Mapping and Routing of Application Flowgraphs onto Network-on-Chips

Dual Degree Project Dissertation

Submitted in partial fulfilment of requirements
for the degree of

Bachelor of Technology
and
Master of Technology

with
specialization in

Microelectronics

by

Shashank Gangrade
Roll No: 12D070048

under the guidance of

Prof. Sachin Patkar

Department of Electrical Engineering

Indian Institute of Technology - Bombay

June 2017

Approval Sheet

This dissertation entitled "**Mapping and Routing of Application Flow-graphs onto Network-on-Chips**" by Shashank Gangrade (12D070048) is approved for the degree of Bachelor of Technology and Master of Technology in Electrical Engineering with specialization in Microelectronics.

Examiners

VIRENDRA SINGH

Supervisor

Chairperson

Date: 04-07-2017

Place: IIT BOMBAY

Declaration of the Academic Ethics

I declare that this written submission represents my ideas in my own words and where others' ideas or words have been included, I have adequately cited and referenced the original sources. I declare that I have properly and accurately acknowledged all sources used in the production of this thesis.

I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/ data/ fact/ source in my submission. I understand that any violation of the above will be a cause for disciplinary action by the Institute and can also evoke penal action from the sources which have not been properly cited or from whom proper permission has not been taken when needed.

Shashank

Date: 04-07-2017

Shashank Gangrade

(Roll No. 12D070048)

Abstract

With the ever increasing number of processing cores, network on chip (NoCs) has emerged as a viable answer to solve the challenges in multi processor systems on a chip. Communication between these cores are becoming an important issue and can become a bottleneck. As part of this work, we have implemented a NoC design flow, which takes user specification of data flow graph for a given application to generate mapping and routing for a given application. The design flow model for mapping to a mesh network includes the communication paradigm between the nodes, by modeling the system as a quadratic assignment problem, solved using a meta-heuristic probabilistic method. This is followed by routing algorithm, which models the network data flow as a multi-commodity flow problem, formulated as an LP and solved using gurobi. We build this NoC design flow on an existing system of NoC infrastructure designed with bluespec with updated mapping and routing logic. The results of cycle accurate simulation show a 37 % drop in packet-in-flight time for a given matrix vector multiplication application.

Acknowledgements

I would like to thank my guide Prof. Sachin Patkar, Electrical Engineering, IIT Bombay for his continued support and guidance throughout the course of my project. His inputs have always served as guiding lights in times of confusion.

I want to extend special thanks to Mandar Datar and Vinay B.Y. Kumar, PhD students in HPC Lab, Electrical Engineering, IIT Bombay for their valuable insights during the numerous discussions I had with them. I also wish to acknowledge the contribution of Gururaj Shaileshwar, an alumnus of HPC Lab to this project, whose work I have partially utilized and built on, in my Dual Degree Project.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Thesis Objective

The objective of this thesis is to explore the effectiveness of static routing and mapping algorithms to model a computation load on a multi processor system. The end goal is to build a custom system to model a given computation load onto a network on chip system architecture of any topology, using static mapping and routing algorithms, and bench-marking them over conventional ways using cycle accurate simulations.

1.2 Emergence of Chip Multi-processor Architecture

Majority of current embedded systems are using more than one processor core. This enables the designer to enhance functionality and performance of existing embedded systems. The driving force behind this trend is the capacity of integrated chips which is still growing exponentially according to Moore's law. With the current CMOS technology it is possible to integrate more than a billion transistors on the same chip. This capacity is enough to integrate hundreds of computing and memory cores on a single chip. Now that the Moore's law scaling has plateaued, frequency scaling doesn't seem to be working anymore and approach seems to be chip multiprocessor way. Designing and using such a system with a large number of cores offers a large design space and many research challenges. One such challenge is the design of an efficient on-chip communication infrastructure for these Systems on

Chip (SoCs). Network on Chip (NoC) paradigm has emerged as a competitive candidate for implementing communication in SoCs. In a NoC, cores are interconnected to each other through packet switched infrastructure consisting of a network of routers. The computation power of a multi-core SoC will depend on the number and type of computing cores and size of on-chip memory. The computational capability of such systems will also be affected by the communication capability of the on-chip communication infrastructure (NoC).

1.3 Existing work in routing and mapping

Topology and routing algorithm are two important features which distinguish various NoC platforms. Communication performance of a NoC depends heavily on the routing algorithm used. A recent survey talks about various routing algorithms [1]. Routing algorithms of NoC can be generally divided into two main categories, i.e., oblivious algorithms and adaptive algorithms, depending on whether the current network status affects path selection. Oblivious algorithms select the routing path in a stationary way, while adaptive algorithms takes current workload of network into consideration when routing. Deterministic algorithms is a subset of oblivious ones, in which the package path selection can be regarded as a function of the source nodes and destination nodes. The chosen path is preordained once the source node and destination node are determined. For example, dimensional ordered routing (DOR) is a typical oblivious routing algorithm. In adaptive routing algorithms, the chosen package paths varies by the current network status.

Source routing has been proposed to guaranteed throughput in multi processor. When source routing is used in a NoC, each core (or its interface to the network) contains a table that includes complete route information to reach all the other cores in the network to which it needs to communicate. In order to route a packet through the network, a sender resource looks up the table and adds complete path from source to destination in the packet header. The advantages being no need to calculate routing logic, but each source node has to store a lot of data, which grows as the size of network increases.

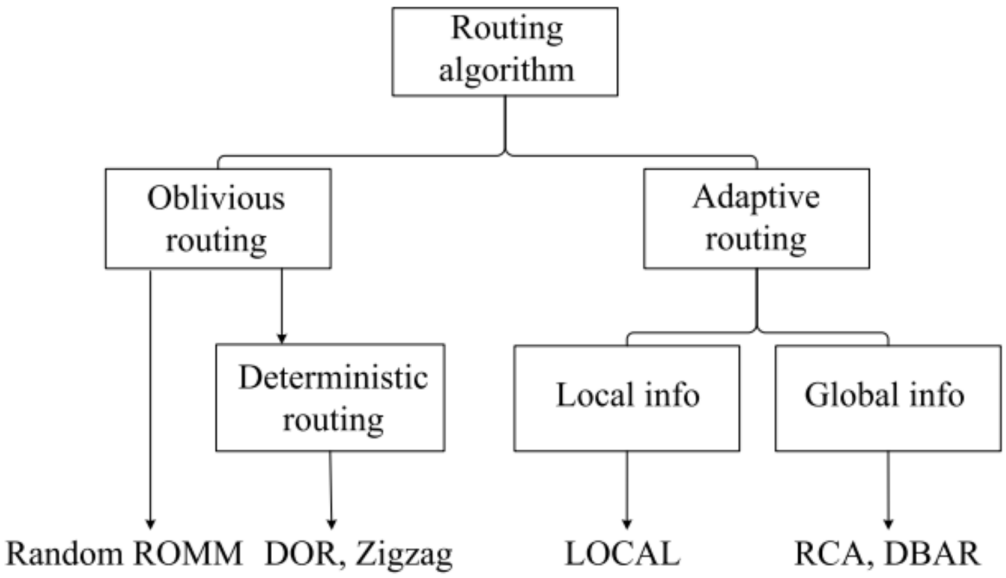

Figure 1.1: Taxanomy of routing algorithms from [1]

1.4 Organisation of Thesis

Further, the thesis goes on to describe the different aspects of the work done in **Chapters 2,3 and 4**. **Chapter 2** focusses on a algorithms for mapping and routing for a given data graph to a mesh NoC. **Chapter 3** describes the multiprocessor system developed to benchmark different routing algorithms in a network on chip architecture, using cycle accurate simulations done in bluespec. **Chapter 5** refers to DDP stage 1 work, where we looked at matrix inversion application. We had used a bordered block diagonal form to appropriately map it on parallel processing elements. The **Appendix** at the end includes code for the algorithms developed in this thesis.

Chapter 2

Mapping and Routing Algorithm

Given the current state of research in the area of projective geometry based networks, questions are being raised as to how placement and routing can be done for it. We want our system to generate placement and routing for a given application based on the data flow graph. Most of the multiprocessor system map to a mesh based interconnect network, hence we look into algorithms to map the network on an equivalent mesh network. In this chapter we discuss a complete flow for generating an optimum mapping and routing of a projective geometry based network on an equivalently sized mesh network architecture. We break this into two separate problems of placement and routing, build a flow which takes a general data flow graph and produces optimum mapping and routing. We formulate the placement problem as a quadratic assignment problem, which uses a simulated annealing approach to get an approximate result. The routing is done by formulating it as a multi commodity max flow problem, which is further solver using gurobi LP solver. We discuss a flow that automates this all to generate the resultant bluespec file (.bsv), which can directly be used for simulation and synthesis.

2.1 Background and Theory

Similar work described in a previous thesis here [2], formulates this as an optimization problem which takes manhattan distances between the mapped nodes as the cost criteria for the optimization problem. This assumes a uniform communication between all nodes. Since this doesn't completely model the network, we look into ways to add packet communication and flow as

criteria for optimization. In the next section we discuss each of this individually and look at formalism of stating the problem as a optimization problem.

Here, we take a look into graph embedding [3], it is a formalism useful to describe simulation of one network (guest) on another (host). As discussed in the book, consider a (guest) graph $G = (V_G, E_G)$ and a (host) graph $H = (V_H, E_H)$. An embedding of graph G over H is described by the following functions,

$$f_V : V_G \rightarrow V_H$$

$$f_E : E_G \rightarrow Path\ in\ H$$

with the condition that if $(u, v) \in G$ then $f_E(u, v)$ is required to be a path from $f_V(u)$ to $f_V(v)$. Given an embedding three things are defined, load, congestion and dilation. In a more general sense we can say the following about the given three terms. Load is defined as the number of nodes of graph G placed on an node in Graph H, for all nodes in H. Dilation is defined as lenght of mapped path from graph G to graph H. Congestion is defined as number of mapped paths of graph G on a given path of H. In more formal terms this is defined as.

$$\forall w \in V_H : LOAD(w) = |\ Vertices\ of\ G\ placed\ on\ w\ |$$

$$\forall e \in E_G : DILATION(e) = Length\ of\ f_E(e)$$

$$\forall e \in E_H : CONGESTION(e) == |\{e' | e \in f_E(e')|$$

The load is assumed to be one for for our system. We formulate the mapping and routing problem as an optimization problem, where the cost is to minimize the dilation and congestion of overall network.

2.2 Placement of Flow-graph Nodes

In this section we will discuss the placement problem of placing a given number of flowgraph nodes on an equivalent square network of mesh nodes. The problem here can be modelled in many ways. One of the earlier works, from a previous master's thesis [2] took the manhattan distance between the mapped locations as the criteria for cost calculation. We try to improve upon the earlier approach by adding the information about packet communication

between nodes as well to the cost criteria. A data flow graph will give the information about packet communication between each pair of nodes in the flowgraph. So for any pair of nodes P_i and P_j in the flowgraph network, mapped to (x_i, y_i) and (x_j, y_j) respectively in the mesh network, and number of packets being communicated between the two nodes is c_{ij}, the individual cost will be

$$cost_M(P_i, P_j) = c_{ij} \times (|x_i - x_j| + |y_i - y_j|)$$

2.2.1 Quadratic Assignment Problem

In the process of formulating this as optimization problem, we look into one of the fundamental combinatorial optimization problem, known as quadratic assignment problem. The definition of the quadratic assignment problem (QAP) is to allocate n facilities to n locations, in a way that has the minimum cost of all possible permutations. The assignment cost is the sum, over all pairs, of the flow between a pair of facilities multiplied by the distance between their assigned locations. This specifies intuitively that the nodes with higher flow should be placed closed together. The problem is stated as "quadratic" because the objective function once specified is quadratic with the given variables.

Given a set $N = \{1, 2, 3, ..., n\}$ and $n \times n$ matrices $F = (f_{ij})$ and $D = (d_{kl})$, the quadratic assignmemnt problem can be stated as follows:

$$\min_{p \in \Phi_N} \sum_{i=1}^{n} \sum_{j=1}^{n} f_{ij} d_{p(i)p(j)}$$

where Φ_N is the set of all permutations of N. $F = (f_{ij})$ is flow matrix, ie flow of materials from facility i to facility j, and $D = (d_{kl})$ is the distance matrix, ie d_{kl} represents the distance from location k to location l. A simple example of the same is the gicen figure 2.1

The quadratic assignment problem (QAP), one of the most difficult problems in the NP-hard class, models many applications in several areas such as operational research, parallel and distributed computing, and combinatorial data analysis. Since the problem is of the NP-hard class in computational complexity, there is no known algorithm that can solve this is polynomial time.

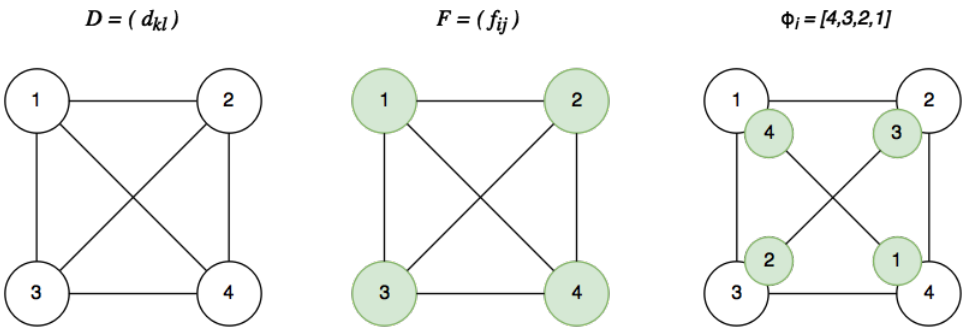

Figure 2.1: Quadratic assignment example

A very straight forward way of placing n facilities on n locations can have a total of $n!$ permutations. Each of these permutation will be used to calculate the cost based on QAP formulation given before. Hence minimum cost solution will take O($n!$), and there is no means to verify that any given solution found is the best one. Hence the problem is an NP-hard problem, and we will look into meta-heuristics in the next section to solve this in reasonable time.

2.2.2 Simulated Annealing Approach

As discussed in the previous section, we find that finding an optimal solution for the given optimization problem can be an incredibly difficult task. A recent survey [4] on quadratic assignment problem, identifies simulated annealing as one of the most widely used technique for solving quadratic assignment for different applications. As the problem sizes grows large, we have to search through a lot of possible solutions to find the optimum one. There is no known algorithm that can solve the quadratic assignment problem in polynomial time and computation times grow to fairly high values. We look into a meta-heuristic of solving this problem, using a technique known as simulated annealing.

Simulated annealing is a probabilistic method for finding the approximate global optimal value for a given function with a large discrete search space. The simulated annealing algorithm was originally inspired from the process of annealing in metal work and the algorithm employs a similar approach to

find the optimal solution.

We will discuss a simulated annealing approach described in [5] to solve quadratic assignment problem. The standard approach in simulated annealing is , a neighbourhood move consisting of swapping two facilities i and j, is analysed by computing the resultant change in objective function δ. All the moves improving the current objective function ($\delta <= 0$ are accepted, and the uphill steps of size $\delta > 0$ are accepted with probability $exp^{-\delta/T}$. As discussed in [5] above algorithm is used with a minor variation. For the fraction of starting iterations, the algorithm goes on randomly, assessing what are the maximum and minimum values of δ values. After which the temprature parameter followes a itertative decreasing rule to slowly converge at a solution. After that it follows a sequential neighbourhood search, coupled with standard negative exponential rule discussed above.

Algorithm 1: Simulated annealing using sequential search

```
Data: n, nb_iter
initialization;
nb_iter_init = nb_iter/100;
random solution;
for i=0 to nb_iter_init do
    choose random r,s in 1 to n;
    delta = change in cost for swap of r and s
end
d_min = min val of delta;
d_max = max val of delta;
t0 = dmin + (dmax - dmin)/10.0;
tf = dmin;
beta = (t0 - tf)/(nb_iter*t0*tf);
temp = t0;
for i=nb_iter_init to nb_iter do
    temp = temp / (1.0 + beta*temp);
    choose sequential r,s;
    delta = change in cost of swap r and s ;
    if delta<0 OR rand(0,1) < exp(-delta/temp) then
        Cost = Cost + delta;
        swap r,s in solution
    end
end
```

An implementation of this algorithm from [6] is used here, the solver takes a text file as input, along with number of iterations and number of retries. Some changes were made in this solver CPP code to add this to our work-flow, such as fixing a constant for number of iterations and number of retires.

2.2.3 Formulation as QAP

In this section, we formulate the placement problem at hand for a given Matrix-Vector application and formulate it for the quadratic assignment solver described in the previous section. We make some changes in the existing solver for integrating it with our work-flow of generating the mappings

directly.

The simulated annealing based solver for quadratic assignment problem takes in two matrices for distance and flow respectively, and calculates the cost basis of this. We use a python script to generate these two matrices, for all values of pg size p. Hence our work-flow is parametrized for larger mesh sizes as well. Although this can be solved for any given mesh and pg network, for our application we generate these matrices internally using Python scripts. For our problem here we consider only square mesh interconnects for simplicity. The projective geometry based data flow graph will be generated for the matrix vector multiplication, but in general the flow can take any general data flow graph. The distance matrix is constructed by calculation the manhattan distances between each of the mesh nodes in the mesh network.

2.2.4 Comparison of Results

As a rule we can compare the location results for this on the basis of two criteria, the computed cost of result and time taken to reach it. The core solver used for solving QAP here was a simulated annealing based solver, there are some other probabilistic meta heuristic solver, which have been proposed recently, for example one fast ant system. The difference in result is not really relevant at small problem sizes, but as we solve larger problem sizes we can look into comparing results from those two algorithms.

2.3 Routing of Data Packets

As discussed in one of the previous sections about graph embedding, we look into the criteria of congestion in this section. Congestion as a formal definitions refers to number of path in the PG network mapped on a given edge of mesh network. As a rule, we would want the congestion to decrease and spread out in a more uniform way. This will ensure that in hardware packets don't have to a wait a lot for the routers.

The earlier work on PG network in the thesis [2] used a simple X-Y routing for packets originating from source to destination. We wish to update this with an smart criteria of using different paths from source to destination. Ideally in a PG network for a given source and destination there is

only one path, but since we have mapped our network on a mesh network, there can be multiple paths. And instead of all the networks congesting on the same path and blocking the router, the packets can take different path from source to destination. For applications with packet size larger than the bandwidth of network interconnects, the packet has to be split into multiple smaller packets that can go on the given bandwidth of network interconnect.

In the next few sections we will discuss the flow for formulating this as a multi commodity max flow problem and discuss ways to solve it and generate final compiler readable bluespec files.

2.3.1 Multi Commodity Flow Problem

The multi commodity flow problem is a network flow problem which involves finding appropriate flows for a number of commodities from a given source to a given destination in the network.

Let $G = (V, E)$ be a directed network with cost c_{ij} and a capacity u_{ij} for each of the arc $(i, j) \in E$. For each node $i \in V$ and each commodity $k \in K$, specify the $b(i, k)$, which is supply or demand of commodity k at node i. Let x_{ij}^k denote the amount of flow of commodity k sent on arc (i, j). LP formulation for the problem can be written as,

$$\min \sum_{k \in K} \sum_{(i,j) \in A} c_{ij}^k x_{ij}^k$$

Subject to

$$\sum_{j:(i,j) \in E} x_{ij}^k - \sum_{j:(j,i) \in A} x_{ji}^k = b(i, k) \quad \forall i \in V, \ \forall k \in K$$

$$\sum_{k \in K} x_{ij}^k <= u_{ij} \quad \forall (i, j) \in E$$

The first constraint specifies the inflow and outflows at all possible nodes for a given commodity. The second constraint here specifies the limitation of totak flow to be less than the capacity of that channel.

As per the objective function and constraints given above, we formulate the multi commodity flow problem as an LP and the use the inbuilt LP solver

from gurobi, for solving this.

2.3.2 Formulation as multi commodity flow

As discussed before we descibe a work-flow for a MatrixVector Multiplication application. For formulating the problem as multi commodity flow problem, we have to reformulate it to add all the contraints and variables. As discussed before, the work-flow is paramatrized for pg of any size. We use the data distribution for matrix vector multiplication specified here [2] for generating data flow graph.

Restructuring for multi commodity flow

A general multi-commodity flow needs some initial information about the graph and commodities for solving the multi-commodity flow. The flow-graph can be parametrized for larger value of PG networks as well hence we can generate arbitrarily large sized PG and there flows here. Each data packet is assumed as an commodity for our problem. Each of vertices of a mesh node are specified as node in python script. We use the placement information from resultant text files generated using QAP in the previous section. We assume constant capacities for each arc and constant cost of flow for each of the commodity across all the arcs. For our problem we assume each commodity has to send 8 packets, hence inflow values are specified appropriately.

It is important to note that these values might not be non uniform in some of the cases of NoC and that permits us to model our network in a better way. Hence an on chip network with different packet sizes and different kind of interconnects across different arcs can be easily modelled here. It can be mapped to any complexities in mesh network like multi-FPGA or different data bandwidths across different interconnects by appropriately changing these values.

After populating appropriate data structures above, we specify the LP variables and constraints in gurobi, which generates gives the flow at each node for each commodity.

Generating Arcs from multi-commodity flow solution

The solution from the last part gives values of flows at each of the arcs for each of the commodity in a dictionary based data structure. By identifying the positive flows for a commodity we identify the arc path taken by one commodity. The arc is defined as list of edges taken by a given packet of commodity to go from source to destination node. This is then transformed to bluespec code with specification for path of each of the split packet.

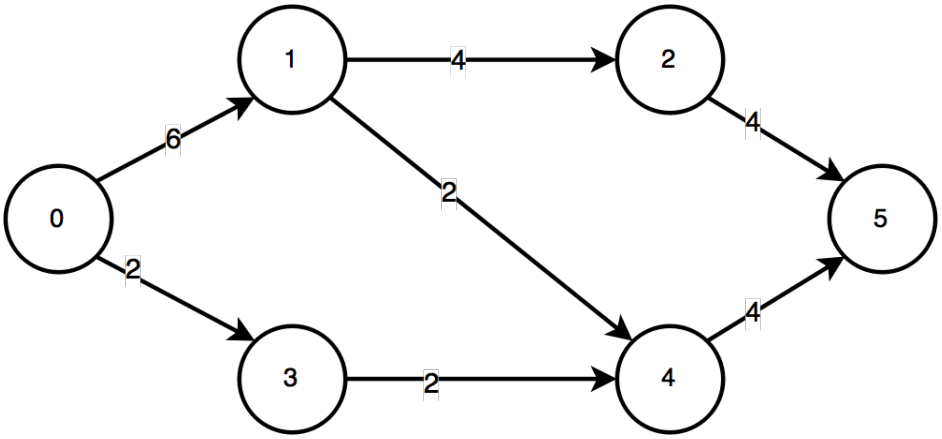

Figure 2.2: Min flow generation example

We use a python script to generate the .bsv files for this. For each commodity, we find the different arcs that the packets are taking and list all possible arcs in the mesh network. Since each commodity can have as many as 8 packets, we might have packets taking different arcs for same source and destination pair. For example in the figure 3.2, we have an example flowgraph for a given commodity with common source and destination. To find all possible arcs in this, we take an iterative approach. Find the path with minimum flow, remove it from the graph, repeat. In that way, the following flowgraph will result in the following graphs.

$$Arc0 = [0, 1, 4, 5]\ Capacity = 2$$

$$Arc1 = [0, 3, 4, 5]\ Capacity = 2$$

$$Arc2 = [0, 1, 2, 5]\ Capacity = 4$$

Hence we generate the list of all possible arcs in the network. Now for generation of hardware files, each arc can carry a given number of packets from a given source to a given destination. So at each source processor we need a lookup table to allot arc_id for every flit that leaves the node. At each intermediate node we need the destination direction for a flit of given arc_id. We populate these two tables for each source processor and intermediate mesh node respectively. These tables are added in the form of bluespec files which can be read by the bluespec simulator tool.

2.3.3 Comparison of Results

For comparing our results we can use the cost criteria specified for the LP formulation, and compare costs for both arc-routing and a simple XY routing. Since we are incurring extra logic for storing `arc_id` and directions, this can be used to serve as an benchmark deciding between the two.

Runtimes for gurobi LP solver

Here are the run times for time taken for time taken by LP solver, This does not include the preprocessing and post processing done before and after but just includes the time taken by gurobi optimize stage. The run times are generated by the solver after it finds a solution.

PG Size	Number of Mesh Nodes	Runtime in seconds
2	9	0.0017
3	16	0.0069
4	25	0.0388
5	36	0.1069
7	64	0.7793
8	81	1.6725
9	100	4.4354
11	144	16.5353
13	196	114.06212
16	289	1097.7584

Table 2.1: Runtime for gurobi solve

The run times for larger pg networks can't be calculated as the data structures to formulate LP grow to larger values, much higher than the current

resources.

Comparison of average congestion

In the table 2.2 here, we present a comparison of congestion values in the network with different routing strategy, ie XY and arc routing. As per the definition of commodity before, we assume that each commodity has to transfer 8 packets from source to destination node. The congestion value here is the average value of numbers of packets flow per arc in mesh network.

As we see, arc routing is able to spread the congestion across the networks in a more uniform way as compared to XY routing, especially for larger network, where the difference between the max and average congestion rises. This effectively translates to number of packets on a arc simultaneously, since bandwidth is limited this will result in delays and max congestion will affect the overall runtime of the application.

PG	Mesh Size	XY		Arc		
		Avg	Max	Avg	Max	Capacity
2	9	9	24	16	16	16
3	16	31	72	26	32	32
4	25	56	120	42.8	58	58
5	36	86	208	64.8	92	92
7	64	214	504	122.64	172	172
8	81	296	704	158.55	230	230

Table 2.2: Congestion values in mesh network for XY and Arc routing

Chapter 3

System Implementation on a mesh based Network on Chip

In the last chapter we proposed an improvement on the routing algorithm used when sending packets over the PG network, over the existing X-Y based routing in a mesh network. In this chapter we will build an network protocol over an existing NoC infrastructure to compare the runtime of a given application over two different routing algorithms. The network infrastructure was developed as a part of previous thesis work in [2] using bluespec. We will update the network packet design and network interface logic to include for arc based packet routing which we have proposed in the last section. We will refer to it as arc routing in the following sections of the report. We add new instructions to effectively increase the data congestion in the network. This includes a different packet design, appending new instructions to the processor ISA and changes in the network interface. We achieve this using cycle accurate simulations of the network and looking at other parameters like average packet latency. The entire system is implemented in a high level HDL, bluespec system verilog (BSV).

In the next few sections, first we will look at the existing network infrastructure, this includes network nodes, interconnects and processing element. In the next part we discuss the implementation of arc routing algorithm on the network infrastructure and understand the changes required in the network. Further next, we will understand the addition of new instruction for sending multiple packets in the network. In the last section we look at the results of simulation of both the routing algorithms.

3.1 System Architecture and Processor ISA

In this section we will discuss about the overall system architecture of the network on chip system implemented in bluespec. The network will be mesh architecture, and each node is connected to a core. Each core has a processing element, memory interface and a network interface. The processing element runs a primitive SMIPS processor. We will discuss the working of existing network infrastructure and additions made in that to account for updated arc routing algorithm and increase the packet flow in the network.

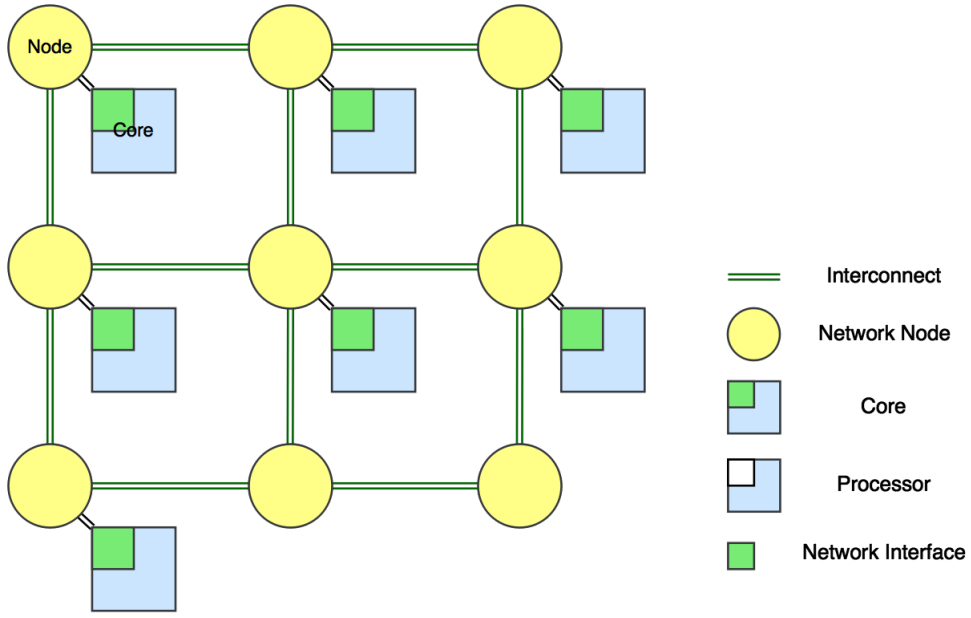

Figure 3.1: NoC System Architecture

The designed system consists of 7 cores, since we are emulating a PG matrix vector application on this network with value of $p = 2$. Each core consists of a SMIPS processor, instruction memory, data memory and a network interface. Several of such cores as per the PG size are connected to the network, which is a square mesh network in this case. The processing core here can send and receive packets from each of the 6 other processing elements. We simulate this network with two different routing strategies to benchmark one over other. The new routing strategy will be known as arc routing here, and this will be benchmarked against X-Y routing. We are also

using one updated communication based placement algorithm for placing processing elements at network nodes.

3.1.1 Processor Description

The core is runs a primitive SMIPS processor which is a derivative of MIPS32 ISA. The SMIPS ISA short for simplified is even more simplified for our use here and the instructions are listed here. Given that SMIPS is a derivative of 32 bit architecture, each of the instructions and registers are 32 bit in length. In the given ISA we implement only 8 instructions. The processor is connected to data and instruction memory through a client server protocol in bluespec, and the `LW` and `SW` instructions read and write to that memory respectively.

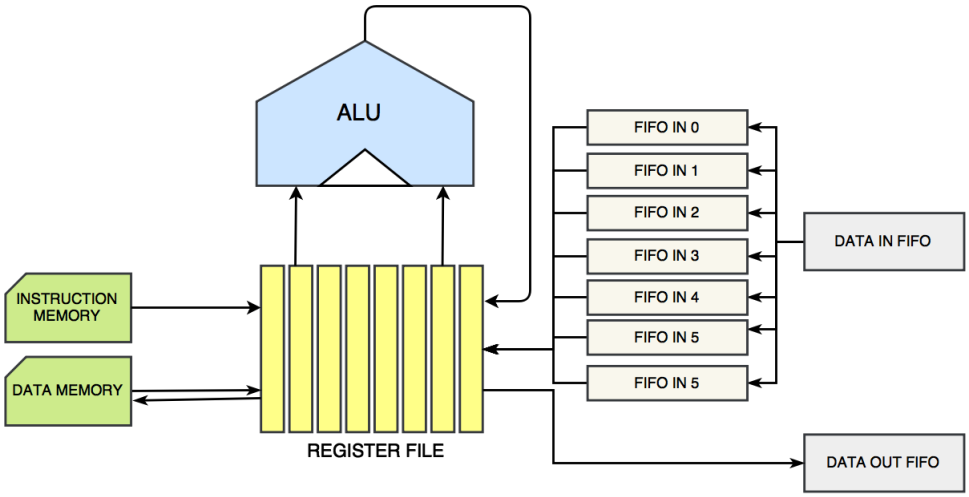

Figure 3.2: Processor Architecture

The ALU instructions take two 32 bit operands from register file and perform the given ALU operation on them. The textttFIFO_READ and `FIFO_WRITE` operation are used to connect with other processors in the network. At each processor there are individual FIFOs for each processor to read from them. The incoming packets from outside come into `DataPacketInFQ` and then copied into the specified FIFO using bluespec arbiter logic. The `FIFO_READ` instruction, as per the source processor id specified in the instruction copies the data from that FIFO to register file destination speci-

fied. The `FIFO_WRITE` instruction adds the destination processor id to the data packet as specified in the instruction and pushes the data packet to `DataPacketOutFQ`.

```
struct { Rindx rbase; Rindx rdst; Simm offset; } LW;
struct { Rindx rbase; Rindx rsrc; Simm offset; } SW;
struct { Rindx rsrc1; Rindx rsrc2; Rindx rdst; } ADD;
struct { Rindx rsrc1; Rindx rsrc2; Rindx rdst; } SUB;
struct { Rindx rsrc1; Rindx rsrc2; Rindx rdst; } MULT;
struct { Rindx rsrc; ProcID destProc; }          FIFO_WRITE;
struct { Rindx rdst; ProcID srcProc; }           FIFO_READ;
void                                             HALT;
```

We have modified the working of `FIFO_READ` and `FIFO_WRITE` instructions here to effectively send 256 bit long data over a system designed for 32 bit packet. This is done so as to benchmark out updated flow based arc routing over traditional X-Y routing. We will discuss these later in the thesis. We can assume for now that each processor will send a `DataPacket`, which is 32 bit of useful payload data and 10 bits of source and destination processor ids.

ProcID src	ProcID dest	Payload Data
5 Bits	5 Bits	32 Bits

Table 3.1: DataPacket Contents

3.1.2 Network Nodes and Interconnect

Each network core has some additional piece of logic which is the network interface, before data is sent into the network. We call this the `Netconnect` module. The `DataPacket` starts from processor and before entering the network, the `Netconnect` wrapper module adds the routing information to the `DataPacket`, this routing information can be X-Y Mesh address of source and destination packets, or arc index of a given arc as discussed in the next few sections. After adding a wrapper of routing information, the `NoCPacket`

is formed. In the network each node has a router which has two channels in each of the five directions, the four initial representing the four directions in along which a mesh node is connected to. The fifth direction is towards the processing element, discussed in the previous section. The router implements an arbitration based logic to route inputs packets to one of the output channels, packet routing is using the routing logic specified using the routing information in `NoCPacket`. The specifications for `NoCPacket` for our design will be discussed later.

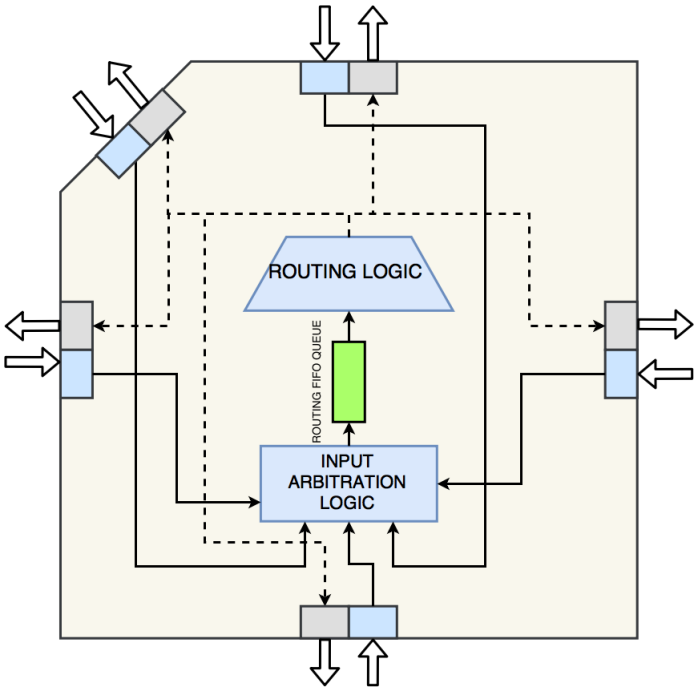

Figure 3.3: Network Node

3.2 Network Infrastructure Appends

In this section we will discuss the network infrastructure implemented, which includes specific router at each node routing logic for arbitration and interconnect with the processing element core. This will specifically focus on the flow based arc routing algorithm which we are implementing in Bluespec HDL. As seen later X-Y routing additions can be made in the same network

without changing anything in the hardware.

3.2.1 Implementing Flow based Arc Routing

As discussed in the previous chapter to reduce the overall congestion the network we implement flow based arc routing in the given mesh NoC. The idea is that each packet should go on the specified arc which minimizes the overall congestion and results in increased data flow. From the solutions of multi-commodity flow in previous chapter we have arc results for each packets. In some cases we see that packets from same source and destination go on different arcs so as to minimize congestions. This is core of what we are trying to achieve.

The solutions for multi-commodity flow result in several arcs for each of the packets to be sent. Each network arc is basically a list of NoC nodes. And each packet for a given source and destination can take multiple different arcs. This will be a lot of information to store on the hardware, and this wont be scalable as number of nodes increase for larger networks. We specify a different approach to mitigate this. Each arc is allotted one unique arc index. Each `NoCPacket` will only have this information. Based on this information, the packet will be routed through the network to its destinations.

NoCArcId	Payload DataPacket
6 Bits	42 Bits

Table 3.2: NoC Packet Contents

The `NOCPacket` is made up of just two fields. The payload `DataPacket` design has been discussed before and it is carried forward from the processor to the network. Before entering the network, the `NetConnect` module is responsible for adding appropriate `NoCArcId` to each `DataPacket`.

At Source

Using Python preprocessing discussed in appendix, the solution from multi-commodity flow are used to list all possible arcs in the network. We have built a lookup table which has to be initiated at each node. This lookup table populates the `lookupNoCArcId` function in the given code. At the given source processor, for each packet to a given destination there has to be

one unique `NoCArcId`, this arc index is basically specification of the whole path that the packet is supposed to take. It has to be noted we only store the arc indexes which originate at the node and not all the arc index.

This logic is added in the `NetConnect` module, and the `NoCArcId` is once added forms the `NoCPacket`, and the 48 bit packet is sent to the network.

```
rule addLayer ;
  dataPacketInQ.deq();
  let datapacket = dataPacketInQ.first();

  ProcId src = datapacket.src;
  ProcId dest = datapacket.dest;
  PacketLoc rindex = datapacket.data.pack_add;

  NoCArcId arc_id = lookupNoCArcId(src,dest,rindex) );
  nocPacketOutQ.enq( NoCPacket { arcid: arc_id, payload:
     datapacket} );

  $display("Packet (%d,%d)Sending at network interface: %d,
     for arc_id %d, loc %d",src, dest, src, arc_id, rindex);
endrule
```

At Intermediate Node

Each `NoCArcId` represents one unique path in the network. But storing paths for all the arcs at all the NoC nodes will be a lot of data to be stored per network node. But one node of the path doesn't need to know whole path and hence we don't store the whole path anywhere. Also the paths not passing thorough a node doesn't need to be bothered with.

Using Python preprocessing discussed in appendix, we take a given node, find the arcs passing through that node, and for each of that arc we find the destination direction for an incoming node. In this way we only have to store the destination direction of the arc index passing through this given node. This lookup table is stored as `lookupArcDest` function in bluespec. The destination can be either one of the four directions ("N", "S", "E", "W")

for intermediate node in the path, or just the home ("H") for final node in the path. This logic is added in the module, which was earlier using an X-Y based routing, instead will just look at the `arcid` for that processor and find that destination direction.

```
// Algorithm: packet is sent according to the arc path is
   supposed to be sent on
rule routePacketNorth (routePacketQ.notEmpty() && (
   lookupArcDest( thisRowAddr, thisColAddr, routePacketQ.
   first().arcid ) == "N" ));
    routePacketQ.deq();
    channelOutQ[1].enq(routePacketQ.first());
endrule
```

At Destination

Once the NoC Packet reaches the destination core, we just strip off the NoCArcId from NoCPacket and transfer the DataPacket to the processor.

3.3 Packet conversion for 256 bit data

As we are benchmarking our design for higher network traffic, we want to increase the communication traffic in the network. We assume that each data packet from processing core has to send in 256 bit of data instead of 32 bit that it is sending right now. Each transfer of packet from a given source node to a destination node will be effectively, 8 such 32 bit `DataPackets`. This will increase the network traffic eightfold, on which we will be used to benchmark the performance of both the designs.

So now to transfer data packet from a given source node to a given destination node, we will have to send 8 data packets from the processor. With each data packet enclosing a part of the 32 bit data that it needs to send. Since we are using a simple SMIPS processor, which uses 32 bit registers for all calculations, we build on the existing infrastructure to split a 32 bit data packet into 8 different packets so that effectively network has to deal with

much higher communication traffic, but for benchmarking the processing element can remain the same.

We want to implement this without making multiple changes in the already existing implementation of system. This includes the hardware Bluespec code, software assembly that runs on SMIPS and data and instruction memory which have been initialized. The 32 bits register data starts from the register file of the source processor and reaches the register file of destination processor. In the next few subsections we will specify the movement of packet along this path.

3.3.1 At source processor

At source processor we modify the `FIFO_WRITE` instruction of ISA, the instruction take two operands, the Register from which data has to be taken `rsrc` and the processor to which data has to be sent `destProc`. It adds source and destination processor index to the 32 bit payload to form the 42 bit `DataPacket`. The `DataPacket` is sent on, pushed to `DataPacketOutFQ` FIFO, which is connected to the `Netconnect` module.

```
FIFO_WRITE { Rindx rsrc, ProcID destProc}
```

So without changing anything in assembly code and data and instruction memory, we modify the `FIFO_WRITE` instruction to split the packet and push 8 packets for each execution of this instruction. Each 32 bit `DataPacket` now contains 4 bit of useful data, 3 bit of location in original data register and rest is garbage for now. We add another stage after `FIFO_WRITE` instruction which fires 8 times to send these 8 packets per data register to the `DataPacketOutFQ`.

```
tagged FIFO_WRITE .it : begin
  fifoWriteCount <= 0 ;
  fifoWriteDestProc <= it.destProc ;
  fifoWriteSrcRindx <= it.rsrc ;
  next_stage = FIFOWrite;
end
```

```
// Rules to specify packets written to output
   DataPacketOutFQ one after another as a part of FIFO_WRITE
    Instruction
Rules fifoWriteRuleSet = emptyRules;
 for (Integer i = 0; i<valueof(NumPackets); i=i+1) begin
   Rules nextRule = rules
   rule fifoWritePacketi(stage == FIFOWrite && fifoWriteCount
       == fromInteger(i) );
    Payload payload32 = ?;
    payload32.pack_data = rf.rd1(fifoWriteSrcRindx)[ 4*(
       fifoWriteCount)+3 : 4*fifoWriteCount ] ;
    payload32.pack_add = fifoWriteCount ;

    dataPacketOutFQ.enq( DataPacket { src:procId, dest:
       fifoWriteDestProc, data:payload32, isBroadcast:False }
        );
    dumpFIFOWriteDest <= fifoWriteDestProc;
    dumpFIFOWriteLoc <= fifoWriteCount;

     if (fifoWriteCount == 7) begin
         fifoWriteCount <= 0;
         stage <= PCgen;
      end
      else begin
          fifoWriteCount <= fifoWriteCount+1 ;
          stage <= FIFOWrite;
       end
     endrule
    endrules;
  fifoWriteRuleSet = rJoinMutuallyExclusive(fifoWriteRuleSet,
     nextRule);
  end
addRules(fifoWriteRuleSet);
```

3.3.2 At source NetConnect Module

One split packet can take different paths to reach to the same processor along different arcs, hence for a given source and destination processor indexes, we also need to look at the max packet count to allot arc index, we update the code to take this into account. The python preprocessing generates a look up table, which specifies a given arc id has to be allotted to a fix number of DataPackets. To implement this we simply compare the location with value given by Lookup table. This way we allot the DataPackets with correct arc index before sending them to the Network.

3.3.3 At destination NetConnect Module

Once the NoCPacket has travelled the network it reaches the destination node. In the Netconnect module we remove the NoC arc index information from the NoCPacket and send the DataPacket to the processor connected.

3.3.4 At destination processor

We modify the `READ_FIFO` instruction in the ISA to read the packets arriving from different arcs in the FIFO. As we had seen before, for packets received from each processor we had a dedicated FIFO which accumulates packets received from that processor. Here we add one 32 bit register and one count variable for packets received from each processor.

The instruction waits for the packets to arrive and as each `DataPacket` arrives, using the location data in the packet it reconstructs the 32 bit packet back again that was transmitted before. A packet count variable keeps track of how many packets have been received and `packetReg` is the 32 bit register which stores the data at specific locations. Once the `packetCount` reaches 8, the register is copied to appropriate register in register file and count is reset to 0.

```
tagged FIFO_READ .it : begin
  pendingFIFORead.enq(inst);
  next_stage = ReadFIFO;
end
```

```
 Rules readFIFORuleSet = emptyRules;
 for (Integer i=0; i<valueof(NumNodes); i=i+1) begin
   Rules nextRule = rules
     rule readFromFIFOi(stage == ReadFIFO && readFIFOArbiter
         .clients[i].grant);
       let regDest = case ( pendingFIFORead.first() )
          matches
         tagged FIFO_READ .it: return it.rdst;
         tagged FIFO_READ_BROADCAST .it: return it.rdst;
       endcase;
       dataPacketInQ[i].deq();
       DataPacket readDataPacket = dataPacketInQ[i].first();

       fifoReadDestRindx[i] <= regDest;

   if (fifoReadPacketCount[i] == 7) begin
       fifoReadPacketCount[i] <= 0 ;
       wba( fifoReadDestRindx[i], fifoReadDataReg[i]);
       stage <= PCgen;
         pendingFIFORead.deq();
   end
   else begin
    fifoReadPacketCount[i] <= fifoReadPacketCount[i]+1 ;
    fifoReadDataReg[i][3 : 0] <= readDataPacket.data.
        pack_data ;
    stage <= ReadFIFO;
    sreadDataPacket.data.pack_data ;
   end
  endrule
 endrules;
 readFIFORuleSet = rJoinMutuallyExclusive(readFIFORuleSet,
    nextRule);
end
addRules(readFIFORuleSet);
```

3.4 Simulation Results

3.4.1 Matrix vector multiplication

We run the application of a matrix vector multiplication over the designed network. The matrix vector multiplication is designed to run on PG network of parallel processing elements, where data distribution is based on PG line and point node connections based on criteria discussed here [7]. The given figure 3.4 shows a pg with p=2, showing cyclic nature of edge connections.

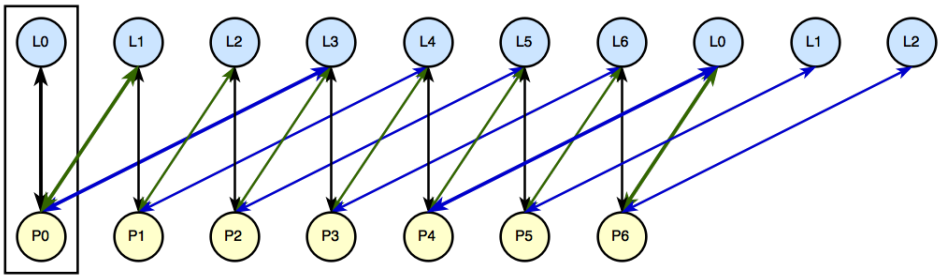

Figure 3.4: PG Network with cyclic edges

While embedding this PG network on the mesh network, the point nodes and line nodes are kept on the same mesh location and effectively merged. The data distribution and communication paradigms are based on PG connections. For example in the given PG network lets take merged point and line node P0 and L0,

1. The matrix block A_{00} and X_0 are saved at node 0 corresponding to node number.

2. The other native data for this node will be A_{46}, A_{64}, A_{40}, A_{04}, A_{60} and A_{06}. Basically all possible permutations of point nodes connected to this line node L0.

3. Send saved X_0 to node 1 and node 3 corresponding to the line nodes connected to point node P0 in PG network.

4. Send partially computed Y_4 and Y_6 to node 4 and node 6 respectively.

Following this algorithm we can build a matrix vector data flow graph for and matrix size and PG with any number of nodes. Refer to Table 3.3 for

detailed distribution.

Processor	Native Data	Computations Performed	Data Sent						
			P0	P1	P2	P3	P4	P5	P6
P0	A_{00}, A_{46}, A_{64}, A_{04}, A_{06}, A_{40}, A_{60}, x_0	$y0_{part0} = A_{00}x_0 + A_{04}x_4 + A_{06}x_6$, $y4_{part0} = A_{46}x_6 + A_{40}x_0$, $y6_{part0} = A_{64}x_4 + A_{60}x_0$		x_0		x_0	$y4_{part0}$		$y6_{part0}$
P1	A_{11}, A_{50}, A_{05}, A_{01}, A_{10}, A_{15}, A_{51}, x_1	$y1_{part1} = A_{11}x_1 + A_{10}x_0 + A_{15}x_5$, $y5_{part1} = A_{50}x_0 + A_{51}x_1$, $y0_{part1} = A_{05}x_5 + A_{01}x_1$	$y0_{part1}$		x_1		x_1	$y5_{part1}$	
P2 P3 P4 P5 P6	⋮								

Table 3.3: Matrix Vector Multiplication with Data Partitioning optimised for PG from [2]

Simulation Results

This system described above is used to run a matrix vector multiplication application distributed over 7 processors. We perform a cycle accurate simulation on this network using bluespec simulator to compare the performance. We compare the two parameters namely number of cycles taken and average packet hop. Here is a table with results from both.

We have also included the results for a slight variation of arc routing, timed arc routing by including a timing component as well. As we see that communication and computation occur in burst for a given application. Hence we solve individual multi-commodity problem given the flows at that time and generate arc lookup tables from that. Here are the results.

Type of routing	Number of Cycles	Average Packet inflight
Arc routing	362	12.3
Timed X-Y routing	365	12.8
X-Y routing	400	19.3

Table 3.4: Simulation Results

As per the results mentioned in the table, we see about 10 % improvement in number of cycles taken and packet in-flight time improves by 37 %, this

validates the effectiveness of our model for mapping and routing of a data flow graph.

Synthesis and Placement

The generated design is used to view post synthesis resource usage and mapping results using Xilinx Vivado. Using bluespec we convert the top module mkSystem to verilog, which along with data files are added to source of Vivado project.

We run synthesis of the design for Zedboard XC7Z020-CLG484-1 [8] target core. We present the synthesis resource usage as well as the mapping of modules on FPGA and analyze how mesh network is implemented on FPGA by commercial tools.

Resource	Utilization	Available	Utilization %
LUT	3751	53200	7.05
LUTRAM	140	17400	0.80
FF	3062	106400	2.88
IO	44	200	22.00
BUFG	1	32	3.12

Table 3.5: Utilization post implementation in Vivado

Also added are the mapping of modules on FPGA after implementation, we have highlighted the cores and nodes as per their connections. The node here are connected in mesh topology.

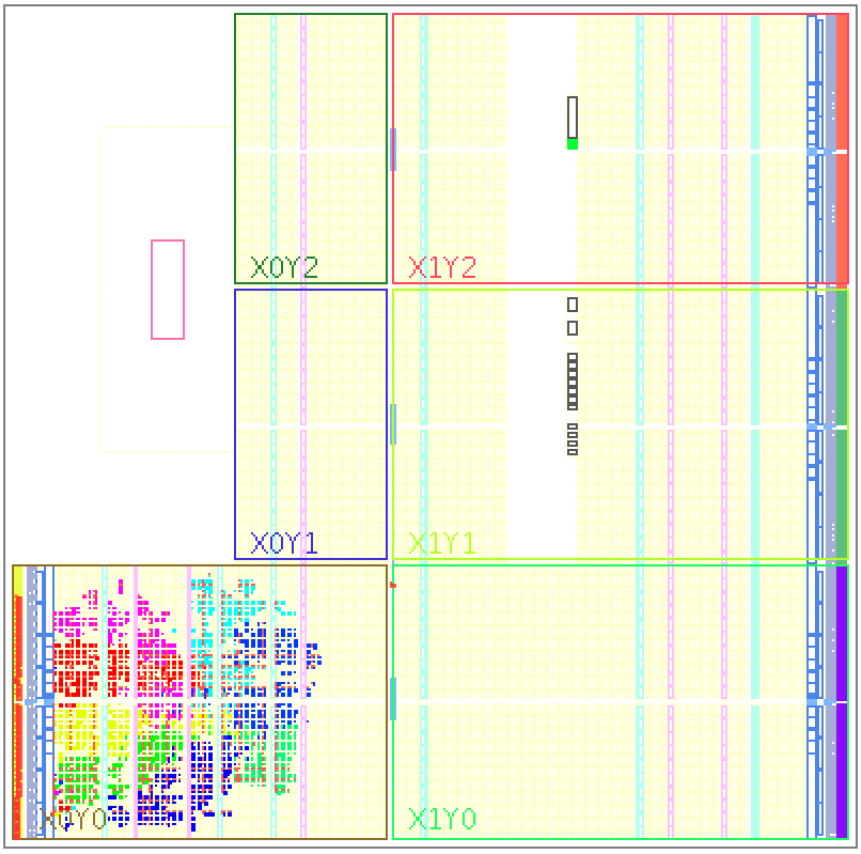

Figure 3.5: Post Implementation Deivce Map in Vivado

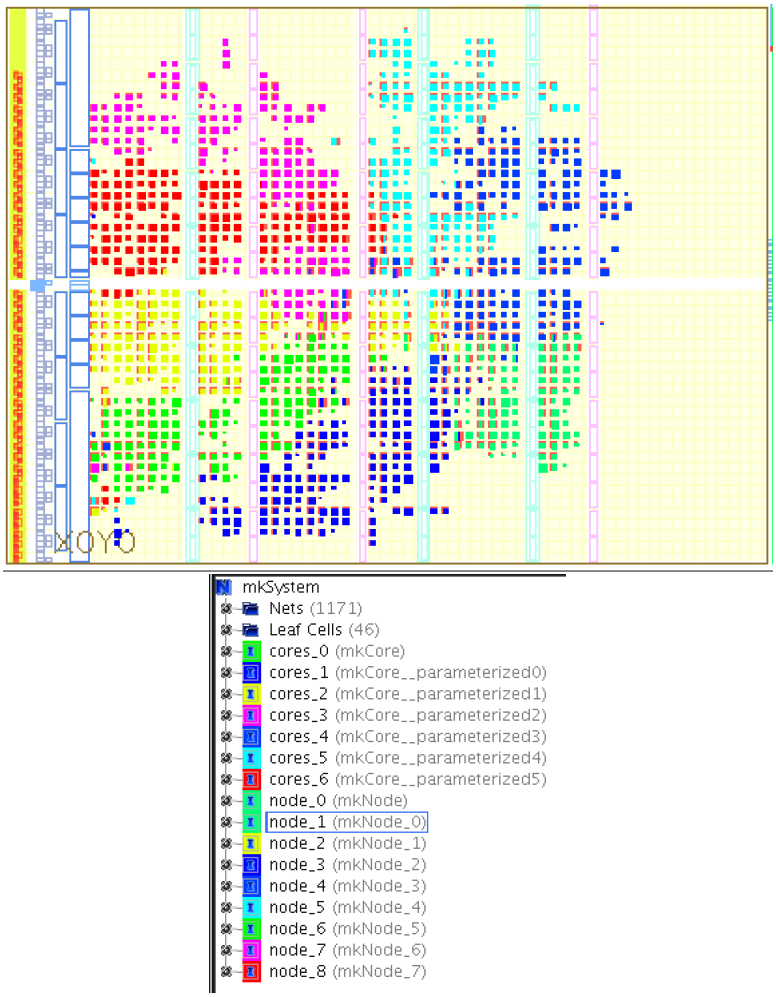

Figure 3.6: Highlighted nodes and cores with legend

Chapter 4

Application mapping of a parallel bordered block diagonal matrix solver [1]

The core idea of the work described in this report is to identify suitable partitions in circuit networks, and represent this in a special form of sparse matrix system, known as bordered block diagonal matrix. The inherent parallelism once identified can be used to run this application on an NoC based system, which we had discussed before. The bordered block diagonal matrix consists of several independed diagonal blocks. Efficient solvers for large circuit networks are important tool in VLSI circuit simulation, power system analysis and several other domains. Solving sparse linear system is the single most dominating problem in scientific computing.We describe two different flows to solving a BBD matrix equation in this paper by extracting parallelism in the BBD matrix formulation. In network analysis we may encounter different types of network elements, both linear and non-linear. We use node tearing nodal analysis greedy [9] algorithm to find node ordering of a MNA matrix such that it creates bordered block diagonal structure.

In section III, we formalize the solution of BBD matrix by doing appropriate matrix operations. In sections IV, we look at software implementation of BBD matrix solvers, extracting inherent parallelism using the multiple processing units. In section V, we present a hardware system to be imple-

[1]The work discussed in this chapter was done as a part of stage 1 of Dual Degree Project, whereby a different application for NOC was discussed.

mented on a FPGA based NoC system, describing the individual blocks and comparing results with software implementations.

4.1 Background

In this section we will refer some of the previous work done in the domain of BBD matrix solvers. In a recent paper [10], they have focussed on efficient matrix ordering strategy such that the BBD matrix formed has minimum border width. In another previous work [11], the authors have used a nested approach while formulating the BBD matrix, instead of just minimizing the width of borders of BBD matrix.

In our approach we focus on the BBD matrix generated from linear network graphs, but the solution should be valid for BBD matrix generated from any other method for any other application as well. The formulation of Bordered Block Diagonal matrix, is being described in detailed in the referred report [12], a brief description is given below. In our formulation of this problem, we take a linear circuit network, formulate the linear circuit network as MNA matrix by performing Modified Nodal Analysis. The resultant matrix equation is of the following standard form.

$$AX = B,$$

where X is the solution set of conductance matrix. So given a linear circuit represented as a graph, with n nodes and m independent voltage sources, then X forms a vector of size $[(n + m) \times 1]$, the top n variables denote the potentials at each of the n nodes and bottom m variables denotes the current through each of the m voltage sources. Now generating the elements of A matrix, which can be broken down into 4 such sub matrices. The size of overall A matrix is $[(m + n) \times (m + n)]$.

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} B = \begin{bmatrix} B_1 \\ B_2 \end{bmatrix}$$

a A_{11} is a $n \times n$ matrix whose diagonal enteries are the sum of conductance connected with node associated with the diagonal. Off diagonal entries are the negative of the conductance between nodes associated with row

and column.

b A_{22} is a $m \times m$ matrix whose all entries are zero.

c A_{21} and A_{12} of size $m \times n$ and $n \times m$ respectively are transpose to each other and consist of 0, -1 and 1 only.

d B is a $[(n+m) \times 1]$ sized matrix.

e Top n enteries consists of sum of currents corresponding to the node associated with each entry.

f Bottom m entries consists of voltage source value for each of the voltage source.

The matrix A obtanined here is not in bordered block diagonal form. We further use node tearing nodal analysis described in [9] to partition the linear network appropriately.

4.2 Problem Formulation

From previous section, we can infer that solutions for linear circuits represented as graphs with a significantly large number of nodes can be formulated as solutions of bordered block diagonal matrix. A general form of matrix is given below.

$$\begin{bmatrix} A_1 & 0 & 0 & \dots & 0 & B_1 \\ 0 & A_2 & 0 & \dots & 0 & B_2 \\ 0 & 0 & A_3 & \dots & 0 & B_3 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & A_{N\text{-}1} & B_{N\text{-}1} \\ C_1 & C_2 & C_3 & \dots & C_{N\text{-}1} & A_N \end{bmatrix} \begin{bmatrix} X_1 \\ X_2 \\ X_3 \\ \vdots \\ X_{N\text{-}1} \\ X_N \end{bmatrix} = \begin{bmatrix} G_1 \\ G_2 \\ G_3 \\ \vdots \\ G_{N\text{-}1} \\ G_N \end{bmatrix}$$

Here each of A_i , B_i, C_i are block matrices and X_i and G_i are vectors respectively of appropriate size. The number of rows is same of each of the matrices in the set A_i, B_i, X_i and G_i. And similarly the number of columns is same for A_i and C_i. Also in a general matrix of this form, the size of block matrix may be different for different values of i, but later in the

implementation we will use a common value for size of block matrices. Our aim is to solve for each of the X_i vectors given all other values. Each of the block matrices can be assumed to be sufficiently dense, and further matrix operation on blocks will be dense matrix operations.

Now solving for individual X_i, we can form $n-1$ equations of the form given below from each of the $n-1$ rows of the matrices defined above,

$$A_i X_i + B_i X_N = G_i$$

which simplifies to,

$$X_i = A_i^{-1}(G_i - B_i X_N)$$

So for i in range $[1, N-1]$ we can find each of the X_i matrices in terms of X_N. Now to determine the value of X_N, we use the following. From the last row of the matrix equation.

$$C_1 X_1 + C_2 X_2 + \cdots + C_{N-1} X_{N-1} + A_N X_N = G_N$$

Expressing X_N in terms of known variables, we get,

$$A_N X_N = G_N - (C_1 X_1 + C_2 X_2 + \cdots + C_{N-1} X_{N-1})$$

which further simplifies to

$$X_N = A_N^{-1}(G_N - \sum_{i=1}^{N-1} C_i X_i)$$

Copying the pre-calculated value of X_i from above in this equation to get,

$$A_N X_N = G_N - \sum_{i=1}^{N-1} (C_i A_i^{-1}(G_i - B_i X_N))$$

Taking X_N related terms on one side we get,

$$A_N X_N - \sum_{i=1}^{n-1} (C_i A_i^{-1} B_i X_N) = G_N - \sum_{i=1}^{n-1} (C_i A_i^{-1} G_i)$$

which further simplifies to

$$X_N = A_\sigma^{-1} G_\sigma$$

where

$$G_\sigma = G_N - \sum_{i=1}^{n-1} C_i A_i^{-1} G_i$$

$$A_\sigma = A_N - \sum_{i=1}^{N-1} C_i A_i^{-1} B_i$$

As we can see that for extremely large values of N, calculating X_N is the most complex and time consuming step, as it needs to calculate $N-1$ summations for calculating the value. The other X_i matrices can be promptly calculated once the value of X_N is known. Hence throughout the course of this document we will focus on the speed-up in calculation of X_N. Once X_N is calculated the values of each individual X_i can be calculated in the one easy step, independent of size N.

To achieve parallelism in calculation of X_N, we need to see that each individual term of sigma can be calculated independently. Let G_i^* denote the individual sigma term in numerator and B_i^* denote the individual sigma term in denominator, such that

$$G_i^* = C_i A_i^{-1} G_i, B_i^* = C_i A_i^{-1} B_i$$

Then we can clearly see the relation in calculation of individual G_i^* and B_i^* terms, as $C_i A_i^{-1}$ term is common for them. Furthermore it is to be noted that during the complete formulation of equations, all the solutions are dependent on A_i^{-1} and not on the predefined value of block matrix A_i, this gives us ample opportunities to explore for hardware software co-design to effectively calculate the solution.

4.3 Software Implementation

This sections covers the software implementation of bordered block diagonal matrix solver. The major purpose of developing a software API here is to serve as a benchmark for comparison with the performance (execution time)

results of FPGA implementation. This can also help in profiling the execution time of several steps of MNA matrix solver, clearly separating the serial and parallel parts of the program. Further in this section we will talk about the external libraries used, implementation of C code and generated results.

4.3.1 Use of External Libraries and APIs

We use external libraries for linear algebra operations. BLAS is a set of low level routines, used here for matrix addition and matrix multiplication. LAPACK is a set of advanced functions built up upon BLAS libraries, used here for matrix inversion, both written in low level FORTRAN. The compiler directives $-lblas$ and $-llapack$ are used for running such programs in C.

OpenMP refers to simple compiler extension that allow us to add parallelism in the existing code by using multiple processing units available. All OpenMP constructs in C and C++ are indicated with a #pragma omp followed by parameters. The complier directive for using OpenMP routines is $-fopenmp$.

4.3.2 Software API

In this section we describe the software implementation of solution of BBD matrix, using the libraries and APIs described in previous sections. The most important task in implementing this algorithms is understanding memory allocations and data structures to be used. We need to pass a number of matrix blocks. Each of matrix operation is carried out using the lapack and blas libraries.

Each of the block matrix is stored as an object of $Matrix$ struct. It defines three elements in the struct definition, integer variables $nrow$ and $ncol$, which define the number of rows and number of columns in that particular matrix and $matval$, which is an array of double variables of predefined size. The complete MNA matrix data is stored as arrays of objects of the $Matrix$ structs. The pointer to each of these array is passed as a variable throughout our bbd_solve API.

As stated before, we need to calculate the value of X_N first, as defined in

the previous section. We had earlier defined X_N as

$$X_N = A_\sigma^{-1} G_\sigma$$

where

$$G_\sigma = G_N - \sum_{i=1}^{n-1} C_i A_i^{-1} G_i$$

$$A_\sigma = A_N - \sum_{i=1}^{N-1} C_i A_i^{-1} B_i$$

$$G_i^* = C_i A_i^{-1} G_i, B_i^* = C_i A_i^{-1} B_i,$$

$$X_i = A_i^{-1}(G_i - B_i X_N)$$

In the above formulation, the each of A_i matrices are of the size $m \times m$, each of the B_i matrices are sized $m \times n$ and each of C_i matrices are $n \times m$. Each of the G_i and X_i are vectors of size $m \times 1$ and the number of bordered blocks is N. Each of these variables N, m and n can be varied in the code.We can see that the flow can be divided as shown in figure 4.1.

The C code proceeds in following steps:

a Define appropriate data structures to store the matrices a given standard format. We define arrays for each of A, B, C, X and G matrices. These are passed as pointers to the API function.

b Write a function to caculate individual G_i^* and B_i^*, from invidual values of A_i, B_i, C_i and G_i. This can be called independently for each of the $(n-1)$ such matrices. We use OpenMP #pragma directives here to parallelise this for loop.

c As per the standard usage, in the above step when the pointer to A matix is passed for inversion, the resultant inverted A matrix is stored in the same variable, hence we need not invert the matrix again when used later.

d Further we find summation of each of the G_i^* and B_i^* values, subtracting them from G_N and A_N respectively, which after appropriate inversion gives the values of X_N.

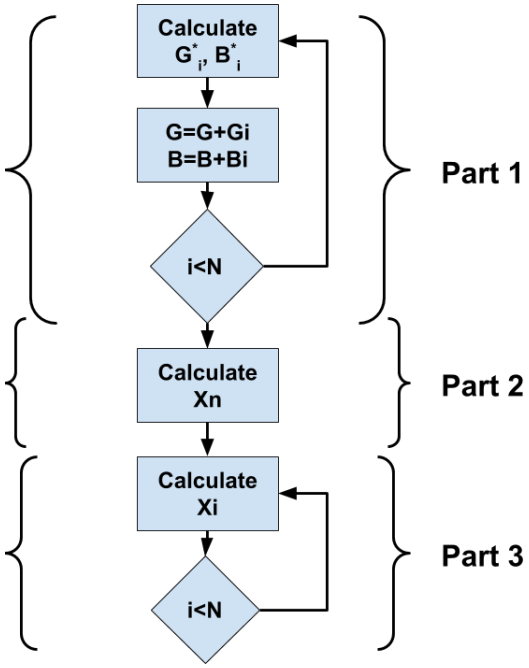

Figure 4.1: System Block Diagram

e Next write a function to calculate individual X_i from given values of X_N, G_i, B_i and A_i^{-1}. This function can be enclosed in an OpenMP #pragma for loop, for each of the $N-1$ such calls.

4.3.3 Results

To test the working of this code, the API was initially tested with small sized matrix to verify the results. Later to profile the time taken to execute, we generate invertible A block matrices, along with appropriately sized B, C and G matrices, initiated with random double precision values written in a text file. Through this we can generate test of any size as required, varying the sizes of block matrices as well and number of blocks in the MNA matrix. GCC compiler by default has a stack size of 8MB which is not suffient large values of N (>1000), so stack size was increased to take care of large sized BBD matrices.

We tabulate the time taken to execute for two different values of block matrix sizes as well the size of BBD matrix defined by variable N. The

Size of Block		N=1000	N=10000	N=100000
4x4	Part1	37.25ms	151.60ms	1463.08ms
	Part2	0.34ms	0.27ms	0.30ms
	Part3	7.45ms	27.74ms	280.82ms
	Total	45.20ms	179.74ms	1744.37ms
8x8	Part1	92.02ms	1193.01ms	11872.77ms
	Part2	0.61ms	0.93ms	0.95ms
	Part3	10.96ms	124.47ms	911.39ms
	Total	103.74ms	1318.66ms	12785.14ms

Table 4.1: Comparison of run times of different parts of the program

program execution is divided into three parts, part one being parallel calculation of individual G_i^* and B_i^*. Part two is the serial calculation of X_N from already computed matrices, and third part is parallel computation of individual X_i from the calculated value of X_N in previous step. The results tabulated profiles the execution time across all three parts of the program.

Further Work

As seen in the data tabulated in table 4.1, part 1 takes the highest % of execution time. We expect that this time be further reduced due to use of OpenMP directives. OpenMP by default support some primitives for reduction operator which can be used for calculation of summation in our API. Moreover we can even compare these results with execution times taken for computing solutions of a complete BBD matrix, directly solving for $AX = G$.

4.4 Hardware Implementation

In the given section we will describe in detail the hardware implementation of bordered block diagonal matrix solver. We expect that this gives significant performance improvement over the traditional software implementations. We will describe the overall block diagram, the basic idea behind design and some light on using Bluespec as hardware description language. The goals for this design were to maximise the parallelism possible, full utilisation of process-

ing elements and overlapping memory I/O and computation such that peak performance is sustained.

The hardware development is done in bluespec system verilog, bluespec is a high level functional programming language which was essentially Haskell extended to handle chip design and Electronic Design Automation in general. Bluespec has inbuilt powerful language constructs, enabling efficient and transparent coding of complex hardware architectures. Further in this section, we will demonstrate the advantages of using bluespec for small block of example code.

4.4.1 Memory Bottle Neck

Ideally speaking we can spawn $N-1$ functional units to calculate each of the partial terms for X_N in parallel, but given size of matrix we are trying to solve here and memory bandwidth considerations, this will be overkill in our case, and hence we need to consider the memory bottle neck in our hardware design. A dual channel DDR2 RAM running at a frequency of 400MHz, with each line of 64 bits will have a theoretical maximum bandwidth of 12.8GB/s. Additionally there is significant latency, and hence data needs to pre-fetched for mitigates delays and waiting times.

4.4.2 Hardware Block Diagrams

In this section we will describe the block diagram which is later implemented in Bluespec. The block diagram has the following major components, and the following sections will discuss in detail the implementation of these functions. We take several assumptions here to simplify the problem and later build upon them.

Since the memory parameters control the number of parallel functional units to be spawned, we do a theoretical analysis of how it depends upon various factors in the system. Let us assume that the number of functional units is k, the memory bandwidth is such that each of the block matrix takes m cycles to fill, α is the clock cycles taken by inversion unit and β is the same number for multiplication unit, where we can state that $\alpha > \beta$. The constraint being that computational blocks should not remain underutilized.

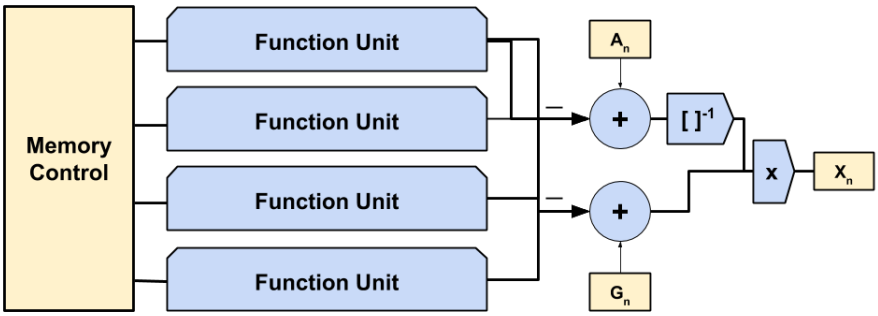

Figure 4.2: System Block Diagram

The memory controller takes care of address, and supplies the appropriate matrices to the each of the functional unit and the individual matrices are stored in a contagious manner as required by the functional units.

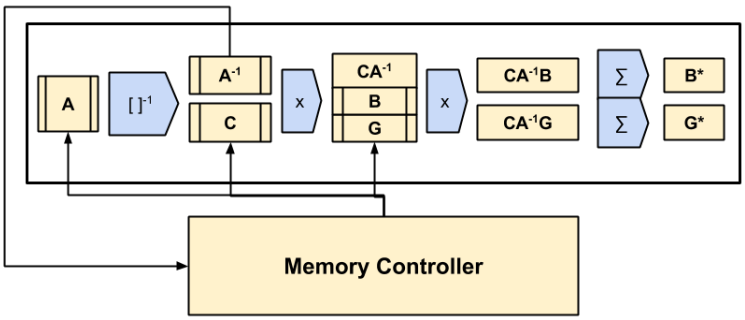

Figure 4.3: Function Unit

Initially when this system starts, in the first m cycles A_1 is filled, similarly in km cycles each of the A_i in the k functional units are filled. Also the inversion units get the data after m cycles and takes α clock cycles to finish computing. In the mean time, we as well start copying C_1 from memory and it is available after $km + m$ cycles from start and A_1^{-1} is available at $\alpha + m$ cycles. For optimal utilization we want simply,

$$km + m = \alpha + m, k = \alpha/m$$

So given the system parameters like memory bandwidth, matrix size and latency of matrix inverse blocks we can identify a suitable value of k, number of functional units.

Memory Controller

As described before memory access is one of the major bottleneck in performance of MNA matrix solver. For this we require a dedicated memory controller to manage the incoming data from the subsequent memory accesses. As per our requirement we want the data to be arranged in specific order in memory, such that the using locality of reference we are able to access more useful data in a given clock cycle.

Following table gives an idea about subsequent memory accesse for a hardware with k=4

Access#	Value	Access#	Value
0-1	A1	16-17	B1
2-3	A2	18	G1
4-5	A3	19-20	B2
6-7	A4	21	G2
8-9	C1	22-23	B3
10-11	C2	24	G3
12-13	C3	25-26	B4
14-15	C4	27	G4

Matrix Multiplication Module

We use the rank-one update matrix multiplication algorithm from a previous work on matrix matrix multiplication in hardware [13]. Take two matrix A and B of size $M \times N$ and $N \times R$ respectively. Matrix A is fetched in column major form, and Matrix B is fetched in row major form. Now considering first column A_1 of A and first row B_1 of B to form the first intermediate C_1 matrix.

$$\begin{bmatrix} A_{11} \\ A_{21} \\ A_{31} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} & B_{13} \end{bmatrix} = \begin{bmatrix} A_{11}B_{11} & A_{11}B_{12} & A_{11}B_{13} \\ A_{21}B_{11} & A_{21}B_{12} & A_{21}B_{13} \\ A_{31}B_{11} & A_{31}B_{12} & A_{31}B_{13} \end{bmatrix}$$

The matrix of right is C_1 one of the value of C_is. We can get N such values of C_i from N columns of A and N rows of B, which all added together will give the matrix product C.

As seen in Fig. 3., block diagram of matrix matrix multiplier is presented. Once the results for a given matrix-matrix multiplication have been

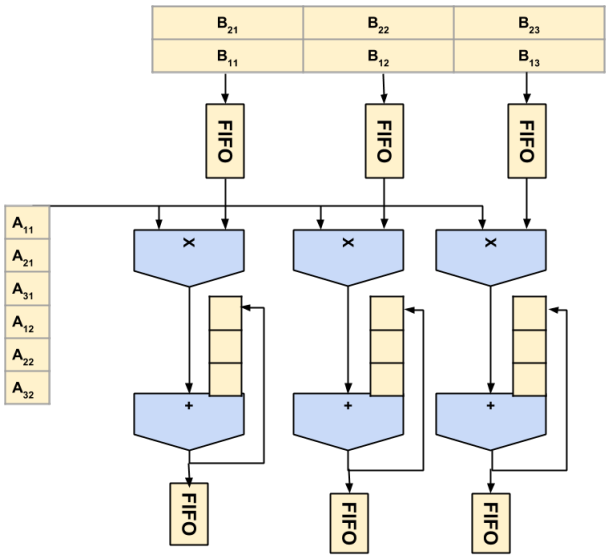

Figure 4.4: Matrix Matrix Multiplier

computed, we flush the registers before adder units and connect the FIFO appropriately such that out matrix C is in a ordered form.

Matrix Inversion Module

We use the ideas of Gauss Jordan Elimination to compute the inverse of a given matrix, it is assured that inverse for a such a matrix of size $m \times m$ will be found in at most m^2 steps. Lets take an example of this algorithm. Take the matrix A and define an identity matrix I of same size.

$$\begin{bmatrix} 12 & 11 & 10 \\ 2 & 13 & 14 \\ 15 & 3 & 9 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Now we will perform some specific set of row operation on matrix A and I identically such that A is now transformed to an Identity matrix and the similar operations done on I gives the inverse of A matrix. To form a one divide the corresponding row by appropriate constant, and to get a zero element subtract row with one element in that column from the given row. In every column, we first create a one and then go on to create a zero, it is guaranteed that such row operations can be done for any general matrix

of size $m \times m$. Conversion of each element to identity matrix form can be implemented in a sequential way, and hence this can be easily be formulated in pipelined manner in hardware, with number of stages being m^2.

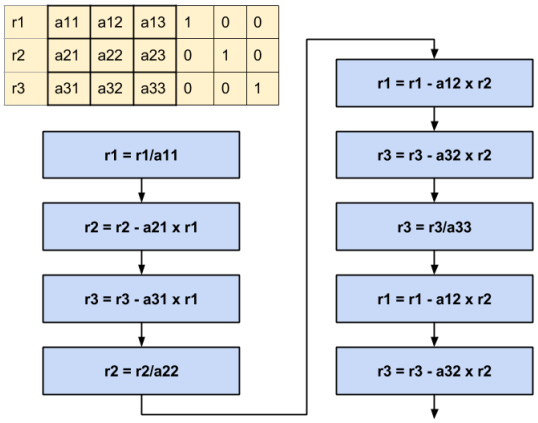

Figure 4.5: Matrix Inversion using Gauss Jordan Elimination

Each of the row operations mentioned in the block diagram are operated on both the A and I matrix. Additionally it may be noted that the computation can start with just first row of matrix A and subsequent rows are used in subsequent clock cycles.

4.5 Future Work

The hardware design are currently in simulation phase, we plan to test this on FPGA using Bluespec emulation platform SCE-MI. We can further improve the memory performance by integrating the smart memory controller to prefetch required data beforehand and sustaining the required bandwidth. The hardware only supports fixed matrix size for A, B and C matrix, although we can have clusters of different size in our network. The performance of software API is not at par with the expected results of multi core CPU systems, and can also be improved by inserting appropriate reductions directives in for loops when shared variables are used.

Chapter 5

Conclusion

In conclusion, we list out the major contributions of this project. We also outline the scope of further activities that can build upon the work presented here.

a We developed a flow to generate mapping and routing for a data flow graph of given application to a mesh based network on chip. This gave us insight into how to model various workloads effectively for mapping and routing. We build this model using modelling the placement problem as a quadratic assignment problem and routing problem as a multi commodity flow problem.

b We worked on an existing system of network on chip architecture to run the updated arc routing algorithm here. Adding logic at source and in-path routers to run the updated arc-routing on the system. We added a multipacket Read and Write instruction to run the network with high traffic scenario. We get 37% improvement in in-flight time compared to XY routing using cycle accurate simulations for matrix vector multiplication application.

c We also synthesized our design using Xilinx Vivado and analysed the automatic placement and routing of cores and nodes done by the elaboration tool.

d We also looked at a different application NoC, matrix inversion based on bordered block diagonal partitions, and designed parallel hardware units for the application that works on large matrix sizes.

5.1 Future Work

This thesis laid the foundation for further exploratory work related to Projective Geometry networks. Future work seems possible on multiple fronts:

a We can look into heuristics based solver for solving multi commodity routing problem, as the run times grow exponentially for larger mesh sizes. The LP formulation can be further tightened and better heuristics can be worked out for faster convergence and better optimatilty.

b Right now we solve the mapping and routing problem independently. We can look into integrating mapping and routing together in the workflow, and run this system for multiple iterations to get the best possible combination of mapping and routing.

c We should look into simulation results of different kinds of workloads with different levels of network congestion with a faster processor. And also compare the resource utilization difference in the updated routing algorithm.

d The hardware units for matrix inversion, should be tested for working on hardware and integrated with the network on chip infrastructure developed here.

References

[1] Yue Wu, Chao Lu, and Yunji Chen. A survey of routing algorithm for mesh network-on-chip. *Frontiers of Computer Science*, 10(4):591–601, 2016.

[2] Gururaj Shaileshwar. Effectiveness of projective geometry noc for cmp systems. Master's thesis, IIT Bombay, 2014.

[3] Abhiram Ranade. Foundations of parallel computation. chapter 9, pages 50–54. 2015.

[4] Eliane Maria Loiola, Nair Maria Maia de Abreu, Paulo Oswaldo Boaventura-Netto, Peter Hahn, and Tania Querido. A survey for the quadratic assignment problem. *European Journal of Operational Research*, 176(2):657 – 690, 2007.

[5] David T. Connolly. An improved annealing scheme for the qap. *European Journal of Operational Research*, 46(1):93 – 100, 1990.

[6] E. Taillard. Simulated annealing procedure for the quadratic assignment problem, 1998.

[7] Shreeniwas Sapre, Hrishikesh Sharma, Abhishek Patil, B. S. Adiga, and Sachin Patkar. Finite projective geometry based fast, conflict-free parallel matrix computations. *CoRR*, abs/1107.1127, 2011.

[8] *ZedBoard Zync Evaluation and Development Hardware User's Guide.*

[9] David P. Koester. *Parallel Block-Diagonal-Bordered Sparse Linear Solvers for Power Systems Applications.* PhD thesis, Syracuse University, 1995.

[10] Liuxi Qian, Dian Zhou, Xuan Zeng, Fan Yang, and Shengguo Wang. A parallel sparse linear system solver for large-scale circuit simulation based on schur complement. In *2013 IEEE 10th International Conference on ASIC*, pages 1–4, Oct 2013.

[11] D. D. Šiljak and A. I. Zečević. A nested decomposition algorithm for parallel computations of very large sparse systems. *Mathematical Problems in Engineering*, 1995.

[12] Mahendra S. Burdhak. Efficient simulation of large non-linear circuits using partitioning and parallelism. Master's thesis, IIT Bombay, 2016.

[13] Vinay B. Y. Kumar, Siddharth Joshi, Sachin B. Patkar, and H. Narayanan. Fpga based high performance double-precision matrix multiplication. *International Journal of Parallel Programming*, 38(3):322–338, 2010.

Chapter 6

Appendix

6.1 Code for quadratic assignment problem solver using simulated annealing

The implementation of simulated annealing algorithm from [6] is presented here with slight modifications in assessing best solution.

The code for this algorithm follows:

```
#include <iostream>
#include <fstream>
#include <math.h>
#include <time.h>
#include <string>

using namespace std;

/*
   ************************************************************
   */
/*
   This programme implement a simulated annealing for the
   quadratic assignment problem along the lines describes in
   the article D. T. Connoly, "An improved annealing scheme
      for
   the QAP", European Journal of Operational Research 46,
      1990,
```

```
    93-100.

    Compiler : g++ or CC should work.

    Author : E. Taillard,
             EIVD, Route de Cheseaux 1, CH-1400 Yverdon,
                Switzerland

    Date : 16. 3. 98

    Format of data file : Example for problem nug5 :

5

0 1 1 2 3
1 0 2 1 2
1 2 0 1 2 // Flow
2 1 1 0 1
3 2 2 1 0

0 5 2 4 1
5 0 3 0 2
2 3 0 0 0 // Cost, Distance
4 0 0 0 5
1 2 0 5 0

   Additionnal parameters : Number of iterations, number of
      runs

*/

/*
    ************************************************************
    */


const long n_max = 851;
```

```
const long infini = 1399999999;
const long nb_iter_initialisation = 1000; // Connolly
   proposes nb_iterations/100

typedef long type_vecteur[n_max];
typedef long type_matrice[n_max][n_max];

type_vecteur best_placement;
long best_cost;

/*--------------- choses manquantes -----------------*/
enum booleen {faux, vrai};


long max(long a, long b) {if (a > b) return(a); else return(
   b);};
double max(double a, double b) {if (a > b) return(a); else
   return(b);}
long min(long a, long b) {if (a < b) return(a); else return(
   b);}
double min(double a, double b) {if (a < b) return(a); else
   return(b);}
void swap(long &a, long &b) {long temp = a; a = b; b = temp
   ;} // swap the arguments which are taken by reference

double temps() {return(double(clock())/double(1000));}

void a_la_ligne(ifstream & fichier_donnees)
{char poubelle[1000]; fichier_donnees.getline(poubelle,
   sizeof(poubelle));}
/*-------------------------------------------------*/

/************* random number generators ***************/

const long m = 2147483647; const long m2 = 2145483479;
const long a12 = 63308; const long q12 = 33921; const long
   r12 = 12979;
```

```
const long a13 = -183326; const long q13 = 11714; const long
    r13 = 2883;
const long a21 = 86098; const long q21 = 24919; const long
   r21 = 7417;
const long a23 = -539608; const long q23 = 3976; const long
   r23 = 2071;
const double invm = 4.656612873077393e-10;
long x10 = 12345, x11 = 67890, x12 = 13579,
    x20 = 24680, x21 = 98765, x22 = 43210;

double mon_rand() // Random number generator
 {long h, p12, p13, p21, p23;
  h = x10/q13; p13 = -a13*(x10-h*q13)-h*r13;
  h = x11/q12; p12 = a12*(x11-h*q12)-h*r12;
  if (p13 < 0) p13 = p13 + m; if (p12 < 0) p12 = p12 + m;
  x10 = x11; x11 = x12; x12 = p12-p13; if (x12 < 0) x12 = x12
      + m;
  h = x20/q23; p23 = -a23*(x20-h*q23)-h*r23;
  h = x22/q21; p21 = a21*(x22-h*q21)-h*r21;
  if (p23 < 0) p23 = p23 + m2; if (p21 < 0) p21 = p21 + m2;
  x20 = x21; x21 = x22; x22 = p21-p23; if(x22 < 0) x22 = x22
     + m2;
  if (x12 < x22) h = x12 - x22 + m; else h = x12 - x22;
  if (h == 0) return(1.0); else return(h*invm);
 }

long unif(long low, long high) // Uniform random number
   generator
 {return(low + long(double(high - low + 1) * mon_rand() ));
 }

/************************** sa for qap
   *******************************/

// Read file contents in matrices A and B of size n
// Here n is global
void lire(long &n, type_matrice &a, type_matrice &b, int
```

```
   pg_num)
 {ifstream fichier_donnees;
  char nom_fichier[30];
  long i, j;

  string file_start = "qap_data_pg";
  string file_end = ".txt";
  string pg_num_string = to_string(pg_num);

  string file_name = file_start+pg_num_string+file_end;

// cout << "Data file name : \n";
// cin >> nom_fichier;
// cout << file_name << '\n';
  fichier_donnees.open( file_name.c_str() );
  fichier_donnees >> n; a_la_ligne(fichier_donnees);
  for (i = 1; i <= n; i = i+1) for (j = 1; j <= n; j = j+1)
    fichier_donnees >> a[i][j];
  for (i = 1; i <= n; i = i+1) for (j = 1; j <= n; j = j+1)
    fichier_donnees >> b[i][j];
  fichier_donnees.close();
 }

// In the given permutation p, swap location r and s
// Return updated change in cost = dist*flow
long calc_delta_complet2(long n, type_matrice & a,
   type_matrice & b,
                        type_vecteur & p, long r, long s)
 {long d;
  d = (a[r][r]-a[s][s])*(b[p[s]][p[s]]-b[p[r]][p[r]]) +
      (a[r][s]-a[s][r])*(b[p[s]][p[r]]-b[p[r]][p[s]]);
  for (long k = 1; k <= n; k = k + 1) if (k!=r && k!=s)
    d = d + (a[k][r]-a[k][s])*(b[p[k]][p[s]]-b[p[k]][p[r]]) +
            (a[r][k]-a[s][k])*(b[p[s]][p[k]]-b[p[r]][p[k]]);
  return(d);
 }
```

```
// Calculate cost for that particular p with A and B
   matrices given
long calcule_cout(long n, type_matrice & a, type_matrice & b
   , type_vecteur & p)
 {long i, j;
  long c = 0;
  for (i = 1; i <= n; i = i + 1) for (j = 1; j <= n; j = j +
     1)
    c = c + a[i][j] * b[p[i]][p[j]];
  return(c);
 }

// Get a new permutation p for each of the different
   solution or retries
void tire_solution_aleatoire(long n, type_vecteur & p)
 {long i;
  for (i = 1; i <= n; i = i+1) p[i] = i;
  for (i = 2; i <= n; i = i+1) swap(p[i-1], p[unif(i-1, n)]);
 }

//
   ****************************************************************************

// ************************ Final Solver Function
   ***************************
//
   ****************************************************************************

// This solves for each retry, this is simply a solver which
    takes number of
// iterations as one of the input

void recuit(long n, type_matrice & a, type_matrice & b,
           type_vecteur & meilleure_sol, long & meilleur_cout
              ,
           long nb_iterations)
```

```
{type_vecteur p;
 long i, r, s;
 long delta;
 double cpu = temps();
 long k = n*(n-1)/2, mxfail = k, nb_fail, no_iteration;
 long dmin = infini, dmax = 0;
 double t0, tf, beta, tfound, temperature;

 for (i = 1; i <= n; i = i + 1)
   p[i] = meilleure_sol[i]; // an initial solution generated
        by uniform permutation
 long Cout = calcule_cout(n, a, b, p);
 meilleur_cout = Cout; // an initial cost by uniform
    solution generated


 // nb_iter_initialisation is global variable initialized
    above
 // Right now to 1000
 for (no_iteration = 1; no_iteration <=
    nb_iter_initialisation;
     no_iteration = no_iteration+1)
  {r = unif(1, n);
   s = unif(1, n-1);
   if (s >= r) s = s+1;

   delta = calc_delta_complet2(n,a,b,p,r,s);
   if (delta > 0)
    {dmin = min(dmin, delta); dmax = max(dmax, delta);};
   Cout = Cout + delta;
   swap(p[r], p[s]);
  };

 t0 = dmin + (dmax - dmin)/10.0;
 tf = dmin;
 beta = (t0 - tf)/(nb_iterations*t0*tf);
```

```
  nb_fail = 0;
  tfound = t0;
  temperature = t0;
  r = 1; s = 2;
  for (no_iteration = 1;
       no_iteration <= nb_iterations - nb_iter_initialisation
          ;
       no_iteration = no_iteration + 1)
    { temperature = temperature / (1.0 + beta*temperature);

      s = s + 1;
      if (s > n)
       {r = r + 1;
        if (r > n - 1) r = 1;
        s = r + 1;
       };

      delta = calc_delta_complet2(n,a,b,p,r,s);
      if ((delta < 0) || (mon_rand() < exp(-double(delta)/
         temperature)) ||
           mxfail == nb_fail)
       {Cout = Cout + delta; swap(p[r], p[s]); nb_fail = 0;}
      else nb_fail = nb_fail + 1;

      if (mxfail == nb_fail) {beta = 0; temperature = tfound
         ;};
      if (Cout < meilleur_cout)
       {meilleur_cout = Cout;
        for (i = 1; i <= n; i = i + 1) meilleure_sol[i] = p[i
           ];
        tfound = temperature;
// cout << "Iteration = " << no_iteration
// << " Cost = " << meilleur_cout
// << " Computational time = " << temps() - cpu << '\n';
       };

   };
```

```
// cout << "Best solution found : \n";
// for (i = 1; i <= n; i = i + 1) cout << meilleure_sol[i]
   << ' ';
// cout << "\nCost of this solution is:";
// cout << meilleur_cout ;
// cout << '\n';

  if (meilleur_cout < best_cost) {
    best_cost = meilleur_cout;
    for (i = 1; i <= n; i = i + 1) best_placement[i] =
       meilleure_sol[i];
  }
 }


long n, nb_iterations, nb_res, no_res;
long Cout;
type_matrice a, b;
type_vecteur p;

int main( int argc, char *argv[])
 {
  nb_iterations = 5000;
  nb_res = 20;
// cout << "nr iterations :" << nb_iterations << endl;
// cout << "nr resolutions :" << nb_res << endl;

  best_cost = 9223372036854775807 ; //LONG MAX VALUE
  int pg_num = atoi(argv[1]);
  lire(n, a, b, pg_num);

  for (no_res = 1; no_res <= nb_res; no_res = no_res + 1)
   {tire_solution_aleatoire(n, p);
    recuit(n,a,b,p,Cout, nb_iterations);
   };
```

```
// cout << " " << "\n" ;
// cout << "Final Placement: \n";
// cout << "Best cost: " << best_cost << "\n";
// cout << "Best solution found : \n";
// for (int i = 1; i <= n; i = i + 1) cout << best_placement
    [i] << ' ';
// cout << "\n";

  ofstream file_sol;

  string file_start = "qap_sol_pg";
  string file_end = ".txt";
  string pg_num_string = to_string(pg_num);

  string file_name = file_start+pg_num_string+file_end;

// cout << "Solution file name : \n";
// cout << file_name << '\n';
  file_sol.open( file_name.c_str() );

  for (int i=1; i<= n; i = i + 1) file_sol << best_placement[
     i] << '␣';

  file_sol.close();

  return 0;
 }
```

6.2 Code for solving routing problem using multicommodity flow

The code for solving the multi commodity flow problem using gurobi based LP solver for a PG based matrix vector multiplication network. The dependency matrix generated here serves as the data flow graph for the problem.

The code for this algorithm follows:

```
from gurobipy import *
import math
import sys
import copy

def read_placement_files () :
  placementAll = {}
  for p in p_values:
    fileobj = open("qap_scripts/qap_sol_pg"+ str(p) +".txt")
    file_string = fileobj.read()
    placement_list = map(int, file_string.split())
    placement_list = [x - 1 for x in placement_list]
    placementAll[p] = placement_list
    fileobj.close()

  return placementAll

def getDependencyList():
  ## This list basically means ith nodes is connected to pg
     nodes mentioned in the list dependency[i]
  dependency = []
  # Inserting the connections between Point Node i and all
     the Line nodes it is connected to, except itself
  for i in range(numCores):
    dependency.append([])
    for j in incidence:
      if (i != (i+ j)%numCores ):
        dependency[i].append((j+i)%numCores)

  # Inserting the connections between Line node i and all the
      Point nodes it is connected to, except itself
  for i in range(numCores):
    for j in incidence:
      if (i != (i - j)% numCores ):
        dependency[i].append((i - j)% numCores)
```

```
  return dependency

  dependency_out = []
  # Inserting the connections between Point Node i and all
     the Line nodes it is connected to, except itself
  for i in range(numCores):
      dependency_out.append([])
      for j in incidence:
          if (i != (i+ j)%numCores ):
              dependency_out[i].append((j+i)%numCores)

  dependency_in = []
  # Inserting the connections between Line node i and all the
      Point nodes it is connected to, except itself
  for i in range(numCores):
      dependency_in.append([])
      for j in incidence:
          if (i != (i - j)% numCores ):
              dependency_in[i].append((i - j)% numCores)

# return dependency_in

  # print "Data sent at t=0, x0, x1, ...: "
  # print "Dependency_out: "
  # print dependency_out

  # print "Dependency sent after t = t1, y0part0, y1part0,
     ..... ",
  # print "Dependency_in: "
  # print dependency_in


def init_commodity():
  commodities = [] # start with empty list for commodities
  for i in range(len(dependency)):
    for j in range(len(dependency[i])):
      commodities.append('s'+ str(i) +'d'+ str(dependency[i][
```

```
          j]))

  return commodities

def get_commodity_src_dst (commodity) :
  ## Returns source and destination nodes for a commodity

  index_s = commodity.index('s') + 1
  index_d = commodity.index('d')
  index_d1 = commodity.index('d') + 1
  index_end = len(commodity)

  src = int(commodity[index_s:index_d])
  dest = int(commodity[index_d1:index_end])

  return src , dest

def get_node_proc_mesh (node):
  index_p = node.index('p') + 1
  index_m = node.index('m')
  index_m1 = node.index('m') + 1
  index_end = len(node)
  proc_num = int ( node[index_p :index_m ] )
  mesh_num = int ( node[index_m1:index_end] )
  return proc_num, mesh_num


def init_nodes():
  nodes = [] # start with empty list for nodes

  for i in range(len(placement)):
    string = 'p'+str(i)+'m'+str(placement[i])
    nodes.append(string)
  return nodes


def init_inflow():
```

```
  inflow = {} # start with empty dictionary for inflow
     specification

  for i in commodities:
    src , dest = get_commodity_src_dst(i)
    for j in nodes:
      p_num , m_num = get_node_proc_mesh (j)
      if p_num == src:
        inflow[(i, j)] = PACK_LENGTH
      elif p_num == dest:
        inflow[(i, j)] = -PACK_LENGTH
      else :
        inflow[(i, j)] = 0
  return inflow

def get_node(mesh_num):
  "Outputs␣the␣node␣string␣for␣a␣given␣location␣in␣mesh,␣
     using␣the␣placement␣list"
  proc_num = placement.index(mesh_num)
  return 'p'+str(proc_num)+'m'+str(mesh_num)

def init_arc_list():
  arc = []
  for i in range(numMeshnodes):
    if i >= meshEdge : #up
      arc.append( ( get_node( i ) , get_node (i-meshEdge) ) )
    if i%meshEdge > 0 : #left
      arc.append( ( get_node( i ) , get_node (i-1) ) )
    if i < numMeshnodes - meshEdge : # down
      arc.append( ( get_node( i ) , get_node (i+meshEdge) ) )
    if i%meshEdge < meshEdge-1 : # right
      arc.append( ( get_node( i ) , get_node (i+1) ) )
  return arc

def init_cost():
  cost = {}
  for i in commodities:
```

```
    for j in arc:
      j_list = list(j)
      j_tup = [i]+j_list
      tup = tuple(j_tup)
      cost[tup] = 1
  return cost

#
    ********************************************************************************
     #
#
    ------------------------------------------------------------------------------
     #
#
# MAIN SCRIPT STARTS HERE
#
#
    ------------------------------------------------------------------------------
     #
#
    ********************************************************************************
     #


######### GETTING PG SIZE FROM COMMAND LINE ############

#if len(sys.argv) >= 2:
# p = int(sys.argv[1])
# reportFileName = "report_lp_hungarian_fixedPG"+str(p)+".
    txt"
# if (p < 2 or p > 19):
# print "Size of PG has to be between 2 and 19\n"
# exit()
#else :
# print "Specify Size of PG to be evaluated with random
    mappings to be generated for mesh \n "
# exit()
```

```
p =2

p_values = [2, 3, 4, 5, 7, 8, 9, 11, 13, 16, 17, 19]

numPGnodes = p*p +p +1
numCores = numPGnodes
# Decide Mesh Size here, largest perfect square number
    greater than numCores
meshEdge = 1
numMeshnodes = meshEdge * meshEdge
while (numPGnodes > numMeshnodes):
    meshEdge += 1
    numMeshnodes = meshEdge *meshEdge


## This is the inflow and outflow at source and destination
    respectively for all comoodities
## This represents how many parts the packet can be broken
    into
PACK_LENGTH = 8;

## This is the fixed capacity for all arcs
## This has to be changed to get a feasible soluton usually
CAPACITY_CONST = 16;

MAX = sys.maxsize

incidenceAll = {2 : [0,1,3],
               3 : [0,1,3,9],
               4 : [0,1,4,14,16],
               5 : [0,1,3,8,12,18],
               7 : [0,1,3,13,32,36,43,52],
               8 : [0,1,3,7,15,31,36,54,63],
               9 : [0,1,3,9,27,49,56,61,77,81],
               11 : [0,1,3,12,20,34,38,81,88,94,104,109],
               13 :
```

```
                [0,1,3,16,23,28,42,76,82,86,119,137,154,175],

            16 :
                [0,1,3,7,15,31,63,90,116,127,136,181,194,204,233,238,255],

            17 :
                [0,1,3,30,37,50,55,76,98,117,129,133,157,189,199,222,293,299

            19 :
                [0,1,19,28,96,118,151,153,176,202,240,254,290,296,300,307,33

            }

incidence = incidenceAll[p]

#
    ##########################################################################

placementAll = read_placement_files()
placement = placementAll[p]
print "placement", placement
## This means that ith value in list is location of ith
    processor

## Actual output from sa_qap is different
## In the list generated, the ith value in the list is the
    position of ith processor in mesh

dependency = getDependencyList() # This is the equvalent of
    data flow graph, here generated for the MatrixVector
    Application
commodities = init_commodity() # Initiate list of commodity,
     data structure for lp, of the form 's2d4' source
    processor 2, destination 4
nodes = init_nodes() # Initiate list of nodes, data
    structure for lp, of the form 'p2m4' processor 2 placed
    at mesh loc 4
```

```
inflow = init_inflow() # Specify source and destination for
    each commodity
arc = init_arc_list() # List of arc tuples generated from
    mesh network specification

multi_dict = {}
for i in arc:
  multi_dict[i] = CAPACITY_CONST

arcs, capacity = multidict(multi_dict) # Initiate arc and
    capacity data structures for lp
cost = init_cost()

m = Model('netflow')

# Create variables
flow = m.addVars(commodities, arcs, obj=cost, name="flow")

# Arc capacity constraints
m.addConstrs(
  (flow.sum('*',i,j) <= capacity[i,j] for i,j in arcs), "cap"
     )

# Flow conservation constraints
m.addConstrs(
  (flow.sum(h,'*',j) + inflow[h,j] == flow.sum(h,j,'*')
  for h in commodities for j in nodes), "node")

# Dont print log to console
# m.Params.LogToConsole = 0

# Compute optimal solution
m.optimize()

#filename0 = "Optimize_Runtimes.txt"
#f0 = open(filename0, 'a')
```

```
# Print solution
if m.status == GRB.Status.OPTIMAL:
# print " Runtime printed here:"
# runtime = str(m.Runtime)
  solution = m.getAttr('x', flow)
```

6.3 Python script to generate bluespec files

This python script takes the arc solutions generated by the gurobi LP solver and generates the bluespec readable file.

```
from gurobipy import *
import math
import sys
import copy


def check_flow ( commodity ):
  minimum_flow = MAX # supposed to be infinity

  for i,j in arcs:
    if (arc_solution[commodity, i, j] > 0) & (arc_solution[
       commodity, i, j] < minimum_flow) :
      minimum_flow = arc_solution[commodity, i, j]

  if minimum_flow == MAX: # Means no non-zero flow was found
    minimum_flow = 0

  return minimum_flow

def get_src_dst (commodity) :
  ## Returns source and destination nodes (in string form)
     for a commodity
  src , dest = get_commodity_src_dst(commodity)
  for j in nodes:
    p_num , m_num = get_node_proc_mesh (j)
```

```
    if src == p_num :
      src_nd = j
    if dest == p_num :
      dest_nd = j
  return src_nd , dest_nd

def get_next_node (commodity, previous_node):
  ''' Get the next node which has minimum flow'''
  min_val = MAX
# print "previous_node is ", previous_node
  for x,y in arcs:
    if (x==previous_node):
# print 'x is', x
# print 'y is', y
      sol_tmp = arc_solution[commodity, x, y]
# print "sol_temp", sol_tmp
# print "(sol_tmp > 0) :" , (sol_tmp > 0)
# print "(sol_tmp < min_val)" , (sol_tmp < min_val)
      if (sol_tmp > 0) & (sol_tmp < min_val) :
# print "Value of next node", y
        min_val = arc_solution[commodity, x, y]
        min_dst = y

# print "min_dst before return of function", min_dst
  return min_dst

def init_arc_list(arc_solution):
  arc_list = []
  # arc_list [3] = ['p0m6', 'p4m4', 8.0, ['p0m6', 'p1m3', '
     p7m0', 'p8m1', 'p4m4']]
  # arc_list[arc_id] = [ src_node, dest_node, count,
     path_list]

  for h in commodities:
    src_node, dest_node = get_src_dst(h)
    while check_flow(h)>0 :
      min_flow = check_flow(h)
```

```
      path_list = []
      path_list.append(src_node)

      prev_node = src_node
      next_node = src_node # Initiating

      while (next_node != dest_node ):
  # print "Prev Node is:" , prev_node
        next_node = get_next_node(h, prev_node)
  # print "Next node is;" , next_node
        path_list.append(next_node)
  # print path_list
        arc_solution[h, prev_node, next_node] = arc_solution[
           h, prev_node, next_node] - min_flow
        prev_node = next_node
  # print "Outside while now"

      path_length = len(path_list)
      path_source = path_list[0]
      path_dest = path_list[path_length-1]

      arc_list.append([ path_source, path_dest, min_flow,
         path_list ])

  # print "Commodity done:", h
  # print ""
  return arc_list


def printToFile(arc_list, filename):

  filename = filename + ".bsv"
  f = open(filename, 'w')

  print "**********************************************"
  print "PACKET␣SPECIFICATION␣AT␣INPUT␣NODES"
  print "**********************************************"
```

```
f.write("import␣MemTypes::*;\n")
f.write("import␣ProcTypes::*;\n")
f.write("\n")
f.write("//␣Python␣generated␣code␣which␣returns␣arc_id␣for␣
   each␣pair␣of␣source␣and␣destination␣of␣packets␣\n")
f.write("\n")
f.write("function␣NoCArcId␣lookupNoCArcId(ProcID␣srcProcId,
   ␣ProcID␣destProcID,␣PacketLocation␣packLoc);\n")
f.write("␣␣NoCArcId␣arc_id␣=␣0;\n")
f.write("\n")

count = 0
flag = True

for src in nodes:
  pnum_src, mnum = get_node_proc_mesh(src)

  if pnum_src < numCores :

    if flag :
      f.write("␣␣if␣(srcProcId␣==␣" + str(pnum_src) + ")␣
         begin\n")
      flag = False
    else : # Logic for if and else if in Bluespec code
      f.write("␣␣else␣if␣(srcProcId␣==␣" + str(pnum_src) +
         ")␣begin\n")

    flaga = True

    for dest in nodes:
      pnum_dest, mnum = get_node_proc_mesh(dest)

      flagc = False
      for arc_id in range(len(arc_list)):
        if ( arc_list[arc_id][0] == src) & ( arc_list[
           arc_id][1] == dest ) :
```

```
            flagc = True

        if flagc:
          if flaga :
            f.write("␣␣␣␣if␣(destProcID␣==␣"+ str(pnum_dest) +
                ")␣begin␣\n")
            flaga = False
          else : # Logic for if and else if in Bluespec code
            f.write("␣␣␣␣else␣if(destProcID␣==␣"+ str(
                pnum_dest) +")␣begin␣\n")


        flagb = True
        for arc_id in range(len(arc_list)):
          if ( arc_list[arc_id][0] == src) & ( arc_list[
              arc_id][1] == dest) :


            count = count + arc_list[arc_id][2]
            print "For␣Source␣", arc_list[arc_id][0] ,
            print "For␣Destination:␣", arc_list[arc_id][1] ,
            print "arc_id␣is", arc_id,
            print "upper␣limit␣is", count
            ret_arc_id = str(arc_id)

            if flagb:
              f.write("␣␣␣␣␣␣if␣(␣packLoc␣<␣"+str(int(count))+
                  ")␣arc_id␣=␣" + ret_arc_id + ";␣\n")
              flagb = False
            else:
              f.write("␣␣␣␣␣␣else␣if␣(␣packLoc␣<␣"+str(int (
                  count))+")␣arc_id␣=␣" + ret_arc_id + ";␣\n")

        count = 0
        if flagc:
          f.write("␣␣␣␣end\n")
          f.write("\n")
```

```
    f.write("␣␣end\n")
    f.write("\n")

f.write("␣␣return␣arc_id;\n")
f.write("\n")
f.write("endfunction:␣lookupNoCArcId\n")


print ""
print "*********************************************"
print "PACKET␣ROUTING␣AT␣INTERMEDIATE␣NODES"
print "*********************************************"

f.write("\n")
f.write("\n")

f.write("//␣Lookup␣function␣for␣destination␣node␣at␣each␣
   mesh␣node␣corresponding␣to␣the␣arc␣id␣and␣source␣mesh␣\
   n")
f.write("function␣String␣lookupArcDest␣(␣NoCAddr2D␣
   thisRowAddr,␣NoCAddr2D␣thisColAddr,␣NoCArcId␣arc_index)
   ;␣\n")
f.write("␣␣String␣dest_direction␣=␣\"N\";\n")

for j in nodes:

  print ""
  print "Packet␣routing␣direction␣at␣node:", j

  proc_num, mesh_num = get_node_proc_mesh(j)

  mesh_row = int(mesh_num/meshEdge)
  mesh_col = mesh_num%meshEdge

  f.write("␣␣if␣((thisRowAddr␣==␣" + str(mesh_row) + ")␣&&␣
     (thisColAddr␣==␣" + str(mesh_col) + "))␣begin␣\n")
```

```
    for arc_id in range (len(arc_list)):
      arc_path = arc_list[arc_id][3]
      len_arc_path = len(arc_path)
      for i in range(len_arc_path):

        if arc_path[i]==j:


          if i == len_arc_path-1 :
            direction = 'H'
          else :
            current_node_in_path = arc_path[i]
            next_node_in_path = arc_path[i+1]

            curr_index_m = current_node_in_path.index('m')
            curr_index_last = len(current_node_in_path)

            next_index_m = next_node_in_path.index('m')
            next_index_last = len(next_node_in_path)

            current_node_mesh = int(current_node_in_path[
               curr_index_m+1:curr_index_last])
            next_node_mesh = int(next_node_in_path[
               next_index_m+1:next_index_last])

            if(next_node_mesh == current_node_mesh +1 ):
              direction = 'E'
            elif (next_node_mesh == current_node_mesh - 1 ):
              direction = 'W'
            elif ((next_node_mesh/3) == (current_node_mesh/3)
               +1):
              direction = 'S'
            elif ((next_node_mesh/3) == (current_node_mesh/3)
               -1):
              direction = 'N'

          print "For␣arc_id:␣", arc_id ,
```

```
          print "direction␣is", direction
          f.write("␣␣␣␣if␣(arc_index␣==␣"+ str(arc_id)+")␣
             dest_direction␣=␣\"" + direction+"\"␣␣;\n")
    f.write("␣␣end␣\n")
  f.write("␣␣return␣dest_direction;\n")
  f.write("endfunction\n")


  f.write("\n")
  f.write("\n")

  f.write("//␣This␣is␣device␣placement␣generated␣from␣the␣qap
     ␣solver␣used␣before␣hardcoded␣in␣mcmf.py␣file␣right␣now
     ␣␣\n")
  f.write("function␣MeshID␣lookupNoCAddr(ProcID␣currProcId);␣
     \n")
  f.write("␣␣case␣(currProcId)\n")

  for i in range(numCores): # i is processor_id
    f.write("␣␣␣␣"+ str(i) + ":␣return␣" + str(placement[i])
       + ";␣\n")



  f.write("␣␣endcase␣\n")
  f.write("endfunction␣\n")
  f.close()

def get_node_from_mesh(mesh_num):

  for i in range(len(placement)):
    if placement[i] == mesh_num:
      proc_num = i

  node_op = 'p' + str(proc_num) +'m' + str(mesh_num)
  return node_op
```

```
def xypath_gen(src_mesh_num, dest_mesh_num):
  pack = [src_mesh_num]

  while (src_mesh_num != dest_mesh_num):

    if(src_mesh_num%3 == dest_mesh_num%3): # check if x is
       same

      if(src_mesh_num/3 == dest_mesh_num/3): # check if y is
         same
        pack.append(src_mesh_num)

      else:
        if(src_mesh_num/3 > dest_mesh_num/3):
          src_mesh_num = src_mesh_num-3
        elif(src_mesh_num/3 < dest_mesh_num/3):
          src_mesh_num = src_mesh_num+3
        pack.append(src_mesh_num)

    else:
      if(src_mesh_num%3 > dest_mesh_num%3):
        src_mesh_num = src_mesh_num-1
      elif(src_mesh_num%3 < dest_mesh_num%3):
        src_mesh_num = src_mesh_num+1
      pack.append(src_mesh_num)

  return pack


def init_xy_list():

  arc_list = []
  # arc_list [3] = ['p0m6', 'p4m4', 8.0, ['p0m6', 'p1m3', '
     p7m0', 'p8m1', 'p4m4']]
  # arc_list[arc_id] = [ src_node, dest_node, count,
     path_list]
```

```
  for h in commodities:
    src_node, dest_node = get_src_dst(h)
    path_list = []

    src_pnum, src_mnum = get_node_proc_mesh(src_node)
    dest_pnum, dest_mnum = get_node_proc_mesh(dest_node)

    mesh_list = xypath_gen(src_mnum, dest_mnum)

    for i in mesh_list:
      path_list.append(get_node_from_mesh(i) )
    arc_list.append( [src_node, dest_node, PACK_LENGTH,
       path_list] )

  return arc_list

arc_solution = copy.copy(solution)

print "Populating␣arc␣list␣here"
arc_list = init_arc_list(arc_solution)
print "Arc_list␣generated:␣"

for j in range(len(arc_list)):
  print "arc_id", j ,
  print ":",
  print arc_list[j]

print "Populating␣xy␣list␣here"
xy_list = init_xy_list()
print "X-Y␣List␣generated"

for i in range(len(xy_list)):
  print "arc_id", i ,
  print ":",
  print xy_list[i]

# Populating data-structures at source node
```

```
##
   ************************************************************
    ##
##
   ------------------------------------------------------------
    ##
## FILE WRITING STARTS HERE
##
   ------------------------------------------------------------
    ##
##
   ************************************************************
    ##


printToFile(arc_list, "Lookup")
printToFile(xy_list, "Lookupxy")
```