbuilt Python library `unittest` [28]. Each test case can then be run automatically via the command line. Each test case consists of at least three methods: `setUp()`, `tearDown()`, and any method that starts with `test`. Testing on each case can then be done via asserting a return value that should be of a given type or has a given value. Unfortunately, since most of our program is UI-based and therefore would need some way to simulate user input across the program, not all testing can be automated this way.

5.6.2 | Manual Testing

Manual testing (or integration testing) is done by hand via testing specific user interactions in different orders to see how the program handles it. For example, this may be trying to assign a task to a node before a task graph has been loaded. The main focus to test is any of the fields for user input, such as loading a file, choosing an export option, or modifing a value on the edit pane. Manual testing is done once a feature is near completion to check for any bugs to fix, along with near the end of the project as a whole.

6 | Conclusion

6.1 | Feature Achievement

In conclusion, the final product has met a large amount of the sets of features. It can be seen that all of the minimal features have been achieved in full. The reasonable features specified have met with one exception. Feature 8 was not met in full, mainly due to the additional work required to allow for panning[1]. It can be seen that only one extra feature has been added, the ability to add a maximum bandwidth for communication links (Feature 15).

6.2 | Criteria for Success Achievement

Criteria	Met
UI structure	Partially
Distinct Mesh & TaskGraph Visualisation	Yes
Clear indication of actions	Yes
Usable with visual impairments	Partially
Easily packaged with distribution	No
No extra programs needed	Partially
User documentation	Yes
Developer documentation	Partially
Structured codebase	Yes
Full automatic testing	No

From this table, it can be seen that a majority of the criterion have been met either fully or in part. Criterion 1 has only been met partially, mainly due to the lacking UI final product. Criterion 4 has been met for those with colour-blindness, but not for those with partial blindness. Criterion 8 has been met with internal comments and documentation, but not in a centralised place for easy reference. Criterion 5 was not able to be met due to the lack of time to bug-fix packaging. Finally, criterion 10 was not able to be met, due to the UI needing manual testing.

[0]Panning requires the mesh to 'share' the right click function between opening a context menu and checking for dragging, along with needing major refactoring of the drawing and position checking methods.

6.3 | Other Successes & Failures

I believe that the visualisations of the mesh and task graph are fit for purpose. As they are the main focus of the program and what an average user will spend the majority of their time looking at, a large amount of time was spent making sure they were clear and concise with the information presented.

In terms of time management, I believe that this project was a failure. While time management went okay for the first half of the project, the features specified to be implemented in the second half were more design-intensive than previously thought. This lead to may parts of the program having to be refactored, causing a drift in the planned project timescale which was left unchecked. It is also my belief that the two weeks allotted for catch up were too minimal, as by that point work had to be started on this report along with exam preparations starting. This could have been rectified by sticking more closely with the chosen project management scheme.

In terms of the UI design, I believe that the final product is lacking. Unfortunately, while Tkinter is able to make some good-looking designs, it suffers from a much more rigid structure than previously thought. For example, padding may only be added to the left and top of a widget, rather than all around. Along with this, there is no formal method to hide or show a widget and its contents, so the final result of the edit pane suffers greatly from this. Along with this, the final product suffers from a lack of skill and time in refining the UI to look better. As described previously, the time allotted for UI refinement was taken up by catch-up work, so the work done was limited and focused on the visualisations of the mesh and task graph.

6.4 | Further Work

There is a variety of additional work that could extend the final program further. For example, the current work done on routing algorithms could be further extended with smarter route choosing in cases where there are more than one route to a given source-destination pair. If this were to be implemented, it would need a rewrite of how the routing algorithm files themselves are called and how they interface with the mesh, since no knowledge of routes is kept from any previous source-destination pairs.

It would also benefit the project to allow for the implementation of more attributes a given core could have, such as the addition of age or computational load limit, that could be then displayed on each core. Along with this, the addition of many of the extra features would aid figure geeration. This could be further refined by including a menu that allows the user to toggle certain attributes on/off the display, so only the relevant information can be exported.

Further work may include the refinement of the UI to allow for the user to open menus and select parts of the UI using keyboard shortcuts. This would aid a potential power-user by streamlining the process of assigning tasks and exporting mappings. Other features that could include the implementation of saving an in-progress mapping, so that the user may return to it at a later date without having to re-do all the work. This could also include the addition of templates for use in a classroom setting, where students are given a specific task graph and mesh combination to map to.

References

[1] Royal Academy of Engineering. "Engineering ethics in practice: A guide for engineers." (2011), [Online]. Available: `https://raeng.org.uk/media/cz5du0gl/engineering_ethics_in_practice_full.pdf` (visited on 12/14/2023).

[2] K. Asanovic, R. Bodik, J. Demmel, *et al.*, "A view of the parallel computing landscape," *Commun. ACM*, vol. 52, no. 10, pp. 56–67, Oct. 2009, ISSN: 0001-0782. DOI: `10.1145/1562764.1562783`. [Online]. Available: `https://doi.org/10.1145/1562764.1562783`.

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

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

[8] C. Glass and L. Ni, "Adaptive routing in mesh-connected networks," in *[1992] Proceedings of the 12th International Conference on Distributed Computing Systems*, 1992.

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

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

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

[12] J. Walker, *Visualising task mapping in many-core systems: Initial report*, 2024.

[13] D. Wells. "Extreme programming: A gentle introduction." (2013), [Online]. Available: `https://www.extremeprogramming.org` (visited on 03/01/2024).

[14] The git team. "Git." (2024), [Online]. Available: `https://git-scm.com/` (visited on 05/23/2024).

[15] M. Lis, P. Ren, M. H. Cho, *et al.*, "Scalable, accurate multicore simulation in the 1000-core era," in *(IEEE ISPASS) IEEE International Symposium on Performance Analysis of Systems and Software*, 2011.

[16] J. H. Ahn, S. Li, S. O, and N. P. Jouppi, "Mcsima+: A manycore simulator with application-level+ simulation and detailed microarchitecture modeling," in *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2013.

[17] The gem5 Team. "The gem5 simulator." (2024), [Online]. Available: `https://www.gem5.org/` (visited on 05/20/2024).

[18] Python Software Foundation. "Python.org." (2024), [Online]. Available: `https://www.python.org/` (visited on 05/19/2024).

[19] Python Software Foundation. "Tkinter - python interface to tcl/tk." (2024), [Online]. Available: `https://docs.python.org/3/library/tkinter.html` (visited on 05/19/2024).

[20] A. C. Guido van Rossum Barry Warsaw. "Pep 8 – style guide for python code." (2013), [Online]. Available: `https://peps.python.org/pep-0008/` (visited on 05/19/2024).

[21] The Graphviz Authors. "Graphviz." (2024), [Online]. Available: `https://graphviz.org/` (visited on 05/19/2024).

[22] The Matplotlib development team. "Matplotlib: Visualization with python." (2024), [Online]. Available: `https://matplotlib.org/` (visited on 05/19/2024).

[23] K. Sugiyama, S. Tagawa, and M. Toda, "Methods for visual understanding of hierarchical system structures," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 11, no. 2, 1981.

[24] P. Nowee, E. Carrera, and S. Kalinowski. "Pydot 2.0.0." (2023), [Online]. Available: `https://pypi.org/project/pydot/` (visited on 05/21/2024).

[25] J. Ellson. "Graphviz-python 2.32.0." (2013), [Online]. Available: `https://pypi.org/project/graphviz-python/` (visited on 05/21/2024).

[26] F. Lundh and J. A. Clark. "Pillow." (2024), [Online]. Available: `https://python-pillow.org/` (visited on 05/23/2024).

[27] Python Software Foundation. "Tkinter dialogs." (2024), [Online]. Available: `https://docs.python.org/3/library/dialog.html#module-tkinter.simpledialog` (visited on 05/21/2024).

[28] Python Software Foundation. "Unittest - unit testing framework." (2024), [Online]. Available: `https://docs.python.org/3/library/unittest.html` (visited on 05/19/2024).

[29] R. Tamassia, *Handbook of Graph Drawing and Visualization*. CRC Press, 2016.

[30] gml4gtk. "Pysugiyama." (2021), [Online]. Available: `https://github.com/gml4gtk/pysugiyama` (visited on 05/23/2024).

[31] C. Gotz. "The sugiyama method - layered graph drawing." (2017), [Online]. Available: `https://blog.disy.net/sugiyama-method/` (visited on 05/23/2024).

A | Adding a Routing Algorithm

Custom algorithms may be added to the program by adding a new python file to the `src/routing-algos/` folder. Any python file in this folder is dynamically imported upon program startup. This file should contain two components:

- `ALGO_NAME`
- `calculate_route()`

`ALGO_NAME` is the name to be displayed when selecting a routing algorithm from the drop-down menu. `calculate_route()` is to be called for every source-destination pair assigned onto the many-core topology. No imports are needed within the file itself, as they should all be loaded in by the main program on startup.

A.1 | Route Calculation

This function is called for every source-destination pair that has been assigned to the mesh. This function is supplied the arguments:

- `mesh` (of type `Mesh`)
- `source` (of type `MeshCore`)
- `dest` (of type `MeshCore`)
- `edge` (of type `TGEdge`)

Mesh is the entire 2D mesh. It has the following methods that may be useful:

`get_core(row, col)` Gets the `MeshCore` at that position.

MeshCore is the core with the assigned task. It has the following attributes that may be necessary:

`row` the row it is on

`col` the column it is on

`north, south, east, west` the outgoing communication load at each of the directions.

TGEdge is the edge connecting the two nodes together. It has the following attributes that may be necessary:

`comm_load` the communication load between the two nodes.

In order to see a simple routing algorithm to reference, the file `xy.py` should be used.

B | Drawing Pretty Task Graphs

In the program, I have used a modified version of the Sugiyama method for drawing the task graph in a layered way. While the details of this algorithm are not important to the main body of the report, they may be of interest to general readers.

The Sugiyama method (or framework), first described in 1980[23], breaks down the algorithm for drawing readable directed graphs (such as our task graph) by separating nodes into layers and reducing the number of crossed edges. It is described in four general steps[29]:

1. The removal of cycles
2. Assignment of layers to nodes
3. Re-ordering of nodes to minimize crossed edges
4. Assignment of x-coordinates

Our project uses a much simpler version of this method. Any task graph imported into the program is assumed to contain no cycles, so step 1 does not need to be done. Along with this, the re-ordering of nodes is not crucial as it is unlikely that any task graph used will have more than 4 nodes per layer. Our implementation is based upon the work done by gml4gtk[30] and disy[31].

B.1 | Layer Assignment

Each node in the graph is sorted into layers via iteratively separating any nodes that have no outgoing edges and removing any edges containing those nodes. This generates a list of layers from bottom-to-top of the graph. This list may then be reversed to get the top-to-bottom sorted nodes in each layers.

B.2 | Dummy Node

In order to keep our edges from crossing more than one layer (and therefore increasing the chance that it looks ugly), we place in 'dummy' nodes between layers. This is done via iterating over all edges in a graph and checking if the source and destination nodes have a layer difference of more than 1 layer. If they do, a dummy node is placed

```
func generate_layers(graph) {
        % list of all layers in a graph
        list layers = {};

        % while there are nodes left in graph
        while size(graph.nodes) > 0 {
                % make a list of all nodes with no outgoing edges,
                % called 'sinks'
                list sinks;
                for node in graph.nodes {
                        if size(node.outgoing) = 0 {
                                sinks.add(node);
                        }

                % add sinks to layers
                layers.add(sinks);

                % remove edges where node in sinks is a child
                for node in graph.nodes {
                        for edge in node.outgoing {
                                if edge.destination in sinks {
                                        node.outgoing.remove(edge);
                                }
                        }
                }
                % loop until no nodes left
        }

        return reverse(layers) % flip layers so top-to-bottom
}
```

Figure B.1: *Pseudo-code for layer assignment in a graph.*

inbetween, with corresponding edges pointing from source to dummy, then dummy to destination.

C | User Guide

The program is split up into 4 sections:

- The menu-bar
- The task graph
- The mesh
- The edit pane

The menu-bar is responsible for initialisation, import and export of task mappings. The task graph is responsible for displaying the currently loaded task graph for easy reference when mapping. The mesh is responsible for facilitating task mapping onto a many-core system. The edit pane is for modifying additional options about the mesh or the task graph.

Typical workflow:

1. The user loads a task graph from disk via the 'Load Taskgraph' button. Some example task graphs are given in `assets/graphs/`.
2. The user creates a mesh using the 'New Mesh' button and is prompted about the size to use.
3. The user fills out the size and the mesh and task graph are visible in their respective sections.
4. The user right-clicks on a core in the mesh to bring up options about mapping a task.
5. The user clicks an available task to assign it.
6. The user repeats this for all available tasks.
7. The user checks in what direction the communication will be in by toggling the 'Show Taskgraph Overlay' toggle in the edit pane..
8. The user chooses a routing algorithm from the dropdown menu in the edit pane.
9. The user is satisfied with the mapping and decides to export it in a specified format by clicking the 'Export Mapping' button in the menu-bar.