A MESSAGE-PASSING MANY-CORE ARCHITECTURE FOR AN ASYNCHRONOUS GRAPH PROGRAMMING MODEL

By

Sebastien Cook

A thesis submitted to the Faculty of Graduate and Postdoctoral Affairs in partial fulfillment of the requirements for the degree of

Master of Applied Science

In

Electrical and Computer Engineering

Carleton University

Ottawa, Ontario

© 2022, Sebastien Cook
Noticeable improvements in processor performance have been achieved by researching programming models, control flow parallelization, general architecture, memory access, and code compilation [53][4]. In this thesis, we seek to improve general processing by applying a many-core message passing (MPMC) architecture with a novel Asynchronous Graph Programming model (AGP).

AGP abstracts higher-level languages into a graph of single instructions providing very high levels of parallelism and asynchronicity. The MPMC architecture utilizes a novel method of segmenting a graph among cores in tandem with the many-core model to exploit AGPs parallelism.

We evaluate the MPMC architecture implementing a functional simulation that, although incapable of providing empirical measurements, provides an efficient method of evaluation that helps accelerate the development cycle. We found that the MPMC architecture can reach a 97% improvement in execution time from a single-core configuration, with room to improve given more cores and better node allocation strategies.
Many thanks to my supervisor Dr. Paulo Garcia for providing me with this opportunity. I’m further grateful for all of his guidance and support during the writing of this thesis. He has both inspired and ignited a passion for research and development in me.

I would like to also extend my gratitude to my mother, Dr. Nathalie Major who has pushed me to pursue this opportunity and provided invaluable support and guidance during my thesis. I would like to thank Mahsa Layeghi, a colleague in my lab who provided fresh ideas and relatable experiences, easing the writing process. I would also like to thank Pat Faris for his deep wisdom and helpful advice regarding both the thesis and what’s next. I would like to acknowledge my incredible group of friends who helped maintain a positive mental space during this difficult task. Finally, I’d like to thank my partner Lucia Wang who, despite coming from an entirely different scientific field, dispensed invaluable feedback and support. Her incredible intelligence, support and affection helped pave the way to success and I strive to give it all back in double during her doctorate program.
## CONTENTS

List of Tables vii  
List of Figures viii  
List of Appendices xi  
Acronyms xii  

1 INTRODUCTION 1  
1.1 Our Approach ........................................... 5  

2 BACKGROUND 8  
2.1 Models and Architectures .............................. 8  
2.2 Graph theory .............................................. 11  
2.3 Asynchronous Graph Programming .................. 13  
2.4 AGP Syntax ................................................ 18  

3 RELATED WORK 19  

4 MESSAGE-PASSING MANY-CORE ARCHITECTURE 28  
4.1 General Architecture ..................................... 28  
4.2 Node Structure and allocation ....................... 30  
4.3 Single Core Architecture .............................. 33  
4.4 Single Core Execution .................................. 35  
4.5 Parallel Execution ....................................... 39  
4.6 Previous Iterations ..................................... 42  

5 EXPERIMENTAL METHOD 44  

6 RESULTS AND DISCUSSION 47  
6.1 General Performance .................................... 47  
6.2 The Impact of Operation Types ..................... 49  
6.3 The Impact of Communication ....................... 53
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>CONCLUSION</td>
<td>56</td>
</tr>
<tr>
<td></td>
<td>BIBLIOGRAPHY</td>
<td>59</td>
</tr>
<tr>
<td>A</td>
<td>APPENDIX A</td>
<td>68</td>
</tr>
<tr>
<td>B</td>
<td>APPENDIX B</td>
<td>72</td>
</tr>
<tr>
<td>C</td>
<td>APPENDIX C</td>
<td>79</td>
</tr>
</tbody>
</table>
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 1</td>
<td>Average Execution Time</td>
<td>48</td>
</tr>
<tr>
<td>Table 2</td>
<td>Average Speedup of Multi Core Configurations in Reference to Single-Core Execution</td>
<td>49</td>
</tr>
<tr>
<td>Table 3</td>
<td>Operation vs execution ticks correlation in a 2x2 configuration</td>
<td>51</td>
</tr>
<tr>
<td>Table 4</td>
<td>Operation vs execution ticks correlation in a 2x2 configuration</td>
<td>52</td>
</tr>
<tr>
<td>Table 5</td>
<td>Operation vs execution ticks correlation in a 4x4 configuration</td>
<td>68</td>
</tr>
<tr>
<td>Table 6</td>
<td>Operation vs execution ticks correlation in a 4x4 configuration</td>
<td>68</td>
</tr>
</tbody>
</table>
**LIST OF FIGURES**

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>Data Parallelism and Task Parallelism represented visually</td>
<td>3</td>
</tr>
<tr>
<td>Figure 2</td>
<td>The size and speed of common memory modules used in a memory hierarchy model</td>
<td>4</td>
</tr>
<tr>
<td>Figure 3</td>
<td>AGP is an intermediate representation of a program that introduces instruction level parallelism</td>
<td>6</td>
</tr>
<tr>
<td>Figure 4</td>
<td>Uniform Parallel Processing Memory Hierarchy</td>
<td>10</td>
</tr>
<tr>
<td>Figure 5</td>
<td>Batch Synchronous Processing model</td>
<td>10</td>
</tr>
<tr>
<td>Figure 6</td>
<td>Basic graph examples: (a) non-directional non-weighted graph, (b) directional non-weighted graph, (c) non-directional weighted graph</td>
<td>12</td>
</tr>
<tr>
<td>Figure 7</td>
<td>Tree graph</td>
<td>13</td>
</tr>
<tr>
<td>Figure 8</td>
<td>AGP graph with application, destruction and push operations</td>
<td>14</td>
</tr>
<tr>
<td>Figure 9</td>
<td>Application Application graph example</td>
<td>15</td>
</tr>
<tr>
<td>Figure 10</td>
<td>Application Destruction graph example</td>
<td>15</td>
</tr>
<tr>
<td>Figure 11</td>
<td>Destruction Destruction graph example</td>
<td>16</td>
</tr>
<tr>
<td>Figure 12</td>
<td>Application Push graph example</td>
<td>17</td>
</tr>
<tr>
<td>Figure 13</td>
<td>Destruction Push graph example</td>
<td>17</td>
</tr>
<tr>
<td>Figure 14</td>
<td>Push Push graph example</td>
<td>17</td>
</tr>
<tr>
<td>Figure 15</td>
<td>Common CGRA reconfigurable array interconnections</td>
<td>21</td>
</tr>
<tr>
<td>Figure 16</td>
<td>Simplified recreation of the ADRES and MorphSys architectures.</td>
<td>22</td>
</tr>
<tr>
<td>Figure 17</td>
<td>Performance, flexibility, and power efficiency of different common architectures. Recreation of an image found in [35]</td>
<td>24</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>Graph processor architecture. This design forgoes shared main memory by using a message-passing design. [47]</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>Message passing many-core architecture</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>Node structure: Each node in the image contains two inputs and outputs. However, a node may contain more or less of either. A node may have no inputs if it’s a variable. A node will always have at least one output.</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>Single core block diagram.</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>list structure</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>Flow chart of a single cores’ logic. Note that this flow chart doesn’t represent a single execution step</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>Flow chart of a single core expansion node execution (a) and expansion broadcast execution (b). Note that this flow chart doesn’t represent a single execution step</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>list representation of two concurrent application operations: before (left), after (right)</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>list representation of two concurrent application and destruction operations</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>list representation of two concurrent destruction operations</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>list representation of two concurrent application and expansion operations</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>list representation of two concurrent destruction and expansion operations</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>list representation of two concurrent expansion operations</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>Full graph expansion representation of benchmark programs</td>
<td></td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>32</td>
<td>Execution time in single-core, 2x2, and 4x4 configurations. Green represents single-core execution, red represents the two-by-two configuration and blue represents the four-by-four configuration.</td>
<td>48</td>
</tr>
<tr>
<td>33</td>
<td>Core utilization in a 2x2 configuration of the fastest and slowest execution.</td>
<td>50</td>
</tr>
<tr>
<td>34</td>
<td>Number of ticks spent looking for nodes vs the total execution ticks.</td>
<td>51</td>
</tr>
<tr>
<td>35</td>
<td>Visualization of the impact of the node distribution from allocation on the total execution time.</td>
<td>53</td>
</tr>
<tr>
<td>36</td>
<td>Bytes processed in relation to execution time.</td>
<td>54</td>
</tr>
<tr>
<td>37</td>
<td>Bytes processed in relation to execution time.</td>
<td>55</td>
</tr>
<tr>
<td>38</td>
<td>Core utilization in a 4x4 configuration of the fastest and slowest execution.</td>
<td>69</td>
</tr>
<tr>
<td>39</td>
<td>Number of ticks spent looking for nodes vs the total execution ticks.</td>
<td>70</td>
</tr>
<tr>
<td>40</td>
<td>Bytes processed in relation to execution time.</td>
<td>71</td>
</tr>
<tr>
<td>41</td>
<td>Application in relation to execution time.</td>
<td>72</td>
</tr>
<tr>
<td>42</td>
<td>Destruction in relation to execution time.</td>
<td>73</td>
</tr>
<tr>
<td>43</td>
<td>Local expansion in relation to execution time.</td>
<td>74</td>
</tr>
<tr>
<td>44</td>
<td>Expansion call in relation to execution time.</td>
<td>75</td>
</tr>
<tr>
<td>45</td>
<td>Result propagation in relation to execution time.</td>
<td>76</td>
</tr>
<tr>
<td>46</td>
<td>Routing in relation to execution time.</td>
<td>77</td>
</tr>
<tr>
<td>47</td>
<td>Garbage collection in relation to execution time.</td>
<td>78</td>
</tr>
</tbody>
</table>
LIST OF APPENDICES

A Appendix A: 4x4 Core Utilization 68
B Appendix B: Execution time vs each process 72
C Appendix C: AGP Code 79
ACRONYMS

AGP  Asynchronous Graph Programming

MPMC  Message Passing Many Core

ASIC  Application Specific Integrated Circuit

FPGA  Field Programmable Gate Array

PIM  Processing In Memory

AMP  Asynchronous Multi Processor

CGRA  Coarse Grained Reconfigurable Architecture

HLS  High-level Synthesis

CPU  Computer Processing Unit

SOC  System On Chip

DRCA  Dynamically Reconfigurable Computer Architecture

VLIW  Very Long Word Instruction

RISC  Reduced Instruction Set Computer

GPU  Graphical Processing Unit

API  Application Programming Interface
Moore’s Law, which went through several revisions, dictates that an integrated circuit’s (IC) complexity will double every 18 months \[^{55}\][^{50}]. Version one was focused on scaling up the number of components on a single chip; version two focused on scaling the performance by increasing the clock speeds; and version three focused on the integration of multiple functional units. Each version of Moore’s Law predicted the widespread advancements such as mainframes, microcomputers, and tablets.

Moore’s Law is often linked to technological advancements, social changes and economic growth \[^{55}\]. Reducing transistor sizes, reducing channel length, and increasing clock frequencies allow for more functions to be integrated into a single IC \[^{50}\]. Unfortunately, the physical limitations of scaling have led to Moore’s Laws predictions to wane. More specifically, any further deduction of atomic size will lead to quantum tunnelling, often causing the IC to short circuit \[^{50}\]. Furthermore, increasing the clock frequency leads to thermal temperatures and energy consumption too high for commercial products \[^{55}\].

In order to maintain performance speedups, researchers and industry look towards parallelism. However parallelism becomes very difficult to program as the complexity and size of a program increases. There are two common approaches to programming parallelism: auto-parallelization and parallel programming \[^{13}\][^{31}].

Auto-parallelization parallelizes previously sequential programs using instruction-level parallelism or parallel compilers. Although the parallelization levels that can be reached are limited due to the program’s sequential nature, it only requires recompilation. This significantly reduces the burden and complexity for
programmers. Parallel Programming on the other hand is a more complicated and involved approach, requiring active parallelization efforts. However, the parallel nature of the models allows programs to reach much higher levels of parallelism.

Multithreading is a common example of a parallel programming model, allowing programmers to separate tasks as threads that can be executed in parallel or concurrently. Concurrency is the capability of completing more than one task simultaneously while parallelism is the capability of executing more than one task simultaneously. Parallelism requires concurrency while concurrency does not require parallelism. A great example is an OS (operating system) which is concurrent by design but may offer parallelism depending on the number of cores it is operating on [41].

As processing demands become greater, concurrency is no longer able to keep up with performance demands. Parallelism is the next logical step that has given rise to multi-core and many-core designs. Parallel computing is the act of solving a problem by dividing it into multiple domains and solving each on different physical processors [41]. However, identifying the correct parallelizable domain is important.

Data-parallel problems are composed of repetitive functions that must be processed across a large data set [5]. These are usually problems assigned to GPUs in a heterogeneous architecture. Task-parallel problems require the application of different functions to a common stream of data [5]. This is commonly associated with the CPU, where separate tasks can be associated with individual threads. It is important to exploit both domains if possible to achieve the best possible performance [5][49].

These parallelism strategies require either the revision or new design of computer architectures, both of which further require new programming models which support parallelism. Noticeable improvements have been achieved by
introduction

researching programming models, control flow parallelization, general architecture, memory access, and code compilation [53][4].

A common performance bottleneck is memory. A processor may only achieve maximum performance when the latency, referring to the time for memory to respond, is approximately zero and the bandwidth, referring to the rate at which data can be transferred, is infinite [14]. The distance between memory and processor, as well as the memory’s performance, affects the latency, while the channel width affects the bandwidth. Increasing the width of the memory package poses drawbacks such as price, size, and power consumption. Therefore, research focuses on reducing memory latency to make up for the reduced bandwidth [14][45]. The most common technique to reduce latency is a multi-level memory hierarchy model [14] as seen in figure 2. This is implemented using multi-level caches on the processor chip, DRAM as main memory, and mass storage devise (SSD,HDD). Other units such as the scratchpad memory [7] have also been proposed to supplement this design. The scratchpad acts similar to a post-it note that the processor can use to store memory for works in progress or data/results it may reuse soon. To support these memory hierarchies and utilize them efficiently, complex compiler support and/or novel programming models are needed to control the data flow and take advantage of the hardware.
A programming model describes an abstract machine with its purpose, goals and general functions [13][37]. It defines a model of execution from which a programming language can implement. The actual implementation of the programming language or the machine used for a given programming model can vary significantly. The Von Neumann model used in sequential computers is a good example of a model with multiple implementations [37]. The Von Neumann model describes that an instruction fetch and a data operation may not occur simultaneously as they share a common bus. The implementation of the bus or operations used is independent of the model. In parallel computing, there are many different models describing multi-core, heterogeneous, and distributed designs integrated with shared or distributed memory models [13][37], describing the methods of parallelization and data flow. Parallel models can significantly increase the performance of a program, however, current models often require programs to be programmed to exploit parallelism.

Compilers are a critical part of computing, taking user code and compiling it into a set of instructions the target hardware can interpret. Many critical parts such as the control flow, memory storage, and efficiency are affected by the quality of compilation. Additionally, a multitude of optimizations can be packed within the compiler to further improve performance before the application even reaches the target hardware. [26] studied the impact of High-Level
Synthesis (HLS) compilers and found that well-educated applications of compiler optimizations can improve performance by 16% and reduce power consumption. [29] finds similar results when evaluating the power consumption of embedded systems. It is important to note that compiler optimizations are not limited to improving throughput or power consumption. As seen in [27], GCC offers many optimization options such as code size optimization that merges repeated values to reduce the memory consumption for memory-constrained systems. However, in the end, compilers are a slave to the programming model and target hardware.

1.1 OUR APPROACH

Asynchronous Graph Programming (AGP)[20], is a novel programming model developed to offer high levels of parallelism and asynchronicity while requiring the programmer to spare little to no thought of parallelization. This is achieved as an intermediate representation, which compiles a higher-level language like C into a graph of single instructions. A single instruction in AGP is a coarse grain representation of either a constant, datum, operation, or expansion, which is saved as a single assignment node. These concepts will be visited later in the text. Representing a program as a graph of coarse grain single instruction introduces instruction level parallelism (ILP) and asynchronicity to the program.

ILP is the parallelization of operations while TLP is the parallelization of inherently parallel threads, which are comprised of numerous operations. While TLP requires a programmer to think parallel to achieve ideal performances, ILP parallelizes operations that are independent of one another, regardless of their thread or position in a program.

Asynchronicity at the software level is the separation of tasks from the main process of execution as it awaits a result from a different resource or process.
This permits the main process to execute other segments of a program until the given tasks receive a result or resource. This separation of tasks can happen at the thread level or instruction level, which AGP inherently does. Additionally, to reducing the amount of time that cores spend idle, asynchronicity offer many benefits such as flexibility (allowing the system to scale and match the program size and/or type (i.e., sequential, multithreaded)), providing better hardware utilization, improving throughput and power consumption, as well as resilience to errors [39].

AGP is an intermediate representation of a program. This means that a program written in a higher level language is compiled into the AGP representation, then later compiled into the target hardware language as can be seen in Figure 3.

![AGP Diagram](image)

**Figure 3:** AGP is an intermediate representation of a program that introduces instruction level parallelism

The effectiveness and performance of AGPs intermediate representation can be heavily affected by the target hardware. This thesis explores the application of a message passing programming model in tandem with a many-core architecture to exploit ILP and the asynchronicity provided by AGP.

ILP mates very well with the many-core architecture as abstracting a program into a series of single instructions generate an incredible number of in-
instructions, and in the end, true parallelism can only be achieved given separate resources. Furthermore, AGP nodes are saved as single assignments which remove the need for shared memory, allowing us to utilize a message passing programming model which requires less synchronization than shared memory.

The message passing many-core architecture (MPMC) is evaluated using a functional simulation implementation. Our functional simulation, AGP, and the MPMC architecture is evaluated using five benchmarks written in a bare-bones AGP language. Each benchmark is tested using single core, 2x2 core, and 4x4 core configurations a thousand times each. As there are numerous nodes in a program and even more numerous ways of allocating the nodes to cores, random node allocation was used for each simulation iteration to provide insight into the impact of node allocation.

specifically the thesis contributions are as follows:

• The application of AGP and a message passing many-core architecture to increase the performance of sequential and parallel programs alike.

• The implementation of a message passing many-core functional simulation which provides preliminary performance information, as well as insight on possible bottlenecks that may be encountered in a hardware implementation.

• A novel graph to list mapping method that is capable of retaining and updating the AGPs graph’s original shape while stored in segmented flat memory.
BACKGROUND

Before delving into the design and concepts of MPMC, we must first review some of the concepts discussed above in further detail. In this section, we will explore common computing and programming models, common computing architectures, some graph theory, and finally AGP in more detail.

2.1 MODELS AND ARCHITECTURES

Computer architecture is the final step towards a working computer. Modern computer architecture can be described by one or more computing models which are, in turn, implementing one or more programming models. [18] determined that there are four types of computer architectures based on data streams and instructions:

- **SISD (Single Instruction Single Data Stream)** can only execute one instruction on one data stream. This architecture has no parallelism capabilities, reminiscent of the original CPUs from Von Neumann’s design.

- **SIMD (Single Instruction Multiple Data Streams)** architecture can apply a single instruction to multiple data streams. This is useful for repetitive tasks and represents data parallelism. GPUs, for example, are an evolved design of SIMD architecture[41].

- **MISD (Multiple Instruction Single Data Stream)** can apply multiple instructions to a single data stream. This architecture is not very common unless one considers pipelines to be MISD.
• MIMD (Multiple Instructions Multiple Data Streams) can apply multiple instructions to multiple data streams. Although the most complicated, MIMD is also the most flexible, with the ability to provide both data and task parallelism depending on the program. Modern CPUs and GPUs are of MIMD design, behaving like SIMD as individual cores and MIMD as a whole, multi- or many-core architecture [41].

Computing models are abstract machines that allow theoretical analysis. In parallel computing, there are four common models: PRAM, PMH, BSP, and LogP [60][41]. Each focuses on memory management.

The PRAM [19] (Parallel Random Access Machine) model is composed of p processors operating synchronously over an unlimited memory that is completely visible by each processor. Within this model, it is described that each read/write operation costs O(1). This is considered an unrealistic model since without proper controls there would be possible conflicts when reading or writing. Variations which present read and write constraints have been created to describe and analyze more realistic versions [8][41][60]. The EREW (Exclusive Read, Exclusive Write) model, each processor’s read and write operations are performed exclusively in different parts of the memory. This can be used for tasks that require no shared memory. CREW (Concurrent Read, Exclusive Write), allows processors to read from common parts of the memory, but writes are done exclusively in individual locations. This is useful for computations, like cellular automata, which require information from shared sites for their own operations. ECREW (Exclusive Read, Concurrent Write) has each processor reading from exclusive locations of the memory, but writing to one common location. There are not many applications of this model since concurrent writing doesn’t offer any advantage without concurrent reading. CRCW (Concurrent Read, Concurrent Write) allows processors to read and write from commonly shared memory locations. Although this model has the most appli-
2.1 Models and Architectures

In applications, it requires write controls, such as priorities which can significantly increase complexity and overhead.

PMH [1] (Parallel Memory Hierarchy) can be seen in many shared memory hierarchies. The PMH model describes a hierarchy of memory modules, where memory modules closest to the processor are small but fast, and the farthest modules are large but slow. This model can be seen in most commercial computer designs.

![Uniform Parallel Processing Memory Hierarchy](image)

**Figure 4: Uniform Parallel Processing Memory Hierarchy**

BSP [51] (Batch Synchronous Processing) is a model focused on communication rather than storage. The model has n processors with small local memories, interconnected through a network. Processors can share information through synchronized communication, requiring all processors to reach a barrier before communication.

![Batch Synchronous Processing model](image)

**Figure 5: Batch Synchronous Processing model**

LogP [10] is a model focused on measuring the communication cost across a network, taking into consideration the latency and overhead of sending and receiving messages. LogP is similar to BSP, and deciding between the two is
a matter of whether the communication cost in the real world would have a significant effect.

Programming models are the abstraction of the programmable aspect of a computing model. They are important for analyzing a computing model, and provide further direction for architecture designs. The shared memory is commonly used with PRAM and PMH, while the message passing model is commonly used with BSP, LogP [41].

The shared memory programming model has parallel entities such as threads or cores processing across a common memory. Depending on the computational model used, parallel threads of computation must synchronize using control systems such as semaphores when reading and/or writing. Most general computing processors use a combination of CRCW and PMH as these models allow for flexibility and a reduction of control system delays, respectively. APIs that implement shared memory models are CUDA, OpenCL, and OpenMC [16][41].

The message passing programming model, commonly known as the distributed model, has processors communicating either synchronously or asynchronously across a network. The model focused on communication and synchronization, with the most common application being distributed systems. Considering a distributed system may have multiple different local hardware designs based on different models, a standard communication interface was created to ease integration. MPI (Message Passing Interface) has become the standard communication interface in distributed systems[16][41].

2.2 GRAPHTHEORY

Graph theory refers to a specific type of graph which is composed of vertices and edges as seen in figure 6. Vertices, also commonly referred to as nodes, represent the data within a graph. Edges represent the connection and rela-
relationship between nodes. With this in mind, a graph G is composed of a finite number of vertices and edges: $G = (V, E)$.

![Graph examples](image)

Figure 6: Basic graph examples: (a) non-directional non-weighted graph, (b) directional non-weighted graph, (c) non-directional weighted graph

As seen in figure 6, edges can be directional and weighted [22][40]. Both offer an extra layer of information between nodes. Graph (a) has non-directional and non-weighted edges, meaning the connection between nodes A and B can also be read as a connection between nodes B and A. A directional edge as seen in (b) offers insight into the directional flow of information or relationships. Weighted edges can also be used to offer a numerical weight between a connection. For example, the numbers in (c) could represent the distance between switches in a network.

The organization of information in nodes and edges offers an intuitive and visual representation of the order, flow, and structure. For this reason, many scientific fields, including computer science, have adopted graph theory. Common computer science applications include social network modelling, big data analysis, complex network analysis, and pattern recognition [22].

Figure 6 demonstrates the basics of a graph. However, there exists a multitude of graph types, each designed to represent types of data in a formally described format. One pertinent example is the tree graph. It is an acyclic graph that has a starting node which is called the root node [40]. As seen in figure 7, the information constantly flows down from a single root node without any loops. This presents a natural hierarchy to the information within the graph.
To visit a given node, you must first visit all the preceding nodes in its path. The inherent hierarchy means that tree graphs have numerous applications such as filing systems, compilers, decision trees, and AGP.

![Tree graph](image)

**Figure 7: Tree graph**

### 2.3 Asynchronous Graph Programming

AGP is a graph-based model of computation meant to be processed asynchronously on highly parallelizable hardware. Its goal is to abstract higher-level languages such as C into a graph composed of inherently parallelizable atomic instructions. Although auto-parallelization is often less performant than purely parallel computing models [31], by parallelizing at the instruction rather than the task level and abstracting into a graph, higher levels of parallelism can be achieved.

Within an AGP graph, the nodes are instructions and the edges represent the dependencies. Much like the tree graph described above, the flow of execution travels from top to bottom and all of the dependencies of an instruction must be fulfilled before running. However, unlike tree graphs, there may be multiple starting (root) nodes, and a single node may have more than one dependency.

There are three types of operations in AGP: application, destruction, and push. Application operations refer to instructions that produce a numeric result that can be applied to the dependent nodes. This includes common atomic
instructions like additions and subtractions but also includes successful if/else operations. Destruction refers to operations that remove un-run nodes from the graph. This occurs on failed if/else operations. Following the destruction, all nodes dependent on the failed if/else are no longer needed and must be pruned from the graph. Push refers to operations that add nodes to the graph. These are equivalent to function calls, which are referred to as expansions in AGP. An expansion operation adds sub-graph nodes to the active graph.

Figure 8 is an AGP graph that contains applications, destruction and push operations. Graph (a) contains three possible starting nodes, a, b, and c, which may be executed in any order or in parallel. These are either static variables or I/O input nodes which are both application nodes used to propagate variables. a and b feed into another application that generates z. Once processed, the if/else nodes may be executed. These, like the starting nodes, may be executed in any order or in parallel. As seen in graph (b), the if condition fails. This leads to destruction which prunes all dependent nodes down to the merge. The else node is processed as an application, leading to an expansion. Graph (c) is the graph state after the push operation. It can be seen in (c) that the subgraph, func, doubles the input and then subtracts it by one. The final node of the subgraph then feeds back into the original expansion’s destination: the merge node.
Figure 8 demonstrates a basic sequential execution where application, destruction, and push operations are executed one after another. However, any number of application, destruction or push operations can be applied simultaneously as long as all the node’s dependencies have been fulfilled. This important feature becomes obvious when represented visually.

![Application Graph Example](image1)

Figure 9: Application Application graph example

Figure 9 depicts parallel application operations. Since the addition and subtraction operations are not dependent on one another, they can easily be applied in parallel since the variables a and b are propagated as individual entities to each operation node. This is achieved by saving the variable result directly in the operation node.

![Application Destruction Graph Example](image2)

Figure 10: Application Destruction graph example

Figure 10 depicts parallel application and destruction operations. As previously mentioned, a successful if/else is executed as an application while the failed pair is executed as destruction. Both the if and else nodes contain the guard statement which means they may be executed in parallel. Figure 10 is
split into three sections to demonstrate the destruction process. However, the destruction may be completed during any portion of the program after the failed statement and will then be represented as instantaneous in further representations. The destruction of a node leads to the removal of all dependent nodes until the merge node is reached.

Since both the if and else nodes contain the guard statement, they may be processed in any order. In the case of two if/else statements that are independent of one another, it is possible to have application application, application destruction, and destruction destruction operations in parallel. Figure 11 depicts the double destruction case. Both failed if nodes and their dependents are removed from the graph, leaving the two else nodes to be executed in the next cycle.

A push operation replaces the expansion node with the sub-graph nodes. The inputs of the subgraph are replaced with the expansions inputs, and the final node of the sub-graph is linked to the expansion’s dependent. Since an expansion node doesn’t affect surrounding nodes, a push operation may be applied with any other operation.

Figure 13 depicts destruction and push operations executed in parallel. It is assumed in this image, that the successful if node which led to the expansion
node was executed previously, allowing both the expansion and else to execute in parallel. This is possible only if there was another available node when executing the if node.
Figure 14 depicts simultaneous push operations. This example further demonstrates that, although both sub-graphs use the same variable names, they are provided with subgraph-specific pseudo names. The names unique to the sub-graph also allow the same subgraph to be pushed in parallel and/or more than once within a single program execution, supporting recursion. The unique names are implemented as offsets, however, this will be further elaborated on in the simulation implementation section of the thesis.

Although the examples above only depict two parallel operations, it is possible for an infinite number of parallel operations.

2.4 AGP SYNTAX

Although AGP is a programming model and represents a method of computation, an AGP language was developed for testing.

Syntax describing on an AGP implementation. <op> (operation), <const> (constant)

```
<graph> ::= <subgraph> <graph> | <subgraph>
<subgraph> ::= "subgraph" "(" <identifier> ")" <node_list>
<node_list> ::= <node> <node_list> | <node>
<node> ::= <identifier> "<-" "input" ";"
    | <identifier> "<-" <expr> ";"
    | "output" "<-" <identifier> ";"
<expr> ::= "(" <op> <identifier>|<const> <identifier>|<const>
    | "(" "expands" <identifier> ":" <node_list> ")"
```


Computing is an advancing field, quickly evolving to meet the needs of different industries as they are faced with new and unique problems. The constant, however, is the goal of faster processing and power reduction. Recently, heterogeneous systems have become a common approach to both achieving better performance and power savings. A prime example is computers with both a CPU and GPU. This is likely the most common example of a heterogeneous system. The CPU, which is capable of completing complex instructions, is not very efficient at repetitive and parallel computations. Visual computing, data mining, machine learning, and data analytics have become common computationally heavy programs. The GPU’s unparalleled performance in operations per second, memory bandwidth, and relative energy savings make GPUs ideal for the aforementioned applications [7].

GPUs are of a superscalar design. Superscalar architecture is a pipeline architecture that contains multiple execution units that are assigned independent instructions at runtime to execute in parallel [57]. This requires complex hardware capable of scheduling independent instructions in a timely matter at runtime[48]. Scheduling instructions too slowly, or scheduling dependent instructions can lead to the pipeline stalling, greatly reducing performance. Although complex, potentially large and power-hungry, it’s a very effective method of parallel computing as seen by the GPUs and RISC processor’s success.

However, there exists an alternative solution which maintains simple hardware implementation at the cost of complicating the software, more specifically the compiler. This is of course referring to the Very Long Instruction Word (VLIW) instruction set architecture. VLIW reduces hardware complexity
by using static instruction allocation at compile time[48]. More specifically, the
compiler, aware of all available execution units in a given processor, breaks
down tasks into subsets of fixed-size instructions that can each be processed
in parallel on the available execution units, taking full advantage of ILP[17].
During runtime, a processor need only fetch the next set of instructions, hence
the very long instruction word, and execute it. This allows processors to main-
tain a simple hardware design. However, the major drawback of VLIW is that
programs require recompilation for each different processor type a program
may be run on as they contain different execution units and opcodes [48].

The VLIW instruction set architecture is similar to AGP, as they recompile
code into subset instructions, seeking to take advantage of static scheduling
and instruction level parallelism. However, where VLIW compiles a program
into a series of tightly coupled parallel fine-grained instructions [17], AGP
compiles a program into a graph of nodes, where each node represents a coarse
grain single instruction, that can be loosely scheduled in parallel with other
nodes due to the lack of shared memory. The abstraction of the program into
single instructions presents ILP opportunities in every program. However, like
VLIW, AGP will then require further compilation into the target hardware
instruction set.

Although superscalar and VLIW architectures have become a focus and are
extensively applied, further development of new programming models and
computer architectures which seek to find alternative solutions, such as AGP
and this thesis, still exist. One of these alternative solutions is the dynamically
reconfigurable computing architectures (DRCA) which is considered by some
the future of computing [39][53]. Reconfiguration allows a DRCA to match the
throughput and power consumption of ever-changing program loads. Among
the various DRCA architectures, coarse-grained reconfigurable architecture
(CGRA) is a promising type.
CGRAs are software-driven reconfigurable processors, specifically driven by data flow [53]. Similar to the architecture presented in this thesis, CGRAs use a message passing architecture. However, CGRAs use a master core, either a RISC or a very long instruction word (VLIW) processor, coupled with a reconfigurable matrix. The reconfigurable matrix is comprised of a series of processing elements (PE) with either buses or segmented interconnections as seen in figure 15. Buses offer deterministic timing performance, however, segmented interconnections allow a PE to route to any other PE [2]. The processor is in charge of executing the un-accelerated portions of applications while the matrix provides substantial parallelism and speedups for parallelizable chunks of a program. The complication of this new architecture is that all designs require novel programming models or specialized compilers to take full advantage of the hardware by properly mapping instructions.

Notable examples of CGRAs that demonstrate promise are ADRES [38], MorphSys [46], and BilRC [2]. ADRES uses a VLIW processor tightly coupled to the reconfigurable matrix, sharing functional units which act as a bridge between the two seen in figure 16 (a). The authors report that the tight coupling and use of a VLIW processor improves instruction-level parallelism, reaching 2- to 4-fold speedups over RISC-based CGRA designs. In contrast, MorphSys uses a RISC processor loosely coupled to the reconfigurable matrix, utilizing the matrix as an accelerator as seen in (b). The authors present a code mapping compiler which they report offers significant acceleration to other FPGA
and RISC systems. Finally, BiRRC offers a reconfigurable matrix that, unlike the other two CGRAs, uses segmented communication, a novel programming model, and a compiler. The matrix uses three types of PEs, removing the need for a scheduling processor, similar to the MPMC design presented in this thesis.

Figure 16: Simplified recreation of the ADRES and MorphSys architectures.

Similar to BiRRC, the design presented in this text uses a novel programming model with an accompanying compiler. AGP, a novel graph programming model, converts higher-level languages into a graph. As this is a new approach, there are no hardware designs purposefully made for this problem. However, graph processing is not a new concept. Both industry and researchers utilize graphs to better visualize large, complex data banks. Therefore, we will compare designs purposefully built to improve data graph processing instead.

GPUs, with their impressive throughput on large datasets, once again seem like an ideal accelerator for graph processing. APIs and software frameworks have been developed to better utilize GPU resources and capabilities for graph processing \[21, 33, 34, 44, 52, 61\]. These improvements revolve around memory management and operation scheduling. CuSha \[33\] applies G-shard, a graph organization algorithm, which uses a SIMT model. SIMT, similar to SIMD, applies single instructions to multiple threads of computation. The authors recognize that this model suits GPUs very well and report an increase in GPU utilization by 57% and speedups of 7.21x compared to common GPU accel-
eration techniques. Enterprise [34] is another programming model that uses breadth-first search (BFS). The authors present a novel thread scheduling algorithm, a workload balancer, and BFS optimizations to rectify the fact that BFS often performs poorly on GPUs. They report improvements of 37x compared to other BFS implementations.

Medusa [61] and MapGraph [21] offer APIs to help improve code production and performance. Coding parallelism is very difficult and requires expertise and time. The Medusa API, based on their novel programming model EMV (edge, message, vertices), utilizes message-passing based processing to allow parallelization on a single or multiple GPU systems using the same API and code. On the other hand, MapGraph dynamically chooses different available scheduling strategies based on graph size. Both report increases in coding productivity and reasonable performance increases as well.

Although promising results are seen in software solutions utilizing GPUs, the acceleration is still bound to the hardware. GPUs suffer from control and memory divergence, load imbalance, and superfluous global memory accesses caused by the irregularity of graph processing [24]. Furthermore, although GPUs are considered power efficient [7, 9, 23], they still consume an exorbitant amount of energy. [23] gives, as an example, the Summit supercomputer which uses 27648 NVIDIA Volta GPUs. Capable of a peak performance of 200 petaflops, the Summit requires a 13 million watt power supply. The Volta GPU processing unit has already reduced the power consumption of the previous NVIDIA GPUs by 50% [9]. While this may be an extreme example as Summit is a supercomputer, GPUs are only applicable to well-structured and parallel applications. To achieve speedups in more specific or more power-constrained applications, application-specific integrated circuits (ASIC) and field programmable gate arrays (FPGA) are used.

ASICs became very popular due to their incredible speedups and power efficiency. As the name suggests, ASICs are designed using a hardware language
for a specific application. The speedup capabilities of the design are almost solely implemented in the hardware. This makes them very inflexible once cast on the silicon. Furthermore, the time, expertise, and cost of designing a single ASIC chip is extremely high. As such, many companies and researchers have begun turning to FPGAs to meet the ideal performance, cost, and power consumption balance [4, 53]. It is often only in embedded systems with high-performance needs and very limited power restrictions that the industry turn to ASICs. Despite this, there are still ASIC graph accelerators presented [3, 6, 25, 43, 62]. Each, despite their narrow scope, offer incredible speedups and power savings for the process that they are specifically designed for.

FPGAs are re-programmable chips that offer a variety of resources in a limited capacity. Due to the limited capacity, FPGAs usually adopt a pipelined MISD model. This means multiple data can be processed simultaneously at different pipeline stages with parallelism being achieved by providing multiple pipelines. A big advantage of FPGAs is that they offer the ability to reprogram post-silicon, leading to a cheaper and simpler development cycle as seen by the numerous solutions offered for graph processing alone [11, 12, 15, 28, 30, 32, 36, 42, 47, 56, 58, 59, 63–65].

[64] implemented a processing unit based on a new programming model. The high-level synthesis (HLS) tools used to program FPGAs permit researchers to prototype and test more abstract designs, often better described in higher-
level languages. Similarly, softcore implementations which support distributed graph programming and offer high parallelism were studied in [30]. Alternatively, FPGA clusters, a cloud-oriented implementation offering more scalability, is presented in [47, 58], providing better support and speedups for large graphs. These more general designs often provide flexibility across multiple algorithms as compared to accelerator designs. However, this comes at the cost of performance.

[65] uses the FPGA as an accelerator in their heterogeneous architecture which tightly couples the FPGA and CPU through the cache memory, permitting higher cooperation and parallelism compared to past designs. [11][12][59] increase the performance of existing graph accelerator implementations by increasing data reuse on block RAM, the high bandwidth FPGA onboard memory, by using dedicated placement and fine-grained partitioning. There is also the integration of PIM to make up for the lack of memory resources on board [32].

FPGAs are very good for fine-grain applications, but show their limitations with coarse-grained applications [2]. It is reasonable to believe this is why FPGAs are not often used for general processing improvements. Despite this, [47] designed a prototype graph processor composed of a control processor connected to multiple node processors which are connected through a global communication network.

![Figure 18: Graph processor architecture. This design forgoes shared main memory by using a message-passing design. [47]](image-url)
The design forgoes cache memory and opts to distribute the large matrix representation of the graph across the processor nodes. Rather than use traditional instruction sets, the processor nodes use matrix operations. The communication network is a high-bandwidth, low-power network that, unlike traditional communication networks, sends small messages containing one matrix element, row, and column values. Communication is routed using static routing which, in combination with small messages, greatly increases throughput while reducing overhead and power consumption. The control processor serves only for timing and providing graph instructions. All of the processing, memory, and communications are handled by the node processors which are designed specifically for matrix computations. The authors report magnitudes of speedup compared to conventional processor designs.

Considering the articles presented above, CGRAs and FPGA designs, specifically, the BilRC and FPGA design from [2], are the most similar to the MPMC architecture presented in this thesis. Like BilRC, MPMC uses segmented communication among multiple PEs, removing the need for shared memory. Similarly, our MPMC architecture distributes the graph among the processing nodes rather than using shared memory as in [2]. However, unlike both designs, the graph is abstracted into a list rather than a matrix for more general processing capabilities and seeks to remove the master core in charge of timing and scheduling.

However, to find the most similar design, we must look back in the past to the transputer[54]. The transputer is a computer architecture comprised of memory, processor and point-to-point communication links among all processors. This is reminiscent of the MPMC architecture presented in this text. Additionally, the transputer utilizes an execution list containing current and ready-to-run tasks. However, unlike the MPMC architecture, every processor in the transputer is linked to one another rather than segmented communication. Furthermore, the transputer seeks to take advantage of TLP, while our
MPMC architecture in tandem with AGP uses ILP. It is the granularity of the parallelism that we hope will make the difference in our implementation, allowing it to succeed where the transputer hasn’t.
MESSAGE-PASSING MANY-CORE ARCHITECTURE

This thesis presents an MPMC architecture that seeks to take advantage of the parallelism and asynchronicity that AGP offers. The architecture serves firstly as a method to validate and verify the parallelism of AGP, and secondly as a proof-of-concept of a new hardware design. Should both prove fruitful, it will also provide insight on possible bottlenecks and speedups gained from different node allocations for future node allocation algorithms and hardware designs. AGP[20] is a novel programming model that offers interesting parallelization and asynchronicity possibilities, however, graph processing models are not very common. It takes specialized hardware to take full advantage of the capabilities a model offers. The MPMC architecture uses a SISD model in a many-core format, leading to a MIMD model. The MPMC presented in this thesis is functional simulation, meaning rather than implementing a cycle-accurate model of a particular architecture, we implement a functional implementation that abstracts low-level details, preserving only essential aspects of AGP-processing to evaluate the paradigm outside of its confinement to any particular architecture.

4.1 GENERAL ARCHITECTURE

The MPMC architecture is structured in a grid-like fashion with communication links to all neighbouring processors as seen in figure 19. The number of cores is flexible, able to support any square number of cores greater than 1. A
square number of cores reduces the overhead of otherwise complex communication routing algorithms.

Communication is implemented using FIFO queues. Each core contains a single FIFO queue that all bordering cores may write to. To send a message to a non adjacent core, the sending core refers to its routing table to determine which core is the fastest path. The binary routing strategy used has cores prioritizing horizontal communication before vertical. Vertical may have been prioritized, however, as the current architecture is implemented in software, the direction doesn’t provide any difference.

Messages are sent as 64bit packets; 32bit destination address and 32bit data payload. If a message requires more than a single packet, they are sent consecutively within a single system tick, were a system tick refers to a single loop in the functional simulations state machine. This ensures the order of messages is maintained, removing the need of packet linking.

The message passing model removes the need for shared memory among the cores. Results and any other pertinent information is propagated exclusively to the cores that require them. Results are then stored locally within the consuming nodes.
Cores operate individually from one another; blind to the operation or state of the others, providing a fully asynchronous system. This is attained thanks to the node structure and graph to list translation developed.

4.2 NODE STRUCTURE AND ALLOCATION

AGP provides a compiler which translates an AGP language into a single list representation of the graph. The AGP language is a bare-bones prefix language that was provided for early testing, however, isn’t intended to be programmed by humans.

The list produced by the compiler maintains the shape of the graphs by linking nodes using list offsets. This works well for single core execution, however, poses a problem for the MPMC architecture. As there isn’t any shared memory, the list must be shared among the cores using a different strategy. In an effort to reduce memory usage and runtime scheduling, the node list is segmented among all the cores. No two cores may own the same node. Due to the mutual exclusivity of nodes, cores can process any available nodes regardless of the other cores’ states.

The segmentation of the list does however requires additional efforts to maintain the shape of the graph. This involves linking nodes that reside in separate lists within different cores. It also requires a method of updating any list involved in an expansion, all while avoiding any synchronization. This is achieved by adding the node size, original list offset, relative offset, and destination core numbers as node elements.

All of the nodes within a program are allocated to the cores before the simulation begins. More specifically, a node allocation strategy produces an array that associates a core number to nodes. The advantage of static allocation is that the core position of every node is known. This means that, with a little pre-processing, propagating results becomes trivial by adding the core num-
ber of all destinations to each node. When adding a node to a core’s program memory, the node size and list offset position is recorded and added to the node. Additionally, the sub-graph offset, which serves as the original relative offset, is added at the end of the node. Finally, for each destination of the node, using the destination offsets and colouring array, the core number associated to the destination node is added. The final structure of the node when added to the program memory can be seen in Figure 20. The expansion nodes are much larger than regular nodes because expansion nodes contain the input nodes, subgraph output nodes, and output receiving nodes’ core numbers.

![Node structure](image)

Figure 20: Node structure: Each node in the image contains two inputs and outputs. However, a node may contain more or less of either. A node may have no inputs if it’s a variable. A node will always have at least one output.

Using the relative offset and the added core numbers, it is possible to translate a graph into segmented lists and allocate them among the cores. The graph shape is retained using a combination of the relative offset, original list offset, and original destination offsets.
The relative offset is an offset range each core is given to ensure no offset overlaps and further provides the node version. An expansion requires the addition of nodes to the list. Since it’s possible to expand the same sub-graph multiple times, all the offsets within the sub-graph nodes must be refactored to avoid any overlap. This is simple during single core execution as the largest current offset can be used as the factor of translation. However, during multi-core execution, this would then require the synchronization of every core during every expansion to ensure that the largest offset is up to date. This would pose a large bottleneck, especially in programs that have multiple sub-graph calls.

Instead, a range of offsets that can be used as the translation factor is assigned to each core. This is determined by program size * core number and is incremented by program size * the number of cores. This ensures that each core is given a translation factor that will never overlap with any other core regardless of the sub-graph being expanded. In turn, this allows any number of cores to expand any or the same sub-graphs simultaneously. The relative offset further provides a node version as each expansion, even of the same sub-graph, is guaranteed a unique relative offset.

The graph structure is preserved by adding the relative offset and destination offset. The linked node can then be determined by verifying if the address is between (relative offset + node offset) and (relative offset + node offset + node size). Should the message contain a result, the exact cell within the node can be determined by subtracting the node offset from the destination offset.

The core number is used for routing, and in turn, reduces the scope of the node search to a single list. A message address is then composed of a core number and address determined by adding the relative offset and destination offset. It is important to note that unless it’s a sug-graphs output receiving node, nodes are linked to nodes within the same relative offset.
Since each core operates as a black box to one another, it’s important that the cores used are capable of operating in a single core configuration. A single core is comprised of a program memory, list, routing table, message FIFO, and state registers.

![Figure 21: Single core block diagram.](image)

The program memory is a read-only memory that contains the original version of all nodes allocated to the given core. This in turn means that the greater number of cores, the larger a program can be. During processing, if an expansion is encountered, nodes from the program memory will be copied into the list and refactored accordingly. The list contains all active nodes. Active nodes are all nodes from the main function and any subsequent expansion call. Active nodes may either be ready to run nodes, or be waiting for dependents. Dependents (i.e results from other nodes) are shared using message passing. The routing table contains the communication routing information needed to propagate results to another core as efficiently as possible. The message FIFO queue is the input buffer of a core. A core has a single FIFO, which each bordering core can write into. The state registers contain the current instruction, and later, the next instruction the core must run that cycle, the relative offset, and the three list pointers: sp, sp_top, and lp.
Sp points to the current node being executed. It also serves as a pointer used to traverse the nodes when searching for a ready-to-run node. Sp top always points to the top node in the list. The nodes in the list grow from the bottom up, however, nodes are added in reverse maintaining the graph’s top-to-bottom order. This can be seen in figure 22. The first node is node 1, pointed to by Sp top. Any subsequent nodes added to the list will be added above node 1 in a similar order. Maintaining the top-to-bottom order of nodes means that oftentimes, the node pointed by Sp top is a ready-to-run node.

At the top of the list, growing down is a pointer list, pointed to by lp. The pointer list contains the list position, node relative offset, and size of each active node in the list. This is used to quickly search the list for a specific node. It also functions as a sanity check during the simulation by providing node list positions.
The cores’ execution is implemented generically, meaning that it operates the same regardless of the configuration and or the number of cores. Once the node allocation is complete and the routing table generated, a core continues indefinitely until the end of simulation node is ran. The end of simulation node has, as dependents, every output the user wishes to process. For example, if a program calculates both the multiplication and addition of two inputs, the result of both the multiplication and addition node would be linked to the end of simulation node. This ensures that every required node has been processed since the order of execution is not guaranteed.

Figure 23: Flow chart of a single cores’ logic. Note that this flow chart doesn’t represent a single execution step

Figure 23 depicts the general execution of a core. After adding all nodes from the main program into its list, the core enters a loop. The first step in the loop is communication. Communication has the highest priority as, without it, routing messages would become a major bottleneck. If the queue is not
empty, the core will pop the first message. If the message destination is not the current core, it will route the message towards its destination. Otherwise, it will unpack the message and check its type. There are three types of messages, each updating the list in some fashion: result, expansion, and mark as dead.

- The result message is equivalent to a write request. When encountered, a core will search the node list. If the node is found, the result is written into the corresponding node variable cell. If the node is not found, the message is re-queued. This occurs if one core has expanded and the other has not.

- An expansion message is a broadcast sent by a core that has encountered an expansion node. This notifies all other cores to update their list. The detailed execution of this message can be seen in the right flow chart of figure 24. The receiving cores will continuously pop messages from the queue until the entire expansion message is recorded in a temporary buffer. It will then complete the list updating process. This will soon be elaborated on below.

- Mark as dead messages are used to propagate destruction. A mark as dead message is first sent from a failed if/else node. Until a merge node is reached, a mark as dead message will be sent for every destination of a marked node. The marked node is then removed from the list.

If the message queue is empty, the core will move on to processing a node. There are four node types, namely application, destruction, push, and simulation end. Simulation end, as mentioned above, terminates the simulation when all outputs have been received. Applications, also referred to as atomic operations, are simple operations that calculate a result. This also includes successful if/else. An application node is the simplest of all nodes, calculating a result and sending a result message to all node destinations. Then, it is removed from
the list. Once complete, the core will check its queue once more, and the cycle continues.

If/else nodes, as mentioned, may result in an application or destruction. When an if/else node is encountered, a boolean result is calculated. Both the if and else node contain opposing test conditions to allow simultaneous or out of order execution. If the boolean result is true, the node will propagate its result and be removed from the list. The result of an if/else statement is either a constant or a variable. If the boolean result is false, meaning it is now a destruction node, a mark as dead message is sent to all of its outputs regardless of whether the current core owns one of the destination nodes. This means that during single-core execution, a core may experience some communication due to mark as dead messages. It is a simplification that further supports parallelism by sending out all mark as dead messages before removing some of its own nodes.

Expansions are the most involved process in the simulation. An expansion spans three states, each possibly taking multiple cycles. Figure 24 depicts the process when an expansion node is encountered and when an expansion broadcast is received, on the left and right, respectively.

When a core encounters an expansion node, it pushes all of the nodes it owns in the called subgraph into its list. It will then refactor all of the nodes’ sub-graph offsets with the correct relative offset. Once all of the nodes are added and refactored to the list, the core will search its list for expansion input nodes. If any are found, the corresponding input value will be written into its result cell. The core will then seek to link the subgraph output nodes to the nodes that are dependent on the expansion node by modifying the output list offsets, the relative offset and core numbers of the output nodes. During these three steps, the calling core compiles an expansion message packet containing the subgraph to be expanded, this expansions relative offset, the input nodes that haven’t been found, and all of the outputs that have not been linked. This
message is then broadcast to all cores. The called cores will proceed through the same steps minus the message broadcast. It’s important to note that a calling core may not own any of the subgraphs nodes. In this case, the core continues through all the previously described steps, compiling the broadcast message.

A list update is an asynchronous task. Each core may complete the update on its own time. However, since communication has a higher priority than local node processing, the list update will often execute immediately once received.

The final process after node executions, with the exception of simulation end, is garbage collection. Considering the size of the nodes, and the limited size of the local list on each core, a garbage collection is required to increase the possible program size. A node is removed from the list when it is finished.
processing or has been marked as dead. In either case, the node is no longer needed and can be removed without any repercussions. This greatly increases the size of the programs that can be run on an individual core. Although the majority of the time the top node is the node being executed, if the node removed is not the top node, all nodes above will be shifted down to fill the gap.

4.5 PARALLEL EXECUTION

Cores operate individually, only interacting among each other with message passing. This along with the mutual exclusivity of nodes, provides a highly parallelizable and asynchronous processing environment. To better demonstrate the parallel capacity of AGP in list form, the following examples will depict every possible operation combination on two concurrent cores.

Figure 25 depicts a list containing four application nodes and three variables. Two of the application nodes have yet to receive their inputs, and the two other applications have become ready to run as their dependents are being distributed. The arrows depict the dependencies and the exact cell that the result is to occupy within the receiving node. The right side of the figure depicts the two active lists after both application operations have been applied to their respective list and nodes. Although both application nodes share the variable “b”, the result is written locally to each node, negating any contention for the resource.

Destruction occurs when an if/else node fails. The node itself and its branch are removed from the list up to a merge node. This can be seen in figures 26, 27, and 29, as depicted by the red x. A branch is pruned by following the result offsets of each node until the merge node is reached.

A push operation occurs when an expansion node is processed. The expansion node is similar to a function call. When all dependencies of an expansion
node are fulfilled, all lists will be updated. Input nodes that aren’t linked to an I/O will be added as a variable node. Figures 28, 29, and 30 each demonstrate a push operation along with application, destruction and push, respectively.

Parallel execution is achieved thanks to the mutual exclusivity of nodes in AGP, the novel segmented graph to list translation, along with the MPMC architecture. The nodes’ mutual exclusivity guarantees that any number of any kind can be run simultaneously if all their dependencies are fulfilled. The segmented list structure offers the capacity of multiple processing elements to process and update the same list simultaneously without any synchronization, while also preserving the shape of the original graph. Finally, the MPMC architecture supports asynchronous parallelism by removing the need for shared
memory. This leads to a processor architecture capable of scaling with the degree of parallelism of a program simply by adding more cores.
Although the focus of this thesis is the final version, there are a few previous iterations and lessons worth mentioning for any who are interested in further pursuing this line of research.

In a previous version we explored dynamic node allocation. As a program size increases, it becomes increasingly processing intensive to determine all possible node allocation outcomes. This makes it difficult to determine the optimal node allocation strategy. An alternative is dynamic node allocation which
is capable of allocating based on load and priority. Although more optimal, it posed multiple problems that led to the eventual abandonment.

Dynamic node allocation firstly requires additional hardware in charge of node allocation. A single scheduling unit introduces synchronicity into the system as the scheduler can only allocate a single node or chunk at a time. It was observed that this led to increased core idle time as the number of cores, and in turn requests, increased. Another disadvantage is the lack of information on node locations. When a node is allocated, its destination core positions are unknown as they have yet to be scheduled. This led to increased communication loads, and as a result, could only be propagated using result requests or broadcasts. Chunk allocation, which allocated based on destination links, was explored, however, this increased the allocation time while still requiring result requests or broadcasts.

Ultimately dynamic node allocation was abandoned in favour of static node allocation as it requires no synchronization, proved to have less overhead and required no additional components.

Another design choice that was later abandoned was the use of threads. This has less to do with the architecture, but rather the simulation implementation. Originally each core was launched as individual threads, taking advantage of the parallel processing already offered by common processors. The design was, however, changed to a simpler multiplexed solution. The lack of control in thread scheduling and the need for semaphores were deemed as unnecessary complications. The simulation itself does not need to perform; its controlled execution, however, is important.
EXPERIMENTAL METHOD

To test the architecture described above, we implemented an ISA-agnostic simulation. Should the concept prove promising, it will further serve as a base for future designs. As such, we seek to evaluate three concepts of this simulation to validate the design and provide useful information for hopeful future designs.

- Measure the speedup gained from additional cores.
- Measure and validate the effect of different node allocations among the same configurations.
- Determine causes for bottlenecks and speedups among different configurations by analyzing operation distribution and communication overhead.

Three programs, one of which has three iterations, were developed to evaluate the architecture. Double factorial, which may be referred to as DF, is a recursive program that offers little opportunity for parallel execution. More specifically, double factorial takes two inputs and returns zero if different or the factorial of x and x + 1. Cascade, on the other hand, is a program tailored to have four computationally independent streams of execution. Cascade computes the factorial of 4 independent variables twenty five times each, simulating four independent data processing pipelines. The final program is matrix multiplication of a two-by-two, four-by-four, and eight-by-eight matrix. Matrix multiplication offers many parallel computations with many shared variables. It is important to note that all of these programs were programmed in AGP.
with no computational optimizations in mind. As such, it is possible that these programs could have been programmed more optimally. However, this is not the main focus of the thesis and would provide no real advantage for analysis.

Figure 31: Full graph expansion representation of benchmark programs

The simulation begins by allocating nodes randomly to a colouring array. The nodes are refactored and placed in the program memory of each core. Cores are implemented as structures, containing each element described in block diagram 21. Each core structure is added to an array of cores which is then passed to the processing function. This function cyclically cycles through each core, executing a single execution tick. This means each core completes a single operation per cycle.

Data on the total tick count as well as the operation executed, the FIFO size and the number of bytes sent every tick is recorded. The total tick count provides the execution speed, and in turn, the performance of MPMC architecture. Unfortunately, as this is ISA-agnostic, it is not possible to compare the results to other processor implementations discussed in related work. However, as this serves as a proof-of-concept for a novel programming language, it would also be unfair to compare without further research and optimization. For this reason, single-core execution is used to provide a baseline for comparison.

There are eight operation types recorded: application, destruction, local expansion, expansion call, message routing, result propagation, garbage collection, and looking for node. Local expansion refers to the time spent updating
the list from a local expansion call. An expansion call refers to the time spent
updating the list from an expansion message. Measuring them separately pro-
vides insight into the distribution of subgraphs among cores and its effect on
performance.

Looking for a node represents two possible core states. When a core has
finished processing a node and routing messages, it begins to search its list for
a node. If no active nodes are found, it begins its search anew. This means that
looking for node represents both the time a core takes to find active nodes, and
the time it spends idle.

Data collected on the FIFO size and the number of bytes set out every tick
provides insight into the communication overhead. MPMC architectures are
built upon communication, meaning their performance can be highly affected
by it.

Each program is run one thousand times with random node allocations in a
one-by-one, two-by-two, and four-by-four core configuration. Random node al-
location casts a wide net of possible node allocations, providing critical insight
into the behaviour of the processor design. Additionally, it provides computa-
tional validity as the core must be able to execute the program correctly for
any node allocation. Computational validity was completed manually after
each simulation by verifying that the output from each iteration was identical.
This was done successfully in all programs.

Random node allocation was completed using numbers from one to one
thousand as the seed to ensure each random allocation was unique. Between
each iteration of the simulation, the core structures are freed and recreated to
avoid any possible corruption of the remaining variables. The simulation im-
plementation can be found at https://github.com/sebastienCook/Message_
passing_many_core_simulation.
RESULTS AND DISCUSSION

6.1 GENERAL PERFORMANCE

Figure 32 displays the execution time of all configurations of each program. Green represents the single-core execution, red represents the two-by-two (4-core) configuration, and blue shows the four-by-four (16-core) configuration.

The single-core execution of every program is a vertical line. This means that the core execution is deterministic. This was further tested using additional cores with the same node allocation. In all cases, given the same node allocation, the execution time and behaviour were identical every execution.

Double factorial demonstrates noticeable improvement from a single core to a multi-core configuration. Unsurprisingly, the addition of cores past the initial two-by-two configuration provides little to no benefit as Double factorial provides little opportunity for parallel execution. However, the little offered is capitalized on in a multi-core configuration.

Cascade also benefits from the multi-core configuration, depending on the node allocation. In the two-by-two configuration, it is seen that certain node allocation strategies reduce the performance, highlighting the importance of future node allocation strategies. However, cascade benefits from the addition of cores in general, seeing significant improvement by further increasing the core count.

Interestingly, a four-by-four core configuration for the two-by-two matrix multiplication provides the same, if not worst speedup than a two-by-two core configuration. Considering two-by-two matrix multiplication is the shortest
of the programs, it is likely that the communication overhead and "physical" distance between nodes on different cores proved significant. Studying the four-by-four and eight-by-eight matrix multiplication, performance once again increases with the number of cores.

Table 1: Average Execution Time

<table>
<thead>
<tr>
<th>Program</th>
<th>Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1x1 ticks</td>
</tr>
<tr>
<td>Double Factorial</td>
<td>1248</td>
</tr>
<tr>
<td>Cascade</td>
<td>375170</td>
</tr>
<tr>
<td>2x2 matrix multiplication</td>
<td>286</td>
</tr>
<tr>
<td>4x4 matrix multiplication</td>
<td>2830</td>
</tr>
<tr>
<td>8x8 matrix multiplication</td>
<td>63502</td>
</tr>
</tbody>
</table>

Figure 32: Execution time in single-core, 2x2, and 4x4 configurations. Green represents single-core execution, red represents the two-by-two configuration and blue represents the four-by-four configuration.
Table 2: Average Speedup of Multi Core Configurations in Reference to Single-Core Execution

<table>
<thead>
<tr>
<th>Program</th>
<th>2x2</th>
<th>4x4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Double Factorial</td>
<td>46%</td>
<td>51%</td>
</tr>
<tr>
<td>Cascade</td>
<td>28%</td>
<td>58%</td>
</tr>
<tr>
<td>2x2 Matrix Multiplication</td>
<td>40%</td>
<td>32%</td>
</tr>
<tr>
<td>4x4 Matrix Multiplication</td>
<td>70%</td>
<td>77%</td>
</tr>
<tr>
<td>8x8 Matrix Multiplication</td>
<td>87%</td>
<td>95%</td>
</tr>
</tbody>
</table>

Putting figure 32 into perspective, table 2 displays the average speed up from a single-core execution. Here, it is evident that the four-by-four core configuration offers less of a speedup than the two-by-two configuration for the two-by-two matrix multiplication. Positively, however, a four-by-four core configuration completing the eight-by-eight matrix multiplications offers an average speedup of 95%. Despite the shared variables needed to calculate the multiplication, the message passing multi-core architecture capitalizes on the numerous parallelizable operations.

6.2 THE IMPACT OF OPERATION TYPES

Regardless of the program, unless the node allocation is exceptionally poor, the increase in cores increases the performance. The minimum average speedup of 32% suggests that this design has potential. However, node allocation techniques can significantly impact this number. Poor node allocation may even lead to a performance worse than that of a single core. This leads to the final item we wish to analyze in this thesis: Determining causes for bottlenecks and speedups among different configurations by analyzing operation distribution and communication overhead.
6.2 The Impact of Operation Types

Figure 33: Core utilization in a 2x2 configuration of the fastest and slowest execution.

Figure 33 displays the number of ticks each core spends on a given operation for the fastest and slowest execution. Visually, it’s noticeable that in both cases, all cores spend the majority of it’s time looking for active nodes to run. However, this is more prevalent in slower executions. This becomes even more prevalent when we plot the ticks spent looking for nodes of all thousand iterations against the total tick count.

The linear shape suggests that the execution time is correlated to the time spent looking for nodes. We attempted to correlate the other operations using linear regression, however as tables 3 and 4 demonstrate, no other operation could be correlated directly. This however doesn’t mean that the other operations don’t have any impact on the performance. Time is spent looking for nodes when a node is completed. However, by implementation, the number of ticks it takes is at most equal to the number of nodes in the list. Unless, there are no active nodes, at which point it would cycle back to the top node and
begin its search anew in case a message has updated the list. This means that extended periods of searching for a node can be caused by a lack of parallelized operations, unbalanced node allocation, and/or communication overhead.

Table 3: Operation vs execution ticks correlation in a 2x2 configuration

<table>
<thead>
<tr>
<th>Program</th>
<th>DF</th>
<th></th>
<th>Cascade</th>
<th></th>
<th>2x2 Matrix</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>R^2</td>
<td>p</td>
<td>R^2</td>
<td>p</td>
<td>R^2</td>
<td>p</td>
</tr>
<tr>
<td>Application</td>
<td>0.0037</td>
<td>0.054</td>
<td>0.04</td>
<td>2.91 * 10^-10</td>
<td>0.09</td>
<td>&lt; 2 * 10^-16</td>
</tr>
<tr>
<td>Destruction</td>
<td>0.5</td>
<td>0.14</td>
<td>0.09</td>
<td>&lt; 2 * 10^-16</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Local Expansion</td>
<td>0.0014</td>
<td>0.234</td>
<td>0.02</td>
<td>1.15 * 10^-5</td>
<td>0.006</td>
<td>0.009</td>
</tr>
<tr>
<td>Expansion Call</td>
<td>0.002</td>
<td>0.136</td>
<td>0.02</td>
<td>2.11 * 10^-3</td>
<td>0.087</td>
<td>&lt; 2 * 10^-16</td>
</tr>
<tr>
<td>Result Propagation</td>
<td>0.5</td>
<td>0.144</td>
<td>0.09</td>
<td>&lt; 2 * 10^-16</td>
<td>0.5</td>
<td>0.759</td>
</tr>
<tr>
<td>Routing</td>
<td>0.0001</td>
<td>0.748</td>
<td>0.001</td>
<td>0.241</td>
<td>0.001</td>
<td>0.238</td>
</tr>
<tr>
<td>Garbage collection</td>
<td>0.5</td>
<td>0.144</td>
<td>0.1</td>
<td>&lt; 2 * 10^-16</td>
<td>0.5</td>
<td>0.76</td>
</tr>
<tr>
<td>Looking for Node</td>
<td>0.97</td>
<td>&lt; 2 * 10^-16</td>
<td>0.99</td>
<td>&lt; 2 * 10^-16</td>
<td>0.97</td>
<td>&lt; 2 * 10^-16</td>
</tr>
</tbody>
</table>

Figure 34: Number of ticks spent looking for nodes vs the total execution ticks.
A lack of parallelization, ergo synchronous computations, is caused by poor node allocation. If a given core is allocated most of the atomic operations, parallelization can’t be exploited. However, even with an even distribution of nodes, if nodes of the same graph level are allocated to the same core, parallel computations can’t be exploited. An evident example is the right graph of (c) in figure 33. This is the slowest execution of the two-by-two matrix. Here it can be seen that core four is working noticeably more than the other three cores. It’s specifically completing more applications and destructions. This possibly delays the execution of other application operations that are needed for the other cores to execute in parallel.

As the parallelizability of the node allocation significantly impacts the performance, simply an unbalanced node allocation may have a significant impact. Figure 35 plots the level of balance of the node allocation by taking the standard deviation of the number of nodes allocated to each core. Figure 35 demonstrates that apart from a visually faint increase of time as the deviation becomes larger, it poses no significant impact.

The results of the four-by-four configuration are largely the same. For this reason, the results can be found in the appendix A as the results serve only to reinforce the current analysis.

<table>
<thead>
<tr>
<th>Program</th>
<th>4x4 Matrix</th>
<th>p</th>
<th>8x8 Matrix</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
<td>0.05</td>
<td>7.27 * 10^-13</td>
<td>0.02</td>
<td>2.84 * 10^-6</td>
</tr>
<tr>
<td>Destruction</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Local Expansion</td>
<td>0.005</td>
<td>0.02</td>
<td>0.007</td>
<td>0.006</td>
</tr>
<tr>
<td>Expansion Call</td>
<td>0.046</td>
<td>5.52 * 10^-12</td>
<td>0.03</td>
<td>3.47 * 10^-8</td>
</tr>
<tr>
<td>Result Propogation</td>
<td>0.5</td>
<td>0.26</td>
<td>0.5</td>
<td>0.47</td>
</tr>
<tr>
<td>Routing</td>
<td>6.44 * 10^-4</td>
<td>0.423</td>
<td>0.001</td>
<td>0.28</td>
</tr>
<tr>
<td>Garbage collection</td>
<td>0.5</td>
<td>0.26</td>
<td>0.5</td>
<td>0.47</td>
</tr>
<tr>
<td>Looking for Node</td>
<td>0.99</td>
<td>&lt; 2 * 10^-16</td>
<td>0.99</td>
<td>&lt; 2 * 10^-16</td>
</tr>
</tbody>
</table>
6.3 The Impact of Communication

Communication overhead is another possible culprit of a performance bottleneck. Looking back at figure 33, we can see that routing doesn’t represent a significant portion of operation time. In fact, a common trend that can be seen is that faster executions have more routing. This supports the theory that allocating by node parallelizability increases performance. Nodes allocated in this fashion would incur more message routing as results would need to be propagated to other cores rather than locally for the next parallel operation.

Figure 36 plots the quantity of bytes sent versus the total execution ticks. Specifically, it plots the total number of bytes sent. This serves to determine if communication can directly be correlated to the performance, and if node
allocations which incur in more communication reduces performance. Visually, the increase of messages doesn’t correlate to performance.

Interestingly, in the four-by-four configurations, patterns begin to appear. Figure 40 plots the total bytes sent to the total execution ticks. In it, clustering can be observed in the cascade and matrix multiplication programs. This means that different node allocations result in the same or similar quantity of bytes to be sent.

This is likely due to the number of nodes in a program spread among a larger number of cores. The larger number of cores leads to fewer local computations. The size of the program matters as it reduces the number of allocations where cores have no nodes, leading to a more constant communication pattern. It should be mentioned that although two-by-two matrix multiplication is the shorter program, double factorial is the smallest as it has fewer total nodes. Its recursive nature is what makes it longer.
Figure 37: Bytes processed in relation to execution time.
CONCLUSION

Although Moore’s Law may no longer hold true, the requirement for better-performing computers continues to rise. In this thesis, we explored a novel programming model, AGP, that abstracts higher-level languages into graphs leading to impressive parallelization levels. We then presented a simulation of an unconventional processor design capable of matching the levels of parallelism and asynchronicity offered by AGP. The MPMC design utilized segmented message passing to propagate results, avoiding the common memory bottleneck. MPMC also presented a novel list structure that translates the AGP graph into flat memory that is capable of being allocated across many cores while still retaining the shape of the original graph.

The results, although only comparable to its single-core execution, demonstrate that the addition of cores leads to significant speedups given the correct node allocation. Further analysis demonstrates that the use of message passing is not significantly correlated to performance, supporting the theory that performance is driven by node allocations. Despite further exploration, however, no definite cause other than balanced node loads was found.

Despite this, MPMC has been computationally validated and demonstrates the early capabilities of AGP’s parallelization. The combination of the AGP property of infinite parallel computations given available nodes and cores, and MPMC’s use of message passing, provides a highly parallelizable system that is capable of scaling with a program’s parallelizability.

It is important to note that the MPMC is designed without any optimizations. This leaves a lot of potential for performance enhancements. One of the optimizations discussed was the addition of node sharing. Currently, only a
single core may own and process a given node. This makes the performance of a program largely dependent on the node allocation strategy. Considering static allocation performance is difficult to predict, the capability of sharing nodes based on processing load or node distances would help alleviate the dependency of node allocation. This could then further help cores adapt to varying program loads.

Depending on a dynamic process to fix a poor static process, however, may prove problematic in the future. This is why we decided to first study the effects of static allocation and determine the best node allocation strategies. The dynamic reallocation would then optimize and balance the node allocation rather than fix it.

Once a dynamic node allocation strategy is integrated, we hope to properly integrate I/O. Currently, I/O is implemented in the most rudimentary fashion, requesting input from the user when the node is encountered. However, this is not very representative of common I/O inputs. While this method serves to demonstrate the ease with which I/O can be linked to nodes, it is not practical. We hope that dynamic allocation will allow a timely response of an I/O input with limited effect on other processes by reallocating the current input node to a lightly loaded core.

A prevalent issue, if not the largest, is that the simulation cannot be compared to other hardware designs. It also cannot be used to compare AGPs performance with other programming paradigms. Using an HLS language, a hardware implementation of the simulation could be produced and tested on an FPGA. This would provide real hardware timing data.

Both AGP and MPMC have large potential for growth and optimizations. We hope that the proof of concept presented in this thesis will inspire further research in both graph-based programming and unconventional processor designs. The potential of an MPMC architecture is immense as it is capable of scaling in size to match the performance, program size and power consump-
tion needed. It also has the potential of a heterogeneous design which may be also visited in the future.
BIBLIOGRAPHY


[53] Shaojun Wei et al. ‘The principle and progress of dynamically recon- 
figurable computing technologies.’ In: Chinese Journal of Electronics 29.4 

[54] Colin Whitby-Strevens. ‘The transputer.’ In: ACM SIGARCH Computer 

[55] Liming Xiu. ‘Time Moore: Exploiting Moore’s Law from the perspective 

[56] Pengcheng Yao, Long Zheng, Xiaofei Liao, Hai Jin, and Bingsheng He. 
‘An efficient graph accelerator with parallel data conflict management.’ 
In: Proceedings of the 27th International Conference on Parallel Architectures 

[57] Fahimeh Yazdanpanah, Daniel Jimenez-Gonzalez, Carlos Alvarez-Martinez, 
Yoav Etsion, and Rosa M Badia. ‘Analysis of the task superscalar archi-
tecture hardware design.’ In: Procedia Computer Science 18 (2013), pp. 339– 
348.

[58] Jialiang Zhang, Soroosh Khoram, and Jing Li. ‘Boosting the performance 
of FPGA-based graph processor using hybrid memory cube: A case for 
breadth first search.’ In: Proceedings of the 2017 ACM/SIGDA International 

[59] Jialiang Zhang and Jing Li. ‘Degree-aware hybrid graph traversal on 
FPGA-HMC platform.’ In: Proceedings of the 2018 ACM/SIGDA Interna-

[60] Yunquan Zhang, Guoliang Chen, Guangzhong Sun, and Qiankun Miao. 
‘Models of parallel computation: a survey and classification.’ In: Frontiers 

[61] Jianlong Zhong and Bingsheng He. ‘Medusa: A parallel graph process-
ing system on graphics processors.’ In: ACM SIGMOD Record 43.2 (2014), 
pp. 35–40.


APPENDIX A

Table 5: Operation vs execution ticks correlation in a 4x4 configuration

<table>
<thead>
<tr>
<th>Program</th>
<th>DF</th>
<th>Cascade</th>
<th>2x2 Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>R²</td>
<td>p</td>
<td>R²</td>
</tr>
<tr>
<td>Application</td>
<td>0.05</td>
<td>6.8 * 10^{-12}</td>
<td>0.01</td>
</tr>
<tr>
<td>Destruction</td>
<td>0.5</td>
<td>0.87</td>
<td>0.06</td>
</tr>
<tr>
<td>Local Expansion</td>
<td>0.004</td>
<td>0.04</td>
<td>0.009</td>
</tr>
<tr>
<td>Expansion Call</td>
<td>0.001</td>
<td>0.304</td>
<td>0.03</td>
</tr>
<tr>
<td>Result Propagation</td>
<td>0.5</td>
<td>0.87</td>
<td>0.07</td>
</tr>
<tr>
<td>Routing</td>
<td>0.078</td>
<td>&lt; 2 * 10^{-16}</td>
<td>5.5 * 10^{-4}</td>
</tr>
<tr>
<td>Garbage collection</td>
<td>0.5</td>
<td>0.87</td>
<td>0.07</td>
</tr>
<tr>
<td>Looking for Node</td>
<td>0.99</td>
<td>&lt; 2 * 10^{-16}</td>
<td>0.99</td>
</tr>
</tbody>
</table>

Table 6: Operation vs execution ticks correlation in a 4x4 configuration

<table>
<thead>
<tr>
<th>Program</th>
<th>4x4 Matrix</th>
<th>8x8 Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>R²</td>
<td>p</td>
</tr>
<tr>
<td>Application</td>
<td>0.31</td>
<td>&lt; 2 * 10^{-16}</td>
</tr>
<tr>
<td>Destruction</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Local Expansion</td>
<td>0.003</td>
<td>0.09</td>
</tr>
<tr>
<td>Expansion Call</td>
<td>0.003</td>
<td>0.08</td>
</tr>
<tr>
<td>Result Propagation</td>
<td>0.5</td>
<td>0.128</td>
</tr>
<tr>
<td>Routing</td>
<td>0.01</td>
<td>6.5 * 10^{-4}</td>
</tr>
<tr>
<td>Garbage collection</td>
<td>0.5</td>
<td>0.128</td>
</tr>
<tr>
<td>Looking for Node</td>
<td>0.99</td>
<td>&lt; 2 * 10^{-16}</td>
</tr>
</tbody>
</table>
Figure 38: Core utilization in a 4x4 configuration of the fastest and slowest execution.
Figure 39: Number of ticks spent looking for nodes vs the total execution ticks.
Figure 40: Bytes processed in relation to execution time.
APPENDIX B

Figure 41: Application in relation to execution time.
Figure 42: Destruction in relation to execution time.
Figure 43: Local expansion in relation to execution time.
(a) Double Factorial  
(b) Cascade  
(c) 2x2 Matrix Multiplication  
(d) 4x4 Matrix Multiplication  
(e) 8x8 Matrix Multiplication

Figure 44: Expansion call in relation to execution time.
Figure 45: Result propagation in relation to execution time.
(a) Double Factorial  
(b) Cascade  
(c) 2x2 Matrix Multiplication  

(d) 4x4 Matrix Multiplication  
(e) 8x8 Matrix Multiplication  

Figure 46: Routing in relation to execution time.
Figure 47: Garbage collection in relation to execution time.
APPENDIX C

C.0.0.1 Double Factorial

```plaintext
SUBGRAPH(plus1)
    DATUM(x);
    INPUT(x);
    DATUM(result);
    OUTPUT(result);
    CONST(one,1);
    OPERATOR(result,op_PLUS,x,one);

SUBGRAPH(fact)
    DATUM(x);
    INPUT(x);
    DATUM(result);
    OUTPUT(result);
    DATUM(resulttrue);
    DATUM(resultfalse);
    DATUM(xis0);
    DATUM(iter);
    DATUM(nextiter);
    DATUM(xminus1);
    DATUM(xminus1cond);
    CONST(zero,0);
    CONST(one,1);
    OPERATOR(xis0,op_ISEQUAL,x,zero);
    OPERATOR(resulttrue,op_IF,xis0,one);
    OPERATOR(resultfalse,op_ELSE,xis0,iter);
    OPERATOR(iter,op_TIMES,x,nextiter);
    EXPAND(fact,MAP_IN(x,xminus1cond);MAP_OUT(result,nextiter););
    OPERATOR(result,op_MERGE,resulttrue,resultfalse);
    OPERATOR(xminus1,op_MINUS,x,one);
    OPERATOR(xminus1cond,op_ELSE,xis0,xminus1);

SUBGRAPH(test)
    DATUM(x);
    DATUM(y);
    INPUT(x);
```

[July 25, 2022 at 8:32 - classicthesis version 1.0]
INPUT(y);
DATUM(z);
CONST(zero,0);
DATUM(fact_out);
DATUM(fact_in);
DATUM(are_equal);
DATUM(result_false);
OPERATOR(are_equal,op_ISEQUAL,x,y);
OPERATOR(fact_in,op_IF,are_equal,x);
OPERATOR(result_false,op_ELSE,are_equal,zero);
EXPAND(fact,MAP_IN(x,fact_in);MAP_OUT(result,fact_out));
OPERATOR(z,op_MERGE,fact_out,result_false);
OUTPUT(z);

SUBGRAPH(main)
    CONST(x,2);
    CONST(y,2);
    DATUM(z);
    DATUM(x2);
    DATUM(y2);
    CONST(one,1);
    OPERATOR(x2,op_PLUS,x,one);
    OPERATOR(y2,op_PLUS,y,one);
    DATUM(z2);
    EXPAND(test,MAP_IN(x,x);MAP_IN(y,y);MAP_OUT(z,z));
    EXPAND(test,MAP_IN(x,x2);MAP_IN(y,y2);MAP_OUT(z,z2));
    OUTPUT(z);
    OUTPUT(z2);
    DATUM(end);
    TERMINATE(end,z,z2);

.0.2 Cascade

SUBGRAPH(fact)
    DATUM(x);
    INPUT(x);
    DATUM(result);
    OUTPUT(result);
    DATUM(resulttrue);
    DATUM(resultfalse);
    DATUM(xis0);
    DATUM(iter);
    DATUM(nextiter);
    DATUM(xminus1);
    DATUM(xminus1cond);
EXPAND(fact,MAP_IN(x,fact_in);MAP_OUT(result,t112));

DATUM(t113);
EXPAND(fact,MAP_IN(x,fact_in);MAP_OUT(result,t113));

DATUM(t114);
EXPAND(fact,MAP_IN(x,fact_in);MAP_OUT(result,t114));

DATUM(t115);
EXPAND(fact,MAP_IN(x,fact_in);MAP_OUT(result,t115));

DATUM(t116);
EXPAND(fact,MAP_IN(x,fact_in);MAP_OUT(result,t116));

DATUM(t117);
EXPAND(fact,MAP_IN(x,fact_in);MAP_OUT(result,t117));

DATUM(t118);
EXPAND(fact,MAP_IN(x,fact_in);MAP_OUT(result,t118));

DATUM(t119);
EXPAND(fact,MAP_IN(x,fact_in);MAP_OUT(result,t119));

DATUM(t120);
EXPAND(fact,MAP_IN(x,fact_in);MAP_OUT(result,t120));

DATUM(t121);
EXPAND(fact,MAP_IN(x,fact_in);MAP_OUT(result,t121));

DATUM(t122);
EXPAND(fact,MAP_IN(x,fact_in);MAP_OUT(result,t122));

DATUM(t123);
EXPAND(fact,MAP_IN(x,fact_in);MAP_OUT(result,t123));

DATUM(t124);
EXPAND(fact,MAP_IN(x,fact_in);MAP_OUT(result,t124));

DATUM(t125);
EXPAND(fact,MAP_IN(x,fact_in);MAP_OUT(result,t125));

DATUM(result);
OUTPUT(result);

SUBGRAPH(main)
    CONST(i1,1);
    CONST(i2,1);
    CONST(i3,1);
c.o.o.3  Matrix 2x2

SUBGRAPH(mtrx_multiply)  
  DATUM(a00);  
  INPUT(a00);  
  DATUM(a01);  
  INPUT(a01);  
  DATUM(a10);  
  INPUT(a10);  
  DATUM(a11);  
  INPUT(a11);  
  DATUM(b00);  
  INPUT(b00);  
  DATUM(b01);  
  INPUT(b01);  
  DATUM(b10);  
  INPUT(b10);  
  DATUM(b11);  
  INPUT(b11);  
  DATUM(c00);  
  OUTPUT(c00);  
  DATUM(c01);  
  OUTPUT(c01);  
  DATUM(c10);  
  OUTPUT(c10);

CONST(i4,1);

DATUM(o1);
DATUM(o2);
DATUM(o3);
DATUM(o4);

EXPAND(single,MAP_IN(t11,i1);MAP_OUT(result,o1));
EXPAND(single,MAP_IN(t11,i2);MAP_OUT(result,o2));
EXPAND(single,MAP_IN(t11,i3);MAP_OUT(result,o3));
EXPAND(single,MAP_IN(t11,i4);MAP_OUT(result,o4));

OUTPUT(o1);
OUTPUT(o2);
OUTPUT(o3);
OUTPUT(o4);

DATUM(end);
TERMINATE(end,o1,o2,o3,o4);
TERMINATE(end,c00,c01,c10,c11);