University of Pisa

Master Degree Thesis

Design & Implementation of a Genetic Algorithm for scalable Shortest Path routing in SDN controllers

Author:
David Alberto Lau Gastelo

Supervisor:
Prof. Stefano Giordano
First Co-Supervisor:
Ing. Gregorio Procissi
Second Co-Supervisor:
CF AN Ing. Carlo Roatta

A thesis submitted in fulfilment of the requirements for the degree of Master

in the

Networking Research Group
Department or Information Engineering

April 2015

"A journey of thousand miles starts with a single step and if that step is the right step, it becomes the last step"

Lao Tzu

UNIVERSITY OF PISA

Abstract

Faculty of Telecommunication Engineering
Department or Information Engineering

Master

Design & Implementation of a Genetic Algorithm for scalable Shortest Path routing in SDN controllers

by David Alberto LAU GASTELO

During the last decades, the Internet Network has been growing exponentially. And this growth of the network means also a growth in terms of complexity. One of the most important process that makes Internet happens is the Routing. Routing is the process of selecting the best paths in a network. This thesis aims to develop a genetic algorithm to solve a network routing protocol problem, and implement it on the new Epiphany architecture. In the literature, the routing problem is solved using search graph techniques to find the shortest path. Dijkstra's algorithm is one of the most popular techniques to solve this problem, and it is widely used on network routing protocols, for example the OSPF (Open Shortest Path First). The developed genetic algorithm performed on the Epiphany architecture is compared with Dijkstra's algorithm to solve routing problem. The results highlight the potentiality of the proposed genetic algorithm and the possibility to use this Epiphany architecture as a cost-effective solution for an embedded SDN (Software Defined Networking) controller.

Acknowledgements

Es irónico que el epílogo se encuentre al inicio. Después de todos estos años y kilómetros, llega a la conclusión esta etapa de mi vida. Lejos de casa, he encontrado otro lugar que puedo llamar hogar. Y lo más importante, he conocido personas que han dejado un tatuaje en mi ser.

Inicio agradeciendo a Valeria, mi alma gemela; que ha sabido ser amiga, madre y maestra estos años. Has ya escrito un profundo e indeleble capítulo de mi vida.

Quiero agradecer también a la Sesta Armi Navali, mis compañeros de armas:
a Mattia, amigo que siempre me ha hecho sonreír;
a Mimmo, óptimo compañero de aventuras;
a Andrea, sabio consejero que se ha demostrado un inquebrantable amigo;
a Antonio, para los amigos Pollo, por su infinita bondad;
a Alessio, quien en su silencio guarda una gran amistad.

Un agradecimiento especial a Alessandro, que no obstante todos los problemas que hemos tenido, me has hecho parte de tu familia, convirtiéndote en un hermano.

Agradezco también a los padres de Valeria, quienes me han hecho entrar en su maravillosa familia, y agradezco también a todos y cada uno de los integrantes de ésta.

Mi gratitud va también al Profesor Giordano, al Comandante Roatta y a Giuseppe Portaluri, dado que ellos me han hecho el camino más simple en el desarrollo de esta tesis.

Finalmente, agradezco a mi familia en mi tierra, quienes no obstante la distancia, ha estado siempre cerca de mí.

Contents

Abstract **ii**

Acknowledgements **iii**

Contents **iv**

List of Figures **vii**

List of Tables **viii**

Glossary **ix**

1 Introduction **1**

2 Basic Concepts **3**

2.1 Shortest Path problems . . . 3

2.1.1 Introduction . . . 3

2.1.2 Applications . . . 7

2.1.3 Tree of Shortest Paths . . . 7

2.1.4 Shortest Path Problems in Acyclic Networks . . . 8

2.2 Dijkstra's Algorithm . . . 9

2.2.1 Running Time Dijkstra's Algorithm . . . 10

2.2.2 Reverse Dijkstra's Algorithm . . . 11

2.2.3 Bidirectional Dijkstra's Algorithm . . . 11

2.3 Genetic Algorithms . . . 12

2.3.1 Introduction . . . 12

2.3.2 Robustness of Traditional Optimization and Search Methods . . . 12

2.3.3 The Goals of Optimization . . . 15

2.3.4 Differences between Genetic Algorithms and Traditional Methods . 16

2.3.5 A Simple Genetic Algorithm . . . 18

3 The Parallella Board **22**

3.1 Introduction . . . 22

3.1.1 Overview . . . 22

3.1.2 Technical Specifications . . . 22

3.2 Epiphany Architecture . . . 24

3.2.1 Introduction 24
3.2.2 System Examples 25
3.2.3 Memory Architecture 26
3.3 eCore CPU 27
3.3.1 Overview 27
3.3.2 Data Types 29
3.3.3 Local Memory Map 32
3.3.4 General Purpose Register 32
3.3.5 Epiphany Instruction Set 33
3.3.6 Pipeline Description 36
3.3.7 Interrupt Controller 36
3.3.8 Hardware Loops (LABS) 37
3.3.9 Direct Memory Access (DMA) 38
3.3.10 Memory Protection Unit (LABS) 40
3.4 Software Development Enviroment 40
3.5 Programming Model 41
3.5.1 Programming Model Introduction 41
3.5.2 Parallel Programming Example 41

4 Experimentation 43
4.1 Set Up 43
4.2 Structure of the model Network and Table of Cost 43
4.3 Proposed Algorithm 44
4.3.1 Global Variables 45
4.3.2 main() function 45
4.3.3 init_org() function 45
4.3.4 eval_gen() function 46
4.3.5 prod_next_gen() function 46
4.3.6 select_one() function 47
4.4 Results 48

5 Conclusions 50

A Parallella Board Configuration 51
A.1 Hardware Accessories 51
A.1.1 Headless 51
A.1.2 With Display 51
A.2 Creating a bootable micro-SD card 52
A.2.1 Downloading the Binaries 52
A.2.2 Install 52
A.3 Connect the board to the computer via Ethernet 53
A.3.1 Setting a static IP address 53
A.3.2 Connecting the board to the computer 53

B C Codes 55
B.1 "defs.h" file 55

B.2 "host.c" file 56
B.3 "dev.c" file 60

Bibliography **98**

List of Figures

3.1 Parallella (top view) 23
3.2 Parallella (bottom view) 23
3.3 Parallella (Architecture) 24
3.4 eCore Architecture 25
3.5 Epiphany System Architecture 25
3.6 Epiphany Memory Architecture 26
3.7 Epiphany Memory Map 26
3.8 eCore CPU Overview 27
3.9 IEEE Single-Precision Floating-Point Data Types 31
3.10 eCore Local Memory Map 32
3.11 eCore Pipeline Stage Description 37
3.12 eCore Pipeline Graphical View 37
3.13 eCore Interrupt Service Routine Operation 38
3.14 eCore DMA Transfer Types 39
3.15 Epiphany Software Development Stack 40
3.16 Matrix Multiplication Data Flow 42

4.3 Crossover operator 47
4.4 Roulette wheel sampling 47
4.5 A screenshot of the program results 49

List of Tables

2.1 Sample Problem Strings and Fitness Values 20

3.1 General-Purpose Registers[1] 33

Glossary

GA Genetic Algorithms. 16–18, 21

LSB Least Significant Bit. 30, 31

MSB Most Significant Bit. 30

NAN Not A Number. 30, 31

NP Non-deterministic Polynomial time. 5

OS Operating System. 43, 50–52

OSPF Open Shortest Path First. 2

SDN Software-defined Networking. 1

A quien conocí en un viaje lejano, y quedará para siempre en el lugar más cercano a mí

Chapter 1

Introduction

During the last decades, the Internet Network has been growing exponentially. And this growth of the network means also a growth in terms of complexity.
One of the most important process that make Internet happens is the *Routing*. Routing is the process of selecting the best paths in a network.
With the augmented complexity and the constantly demand of connection and resources, the research aims to new challenging targets to keep functional the network and a efficient way to manage it.
Thanks to the new technologies, there are possible solutions to this problem. One of the is a new design of the network, but in a scalable and dynamic way. These is the concept idea of the Software-defined Networking (SDN). SDN is an architecture purporting to be dynamic, manageable, cost-effective, and adaptable, seeking to be suitable for the high-bandwidth, dynamic nature of today's applications. SDN architectures decouple network control and forwarding functions, enabling network control to become directly programmable and the underlying infrastructure to be abstracted from applications and network services.

Another important part of the routing is the protocol of forwarding, that performs an algorithm under certain rules defined by the type of protocol.
One of the algorithms used in routing is the Dijkstra's algorithm[2]. An equivalent algorithm was developed by Edward F. Moore in 1957[3]. For a given source vertex (node) in the graph in Figure 1.1, the algorithm finds the path with lowest cost (i.e. the shortest path) between that vertex and every other vertex. It can also be used for finding costs of shortest paths from a single vertex to a single destination vertex by stopping the algorithm once the shortest path to the destination vertex has been determined. For example, if the vertices of the graph represent routing nodes and edge path costs represent

time spent to pass from a node to another directly linked node, Dijkstra's algorithm can be used to find the shortest route between one node and all other nodes. The shortest path first is widely used in network routing protocols, most notably Open Shortest Path First (OSPF). OSPF is a dynamic routing protocol. It is a link state routing protocol and is part of the interior gateway protocols group. OSPF keeps track of the complete network topology and all the nodes and connections within that network.
OSPF routing protocol is a very important protocol to consider when setting up routing instructions on the network. As OSPF gives the routers the ability to learn the most optimal (shortest) paths it can definitely speed up data transmission from source to destination.
In the literature, Dijkstra's algorithm is often described as a greedy algorithm. A greedy algorithm is described as "a heuristic algorithm that at every step selects the best choice available at the step without regard to future consequence"[4]

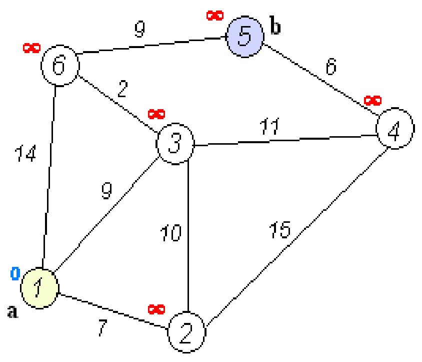


Figure 1.1: Network topology

The optimization of multicore systems now permits the execution on more complex algorithms used in routing that can compete (and sometimes overcome) with the traditional ones.

Chapter 2

Basic Concepts

2.1 Shortest Path problems

2.1.1 Introduction

Shortest path problems lie at the hearth of network flows. They are alluring to both researchers and to practitioners for several reasons:

- They arise frequently in practice since in a wide variety of applications settings where is required to send some material (e.g., a computer data packet, a telephone call, a vehicle) between two specified points in a network as *quickly*, as *cheaply*, or as *reliably* as possible.
- They are easy to solve efficiently.
- As the simplest network models, they capture many of the most salient core ingredients of network flows and so they provide both a benchmark and a point of departure for studying more complex network models
- They arise frequently as subproblems when solving many combinatorial and network optimization problems.

  Even though shortest path problems are relatively easy to solve, the design and analysis of most efficient algorithms for solving them requires considerable ingenuity. Consequently, the study of shortest path problems is a natural starting point for introducing many key ideas from network flows, including the use of clever data structures and ideas such as data scaling to improve the worst-case algorithm performance.

Notation and Assumptions

Consider a directed network G = (N, A) with and *arc length* (or *arc cost*) c_{ij} associated with each arc $(i, j) \in A$. The network has a distinguished node s, called emphsource. Let $A(i)$ represent the arc adjacency list of node i an let $C = max\{c_{ij} : (i,j) \in A\}$. Define the *length of a directed path* as the sum of the lengths of arcs in the path.
The shortest path problem is to determine for every non-source node $i \in N$ a shortest length directed path from node s to node i.
Alternatively, the problem might be view as sending 1 unit of flow as cheaply as possible (with arc flow costs as c_{ij}) form node s to each of the nodes in $N - s$ in an uncapacitated network. This viewpoint gives rise to the following linear programming formulation of the shortest path problem.

Minimize

$$\sum_{(i,j \in A)} c_{ij} x_{ij} \tag{2.1}$$

subject to

$$\sum_{j:(i,j)\in A} x_{ij} - \sum_{j:(j,i)\in A} x_{ji} = \begin{cases} n-1 & for\ i = s \\ -1 & for\ all\ i \in N - \{s\} \end{cases} \tag{2.2}$$

$$x_{ij} \geq 0, \quad \forall (i,j) \in A$$

These assumptions will be done for the study of the shortest path problem:[5]

Assumption 1 *All arc lengths are integers*

The integrality assumption imposed on arc lengths is necessary for some algorithms and unnecessary for others. That is, for some algorithms these condition can be relaxed and still perform the same analysis.
Algorithms whose complexity bound depends on C assume integrality of the data. Note that always can be transformed rational arc capacities to integer arc capacities by multiplying them by a suitably large number. Moreover, is necessarily needed convert irrational numbers to rational numbers to represent them on a computer.
Therefore, the integrality assumption is really not a restrictive assumption in practice.

Assumption 2 *The network contains a directed path from node s to every other node in the network*

This assumption can be always satisfied by adding a "fictitious" arc (s, i) of suitably large cost for each node i that is not connected to node s by a directed path.

Assumption 3 *The network does not contain a negative cycle (i.e., a directed cycle of negative length)*

For any network containing a negative cycle W, the linear programming formulation (2.1 & 2.2) has an unbounded solution because can be send an infinite amount of flow along W. The shortest path problem with a negative cycle is substantially harder to solve than is the shortest part problem without a negative cycle. Indeed, because the shortest path problem is an Non-deterministic Polynomial time (NP)-complete problem, no polynomial-time algorithm for this problem is likely exist.
Negative cycles complicate matters, in part, for the following reason. All algorithms that are capable of solving shortest path problems with negative length arc essentially determine shortest length directed walks form the source to other nodes. If the network contains no negative cycle, then some shortest length directed walk is a path (i.e., does not repeat nodes), since directed cycles from this walk can be eliminated without increasing its length.
The situation for networks with negative cycles is quite different; in these situations, the shortest length directed walk might traverse a negative cycle an infinite number of times since each such repetition reduces the length of the walk.
In these cases in needed to avoid walks that revisit nodes; the addition of this apparently mild stipulation has significant computational implications: with it, the shortest path problem becomes substantially more difficult to solve.

Assumption 4 *The network is directed*

If the network were undirected and all arc lengths were nonnegative, the shortest path problem could be transformed to one on a directed network[5].

Various Types of Shortest Path Problems

Researchers have studied several different types of (directed) shortest path problems:

1. Finding shortest paths form one node to all other nodes when arc lengths are nonnegative
2. Finding shortest path form one node to all others nodes for networks with arbitrary arc lengths
3. Finding shortest paths form every node to every node
4. Various generalizations of the shortest path problem

Analog Solution of the Shortest Path Problem

The shortest path problem has a particularly simple structure that has allowed researchers to develop several intuitively appealing algorithms for solving it. The following analog model for the shortest path problem (with nonnegative arc lengths) provides valuable insight that helps in understanding some of the essential features for the shortest path problem.

Consider a shortest path problem between a specified pair of nodes s and t (this discussion extends easily for the general shortest path model with multiple destination nodes and with nonnegative arc lengths).
Construct a string model with nodes represented by knots, and for any arc $(i,\ j)$ in A, a string with length qual to c_{ij} joining the two knots i and j.
Assume that none of the strings can be stretched. After constructing the model, hold the knot representing node s in one hand, the knot representing node t in the other hand, and pull the hands apart.
One or more paths will be held tight; these are the shortest paths from node s to node t.
Several insights about the shortest path can be extracted from this simple string model:

1. For any arc on shortest path, the string will be taut. Therefore, the shortest path distance between any two successive nodes i and j on this path will equal the length c_{ij} of the arc $(i,\ j)$ between these nodes.
2. For any two nodes i and j on the shortest path (which need not be successive nodes on the path) that are connected by an arc $(i,\ j)$ in A, the shortest path distance from the source to node i plus c_{ij} (a composite distance) is always as large as the shortest path distance from the source to node j. The composite distance might be larger because the string between nodes i and j might not be taut.
3. To solve the shortest path problem, a *maximization* problem have been solved (by pulling the string apart). In general, all network flow problems modeled as minimization problems have an associated "dual" maximization problem; by solving one problem, generally solve the other as well.

Label-Setting and Label-Correcting Algorithms

The network flow literature typically classifies algorithm approaches for solving shortest path problems into two groups: *label setting* and *label correcting*.
Both approaches are iterative. They assign tentative distance labels to nodes at each step; the distance labels are estimates of (i.e.; upper bounds on) the shortest path distances. The approaches vary in how they update the distance labels from step to step and how they "converge" toward the shortest path distances.
Label-setting algorithms designate one label as permanent (optimal) at each iteration.
In contrast, label-correcting algorithms consider all label as temporary until the final step, when they all become permanent. One distinguished feature of these approaches is the class of problems that they solve.
Label-setting algorithms are applicable only to: (1)shortest path problems defined on acyclic network with arbitrary arc lengths and (2) shortest path problems with nonnegative arc lengths.

The label-correcting algorithms are more general apply to all classes of problems, including those with negative arc lengths.
The label-setting algorithms are much more efficient, that is, have much better worst-case complexity bounds; on the other hand, the label-correcting algorithms not only apply to more general classes of problems, but they also offer more algorithmic flexibility. In fact, the label-setting algorithms can be viewed as a special case of the label-correcting algorithms.

2.1.2 Applications

Shortest path problems arise in a wide variety of practical problem settings, both as stand-alone models and as subproblems in more complex problem settings.
For example, they arise in the telecommunications and transportation industries whenever is needed to send a message or a vehicle between two geographical locations as quickly or as cheaply as possible.
Urban traffic planning provides another important example: The models that urban planners use for computing traffic flow patterns are complex nonlinear optimization problems or complex equilibrium models; they build, however, on the behavioral assumption that user of the transportation system travel, with respect to prevailing traffic congestion, along shortest paths form their origins to their destinations. Consequently, most algorithmic approaches for finding urban traffic patterns solve a large number of shortest path problems as subproblems (one for each origin-destination pair in the network)[6]
They are used on applications that includes urban housing, project management, inventory planning and DNA sequencing.
They are also used on generic mathematical applications - approximating functions, solving certain types of difference equations, and solving the so-called knapsack problem - as well as direct applications in the domains of production planning, telephone operator scheduling, vehicle fleet planning and Internet.

2.1.3 Tree of Shortest Paths

In the shortest path problem, the goal is determinate a shortest path from the source node to all other $(n - 1)$ nodes[2]. How much storage would be needed to store these paths? One naive answer would be an upper bound of $(n - 1)^2$ since each path could contain at most $(n - 1)$ arcs.
Fortunately, is not needed this much amount of storage: $(n - 1)$ storage locations are sufficient to represent all these paths. This result follows from the fact that is always

possible find a directed out-tree rooted from the source with the property that the unique path from the source to any node is a shortest path to that node. It is refer to such a tree as a *shortest path tree.*

Each shortest path algorithm is capable of determining this tree as it computes the shortest path distances. The existence of the shortest path tree relies on the following property.

Property 1.3.1. *If the path $s = i_1$ - i_2 - . . . . - $i_h = k$ is a shortest path from node s to node k, then for every $q = 2, 3, . . ., h -1$, the subpath $s = i_1$ - i_2 - . . . . - i_q is a shortest path from the source node to node i_1*

Property 1.3.2. *Let the vector d represent the shortest path distances. Then a directed path P from the source node to node k is a shortest a shortest path if and only if $d(j) = d(i) + c_{ij}$ for every arc $(i, j) \in P$*

2.1.4 Shortest Path Problems in Acyclic Networks

A network is said to be *acyclic* if it contains no directed cycle. In this subsection will be shown how to solve the shortest path problem on an acyclic network in *O(m)* time even though the arc length might be negative.

It always possible to number (on order) nodes in an acyclic network $G = (N, A)$ in *O(m)* time so that *i ¡ j* for every arc $(i, j) \in$ A. This ordering of nodes is called a *topological ordering.* Conceptually, once is determined the topological ordering, the shortest path problem is quite easy to solve by a simple dynamic programming algorithm.

Suppose that are determined the shortest path distances *d(i)* from the source node to nodes $i = 1, 2, . . ., k - 1$.

Consider node *k*. The topological ordering implies that all the arcs directed into this node amanate from one of the nodes 1 trough *k - 1.*

By Property 1.3.1, the shortest path node *k* is composed of a shortest path to one of the nodes $i = 1, 2, . . ., k - 1$ together with the arc (i, k). Therefore, to compute the shortest path distance to node *k*, is needed only select the minimum of $d(i) + c_{ik}$ for all incoming arcs (i, k).

This algorithm is a *pulling* algorithm in that to find the shortest path distance to any node, it "pulls" shortest path distances forward from lower-numbered nodes. Notice that to implement this algorithm, is needed to access conveniently all the arcs directed into each node. Since is frequently store the adjacency list *A(i)* of each node *i*, which gives the arcs emanating out of a node, is also likely to implement a *reaching* algorithm that propagates information from each node to higher-indexed nodes, and so uses the usual adjacency list. Such algorithm is described next.

First, set $d(s) = 0$ and the remaining distance labels to a very large number.

Then, examine nodes in the topological order and for each node i being examined, scan arcs in $A(i)$. If for any arc $(i,\ j) \in A(i)$, is found that $d(j)$ ¿ $d(i) + c_{ij}$, then set $d(j) = d(i) + c_{ij}$.
When the algorithm has examined all the nodes once in this order, the distance labels are optimal.
Theorem 1.1 *The reaching algorithm solves the shortest path problem on acyclic networks in O(m) time*

In this subsection it's shown how can solve the shortest path problem on acyclic networks very efficiently using the simplest possible algorithm. Unfortunately, is not possible to apply this one-pass algorithm, and examine each node and each arc exactly once, for networks containing cycles; nevertheless, it's possible utilize the same basic reaching strategy used in this algorithm and solve any shortest path problem with nonnegative arc lengths using a modest additional amount of work.

2.2 Dijkstra's Algorithm

Dijkstra's algorithm find shortest paths from the source node s to all other nodes in a network with nonnegative arc lengths. Dijkstra's algorithm maintains a distance label $d(i)$ with each node i, which is an upper bound on the shortest path length to node i.[2]
At any intermediate step, the algorithm divides the nodes into two groups: those whic it designates as *permanently labeled* (or permanent) and those it designates as *temporarily labeled* (or temporary).
The distance to any permanent node represents the shortest distance from the source to that node. For any temporary node, the distance label is an upper bound on the shortest path distance to that node.
The basic idea of the algorithm is to fan out from node s and permanently label nodes in the order of their distance from node s.
Initially is given to node s a permanent label of zero, and each other other node j a temporary label equal to ∞. At each iteration, the label of a node i is its shortest distance from the source node along a path whose internal nodes (i.e., nodes other than s or the node i itself) are all permanently labeled. The algorithm selects a node i with the minimum temporary label (breaking ties arbitrarily), makes it permanent, and reaches out from that node - that os, scans arcs in $A(i)$ to update the distance labels of adjacent nodes.
The algorithm terminates when it has designated all nodes as permanent.
The correctness of the algorithm relies on the key observation that is always possible designate the node with the minimum temporary label as permanent.

Dijkstra's algorithm maintains a directed out-tree T rooted at the source that spans the nodes with finite distance labels. The algorithm maintains this tree using predecessor indices (i.e., if $(i,\ j) \in T$, then *pred(j)* $=$ i). The algorithm maintains the invariant property that every tree arc $(i,\ j)$ satisfies the condition $d(j) = d(i) + c_{ij}$ with respect to the current distance labels. At termination, when distance labels represent shortest path distances, T is a shortest path tree (From Property 1.3.2).
In Dijkstra's algorithm, refer to the operation of selecting a minimum temporary distance label as a *node selection* operation. To refer the operation of checking whether the current labels for nodes i and j satisfy the condition $d(j)$ ¿ $d(i) + c_{ij}$ and, if so, then setting $d(j) = d(i) + c_{ij}$ as a *distance update* operation.

2.2.1 Running Time Dijkstra's Algorithm

This subsection studies the worst-case complexity of Dijkstra's algorithm. Might view the computational time for Dijkstra's algorithm as allocated to the following two basic operations:

1. *Node selections.* The algorithm performs this operation n times and each such operation reuqieres that it scans each temporarily labeled node. Therefore, the total node selection time is $n + (n - 1) + (n + 2) + . . . + 1 = O(n^2)$.
2. *Distance updates.* The algorithm performs this operation $|A(i)|$ times for node i. Overall, the algorithm performs this operation $\sum_{i\in N} |A(i)| = m$ times. Since each distance update operation requires $O(1)$ time, the algorithm requires $O(m)$ total time for updating all distance labels.

These operations established the following result.
Theorem 1.2 *Dijkstra's algorithm solves the shortest path problem in $O(n^2)$ time*

The $O(n^2)$ time bound for Dijkstra's algorithm is the best possible for completely dense networks (i.e., $m = \Omega(n^2)$), but can be improved for sparse networks.
Notice that the times required by the node selections and distances updates are not balanced. The node selection requires a total of $O(n^2)$ time, and the distance updates require only $O(m)$ time.
Researchers have attempted to reduce the node selection time without substantially increasing the time for updating the distances. Consequently, they have, using clever data structures, suggested several implementations of the algorithm. These implementations have either dramatically reduced the running time of the algorithm in practice or imporved its worst-case complexity.

2.2.2 Reverse Dijkstra's Algorithm

In the (forward) Dijkstra's algorithm, has been determined a shortest path from node s to every other node in N - $\{s\}$.
Suppose that is required to determine a shortest path from every node in N - $\{t\}$ to a sink node *t*. To solve this problem, its used a slight modification of Dijkstra's algorithm, refer as the *reverse Dijkstra's algorithm.*
The reverse Dijkstra's algorithm maintains a distance *d'(j)* with each node j, which is an upper bound on the shortest path length from node j to node *t*.
As before, the algorithm designates a set of nodes, say *S'*, as permanently labeled and the remaining set of nodes, say $\overline{S}'$, as temporarily labeled. At each iteration, the algorithm designates a node with the minimum temporary distance label, say *d'(j)*, as permanent. It then examines each incoming arc (i, j) and modifies the distance label of node i to min$\{$*d'(i)*, c_{ij} + *d'(j)*$\}$. The algorithm terminates when all the nodes have become permanently labeled.

2.2.3 Bidirectional Dijkstra's Algorithm

In some applications of the shortest path problem, is needed not determine a shortest path from node s to every other node in the network.
Suppose, instead, that is wanted to determine a shortest path from node s to a specified node *t*. To solve this problem and eliminate some computations, is possible to terminate Dijkstra's algorithm as soon as it has selected t from $\overline{S}$ (even though some nodes are still temporarily labeled). The bidirectional Dijkstra's algorithm, which is described next, allow to solve this problem even faster in practice (though not in the worse case)[7]
In the bidirectional Dijkstra's algorithm, is applied simultaneously the forward Dijkstra's algorithm from node s and reverse Dijkstra's algorithm from node *t*. The algorithm alternatively designates a node in $\overline{S}$ and a node in $\overline{S}$' as permanent until both the forward and reverse algorithms have permanently labeled the same node, say node k (i.e., $S \cap S' = \{k\}$).
At this point, let *P(i)* denote the shortest path from node s to node $i \in S$ found by the forward Dijkstra's algorithm, and let *P'(j)* denote the shortest path from node $j \in S'$ to node t found by the reverse Dijkstra's algorithm.
A straightforward argument shows that the shortest path from node s to node t is either the path *P(k)* $\cup$ *P'(k)* or a path *P(i)* $\cup$ $\{(i, j)\}$ $\cup$ *P'(j)* for some arc (i, j), $i \in S$ and $j \in S'$. This algorithm is very efficient because it tends to permanently label few nodes and hence examines the arcs incident to a large number of nodes.

2.3 Genetic Algorithms

2.3.1 Introduction

Genetic algorithms are search algorithms based on the mechanics of natural selection and natural genetics. They combine survival of the fittest among string structures with a structured yet randomized information exchange to form a search algorithm with some of the innovative flair of human search. In every generation, a new set of artificial creatures (strings) is created using bits an pieces of the fittest of the old; an occasional new part is tried for good measure.
While randomized, genetic algorithms are no simple random walk. They efficiently exploit historical information to speculate on new search point with expected improved performance.
Genetic algorithms have been developed by John Holland, his colleagues, and his students at the University of Michigan. The goal of their research have been twofold: (1) to abstract and rigorously explain the adaptive processes of natural systems, and (2) to design artificial systems software that retains the important mechanisms of natural systems. This approach has led to important discoveries in both natural and artificial system science.
The central theme of research on genetic algorithms has been *robustness*, the balance between efficiency and efficacy necessary for survival in many different environments. The implications of robustness for artificial for artificial systems are manifold. If artificial systems can be made more robust, costly redesigns can be reduced or eliminated. If higher levels of adaptation can be achieved, existing systems can perform their functions longer and better. Designers of artificial systems - both software and hardware, whether engineering systems, computer systems, or business systems - can take advantage of the robustness, the efficiency, and the flexibility of biological systems. Features for self-repair, self-guidance, and reproduction are the rule in biological systems, whereas they barely exist in the most sophisticated artificial systems.

2.3.2 Robustness of Traditional Optimization and Search Methods

The current literature identifies three main types of search methods: calculus-based, enumerative, and random.
Calculus-based methods have been studied heavily. These subdivide into two main classes: indirect and direct. Indirect methods seek local extrema by solving the usually nonlinear set of equations resulting from setting the gradient of the objective function

equal to zero. This is the multidimensional generalization of the elementary calculus notion of extremal points, as illustrated in Figure 2.1

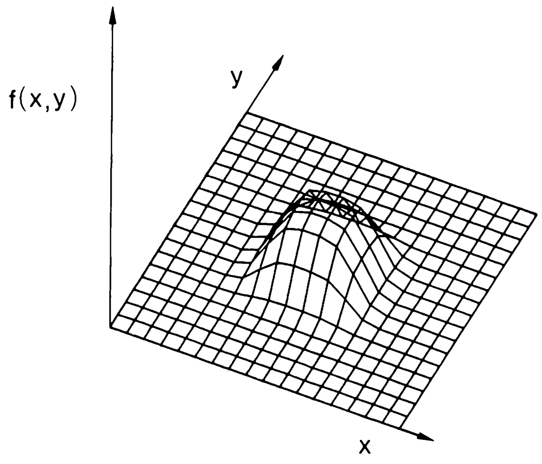


FIGURE 2.1: The single-peak function is easy for calculus-based methods.[8]

Given a smooth, unconstrained function, finding a possible peak starts by restricting search to those points with slopes of zero in all directions. On the other hand, direct (search) methods seek local optima by hopping in the function and moving in a direction related to the local gradient. This is simply the notion of *hill-climbing*: to find the local best, climb the function in the steepest permissible direction. While both of these calculus-based methods have been improves, extended, hashed, and rehashed, some simple reasoning shows their lack of robustness.
First, both methods are local in scope; the optima they seek are the best in a neighborhood of the current point. For example, suppose that Figure 2.1 shows a portion of complete domain of interest; a more complete picture is shown in Figure 2.2. Clearly, starting the search or zero-finding procedures in the neighborhood of the lower peak will cause to miss the main event (the higher peak). Furthermore, once the lower peak is reached, further improvement must be sought through random restart or they trickery. Second, calculus-based methods depend upon the existence of derivates, this is a severe shortcoming. Many practical parameters spaces have little respect for the notion of a derivate and the smoothness this implies.

The real world of search is fraught with discontinuities and vast multimodal. noisy search spaces as depicted in a less calculus-friendly function like the one shown on Figure 2.3. It comes as no surprise that methods depending upon the restrictive requirements of continuity and derivative existence are unsuitable for all but a very limited problem

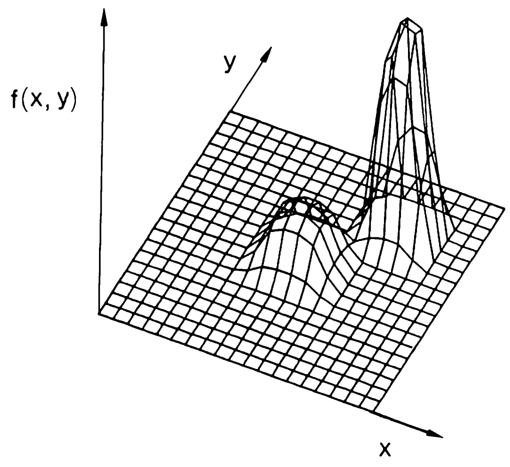


Figure 2.2: The multiple-peak function[8]

domain. For this reason and because of their inherently local scope search, the calculus-based methods are insufficient to solve these type of problems. They are insufficiently robust in unintended domains.

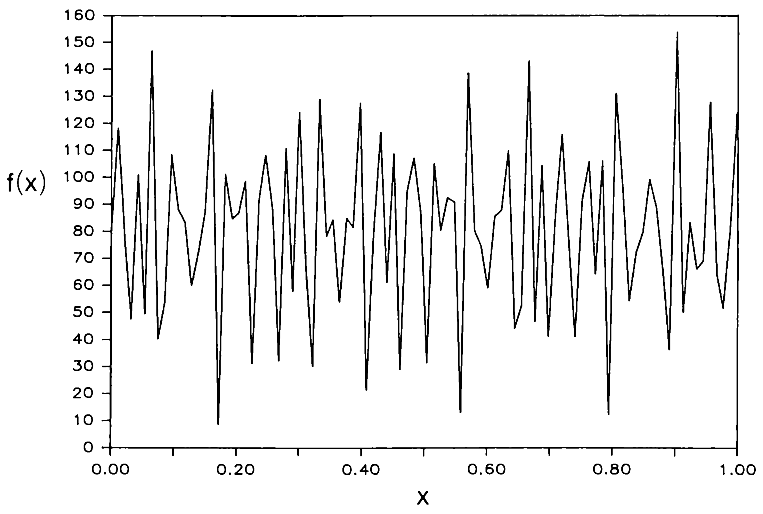


Figure 2.3: A typical function found in real world[8]

Enumerative schemes have been considered in may shapes and sizes. The idea is fairly straightforward; within a finite search space, or a discretized infinite search space, the search algorithm starts looking at objective function values at every point in the space, one at time. Although the simplicity of this type of algorithm is attractive, and enumeration is a very kind of search (when the number of possibilities is small), such schemes must ultimately be discounted in the robustness race for one simple reason: lack of efficiency. Many practical spaces are simply too large to search one at a time and still have a chance of using the information to some practical end. Even the highly touted enumerative scheme *dynamic programming* breaks down on problems of moderate size

and complexity, suffering from a malady melodramatically labeled ”the curse of dimensionality” by its creator (Bellman, 1961). Concluding, less clever enumerative schemes are similarly, and more abundantly, cursed for real problems.
Random search algorithms have achieved increasing popularity as researchers have recognized the shortcomings of calculus-based and enumerative schemes. Yet, random walks and random schemes that search and save the best must also be discounted because of the efficiency requirement. Random searches, in the log run, can be expected to do no better than enumerative schemes. But before discount strictly random search methods, they must be separated from randomized techniques.
The genetic algorithm is an example of a search procedure that uses random choice as a tool to guide a highly exploitative search through a coding of a parameter space. Using random choice as a tool in a directed search process seems strange at first, but nature contains many examples. Another currently popular search technique, *simulated annealing*, uses random processes to help guide its form of search for minimal energy states.

2.3.3 The Goals of Optimization

Optimization seeks to improve performance toward some optimal point or points. Note that this definition has two parts: (1) seek improvement to approach some (2) optimal point. There is a clear distinction between the *process* of imporvement and the *destination* or optimum itself. Yet, in judging optimization procedures, typically the target focused is solely the convergence (does the method reach the optimum?) and forget entirely about interim performance. This emphasis stems form the origins of optimization in calculus. It is not, however, a natural emphasis.
Consider a human decision maker, for example, a businessman. How to judge his decisions? What criteria is used to decide whether he has done a good or bad job? Usually the criteria is make adequate selections within the times and resources allotted. Does he produce a better widget? Does he get it to market more efficiently? With better promotion? Thee businessman is never judged by an attainment of the best criteria; perfection is all too stern a taskmaster.
As a result, the conclusion is that convergence to the best is not an issue in business or in most work scenarios, the only concerned matter is doing better relative to others[9]
This, to reach more humanlike optimization tools, the priorities of optimization should be reordered. The most important goal of optimization is improvement.
Attainment of the optimum is much less important for complex systems. It would be nice to be perfect: meanwhile, we can only strive to improve.

2.3.4 Differences between Genetic Algorithms and Traditional Methods

In order for genetic algorithms to surpass the more traditional methods in quest for robustness, Genetic Algorithmss (GAs) must differ in some very fundamental ways. GAs are different from more normal optimization and search procedure in four ways:

1. GAs work with a coding of the parameter set, not the parameters themselves.
2. GAs search from a population of points, not a single point.
3. GAs use payoff (objective function) information, not derivatives or other auxiliary knowledge.
4. GAs use probabilistic transition rules, not deterministic rules.

GAs require the natural parameter set of the optimization problem to be coded as a finite-length string over some finite alphabet. As an example, consider the optimization problem posed in Figure 2.4

The target is to maximize the function $f(x) = x^2$ on the integer interval [0, 31]. With more traditional methods, we would be tempted to twiddle with the parameter x, until the highest objective function value is reached.

With glsplga, the first step of the optimization process is to code the parameter x as a finite-length string. Let consider an optimization problem where the coding comes a bit more naturally.

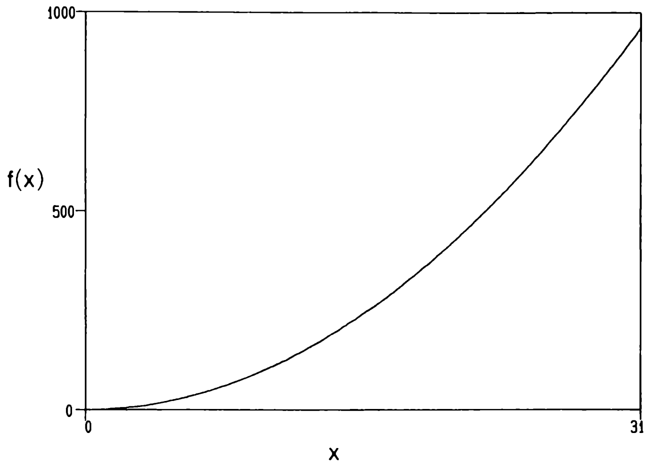


FIGURE 2.4: A simple function optimization example, the function $f(x) = x^2$ on the integer interval [0, 31][8]

Consider the black box switching problem illustrated in Figure 2.5. This problems concerns a black box device with a bank of five input switches. For every setting of the five switches, there is an output signal *f*, mathematically *f* = *f(s)*, where *s* is a particular setting of the five switches.

The objective of the problem is to set the switches to obtain the maximum possible f value. With other methods of optimization, we might work directly with the parameter set (the switch settings) and toggle switches from one setting to another using the transition rules of our particular method.

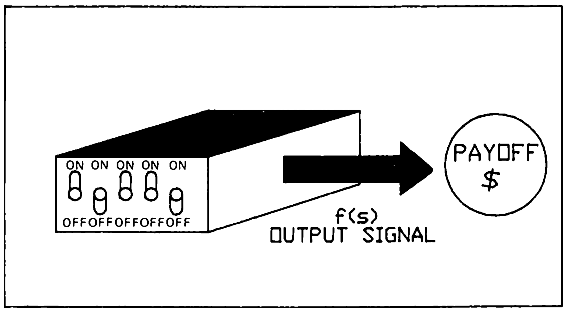


FIGURE 2.5: A black box optimization problem with five on-off switches illustrates the idea of a coding and a payoff measure. Genetic algorithms only require these two things: they don't need to know the workings of the black box[8]

With genetic algorithms, we first code the switches as a finite-length string. A simple code can be generated by considering a string of five 1's and 0's where each of the five switches is represented by a 1 if the switch is on and a 0 is the switch is off. With this coding, the string 11110 codes the setting where the first four switches are in and the fifth switch is off. Some of the codings introduced later will not be obvious, but at this juncture we acknowledge that genetic algorithms use codings.

In many optimization methods, we move gingerly from a single point in the decision space to the next using some transitions rule to determine the next point. This point-to-point method is dangerous because it is a perfect prescription for locating flase peaks in multimodal (many-peaked) search spaces.

By contrast, GAs work from a rich database of points simultaneously (a population of strings), climbing many peaks in parallel; thus, the probability of finding a false peak is reduced over methods that go point to point. As an example, consider the black box optimization problem (Figure 2.5) again.

Other techniques for solving this problem might start with one set of switch settings, apply some transition rules, and generate a new trial switch setting.

A genetic algorithm starts with a population of strings and thereafter generates successive populations of strings. For example, in the five-switch problem, a random start using successive coin flips (head = 1, tail = 0) might generate the initial population of size $n = 4$ (small by genetic algorithm standards):

```
01101
```

```
11000
01000
10011
```

After this start, successive populations are generated using the genetic algorithm. By working form a population of well-adapted diversity instead of a single point, the genetic algorithm adheres to the old adage that there is safety in numbers; this parallel flavor contributes to a genetic algorithm's robustness[10].

Many search techniques require much auxiliary information in order to work properly. For example, gradient techniques need derivates (calculated analytically or numerically) in order to be able to climb the current peak, and other local search procedures like the techniques of combinatorial optimization requires access to most if not all tabular parameters.

By contrast, genetic algorithms have no need for all this auxiliary information: GAs are blind. To perform an effective search for better and better structures, they only require payoff value (objective function values) associated with individual strings. This characteristic makes a GA a more canonical method than many search schemes. After all, every search problem has a metric (or metrics) relevant to the search; however, different search problems have vastly different forms of auxiliary information.

On the other hand, the refusal to use specific knowledge when it does exist can place an upper bound on the performance of an algorithm when it goes head to head with methods designed for that problem.

Unlike many methods, GAs use probabilistic transition rules to guide their search. To persons familiar with deterministic methods this seems odd, but the use of probability does not suggest that the method is some simple random search; this is not decision making at the toss of a coin. Genetic algorithms use random choice as a tool to guide a search toward regions of the search space with likely improvement.

Taken together, these four differences - direct use of a coding, search from a population, blindness to auxiliary information, and randomized operators - contribute to a genetic algorithm's robustness and resulting advantage over other more commonly used techniques.

2.3.5 A Simple Genetic Algorithm

The mechanics of a simple genetic algorithm are surprisingly simple, involving nothing more complex than copying strings and swapping partial strings. The explanation of why this simple process works is much more subtle and powerful. Simplicity of operation and power of effect are two of the main attractions of the genetic algorithm approach.

The previous subsection pointed out how genetic algorithm process populations of strings.

Recalling the black box switching problem, remember that the initial population had four strings:

```
01101
11000
01000
10011
```

Also recall that this population was chosen at random through 20 successive flips of an unbiased coin. We now must define a set of simple operations that take this initial population and generate successive population that (hopefully) improve over time.
A simple genetic algorithm that yields good results un many practical problems is composed of three operators:

1. Reproduction
2. Crossover
3. Mutation

Reproduction is a process in which individual strings are copied according to their objective function values, f (biologist call this function the *fitness* function). Intuitively, we can think of the function f as some measure of profit, utility, or goodness that we want to maximize. Copying strings according to their fitness values means that strings with higher value have a higher probability of contributing one or more offspring in the next generation. This operator, of course, is an artificial version of natural selection, a Darwinian survival of the fittest among string creatures. In natural populations fitness is determined by a creature's ability to survives predators, pestilence, and the other obstacles to adulthood and subsequent reproduction. In our unabashedly artificial setting, the objective functions is the final arbiter of the string-creature's life or death.

The reproduction operator may be implemented in algorithmic form in a number of ways. Perhaps the easiest is to create a biased roulette wheel where each current string in the population has a roulette wheel slot sized in proportion to its fitness. Suppose the sample population has a roulette wheel slot sized in proportion to its fitness. Suppose the sample population of four strings in the black box problem has objective or fitness function values f as shown in Table 2.1.

Summing the fitness over all four strings, we obtain a total of 1170. The percentage of population total fitness is also shown in the table. The corresponding weighted roulette wheel for this generation's reproduction is shown in Figure 2.6. To reproduce, we simply spin the weighted roulette wheel thus defines four times. For the example problem, string

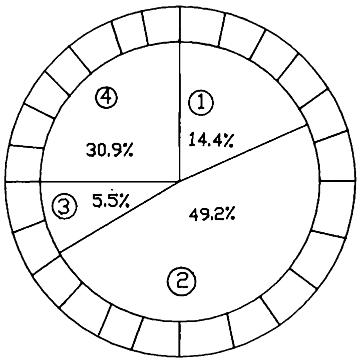


FIGURE 2.6: Simple reproduction allocates offspring strings using a roulette wheel with slots sized according to fitness[8]

No.	String	Fitness	% of Total
1	01101	169	14.4
2	11000	576	49.2
3	01000	64	5.5
4	10011	361	30.9
Total		1170	100.0

TABLE 2.1: Sample Problem Strings and Fitness Values

number 1 has a fitness value of 169, which represents 14.4 percent of the total fitness. As a result, string 1 is given 14.4 percent of the biased roulette wheel, and each spin turns up string 1 with probability 0.144. Each time we require another offspring, a simple spin of the weighted roulette wheel yields the reproduction candidate. In this way, more highly fit strings have a higher number of offspring in the succeeding generation. Once a string has been selected for reproduction, an exact replica of the string is made. This string is then entered into a mating pool, a tentative new population, for further genetic operator action.

After reproduction, simple crossover (Figure 2.7) may proceed in two steps. First, members of the newly reproduces strings in the mating pool are mated at random. Second, each pair of strings undergoes crossing over as follows: an integer position k along the string is selected uniformly at random and the string length less one *[1, l - 1]*. Two new strings are created by swapping all characters between positions k *+ 1* and l inclusively. For example, consider strings A_1 and A_2 from our example initial population:

A_1 *= 0 1 1 0 | 1*

A_2 *= 1 1 0 0 | 0*

Suppose in choosing a random number between 1 and 4, we obtain a $k = 4$ (as indicated by the separator symbol |). The resulting crossover yields two new strings where the prime (') means the strings are part of the new generation:

$A'_1 = \mathit{0\ 1\ 1\ 0\ 0}$

$A'_2 = \mathit{1\ 1\ 0\ 0\ 1}$

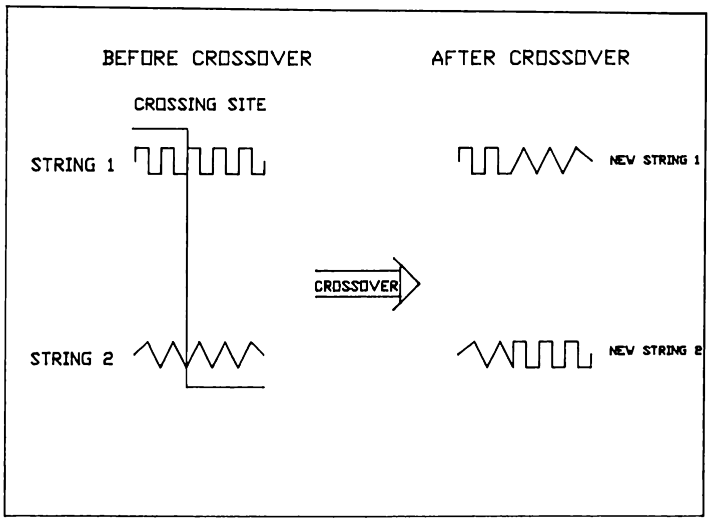


FIGURE 2.7: A schematic of simple crossover shows the alignment of two strings and the partial exchange of information, using a cross site chosen at random[8]

The mechanics of reproduction and crossover are surprisingly simple, involving random number generation, string copies, and some partial string exchanges. Nonetheless, the combined emphasis of reproduction and the structured, though randomized, information exchange of crossover give genetic algorithms much of their power.

If reproduction according to fitness combined with crossover gives genetic algorithms the bulk of their processing power, what then is the purpose of the mutation operator? Mutation is needed because, even though reproduction and crossover effectively search and recombine extant notions, occasionally they may become overzealous and lose some potentially useful genetic material (1's or 0's at particular locations)[10]. In artificial genetic systems, the mutation operator protects against such an irrecoverable loss. In the simple GA, mutation is the occasional (with small probability) random alteration of the value of a string position. In the binary coding of the black box problem, this simply means changing a 1 to a 0 and vice versa. By itself, mutation is a random walk through the string space. When used sparingly with reproduction and crossover, it is an insurance policy against premature loss of important notions.

That the mutation operator plays a secondary role in the simple GA, we simply note that the frequency of mutation to obtain per thousand bit (position) transfers. Mutation rates are similarly small (or smaller) in natural populations, leading us to conclude that mutation is appropriately considered as a secondary mechanism of genetic algorithm adaptation.

Chapter 3

The Parallella Board

3.1 Introduction

3.1.1 Overview

The Parallella computer is a high performance, credit card sized computer based on the Epiphany multi-core chips from Adapteva. The Parallella can be used as a standalone computer, an embedded device or as a component in a scaled out parallel server cluster. The Parallella includes a low power dual core ARM A9 processor and runs several of the popular Linux distributions, including Ubuntu. For the development of the thesis, I've used the Linaro distribution, a lightweight open source OS designed to work on ARM architectures. The unique Epiphany co-processor chips consists of a scalable array of simple RISC processors programmable in bare metal C/C++ or in a parallel programming frameworks like OpenCL, MPI, and OpenMP. The mesh of independent cores are connected together with a fast on-chip-network within a distributed shared memory architecture.

3.1.2 Technical Specifications

The Parallella Board has the below configuration:

- Zynq Z7020 Dual-core ARM A9 CPU
- 16-core Epiphany Coprocessor
- 1GB RAM
- MicroSD Card reader
- USB 2.0

- Up to 48 GPIO signal
- Gigabit Ethernet
- HDMI port
- 54mm x 87mm form factor

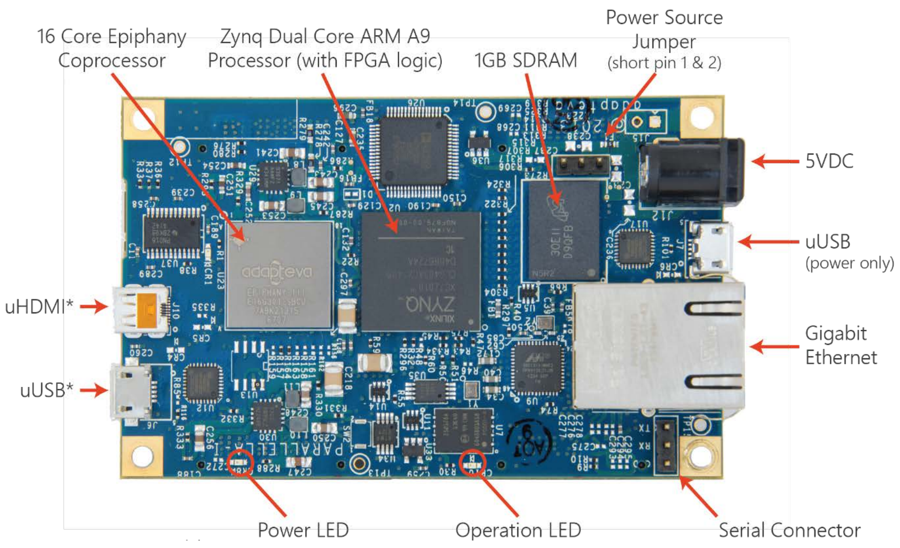

16 Core Epiphany Coprocessor
Zynq Dual Core ARM A9 Processor (with FPGA logic)
1GB SDRAM
Power Source Jumper (short pin 1 & 2)
5VDC
uUSB (power only)
Gigabit Ethernet
uHDMI*
uUSB*
Power LED
Operation LED
Serial Connector

FIGURE 3.1: Parallella Board (top view)[11]

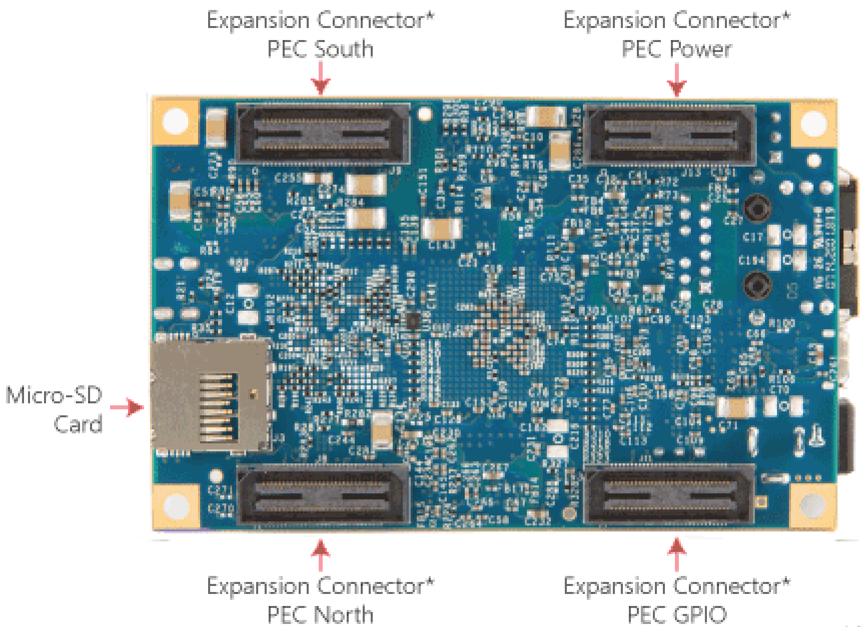

Expansion Connector* PEC South
Expansion Connector* PEC Power
Micro-SD Card
Expansion Connector* PEC North
Expansion Connector* PEC GPIO

FIGURE 3.2: Parallella Board (bottom view)[11]

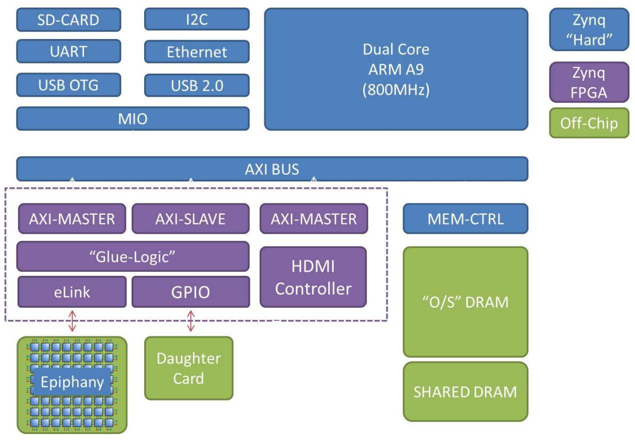


FIGURE 3.3: Parallella High Level Architecture[12]

3.2 Epiphany Architecture

3.2.1 Introduction

The Epiphany architecture defines a multicore, scalable, shared-memory, parallel computing fabric. It consists of a 2D array of compute nodes connected by a low-latency mesh network-on-chip. These are the key components of the architecture:

- **Processor:** 16 superscalar floating point RISC CPUs (eCore), each one capable of two floating point operations per clock cycle and one integer calculation per clock cycle. The CPU has a general-purpose instruction set and is programmable with C/C++.
- **Memory System:** The Epiphany memory architecture is based on a flat shared memory map in which each compute node has up to 1MB of local memory as a unique addressable slice of the total 32-bit address space. A processor can access its own local memory and other processor's memory through regular load/store instructions. The local memory system is comprised of 4 separate sub-banks, allowing for simultaneous memory accesses by the instruction fetch engine, local load-store instructions, and by memory transactions initiated by the DMA engine other processors within system.
- **Network-On-Chip:** The Epiphany Network-on-Chip (eMesh) is a 2D mesh network that handles all on-chip and off- chip communication. The eMesh network uses atomic 32-bit memory transactions and operates without the need for any special programming. The network consists of three separate and orthogonal mesh structures, each serving different types of transaction traffic: one network for on-chip write traffic, one network for off chip write traffic, and one network for all read traffic.

- **Off-Chip IO:** The eMesh network and memory architecture extends off-chip using source synchronous dual data rate LVDS links (”elinks”). Each eCore has 4 independent off-chip elinks, one in each physical direction (north, east, west and south). The off chip links allows for glueless connection of multiple eCore chips on a board and for interfacing to an FPGA.

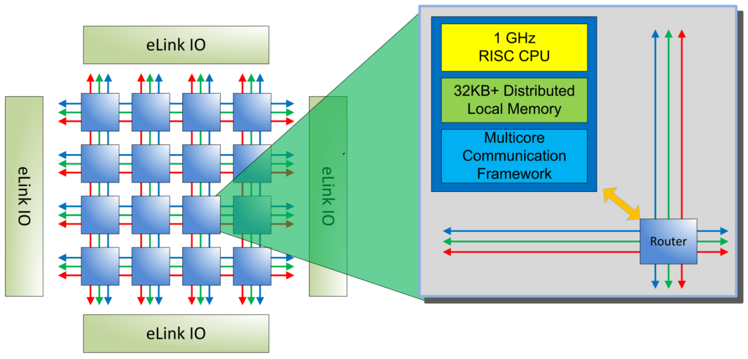

eLink IO
eLink IO
eLink IO
eLink IO
1 GHz RISC CPU
32KB+ Distributed Local Memory
Multicore Communication Framework
Router

FIGURE 3.4: eCore Architecture[12]

3.2.2 System Examples

The Epiphany co-processor can be used in different configurations, some of which are shown in the next figure:

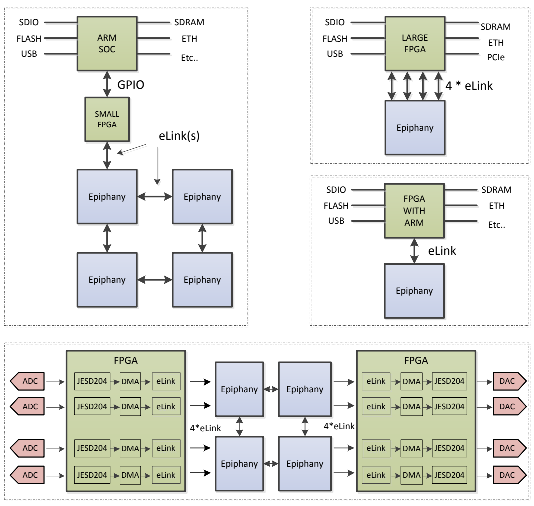


FIGURE 3.5: Epiphany System Architecture[12]

3.2.3 Memory Architecture

To start programming the eCores, is important to know the Memory Architecture. The most restrictives limitation to program the Epiphany co-processor is the lack of a high amount of memory. Nowadays, the maximum capacity of each eCore is 32KB.

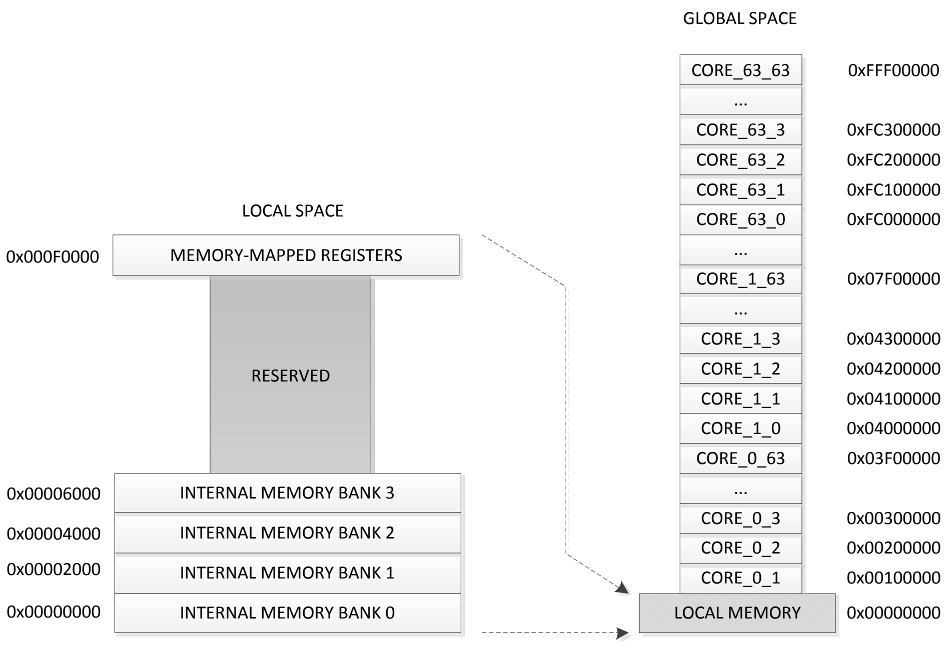
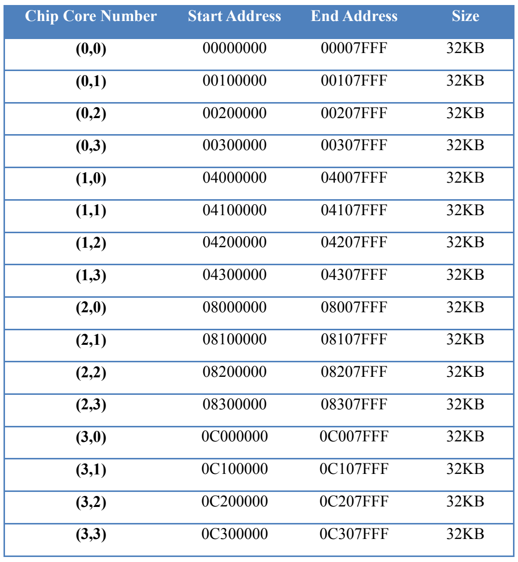


FIGURE 3.6: Epiphany Memory Architecture[12]



Chip Core Number	Start Address	End Address	Size
(0,0)	00000000	00007FFF	32KB
(0,1)	00100000	00107FFF	32KB
(0,2)	00200000	00207FFF	32KB
(0,3)	00300000	00307FFF	32KB
(1,0)	04000000	04007FFF	32KB
(1,1)	04100000	04107FFF	32KB
(1,2)	04200000	04207FFF	32KB
(1,3)	04300000	04307FFF	32KB
(2,0)	08000000	08007FFF	32KB
(2,1)	08100000	08107FFF	32KB
(2,2)	08200000	08207FFF	32KB
(2,3)	08300000	08307FFF	32KB
(3,0)	0C000000	0C007FFF	32KB
(3,1)	0C100000	0C107FFF	32KB
(3,2)	0C200000	0C207FFF	32KB
(3,3)	0C300000	0C307FFF	32KB

FIGURE 3.7: Epiphany Memory Map[12]

3.3 eCore CPU

3.3.1 Overview

The different sub components of the eCore CPU are illustrated in Figure 3.8. The processor includes a general purpose program sequencer, large general purpose register file, integer ALU (IALU), floating point unit (FPU), debug unit, and interrupt controller.

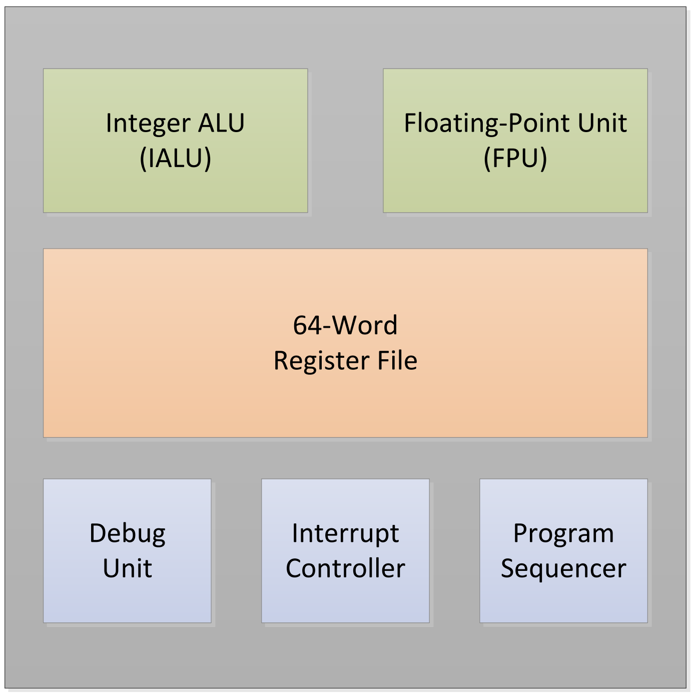


FIGURE 3.8: eCore CPU Overview[12]

Program Sequencer

The program sequencer supports all standard program flows for a general-purpose CPU, including:

- *Loops:* One sequence of instructions is executed several times. Loops are implemented using general-purpose branching instructions, in which case the branching can be done by label or by register.
- *Functions:* The processor temporarily interrupts the sequential flow to execute instructions from another part of the program. The CPU supports all C-function calls, including recursive functions.

- *Jumps:* Program flow is permanently transferred to another part of the program. A jump by register instruction allows program flow to be transferred to any memory location in the 32-bit address space that contains valid program code.
- *Interrupts:* Interrupt servicing is handled by the interrupt controller, which redirects the program sequencer to an interrupt handler at a fixed address associated with the specific interrupt event. Before entering the interrupt service routine, the old value of the program counter is stored so that it can be retrieved later when the interrupt service routine finishes.
- *Idle:* A special instruction that puts the CPU into a low-power state waiting for an interrupt event to return the CPU to normal execution. This idle mode is useful, for example, in signal processing applications that are real-time and data-driven.
- *Linear:* In linear program flows, the program sequencer continuously fetches instructions from memory to ensure that the processor pipeline is fed with a stream of instructions without stalling.

Register File

The 9-port 64-word register file provides operands for the IALU and FPU and serves as a temporary power efficient storage place instead of memory. Arithmetic instructions have direct access to the register file but not to memory. Movement of data between memory and the register file is done through load and store instructions. Having a large number of registers allows more temporary variables to be kept in local storage, thus reducing the number of memory read and write operations. The flat register file allows user to balance resources between floating-point and integer ALU instructions as any one of the 64 registers be used by the floating-point unit or IALU, without restrictions. In every cycle, the register file can simultaneously perform the following operations:

- Three 32-bit floating-point operands can be read and one 32-bit result written by FPU.
- Two 32-bit integer operands can be read and one 32-bit result written by IALU.
- A 64-bit double-word can be written or read using a load/store instruction.

Integer ALU

The Integer ALU (IALU) performs a single 32-bit integer operation per clock cycle. The operations that can be performed are: data load/store, addition, subtraction, logical shift, arithmetic shift, and bitwise operations such as XOR and OR. The IALU's single-cycle execution means the compiler or programmer can schedule integer code without worrying about data-dependency stalls. All IALU operations can be performed in parallel with floating-point operations as long as there are no register-use conflicts between the two instructions. Pre and post modify addressing and double-word load/store capability enables efficient loading and storing of large data arrays.

Floating Point Unit

The floating-point unit (FPU) complies with the single precision floating point IEEE754 standard, executes one floating-point instruction per clock cycle, supports round-to-nearest even and round-to-zero rounding modes, and supports floating-point exception handling. The operations performed are: addition, subtraction, fused multiply-add, fused multiply-subtract, fixed-to-float conversion, absolute, float-to-fixed conversion. Operands are read from the 64-entry register file and are written back to the register file at the end of the operation. No restrictions are placed on register usage. Regular floating-point operations such as floating-point multiply/add read two 32-bit registers and produce a 32-bit result. A fused multiply-add instruction takes three input operands and produces a single accumulated result. A large number of floating-point signal-processing algorithms use the multiply-accumulate operations, and for these applications the fused operations has the potential of reducing the number clock cycles significantly.

Interrupt Controller

The interrupt controller supports up to 10 interrupts and exceptions, with full support for nested interrupts and interrupt masking.

Hardware Loops

Efficient zero overhead loops are supported through built in hardware support.

Debug Unit

The debug unit provide multicore debug capabilities such as: single stepping, breakpoints, halt, and resume.

3.3.2 Data Types

The CPU architecture supports the following integer data types:

- Byte: 8 bits
- Half-Word: 16 bits (must be aligned on 2 byte boundary in memory)
- Double: 64 bits (must be aligned on 8 byte boundary in memory)

The data types can be of signed or unsigned format, as shown below. All register-register operations operate on word types only, but data can be stored in memory as any size. For example, an array of bytes can be stored in memory by an external host, read into the register file using the byte load instruction, operated on as 32-bit integers, and then can stored back into memory using the byte store instruction.

Signed Integer Representation (from Most Significant Bit (MSB) to Least Significant Bit (LSB))
$-\mathbf{a}_{N-1} \cdot 2^{N-1}\mathbf{a}_{N-2} \cdot 2^{N-2}\mathbf{a}_{N-3} \cdot 2^{N-3}\mathbf{a}_{N-4} \cdot 2^{N-4}\mathbf{a}_{N-5} \cdot 2^{N-5} \cdots \mathbf{a}_0 \cdot 2^0$

Unsigned Integer Representation (from MSB to LSB)
$\mathbf{a}_{N-1} \cdot 2^{N-1}\mathbf{a}_{N-2} \cdot 2^{N-2}\mathbf{a}_{N-3} \cdot 2^{N-3}\mathbf{a}_{N-4} \cdot 2^{N-4}\mathbf{a}_{N-5} \cdot 2^{N-5} \cdots \mathbf{a}_0 \cdot 2^0$

Floating-Point Data Types

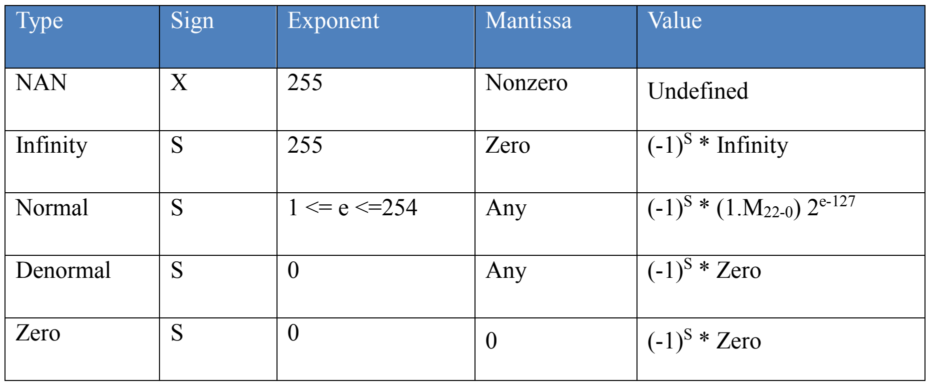
The FPU supports the IEEE754 32-bit single-precision floating-point data format, shown below:

SIGN	EXP[7:0]	MANTISSA[22:0]

A number in this floating-point format consists of a sign bit, s, a 24-bit mantissa, and an 8-bit unsigned-magnitude exponent, e. For normalized numbers, the mantissa consists of a 23-bit fraction, f, and a hidden bit of 1 that is implicitly presumed to precede f22 in the mantissa. The binary point is presumed to lie between this hidden bit and f22. The least-significant bit (LSB) of the fraction is f0; the LSB of the exponent is e0. The hidden bit effectively increases the precision of the floating-point mantissa to 24 bits from the 23 bits actually stored in the data format. This bit also ensures that the mantissa of any number in the IEEE normalized number format is always greater than or equal to 1 and less than 2. The exponent, e, can range between $1 \leq \text{e} \leq 254$ for normal numbers in the single-precision format. This exponent is biased by +127 (254/2). To calculate the true unbiased exponent, 127 must be subtracted from e. The IEEE standard also provides for several special data types in the single-precision floating- point format, including:

- An exponent value of 255 (all ones) with a nonzero fraction is a not-a-number (Not A Number (NAN)). NANs are usually used as flags for data flow control, for the values of uninitialized variables, and for the results of invalid operations such as $0 * \infty$.
- Infinity is represented as an exponent of 255 and a zero fraction. Because the number is signed, both positive and negative infinity can be represented.
- Zero is represented by a zero exponent and a zero fraction. As with infinity, both positive zero and negative zero can be represented. The IEEE single-precision

floating-point data types supported by the processor and their interpretations are summarized in Figure 3.9.



Type	Sign	Exponent	Mantissa	Value
NAN	X	255	Nonzero	Undefined
Infinity	S	255	Zero	$(-1)^S$ * Infinity
Normal	S	1 <= e <=254	Any	$(-1)^S * (1.M_{22-0})\ 2^{e-127}$
Denormal	S	0	Any	$(-1)^S$ * Zero
Zero	S	0	0	$(-1)^S$ * Zero

FIGURE 3.9: IEEE Single-Precision Floating-Point Data Types[1]

The CPU is compatible with the IEEE-754 single-precision format, with the following exceptions:

- No support for inexact flags.
- INAN inputs generate an invalid exception and return a quiet NAN. When one or both of the inputs are NANs, the sign bit of the operation is set as an XOR of the signs of the input sign bits.
- Denormal operands are flushed to zero when input to a computation unit and do not generate an underflow exception. Any denormal or underflow result from an arithmetic operation is flushed to zero and an underflow exception is generated.
- Round to $\pm\infty$ is not supported.

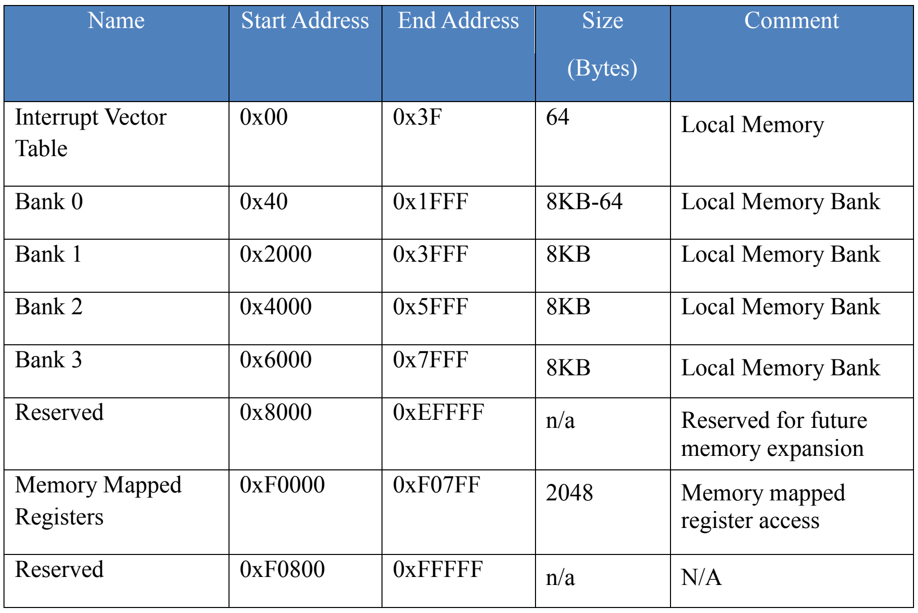
By default, the FPU performs round-to-nearest even IEEE754 floating-point rounding. In this rounding mode, the intermediate result is rounded to the nearest complete number that fits within the final 32-bit floating-point data format. If the result before rounding is exactly halfway between two numbers in the destination format (differing by an LSB), the rounded result is that number which has an LSB equal to zero. Statistically, rounding up occurs as often as rounding down, so there is no large sample bias.

The FPU supports truncation rounding when the rounding mode bit is set in the Core Configuration Register. In truncate rounding mode, the intermediate mantissa result bits that are not within the first 23 bits are ignored. Over a large number of accumulations, there can be a large sample bias in the computation, so truncation rounding mode should be avoided for most applications.

The FPU detects overflow, underflow, and invalid conditions during computations. If one of these conditions is detected, a software exception signal is sent to the interrupt controller to start an exception handling routine. Double-precision floating-point arithmetic is emulated using software libraries and should be avoided if performance considerations outweigh the need for additional precision.

3.3.3 Local Memory Map

Figure 3.10 summarizes the memory map of the eCore CPU local memory. All registers



Name	Start Address	End Address	Size (Bytes)	Comment
Interrupt Vector Table	0x00	0x3F	64	Local Memory
Bank 0	0x40	0x1FFF	8KB-64	Local Memory Bank
Bank 1	0x2000	0x3FFF	8KB	Local Memory Bank
Bank 2	0x4000	0x5FFF	8KB	Local Memory Bank
Bank 3	0x6000	0x7FFF	8KB	Local Memory Bank
Reserved	0x8000	0xEFFFF	n/a	Reserved for future memory expansion
Memory Mapped Registers	0xF0000	0xF07FF	2048	Memory mapped register access
Reserved	0xF0800	0xFFFFF	n/a	N/A

FIGURE 3.10: eCore Local Memory Map Summary[1]

are memory-mapped and can be accessed by external agents through a read or write of the memory address mapped to that register or through a program executing MOVTS/MOVFS instructions. The eCore complete local memory space is accessible by any master within an Epiphany system by adding 12-bit processor node ID offset to the local address locations. Reading directly from the general-purpose registers by an external agent is not supported while the CPU is active. Unmapped bits and reserved bits within defined memory-mapped registers should be written with zeros if not otherwise specified.

3.3.4 General Purpose Register

The CPU has a general-purpose register file containing 64 registers shown in Table 3.1. General-Purpose registers have no restrictions on usage and can be used by all instructions in the Epiphany instruction-set architecture. The only general purpose register written implicitly by an instruction is register R14, used to save a PC on a functional call. The register convention shown in Table 3.1 shows the register usage assumed by the compiler to ensure safe design and interoperability between different libraries written in C and or assembly. The registers do not have default values.

Name	Synonym	Role in the Procedure Call Standard	Saved By
R0	A1	Argument/result/scratch register #1	Caller saved
R1	A2	Argument/result/scratch register #2	Caller saved
R2	A3	Argument/result/scratch register #3	Caller saved
R3	A4	Argument/result/scratch register #4	Caller saved
R4	V1	Register variable #1	Caller saved
R5	V2	Register variable #2	Caller saved
R6	V3	Register variable #3	Caller saved
R7	V4	Register variable #4	Caller saved
R8	V5	Register variable #5	Caller saved
R9	V6/SB	Register variable #6/Static base	Caller saved
R10	V7/SL	Register variable #7/Static base	Caller saved
R11	V8/FP	Variable Register #8/Frame Pointer	Caller saved
R12	-	Intra-procedure call scratch register	Caller saved
R13	SP	Stack Pointer	Caller saved
R14	LR	Link Register	Caller saved
R15	-	General Use	Caller saved
R16-R27	-	General use	Caller saved
R28-R31	-	Reserved for constants	N/A
R32-R43	-	General use	Caller saved
R44-R63	-	General use	Caller saved

TABLE 3.1: General-Purpose Registers[1]

The first four registers, R0-R3 (or A1-A4), are used to pass arguments into a subroutine and to return a result from a function. They can also be used to hold intermediate values within a function. The registers R4-R8, R10, R11 (or V1-V5, V7-V8) are used to hold the values of a routine's local variables. The following registers are set implicitly by certain instructions and architecture convention dictates that they have fixed use.

- **Stack Pointer:** Register R13 is a dedicated as a stack pointer (SP).
- **Link Register:** The link register, LR (or R14), is automatically written by the CPU when the BL or JALR instruction is used. The register is automatically read by the CPU when the RTS instruction is used. After the linked register has been saved onto the stack, the register can be used as a temporary storage register.

3.3.5 Epiphany Instruction Set

The Epiphany instruction-set architecture (ISA) is optimized for real-time signal processing application, but it has all the features needed to also perform well in standard control code. Instruction set highlights include:

- Orthogonal instruction set, with no restrictions on register usage.
- Instruction set optimized for floating point computation and efficient data movement.
- Post-modify load/store instructions for efficient handling of large array structures.
- Rich set of branch conditions, with 3-cycle branch penalty on all taken branches and zero penalty on untaken branches.
- Conditional move instructions to reduce branch penalty for simple control-code structures.
- Instructions with immediate modifies for high code density and low power consumption.
- Compact and efficient floating-point instruction set.

The ISA uses a split width instruction encoding method, which improves code density compared with standard 32-bit width encoding. All instructions are available as both 16 and 32-bit instructions, with the instruction width depending on the registers used in the operation. Any command that uses registers 0 through 7 only and does not have a large immediate constant is encoded as a 16-bit instruction. Commands that use higher numbered registers are encoded as 32-bit instructions. This encoding is transparent to the user, but is carefully integrated with the compiler to minimize C-based code size and power consumption.

Branch Instructions

Unrestricted branching is supported throughout the 32-bit memory map using branch instructions and register jump instructions. Branching can occur conditionally, based on the arithmetic flags set by the integer or floating-point execution unit. The following table illustrates the condition codes supported by the ISA. The architecture supports two sets of flags to allow independent conditional execution and branching of instructions based on results from two separate arithmetic units. The full set of branching conditions allows the synthesis of any high-level control comparison, including: $<, <=, =, ==, != , >=, and >$. Both signed and unsigned arithmetic is supported.

Load/Store Instructions

Load and store instructions move data between the general-purpose register file and any legal memory location within the architecture, including external memory and any other eCore CPU in the system. All other instructions, such as floating-point and integer arithmetic instructions, are restricted to using registers as source and destination operands.

- **Displacement Addressing:** The memory address is calculated by adding an immediate offset to a base register value. The immediate offset is limited to 3 bits for 16-bit load/store instructions or 11 bits for 32-bit load/store instructions. The base register value is not modified by the load/store operation. This mode is useful for accessing local variables.
- **Indexed Addresing:** The memory address is calculated by adding a register value offset to a base register value. The base register value is not modified by the load/store operation. This mode is useful in array addressing.
- **Post-Modify Auto-increment Addressing:** The memory address is taken directly from the base register value. After the memory operation has completed, a register offset is added to the base register value and written back to the base register. This mode is useful for processing large data arrays and for implementing an efficient stack-pop operation.

Byte, short, word, and double data types are supported by all load/store instruction formats. All loads and stores must be aligned with respect to the data size being used. Short (16-bit) data types must be aligned on 16-bit boundaries in memory, word (32-bit) data types must be aligned on 32-bit boundaries, and double (64-bit) data types must be aligned on 64-bit boundaries. Unaligned memory accesses returns unexpected data and generates a software exception. Double data type load/store instructions must specify an even register in the general-purpose register file. The corresponding odd register is written implicitly. Attempts to use odd registers with double data format is flagged as an error by the assembler.

Integer Instructions

General-purpose integer instructions, such as ADD, SUB, ORR, AND, are useful for control code and integer math. These instructions work with immediate as well as register-based operands. The instructions update the integer status bits of the STATUS register.

Floating-Point Instructions

An orthogonal set of IEEE754-compliant floating-point instructions for signal processing applications. These instructions update the floating-point status bits of the STATUS register.

Secondary Signed Integer Instructions

The basic floating point instruction set can be substituted with a set of signed integer instructions by setting the appropriate mode bits in the CONFIG register [19:16]. These instructions use the same opcodes as the floating-point instructions and include: IADD, ISUB, IMUL, IMADD, IMSUB.

Register Move Instructions

All register moves are done as complete word (32-bit) entities. Conditional move instructions support the same set of condition codes as the branch instructions

Program Flow Instructions

A number of special instructions used by the run time environment to enable efficient interrupt handling, multicore programming, and program debug.

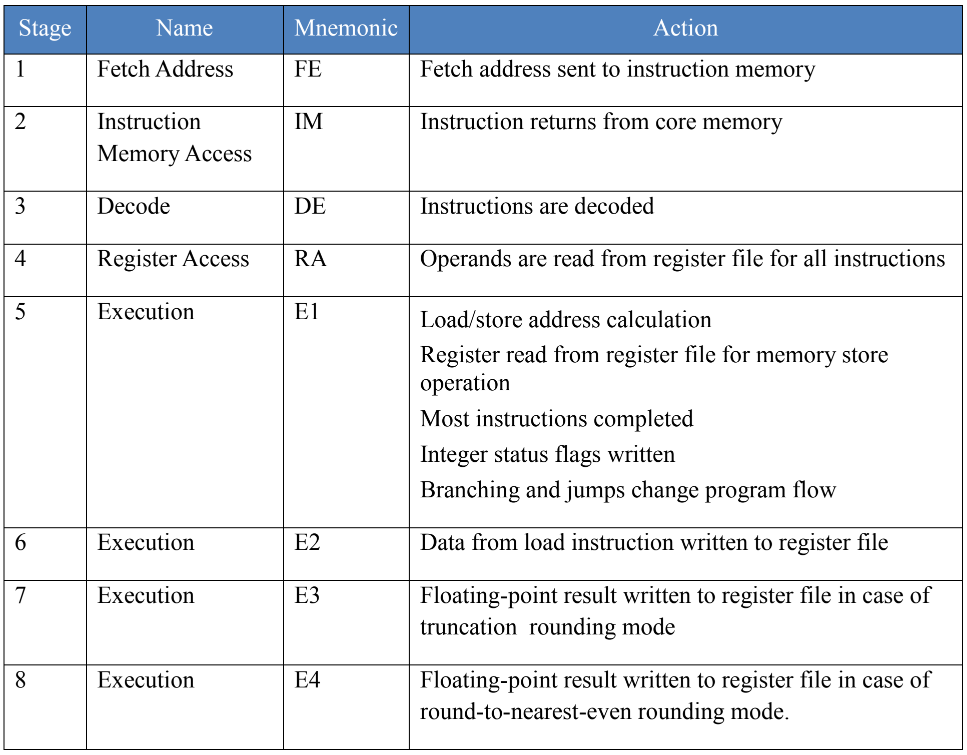
3.3.6 Pipeline Description

The Epiphany CPU has a variable length instruction pipeline that depends on the type of instruction being executed. All instructions share the same instruction pipeline until the E1 pipeline stage, and instructions are guaranteed to complete once they reach that stage. Load instructions complete at stage E2, and floating-point instructions complete at stage E4. All other instructions complete at E1. Instructions are dispatched in-order but can finish out-of-order. The pipeline controller makes sure that the integrity of the program is maintained by stalling the pipeline appropriately if there is a read-after-write (RAW) or write-after-write (WAW) pipeline hazard.

In the execution of instructions, the CPU implements an interlocked pipeline. When an instruction executes, the target register of the read operation is marked as busy until the write has been completed. If a subsequent instruction tries to access this register before the new value is present, the pipeline will stall until the previous instruction completes. This stall guarantees that instructions that require the use of data resulting from a previous instruction do not use the previous or invalid data in the register.

3.3.7 Interrupt Controller

The eCore interrupt controller provides full support for prioritized nested interrupt service routines. Figure 3.13 shows the behavior of the hardware mechanisms within the interrupt controller and how the user can control the behavior of the system through code.



Stage	Name	Mnemonic	Action
1	Fetch Address	FE	Fetch address sent to instruction memory
2	Instruction Memory Access	IM	Instruction returns from core memory
3	Decode	DE	Instructions are decoded
4	Register Access	RA	Operands are read from register file for all instructions
5	Execution	E1	Load/store address calculation Register read from register file for memory store operation Most instructions completed Integer status flags written Branching and jumps change program flow
6	Execution	E2	Data from load instruction written to register file
7	Execution	E3	Floating-point result written to register file in case of truncation rounding mode
8	Execution	E4	Floating-point result written to register file in case of round-to-nearest-even rounding mode.

FIGURE 3.11: Pipeline Stage Description[11]

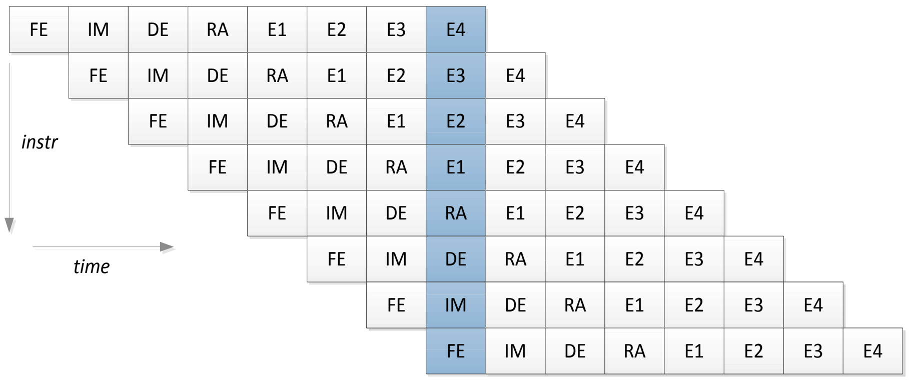


FIGURE 3.12: Pipeline Graphical View[11]

3.3.8 Hardware Loops (LABS)

The Epiphany core supports zero overhead loops with the LC (loop counter), LS (loop start address), and LE (loop end address) registers. The three hardware loop registers must be correctly programmed before executing the critical code section. When the program counter (PC) matches the value in LE and the LC is greater than zero, the PC gets set to the address in LS. The LC register decrements automatically every time the program scheduler completes one iteration of the code loop defined by LS and LE.

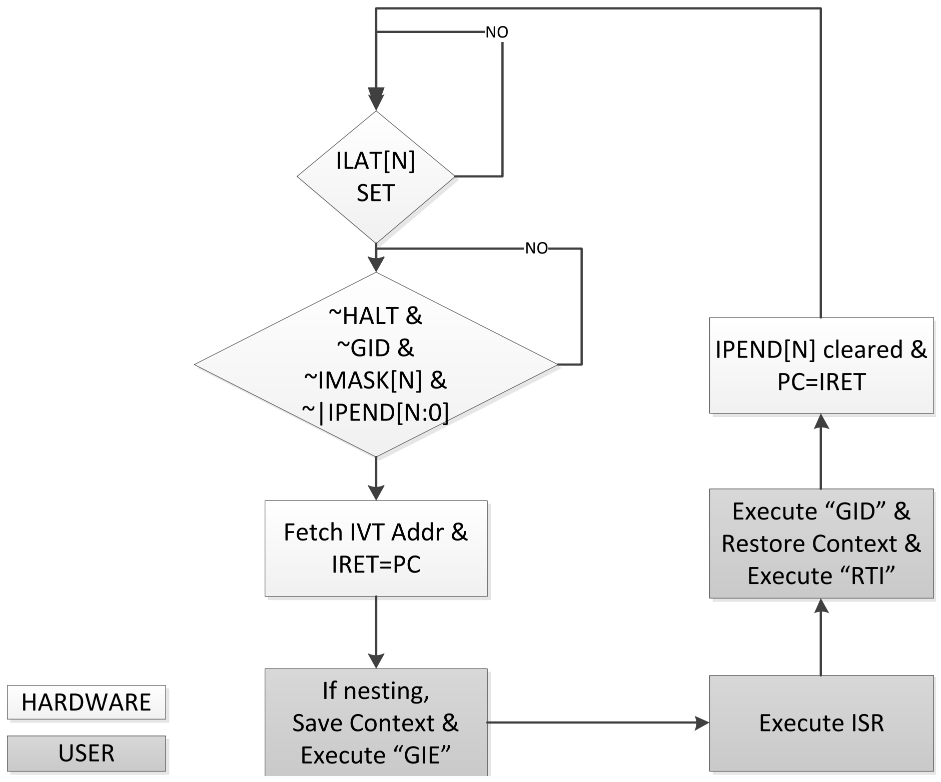

NO
ILAT[N] SET
NO
~HALT & ~GID & ~IMASK[N] & ~|IPEND[N:0]
Fetch IVT Addr & IRET=PC
If nesting, Save Context & Execute "GIE"
Execute ISR
Execute "GID" & Restore Context & Execute "RTI"
IPEND[N] cleared & PC=IRET
HARDWARE
USER

FIGURE 3.13: Interrupt Service Routine Operation[1]

The Epiphany hardware loop does place certain restrictions on the program:

- All interrupts must be disabled while inside a hardware loop.
- The start of the loop must be aligned on a double word boundary.
- The next-to-last instruction must be aligned on a double word boundary.
- All instructions in the loop set as 32 bit instructions using ".l" assembly suffix
- The minimum loop length is 8 instructions.

3.3.9 Direct Memory Access (DMA)

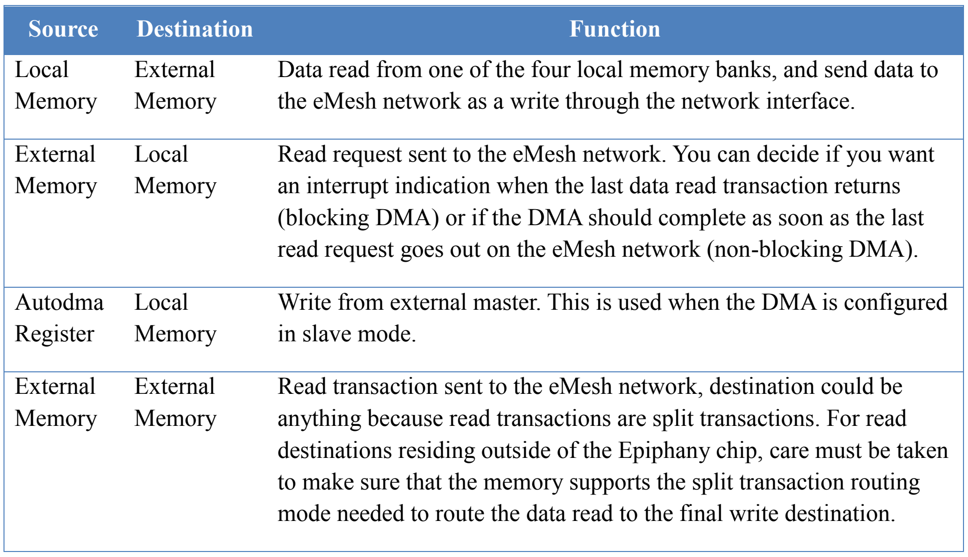
Each Epiphany processor node contains a DMA engine to facilitate data movement across the eMesh network. The DMA engine works at the same clock frequency as the CPU and can transfer one 64-bit double word per clock cycle, enabling a sustained data transfer rate of 8GB/sec. The DMA engine has two general-purpose channels, with separate configuration for source and destination.

The main features of the DMA engine are:

- Two independent DMA channels per processor node.

- Separate specification of source/destination address configuration per descriptor and channel.
- 2D DMA operation.
- Flexible stride sizes
- DMA descriptor chaining.
- Hardware interrupts flagging to local CPU subsystem.

The Figure 3.14 shows the kind of transfers supported by the processor node's DMA engine.



Source	Destination	Function
Local Memory	External Memory	Data read from one of the four local memory banks, and send data to the eMesh network as a write through the network interface.
External Memory	Local Memory	Read request sent to the eMesh network. You can decide if you want an interrupt indication when the last data read transaction returns (blocking DMA) or if the DMA should complete as soon as the last read request goes out on the eMesh network (non-blocking DMA).
Autodma Register	Local Memory	Write from external master. This is used when the DMA is configured in slave mode.
External Memory	External Memory	Read transaction sent to the eMesh network, destination could be anything because read transactions are split transactions. For read destinations residing outside of the Epiphany chip, care must be taken to make sure that the memory supports the split transaction routing mode needed to route the data read to the final write destination.

FIGURE 3.14: DMA Transfer Types[12]

The DMA engine has two complete data transfer channels and supports data movement as a master as well as a slave device. In a slave configuration, the pace of the data transfers is controlled by an external master. In a master configuration, the DMA pushes a transaction every clock cycle if the necessary memory and interface resources are available.

- In the MASTER mode, the DMA generates a complete transfer transaction with a source and a destination address.
- In the SLAVE mode, the source address of a DMA configuration is ignored. The data is always taken from the DMAxAUTO register and transferred to the destination address. The pace of the transaction is driven by another master in the system, which could be an I/O device, a programmable core, or another DMA channel.

3.3.10 Memory Protection Unit (LABS)

The Memory Protection Unit allows the user to specify parts or all of the local memory as read only memory. The 32KB local memory is split into 8 4KB page that can be independently set to read-only. If a write is attempted to a page that has been set to read only, and the memory fault exception bit in the ILAT register is set. The MEMPROTECT register can be used to help debug program faults related to stack overflow and multicore memory clobbering.

3.4 Software Development Enviroment

The Epiphany multicore architecture supports open-source ANSI C/C++ software development flows, using GNU GCC and GDB. The highly optimized GCC compiler enables acceptable real-time performance from pure ANSI-C/C++ applications without having to write assembly code for the vast majority of applications.

The Epiphany Software Development Kit (eSDK) is a state-of-the-art software development environment targeting the Epiphany multicore architecture. The eSDK is based on standard development tools including an optimized C-compiler, debugger, and multicore integrated development environment. The eSDK enables out-of-the-box execution of applications written in regular ANSI-C and does not require any C-subset, language extensions, or SIMD style programming. The Epiphany SDK includes:

- ANSI-C/C++ GCC compiler
- OpenCL SDK
- Multicore GDB debugger
- Runtime library

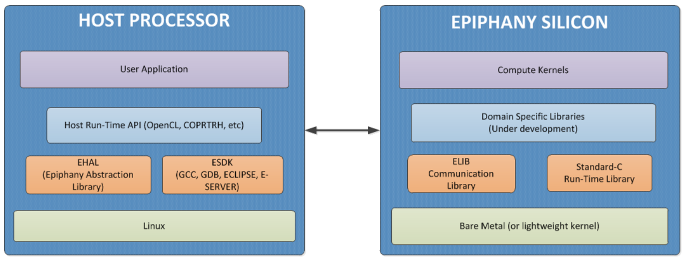

HOST PROCESSOR
User Application
Host Run-Time API (OpenCL, COPRTRH, etc)
EHAL (Epiphany Abstraction Library)
ESDK (GCC, GDB, ECLIPSE, E-SERVER)
Linux
EPIPHANY SILICON
Compute Kernels
Domain Specific Libraries (Under development)
ELIB Communication Library
Standard-C Run-Time Library
Bare Metal (or lightweight kernel)

Figure 3.15: Epiphany Software Development Stack[1]

Basically, to create a program that runs on the Epiphany cores, is required two scripts. One will contain the "Host" part of the program and the other will contain the "Device" part of the program. The ”Host” script runs on the ARM processor of the Parallella Board, and it must initialize the eCores to receive data. The ”Device” script runs on the eCores, and it must be compiled with an special chain-tool provided by Adapteva.

3.5 Programming Model

3.5.1 Programming Model Introduction

The Epiphany architecture is programming-model neutral and compatible with different parallel-programming methods, including Single Instruction Multiple Data (SIMD), Single Program Multiple Data (SPMD), Host-Slave programming, Multiple Instruction Multiple Data (MIMD), static and dynamic dataflow, systolic array, shared-memory multithreading, message- passing, and communicating sequential processes (CSP).

3.5.2 Parallel Programming Example

The following example shows how multiple Epiphany mesh nodes can be combined to improve the overall throughput of a computation. For simplicity, we have chosen matrix multiplication, but the concepts also apply to more complicated programs. Matrix multiplication can be represented by the following formula:

$$C_{ij} = \sum_{k=0}^{N-1} (A_{ik} B_{kj}) \tag{3.1}$$

Where A and B are the input matrices, C is the result, and i and j represent the row-column coordinate of the matrix elements. A naive (but correct) implementation of the matrix multiplication running on a single core is given below:

```
for (i = 0; i < M; i++){
        for(j = 0; j < N; j++){
                for(k = 0; k < K; k++){
                        C[i][j] += A[i][k] * B[k][j];
                }
        }
}
```

The code above can be written in standard C/C++ and compiled to run on a single core, with matrices A, B, and C placed in the core’s local memory. In this simple programming example, there is no difference between the Epiphany architecture and any other single threaded processor platform. To speed up this calculation using several mesh nodes simultaneously, we first need to distribute the A, B, C matrices over P tasks. Due to the matrix nature of the architecture, the natural way to distribute large matrices is by cutting them into smaller blocks, sometimes referred to as ”blocked by row and column”. We then construct a SPMD program that runs on each of the mesh nodes.

Figure 3.16 shows how the matrix multiplication can be divided into 16 sub-tasks and mapped onto 16 mesh nodes. Data sharing between the sub tasks can be done by passing data between the cores using a message passing API provided in the Epiphany SDK or by explicitly writing to global shared memory.

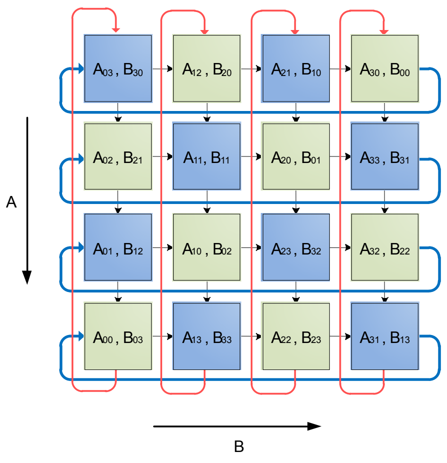


FIGURE 3.16: Matrix Multiplication Data Flow[11]

The parallel matrix multiplication completes in $\sqrt{P}$ steps, (where P is the number of processors) with each matrix multiplication task operating on data sets that are of size $\sqrt{P}$ x $\sqrt{P}$. At each step of the process, contributions to the local C matrix accumulate in each task, after which the local A matrix moves down and the local B matrix moves to the right. The entire example can be completed using standard ANSI programming constructs. Given the algorithm above, a 16-core Epiphany implementation operating at 1GHz can complete a 128x128 matrix multiply in 2ms while achieving 90% of the theoretical peak performance. The matrix multiplication algorithm in this example scales to thousands of cores and demonstrates how the Epiphany architecture's performance scales linearly with the number of cores in the system when proper data distribution and programming models are used.

Chapter 4

Experimentation

4.1 Set Up

To prepare the board and install the Operating System (OS), read the Appendix A. In this section will be explained how to write the algorithm and make it run on the Epiphany cores.
A file containing some global constants and defined structs will be required. All this information will be writed on a file called *"defs.h"*, and it will be included in both scripts, host and device. The file *"defs.h"* is shown in B.1 on the Appendix B. Is possible to change the parameters for the execution of the algorithm.
In the Appendix B is also shown the host program (B.3)
This file must include the Epiphany Hardware Abstraction library *(#include<e-hal.h>)*, that provides functionality for communicating with the Epiphany chip when the application runs on a host. The communication is performed using memory writes to and reads from shared buffers that the applications on both sides should define.
The host application will communicate with the Epiphany device by either accessing the eCore's private memory space (to write from the host the seed for the rand() function), or by using shared buffers in the device external memory (for read the data given by the eCores).

4.2 Structure of the model Network and Table of Cost

The topology of the network that will be considered for the development of the algorithm is shown in Figure 4.1, and the rules and costs of the links are defined by the table in Figure 4.2.

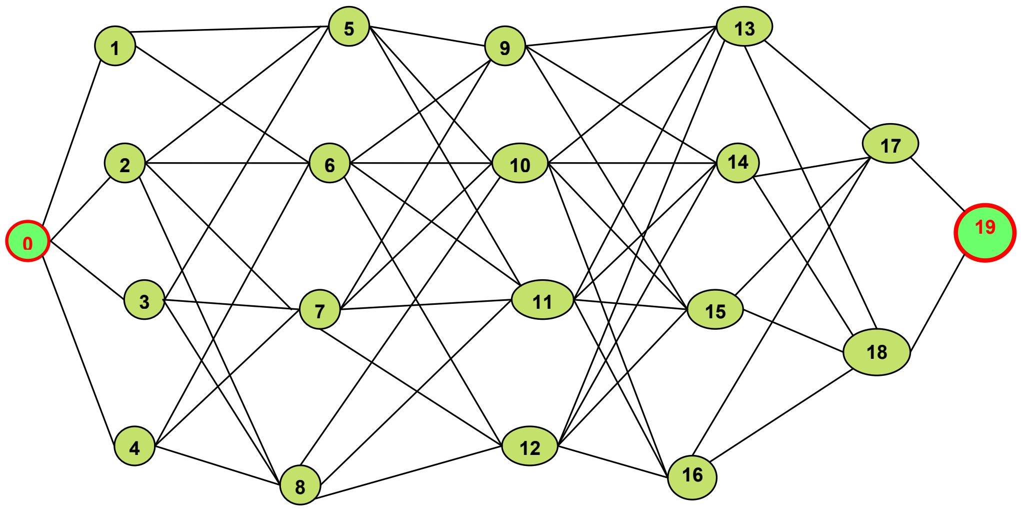
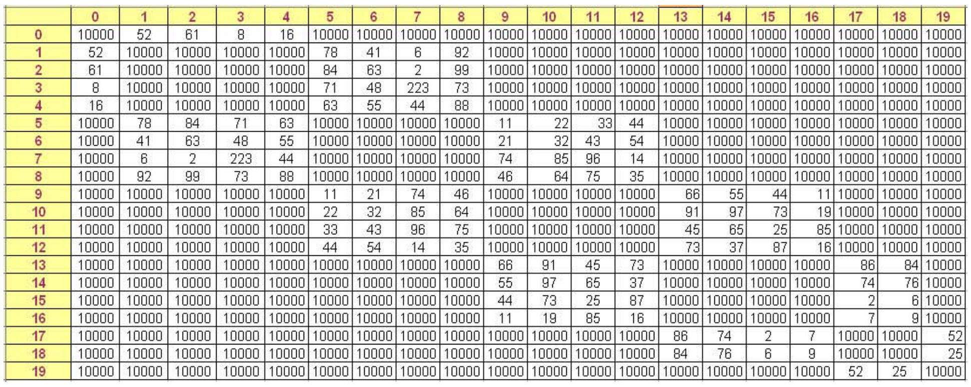


FIGURE 4.1: Network topology used



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
0	10000	52	61	8	16	10000	10000	10000	10000	10000	10000	10000	10000	10000	10000	10000	10000	10000	10000	10000
1	52	10000	10000	10000	10000	78	41	6	92	10000	10000	10000	10000	10000	10000	10000	10000	10000	10000	10000
2	61	10000	10000	10000	10000	84	63	2	99	10000	10000	10000	10000	10000	10000	10000	10000	10000	10000	10000
3	8	10000	10000	10000	10000	71	48	223	73	10000	10000	10000	10000	10000	10000	10000	10000	10000	10000	10000
4	16	10000	10000	10000	10000	63	55	44	88	10000	10000	10000	10000	10000	10000	10000	10000	10000	10000	10000
5	10000	78	84	71	63	10000	10000	10000	10000	11	22	33	44	10000	10000	10000	10000	10000	10000	10000
6	10000	41	63	48	55	10000	10000	10000	10000	21	32	43	54	10000	10000	10000	10000	10000	10000	10000
7	10000	6	2	223	44	10000	10000	10000	10000	74	85	96	14	10000	10000	10000	10000	10000	10000	10000
8	10000	92	99	73	88	10000	10000	10000	10000	46	64	75	35	10000	10000	10000	10000	10000	10000	10000
9	10000	10000	10000	10000	10000	11	21	74	46	10000	10000	10000	10000	66	55	44	11	10000	10000	10000
10	10000	10000	10000	10000	10000	22	32	85	64	10000	10000	10000	10000	91	97	73	19	10000	10000	10000
11	10000	10000	10000	10000	10000	33	43	96	75	10000	10000	10000	10000	45	65	25	85	10000	10000	10000
12	10000	10000	10000	10000	10000	44	54	14	35	10000	10000	10000	10000	73	37	87	16	10000	10000	10000
13	10000	10000	10000	10000	10000	10000	10000	10000	10000	66	91	45	73	10000	10000	10000	10000	86	84	10000
14	10000	10000	10000	10000	10000	10000	10000	10000	10000	55	97	65	37	10000	10000	10000	10000	74	76	10000
15	10000	10000	10000	10000	10000	10000	10000	10000	10000	44	73	25	87	10000	10000	10000	10000	2	6	10000
16	10000	10000	10000	10000	10000	10000	10000	10000	10000	11	19	85	16	10000	10000	10000	10000	7	9	10000
17	10000	10000	10000	10000	10000	10000	10000	10000	10000	10000	10000	10000	10000	86	74	2	7	10000	10000	52
18	10000	10000	10000	10000	10000	10000	10000	10000	10000	10000	10000	10000	10000	84	76	6	9	10000	10000	25
19	10000	10000	10000	10000	10000	10000	10000	10000	10000	10000	10000	10000	10000	10000	10000	10000	10000	52	25	10000

FIGURE 4.2: Cost of the links

With the value of 10000 is indicated a non permitted link between two nodes.

4.3 Proposed Algorithm

The steps of the algorithm are exposed in this section.The whole code of the algorithm implemented is shown on the Appendix B, in the section B.3. The functions that compose the algorithm are:

- *int main(void)*
- *void init_org(void)*
- *eval_gen(void)*
- *prod_next_gen(int organism)*
- *int select_one(void)*
- *Auxiliary functions*

4.3.1 Global Variables

There are variables that are common for all the functions. They are declared before the main function. These variables are:

- **curr_gen[N_org][N_gen]:** This matrix will store the current generation of organism (paths). The first index indicates the number of organism, and the second index indicates the number of gene inside the organism.
- **next_gen[N_org][N_gene]:** This matrix will store the offsprings of the current generation.
- **org_cost[N_org]:**This array will store the cost of the organism that compose the current generation.
- **sort_cost[N_org]:** Array that stores the same cost of the precedent variable, but sorted in ascending order.
- **tot_cost:** Variable that stores the sum of all the path cost of the organisms included in the current generation.
- **min_cost:** Stores the minimum cost value found in the current generation.
- **clon:** Stores the organism index of the path with the lower cost.

4.3.2 main() function

The function reads from the private memory of the eCore the seed to initialize the rand() function, otherwise the entire process will be almost deterministic, because the function rand() gives values according the seed that receives as input. The only way to obtain a random number that can be used as seed is including the <*time.h*> library. Unfortunately, the Epipahny SDK (Software Development Kit) doesn't permit the use of this library, so it have to be given by the host, writing inside the private memory of each eCore.
The main function contains also the table of the cost for the links, recalls the function that creates the first generation of organisms, recalls the function that makes the evaluation of the generation, manages the flags and writes the results on the external memory buffer.

4.3.3 init_org() function

It's a function of type void, that created the first generation of organisms and writes it on the **curr_gen** matrix. A chromosome corresponds to a possible solution of the

optimization problem. Thus, each chromosome represents a path which consists of a set of nodes to complete the feasible solution, as the sequence of nodes with the source node followed by intermediate nodes (via nodes), and the last node indicating the destination, which is the goal.
The default maximum number of chromosome length is equal to the number of nodes. The first gene represents the source, and it's written directly because is specified by the "defs.h" file. The function selects one the neighbors provided that it has not been picked before. It keeps doing this operation until reaches the destination node which is, like the source, specified in the mentioned file.

4.3.4 eval_gen() function

The evaluation stage has two purposes. Primarily, I have to determine the fitness of all the organisms so that later on, in the **prod_next_gen()** function, I'll know which were the better organisms and therefore which should reproduce more often. The function applies the rules to weight the path's cost of each organism, from the source to the destination node. After this cost's calculation, writes it on the **org_cost** vector, and saves the position (that will identify the fittest organism in the current generation) on the **clon** variable.

4.3.5 prod_next_gen() function

Once all the current generation was evaluated, it's possible to select the best organism to reproduce them. The function implements the Crossover and Mutation process.

Crossover
The function takes to organism recalling the **select_one() function**. Then, analyzes both of the organism to found the common points in the parents. The common nodes are where these two parents intersect. Among the common points, the function selects one of them randomly. The genes to the right of the crossover point are copied over from parent one to parent two, while the genes right of the crossover point from parent two are copied to the parent one. The new organism produced are the offsprings of the parents. The crossover operation is illustrated in Figure 4.3

Mutation
With some probability, the function performs the mutation of one gene of the offspring organism. The gene mutated and the one chosen to replace it are randomly selected.

After these two operations, the function calculates the fitness of the two organism newly created, and if they fitness are less than the nodes with maximum fitnesses in the

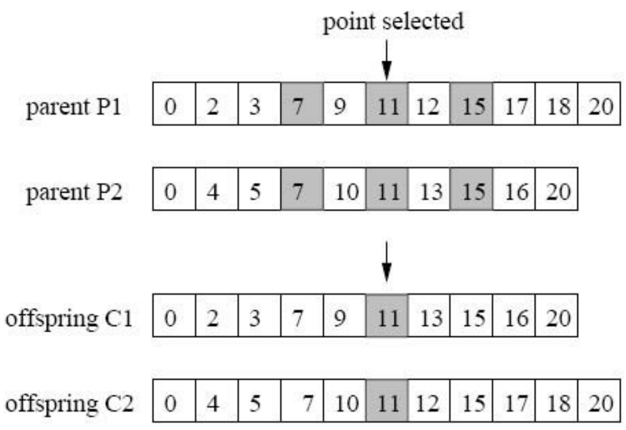


FIGURE 4.3: Crossover operator

population, replaces them with the nodes with the maximum fitnesses. All the organism will be temporarily stored in the **next_gen** matrix. After all the organism have been created, the function copies them into the **curr_gen** matrix.

4.3.6 select_one() function

How to select the organism to reproduce will determine how effective is the algorithm. The method used in this experiment is the Roulette Wheel Sampling, which is illustrated in Figure 4.4. Metaphorically, each organism is ”assigned” a slice of the roulette wheel. The size of the slice each organism gets is proportional to its fitness. Then, the wheel is spun and and whichever slice it lands on, that organism gets selected.

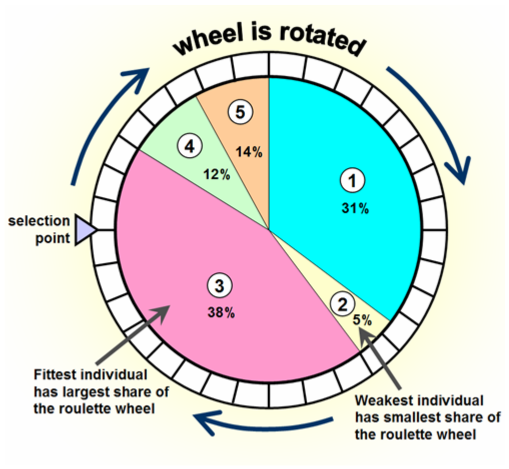

wheel is rotated
5
14%
4
12%
1
31%
selection point
3
38%
2
5%
Fittest individual has largest share of the roulette wheel
Weakest individual has smallest share of the roulette wheel

FIGURE 4.4: Roulette wheel sampling

4.4 Results

This experiment have been performed changing some settings (number of iterations, organism per generation). The tables below show the results obtained. For all the experiments, I have choose to make 10 iterations.

Cost	Time (s)	Path								
122	0.195	0	1	7	12	16	18	19		
166	0.197	0	3	8	12	16	18	19		
233	0.201	0	4	6	11	16	18	19		
195	0.202	0	2	7	12	15	18	19		
276	0.226	0	4	7	10	5	9	15	17	19
240	0.200	0	2	7	12	14	17	19		
168	0.193	0	4	5	11	15	18	19		
223	0.199	0	4	7	10	16	17	19		
223	0.201	0	4	6	9	8	12	16	18	19
255	0.219	0	1	6	11	16	18	19		

Table 4.1: 2 organism per generation

Cost	Time (s)	Path						
188	0.328	0	3	5	9	15	17	19
181	0.301	0	4	6	10	16	17	19
124	0.287	0	4	7	12	16	18	19
168	0.299	0	3	5	11	15	18	19
190	0.299	0	2	6	9	16	18	19
179	0.283	0	4	7	9	16	18	19
168	0.282	0	3	5	11	15	18	19
159	0.297	0	1	6	9	16	18	19
135	0.283	0	3	5	9	16	18	19
152	0.268	0	3	6	9	15	18	19

Table 4.2: 5 organism per generation

Cost	Time (s)	Path								
122	0.448	0	1	7	12	16	18	19		
159	0.442	0	4	7	12	16	18	15	17	19
135	0.433	0	4	5	9	16	18	19		
160	0.416	0	3	5	9	16	17	19		
155	0.470	0	3	6	11	15	18	19		
122	0.412	0	3	6	9	16	18	19		
191	0.421	0	3	5	11	15	17	19		
178	0.421	0	3	6	11	15	17	19		
167	0.385	0	4	6	9	15	18	19		
122	0.469	0	3	6	9	16	18	19		

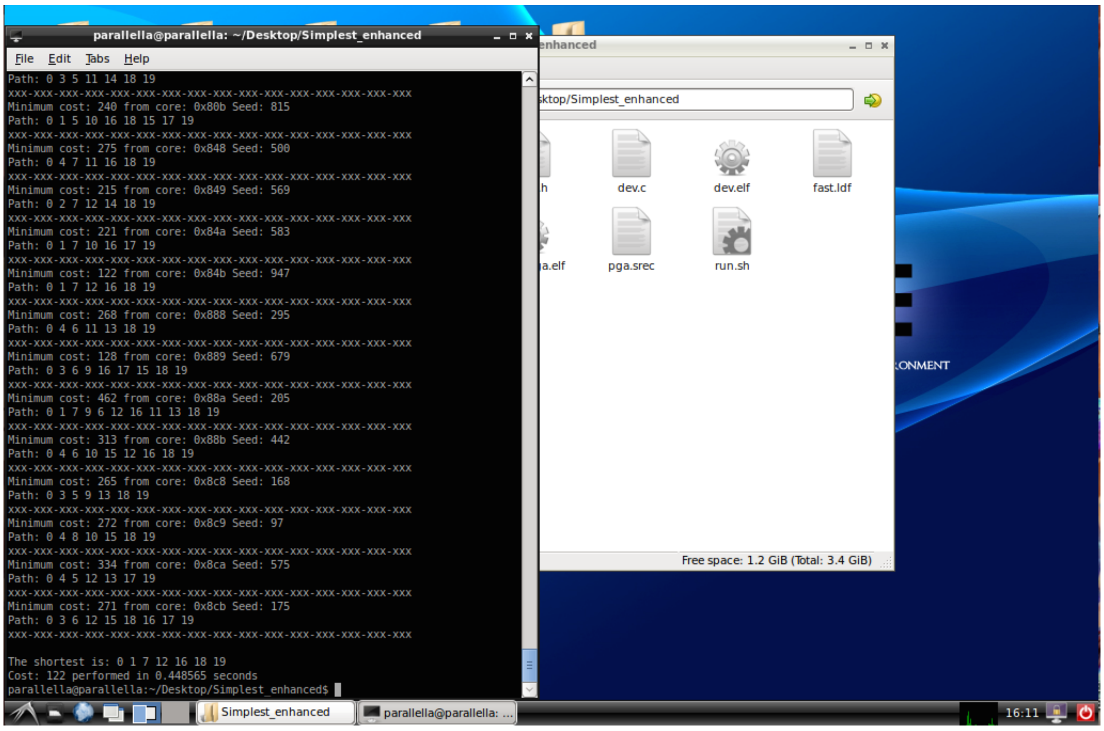
Table 4.3: 10 organism per generation

As shown in the tables, an increase of the of the number of organism per generation make the process slower, but the costs reduce significantly.

The confront was made with a Dijkstra's algorithm performed on the ARM processor. The result is shown in the next table:

Cost	Time (s)	Path						
122	0.281	0	1	7	12	16	18	19

TABLE 4.4: Dijkstra's algorithm result



```
Path: 0 3 5 11 14 18 19
xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx
Minimum cost: 240 from core: 0x80b Seed: 815
Path: 0 1 5 10 16 18 15 17 19
xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx
Minimum cost: 275 from core: 0x848 Seed: 500
Path: 0 4 7 11 16 18 19
xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx
Minimum cost: 215 from core: 0x849 Seed: 569
Path: 0 2 7 12 14 18 19
xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx
Minimum cost: 221 from core: 0x84a Seed: 583
Path: 0 1 7 10 16 17 19
xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx
Minimum cost: 122 from core: 0x84b Seed: 947
Path: 0 1 7 12 16 18 19
xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx
Minimum cost: 268 from core: 0x888 Seed: 295
Path: 0 4 6 11 13 18 19
xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx
Minimum cost: 128 from core: 0x889 Seed: 679
Path: 0 3 6 9 16 17 15 18 19
xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx
Minimum cost: 462 from core: 0x88a Seed: 205
Path: 0 1 7 9 6 12 16 11 13 18 19
xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx
Minimum cost: 313 from core: 0x88b Seed: 442
Path: 0 4 6 10 15 12 16 18 19
xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx
Minimum cost: 265 from core: 0x8c8 Seed: 168
Path: 0 3 5 9 13 18 19
xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx
Minimum cost: 272 from core: 0x8c9 Seed: 97
Path: 0 4 8 10 15 18 19
xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx
Minimum cost: 334 from core: 0x8ca Seed: 575
Path: 0 4 5 12 13 17 19
xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx
Minimum cost: 271 from core: 0x8cb Seed: 175
Path: 0 3 6 12 15 18 16 17 19
xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx

The shortest is: 0 1 7 12 16 18 19
Cost: 122 performed in 0.448565 seconds
parallella@parallella:~/Desktop/Simplest_enhanced$
```

FIGURE 4.5: A screenshot of the program results

Chapter 5

Conclusions

A Genetic Algorithm is developed to find the shortest path routing in a network. It is a flexible algorithm and is possible to change some parameters. The developed algorithm runs on the new multicore Epiphany structure. The length of each organism (chromosome) depends on the number of nodes in the network. The algorithm is simulated to solve a network of 20 nodes, using the firs one (0) as source and the last node (19) as destination. The obtained results affirmed the potential of the proposed algorithm that gave similar results as Dijkstra's algorithm, and the possibility to use the Epiphany structure as a cost-efficient component inside SDN controllers. Knowing that the time of performing of both algorithms increases with the increment of the number of nodes, future works can implement the genetic algorithm using a network with a large number of nodes and compare it with the Dijkstra's algorithm, since the time of performing of the last one increases faster than the one of the genetic algorithm.
There is another consideration that must be observed: the centralized model of controlling performed by the SDN paradigma. Since the genetic algorithm generates possible paths to reach a destination node given a source node, while the Dijkstra's algorithm must wait for the responses of all the neighbors nodes to calculate the best path; even if the genetic algorithm doesn't reach the shortest path, it generates a path that can reach the destination. This can be an advantage in terms of time to solution, that can be exploited using this new type of networking.

Appendix A

Parallella Board Configuration

A.1 Hardware Accessories

The accessories needed depend on the type of OS installed on the Parallella Board. There're two types of OS: Headless and With Display.

A.1.1 Headless

The Headless mode requires only three components:

- A 2000mA rated 5V DC power supply with 5.5mm OD / 2.1mm ID center positive polarity plug.
- A micro-SD card (minimun 4 GB).
- An Ethernet cable.

A.1.2 With Display

The Display mode will use a display with HDMI connection, and a keyboard connected directly to the board to input the commands. The advantage of this mode is that a computer is not necessary to access to the board. There're several accessories that will be required:

- A 2000mA rated 5V DC power supply with 5.5mm OD / 2.1mm ID center positive polarity plug.

- A micro-SD card (minimun 4 GB).
- A micro HDMI to HDMI cable.
- A USB male Micro-B to female Standard-A cable.
- A display with a HDMI port.
- A keyboard.

A.2 Creating a bootable micro-SD card

To create a bootable image of the OS Linaro for the Parallella board, a computer is required. The instructions written below are indicated for a PC running a Linux OS.

A.2.1 Downloading the Binaries

The Binaries can be downloaded from the site:
http://www.parallella.org/create-sdcard/
Choose the distribution (Headless or With Display) that fit the ARM type of the Zynq core of the board.

A.2.2 Install

1. Insert the micro-SD card in the computer.
2. Open a bash window.
3. Unzip the Ubuntu image

   ```
   $ gunzip -d <release_name>.img.gz
   ```

4. Verify the device path of the SD card

   ```
   $ sudo fdisk -l | grep Disk
   ```

5. Unmount the SD card

   ```
   $ umount <SD_device_path>
   ```

6. Burn the Ubuntu disk image on the micro-SD card

```
$ sudo dd bs=4M if=<release_name>.img of=<SD_device_path>
```

This procedure will take 10 minutes approx. At the end of the process:

```
$ sync
```

7. Check the files *uImage*, *devicetree.dtb* and *parallella.bit.bin* inside the partition /boot

A.3 Connect the board to the computer via Ethernet

These instructions are used to connect the Parallella board (with a headless Ubuntu image) to a computer running a Linux OS. The last distributions released by Adapteva have as default an IP address assigned via DHCP. There is also a way to set an static IP, but the advantage between using a dynamic and a static IP is that the first one allows to share the internet connection from the computer to the board.

A.3.1 Setting a static IP address

1. Mount the SD card in the computer.
2. Edit the file *eth0*, located at etc/network/interfaces.d

```
auto eth0
iface eth0 inet static
address 192.168.xxx.yyy
netmask 255.255.255.000
gateway 192.168.xxx.zzz
```

xxx, yyy and zzz are the subnet, the Parallella's address and the gateway address respectively, arbitrary chosen.

A.3.2 Connecting the board to the computer

1. Connect the board to the computer with a regular Ethernet cable.
2. Open a bash window and connect via ssh

```
ssh -X parallella@parallella.local
```

If is set a static IP

```
ssh -X parallella@192.168.xxx.yyy
```

Note: the command includes -X for a graphic display of the applications. Is required a previously installation of X11 on the computer.

Appendix B

C Codes

B.1 ”defs.h” file

```
#define N_org           5       //numer of organism for each generation
#define N_gen           20      //number of nodes
#define Mut_rate        0.01    //probability of mutation

#define iter            30      //number of iterations

#define source          0       //source node
#define destination     19      //destination node

#define max_cost        1000    //when there is no link between a node to
     another

#define SEED            0x7000  //physical address where will be stored
    the seed given by the host

//struct created to transport the info from the eCore to the host
typedef struct{

        int     flag;
        int     core_id;
        int     seed;
        int     mini;
        int     array[N_gen];
}Mailbox;
```

B.2 ”host.c” file

```
/*
This is the HOST side of the code. Reads static info from "defs.h" header
    .
Prepares the epiphany architecture to load the program descripted on "dev
    .c"
Prints the data and close the Epiphany device.
*/
#include <stdlib.h>
#include <stddef.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <time.h>

#include <e-hal.h>

//in the file defs.h are defined constants and structs that will be used
    by Host and Device
#include "defs.h"

#define _Buffsize       (2048)
#define _BuffOffset     (0x01000000)

int main(int argc, char *argv[]){

        unsigned row_loop, col_loop;
        int i;
        int temp[N_gen];

        //epiphany variables
        e_platform_t epiphany;
        e_epiphany_t dev;
        e_mem_t memory;

        //variables for the communication with the epiphany
        Mailbox mail;

        e_return_stat_t result;

        //initialization of the epiphany cores
        e_init(NULL);
        e_reset_system();
        e_get_platform_info(&epiphany);

        //define the buffer where will be stored the information
    processed by the eCores
```

```
        e_alloc(&memory, _BuffOffset, _Buffsize);

        //seed the rand() function
        srand(time(NULL));
        int num = 0;

        int c, d;
        int org_reg[16][N_gen] = {{0}};
        int org_cost[16];

        c = 0;
        for(row_loop = 0; row_loop < 4; row_loop++){
                for(col_loop = 0; col_loop < 4; col_loop++){

                        e_open(&dev, row_loop, col_loop, 1, 1);
                        //e_reset_group(&dev);

                        num = rand()%1000;
                        //hardwrite  on the eCore a random seed
                        e_write(&dev, 0, 0, SEED, &num, sizeof(num));

                        //load on the eCore the program that will be
executed
                        result = e_load("pga.srec", &dev, 0, 0, E_TRUE);
                        if(result != E_OK){
                        fprintf(stderr,"Error Loading the Epiphany
Application %i\n", result);
                        }

                        unsigned int addr = offsetof(Mailbox, flag);
                        //put the flag to 0
                        mail.flag = 0;
                        //read on the eCore until the flag is raised from
 0 to 1
                        while(mail.flag != 1){
                        e_read(&memory, 0, 0, addr, &mail.flag, sizeof(
mail.flag));
                        }
                        //read the information written by the eCore on
the external buffer
                        e_read(&memory, 0, 0, 0x0, &mail, sizeof(mail));

                        //copy the path given by the eCore
                        for(i = 0; i < N_gen; i++){
                                temp[i] = mail.array[i];
                        }
```

```
                                fprintf(stderr, "xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-
    xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx \n");
                                fprintf(stderr, "Minimum cost: %i from core: 0x
    %03x Seed: %i \n", mail.mini, mail.core_id, mail.seed);
                                fprintf(stderr, "Path: ");
                                for(i = 0; i < N_gen; i++){
                                        if(temp[i] != N_gen){
                                                fprintf(stderr, "%i ", temp[i]);
                                        }else{break;}
                                }
                                fprintf(stderr, "\n");

                                //store the cost of the paths
                                org_cost[c] = mail.mini;
                                //store the paths
                                for(d = 0; d < N_gen; d++){
                                        org_reg[c][d] = temp[d];
                                }
                                c++;
                        }


        }
        fprintf(stderr, "xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-xxx-
    xxx-xxx-xxx-xxx \n \n");

        //search for the organism that contain the shortest path between
    all the organism given by the eCores
        int clon;
        int min_cost = org_cost[0];
        for(c = 1; c < 16; c++){
                if(org_cost[c] < min_cost){
                        min_cost = org_cost[c];
                        clon = c;
                }
        }

        //print the shortest path
        fprintf(stderr, "The shortest is: ");
        for(d = 0; d < N_gen; d++){
                if(org_reg[clon][d] != N_gen){
                        fprintf(stderr, "%i ", org_reg[clon][d]);
                }else{break;}
        }
        fprintf(stderr, "\n");

        //print the cost path
        fprintf(stderr, "Cost: %i \n", min_cost);

        //interrupt the communication with the eCores
        e_close(&dev);
        e_free(&memory);
        e_finalize();

        return 0;
}
```

B.3 ”dev.c” file

```c
#include <stdio.h>
#include <stdlib.h>

#include "e_lib.h"
#include "defs.h"

//Global variables
int curr_gen[N_org][N_gen];
int next_gen[N_org][N_gen] ;
int org_cost[N_org];
int sort_cost[N_org];
int tot_cost;
int min_cost;
int clon;

int table[N_gen][N_gen];

//Functions Declarations
void init_org(void);
void eval_gen(void);
void eval_org(int organism);
int bound_rand(int max, int min);
int num_rand(int num, int min, int max);
int rand_rand(int min1, int max1, int min2, int max2);
int select_one(void);
int select_best(void);
void prod_next_gen(void);

Mailbox mail SECTION("shared_dram");

//MAIN
int main(void){

        mail.flag = 0;

        int i, j;

        //get the e_core info
        e_coreid_t id;
        id = e_get_coreid();

        //pointer to the data writed by the ARM
        int *p_seed =   (void*)SEED;
        //read the random number from the ARM and seed the srand
        unsigned seed = *p_seed;
        srand(seed);
        //wipe the curr_gen
        for(i = 0; i < N_org; i++){
                for(j = 0; j < N_gen; j++){
                        curr_gen[i][j] = N_gen;
                }
        }

        //set the table of cost

        table[0][1] = 52; table[0][2] = 61; table[0][3] = 8; table[0][4]
    = 16;

        table[1][0] = 52; table[1][5] = 78; table[1][6] = 41; table[1][7]
     = 6; table[1][8] = 92;

        table[2][0] = 61; table[2][5] = 84; table[2][6] = 63; table[2][7]
     = 2; table[2][8] = 99;

        table[3][0] = 8; table[3][5] = 71; table[3][6] = 48; table[3][7]
    = 223; table[3][8] = 73;

        table[4][0] = 16; table[4][5] = 63; table[4][6] = 55; table[4][7]
     = 44; table[4][8] = 88;

        table[5][1] = 78; table[5][2] = 84; table[5][3] = 71; table[5][4]
     = 63; table[5][9] = 11; table[5][10] = 22; table[5][11] = 33; table
    [5][12] = 44;

        table[6][1] = 41; table[6][2] = 63; table[6][3] = 48; table[6][4]
     = 55; table[6][9] = 21; table[6][10] = 32; table[6][11] = 43; table
    [6][12] = 54;

        table[7][1] = 6; table[7][2] = 2; table[7][3] = 223; table[7][4]
    = 44; table[7][9] = 74; table[7][10] = 85; table[7][11] = 96; table
    [7][12] = 14;

        table[8][1] = 92; table[8][2] = 99; table[8][3] = 73; table[8][4]
     = 88; table[8][9] = 46; table[8][10] = 64; table[8][11] = 75; table
    [8][12] = 35;

        table[9][5] = 11; table[9][6] = 21; table[9][7] = 74; table[9][8]
     = 46; table[9][13] = 66; table[9][14] = 55; table[9][15] = 44; table
    [9][16] = 11;

        table[10][5] = 22; table[10][6] = 32; table[10][7] = 85; table
    [10][8] = 64; table[10][13] = 91; table[10][14] = 97; table[10][15] =
    73; table[10][16] = 19;
        table[11][5] = 33; table[11][6] = 43; table[11][7] = 96; table
    [11][8] = 75; table[11][13] = 45; table[11][14] = 65; table[11][15] =
    25; table[11][16] = 85;

        table[12][5] = 44; table[12][6] = 54; table[12][7] = 14; table
    [12][8] = 35; table[12][13] = 73; table[12][14] = 37; table[12][15] =
    87; table[12][16] = 16;

        table[13][9] = 66; table[13][10] = 91; table[13][11] = 45; table
    [13][12] = 73; table[13][17] = 86; table[13][18] = 84;

        table[14][9] = 55; table[14][10] = 97; table[14][11] = 65; table
    [14][12] = 37; table[14][17] = 74; table[14][18] = 76;

        table[15][9] = 44; table[15][10] = 73; table[15][11] = 25; table
    [15][12] = 87; table[15][17] = 2; table[15][18] = 6;

        table[16][9] = 11; table[16][10] = 19; table[16][11] = 85; table
    [16][12] = 16; table[16][17] = 7; table[16][18] = 9;

        table[17][13] = 86; table[17][14] = 74; table[17][15] = 2; table
    [17][16] = 7; table[17][19] = 52;

        table[18][13] = 84; table[18][14] = 76; table[18][15] = 6; table
    [18][16] = 9; table[18][19] = 25;

        table[19][17] = 52; table[19][18] = 25;

        //initialize and do the first evaluation of the generation
        init_org();
        eval_gen();

        //iterations
        for(i = 0; i < iter; i++){
                prod_next_gen();
                eval_gen();
        }

        mail.core_id = id;
        mail.seed = seed;
        mail.mini = min_cost;

        for(i = 0; i < N_gen; i++){
                mail.array[i] = curr_gen[clon][i];
        }

        mail.flag = 1;
```

```
        return EXIT_SUCCESS;
}

/*

*/

void init_org(void){
        int organism;
        int gene;
        int i;
        int temp;
        int v[N_gen];
        int flag;

        // initialize the organism
        for(organism = 0; organism < N_org; ++organism){
                //initialize the register of used nodes
                for(i = 0; i < N_gen; i++){
                        v[i] = 0;
                }

                //hard write the source in the first bin of each organism
                curr_gen[organism][0] = source;
                v[source] = 1;

                //intialises the organism and stop if the destination is
    reached
                for(gene = 0; gene < N_gen && curr_gen[organism][gene] !=
     destination; ++gene){

                        if(curr_gen[organism][gene] == 0){
                                //set a flag to 0. Write a random
    neighbor and verify if is not previously
                                //included in the organism. If all the
    possibles neighbors are included in
                                //the organism, raise the flag to 1 and
    break the for cycle (error)
                                flag = 0;
                                do{
                                        temp = bound_rand(1,4);
                                        if((v[1]+v[2]+v[3]+v[4]) == 4){
    flag = 1; break;}
                                }while(v[temp] == 1);

                                if(flag == 1){break;}
```

```
                            curr_gen[organism][gene+1] = temp;
                            v[temp] = 1;
                    }

                    if(curr_gen[organism][gene] == 1){
                            flag = 0;
                            do{
                                    temp = num_rand(0,5,8);
                                    if((v[0]+v[5]+v[6]+v[7]+v[8]) ==
5){flag = 1; break;}
                            }while(v[temp] == 1);

                            if(flag == 1){break;}

                            curr_gen[organism][gene+1] = temp;
                            v[temp] = 1;
                    }

                    if(curr_gen[organism][gene] == 2){
                            flag = 0;
                            do{
                                    temp = num_rand(0,5,8);
                                    if((v[0]+v[5]+v[6]+v[7]+v[8]) ==
5){flag = 1; break;}
                            }while(v[temp] == 1);

                            if(flag == 1){break;}

                            curr_gen[organism][gene+1] = temp;
                            v[temp] = 1;
                    }

                    if(curr_gen[organism][gene] == 3){
                            flag = 0;
                            do{
                                    temp = num_rand(0,5,8);
                                    if((v[0]+v[5]+v[6]+v[7]+v[8]) ==
5){flag = 1; break;}
                            }while(v[temp] == 1);

                            if(flag == 1){break;}

                            curr_gen[organism][gene+1] = temp;
                            v[temp] = 1;
                    }

                    if(curr_gen[organism][gene] == 4){
                            flag = 0;
```

```
                                do{
                                        temp = num_rand(0,5,8);
                                        if((v[0]+v[5]+v[6]+v[7]+v[8]) ==
5){flag = 1; break;}
                                }while(v[temp] == 1);

                                if(flag == 1){break;}

                                curr_gen[organism][gene+1] = temp;
                                v[temp] = 1;
                        }

                        if(curr_gen[organism][gene] == 5){
                                flag = 0;
                                do{
                                        temp = rand_rand(1,4,9,12);
                                        if((v[1]+v[4]+v[9]+v[12]) == 4){
flag = 1; break;}
                                }while(v[temp] == 1);

                                if(flag == 1){break;}

                                curr_gen[organism][gene+1] = temp;
                                v[temp] = 1;
                        }

                        if(curr_gen[organism][gene] == 6){
                                flag = 0;
                                do{
                                        temp = rand_rand(1,4,9,12);
                                        if((v[1]+v[4]+v[9]+v[12]) == 4){
flag = 1; break;}
                                }while(v[temp] == 1);

                                if(flag == 1){break;}

                                curr_gen[organism][gene+1] = temp;
                                v[temp] = 1;
                        }

                        if(curr_gen[organism][gene] == 7){
                                flag = 0;
                                do{
                                        temp = rand_rand(1,4,9,12);
                                        if((v[1]+v[4]+v[9]+v[12]) == 4){
flag = 1; break;}
                                }while(v[temp] == 1);
```

```
                                        if(flag == 1){break;}

                                        curr_gen[organism][gene+1] = temp;
                                        v[temp] = 1;;
                                }

                                if(curr_gen[organism][gene] == 8){
                                        flag = 0;
                                        do{
                                                temp = rand_rand(1,4,9,12);
                                                if((v[1]+v[4]+v[9]+v[12]) == 4){
    flag = 1; break;}
                                        }while(v[temp] == 1);

                                        if(flag == 1){break;}

                                        curr_gen[organism][gene+1] = temp;
                                        v[temp] = 1;
                                }

                                if(curr_gen[organism][gene] == 9){
                                        flag = 0;
                                        do{
                                                temp = rand_rand(5,8,13,16);
                                                if((v[5]+v[8]+v[13]+v[16]) == 4){
    flag = 1; break;}
                                        }while(v[temp] == 1);

                                        if(flag == 1){break;}

                                        curr_gen[organism][gene+1] = temp;
                                        v[temp] = 1;
                                }

                                if(curr_gen[organism][gene] == 10){
                                        flag = 0;
                                        do{
                                                temp = rand_rand(5,8,13,16);
                                                if((v[5]+v[8]+v[13]+v[16]) == 4){
    flag = 1; break;}
                                        }while(v[temp] == 1);

                                        if(flag == 1){break;}

                                        curr_gen[organism][gene+1] = temp;
                                        v[temp] = 1;
                                }
```

```
                        if(curr_gen[organism][gene] == 11){
                                flag = 0;
                                do{
                                        temp = rand_rand(5,8,13,16);
                                        if((v[5]+v[8]+v[13]+v[16]) == 4){
    flag = 1; break;}
                                }while(v[temp] == 1);

                                if(flag == 1){break;}

                                curr_gen[organism][gene+1] = temp;
                                v[temp] = 1;
                        }

                        if(curr_gen[organism][gene] == 12){
                                flag = 0;
                                do{
                                        temp = rand_rand(5,8,13,16);
                                        if((v[5]+v[8]+v[13]+v[16]) == 4){
    flag = 1; break;}
                                }while(v[temp] == 1);

                                if(flag == 1){break;}

                                curr_gen[organism][gene+1] = temp;
                                v[temp] = 1;
                        }

                        if(curr_gen[organism][gene] == 13){
                                flag = 0;
                                do{
                                        temp = rand_rand(9,12,17,18);
                                        if((v[9]+v[12]+v[17]+v[18]) == 4)
    {flag = 1; break;}
                                }while(v[temp] == 1);

                                if(flag == 1){break;}

                                curr_gen[organism][gene+1] = temp;
                                v[temp] = 1;
                        }

                        if(curr_gen[organism][gene] == 14){
                                flag = 0;
                                do{
                                        temp = rand_rand(9,12,17,18);
                                        if((v[9]+v[12]+v[17]+v[18]) == 4)
    {flag = 1; break;}
```

```
                                }while(v[temp] == 1);

                                if(flag == 1){break;}

                                curr_gen[organism][gene+1] = temp;
                                v[temp] = 1;
                        }

                        if(curr_gen[organism][gene] == 15){
                                flag = 0;
                                do{
                                        temp = rand_rand(9,12,17,18);
                                        if((v[9]+v[12]+v[17]+v[18]) == 4)
    {flag = 1; break;}
                                }while(v[temp] == 1);

                                if(flag == 1){break;}

                                curr_gen[organism][gene+1] = temp;
                                v[temp] = 1;
                        }

                        if(curr_gen[organism][gene] == 16){
                                flag = 0;
                                do{
                                        temp = rand_rand(9,12,17,18);
                                        if((v[9]+v[12]+v[17]+v[18]) == 4)
    {flag = 1; break;}
                                }while(v[temp] == 1);

                                if(flag == 1){break;}

                                curr_gen[organism][gene+1] = temp;
                                v[temp] = 1;
                        }

                        if(curr_gen[organism][gene] == 17){
                                flag = 0;
                                do{
                                        temp = num_rand(19,13,16);
                                        if((v[19]+v[13]+v[16]) == 3){flag
     = 1; break;}
                                }while(v[temp] == 1);

                                if(flag == 1){break;}

                                curr_gen[organism][gene+1] = temp;
                                v[temp] = 1;
                        }

                        if(curr_gen[organism][gene] == 18){
                                flag = 0;
                                do{
                                        temp = num_rand(19,13,16);
                                        if((v[19]+v[13]+v[16]) == 3){flag
 = 1; break;}
                                }while(v[temp] == 1);

                                if(flag == 1){break;}

                                curr_gen[organism][gene+1] = temp;
                                v[temp] = 1;
                        }

                        if(curr_gen[organism][gene] == 19){
                                flag = 0;
                                do{
                                        temp = bound_rand(17,18);
                                        if((v[17]+v[18]) == 2){flag = 1;
break;}
                                }while(v[temp] == 1);

                                if(flag == 1){break;}

                                curr_gen[organism][gene+1] = temp;
                                v[temp] = 1;
                        }

                }

                //scan the organism, searching if is included the
destination. If included, raise a flag to 1
                flag = 0;
                for(gene = 0; gene < N_gen; gene++){
                        if(curr_gen[organism][gene] == destination){flag
= 1; break;}
                }
                //if the flag is not raised, initializes the current
organism to N_gen and decrease an organism index
                //in the case of error, this procedure will reinitialize
the organism.
                if(flag == 0){
                        for(gene = 0; gene < N_gen; gene++){
                                curr_gen[organism][gene] = N_gen;
                        }
                        --organism;
                }
        }
}

/*

*/

void eval_gen(void){
        int organism;
        int gene;
        int i, temp;
        int currentCost;

        tot_cost = 0;

        for(organism = 0; organism < N_org; ++organism){
                //initializes the cost of each organism
                currentCost = 0;

                //analize only the bins between <0,N_gen> edges excluded
                for(gene = 0; gene < N_gen; ++gene){

                        //verificates if the bin is not 20 or the
    destination node (end of the info), else break the analysis
                        if(curr_gen[organism][gene] != 20 && curr_gen[
    organism][gene] != destination){

                                //if the bin is equal to 0
                                if(curr_gen[organism][gene] == 0){
                                        if(curr_gen[organism][gene+1] ==
    1 || curr_gen[organism][gene+1] == 2 || curr_gen[organism][gene+1] ==
    3  || curr_gen[organism][gene+1] == 4){
                                                if(curr_gen[organism][
    gene+1] == 1){currentCost = currentCost + table[0][1];}
                                                if(curr_gen[organism][
    gene+1] == 2){currentCost = currentCost + table[0][2];}
                                                if(curr_gen[organism][
    gene+1] == 3){currentCost = currentCost + table[0][3];}
                                                if(curr_gen[organism][
    gene+1] == 4){currentCost = currentCost + table[0][4];}
                                        }
                                        else{
                                                currentCost = currentCost
     + max_cost;
                                        }
                                }
```

```
                                        //if the bin is equal to 1
                                        if(curr_gen[organism][gene] == 1){
                                                if(curr_gen[organism][gene+1] ==
0 || curr_gen[organism][gene+1] == 5 || curr_gen[organism][gene+1] ==
6 || curr_gen[organism][gene+1] == 7  || curr_gen[organism][gene+1] ==
 8){
                                                        if(curr_gen[organism][
gene+1] == 0){currentCost = currentCost + table[1][0];}
                                                        if(curr_gen[organism][
gene+1] == 5){currentCost = currentCost + table[1][5];}
                                                        if(curr_gen[organism][
gene+1] == 6){currentCost = currentCost + table[1][6];}
                                                        if(curr_gen[organism][
gene+1] == 7){currentCost = currentCost + table[1][7];}
                                                        if(curr_gen[organism][
gene+1] == 8){currentCost = currentCost + table[1][8];}
                                                }
                                                else{
                                                        currentCost = currentCost
 + max_cost;
                                                }
                                        }

                                        //if the bin is equal to 2
                                        if(curr_gen[organism][gene] == 2){
                                                if(curr_gen[organism][gene+1] ==
0 || curr_gen[organism][gene+1] == 5 || curr_gen[organism][gene+1] ==
6 || curr_gen[organism][gene+1] == 7  || curr_gen[organism][gene+1] ==
 8){
                                                        if(curr_gen[organism][
gene+1] == 0){currentCost = currentCost + table[2][0];}
                                                        if(curr_gen[organism][
gene+1] == 5){currentCost = currentCost + table[2][5];}
                                                        if(curr_gen[organism][
gene+1] == 6){currentCost = currentCost + table[2][6];}
                                                        if(curr_gen[organism][
gene+1] == 7){currentCost = currentCost + table[2][7];}
                                                        if(curr_gen[organism][
gene+1] == 8){currentCost = currentCost + table[2][8];}
                                                }
                                                else{
                                                        currentCost = currentCost
 + max_cost;
                                                }
                                        }

                                        //if the bin is equal to 3
                                        if(curr_gen[organism][gene] == 3){
```

```
                                        if(curr_gen[organism][gene+1] ==
    0 || curr_gen[organism][gene+1] == 5 || curr_gen[organism][gene+1] ==
    6 || curr_gen[organism][gene+1] == 7  || curr_gen[organism][gene+1] ==
     8){
                                                if(curr_gen[organism][
    gene+1] == 0){currentCost = currentCost + table[3][0];}
                                                if(curr_gen[organism][
    gene+1] == 5){currentCost = currentCost + table[3][5];}
                                                if(curr_gen[organism][
    gene+1] == 6){currentCost = currentCost + table[3][6];}
                                                if(curr_gen[organism][
    gene+1] == 7){currentCost = currentCost + table[3][7];}
                                                if(curr_gen[organism][
    gene+1] == 8){currentCost = currentCost + table[3][8];}
                                        }
                                        else{
                                                currentCost = currentCost
     + max_cost;
                                        }
                                }

                                //if the bin is equal to 4
                                if(curr_gen[organism][gene] == 4){
                                        if(curr_gen[organism][gene+1] ==
    0 || curr_gen[organism][gene+1] == 5 || curr_gen[organism][gene+1] ==
    6 || curr_gen[organism][gene+1] == 7  || curr_gen[organism][gene+1] ==
     8){
                                                if(curr_gen[organism][
    gene+1] == 0){currentCost = currentCost + table[4][0];}
                                                if(curr_gen[organism][
    gene+1] == 5){currentCost = currentCost + table[4][5];}
                                                if(curr_gen[organism][
    gene+1] == 6){currentCost = currentCost + table[4][6];}
                                                if(curr_gen[organism][
    gene+1] == 7){currentCost = currentCost + table[4][7];}
                                                if(curr_gen[organism][
    gene+1] == 8){currentCost = currentCost + table[4][8];}
                                        }
                                        else{
                                                currentCost = currentCost
     + max_cost;
                                        }
                                }

                                //if the bin is equal to 5
                                if(curr_gen[organism][gene] == 5){
                                        if(curr_gen[organism][gene+1] ==
    1 || curr_gen[organism][gene+1] == 2 || curr_gen[organism][gene+1] ==
    3 || curr_gen[organism][gene+1] == 4  || curr_gen[organism][gene+1] ==
     9 || curr_gen[organism][gene+1] == 10 || curr_gen[organism][gene+1]
    == 11 || curr_gen[organism][gene+1] == 12){
                                                if(curr_gen[organism][
    gene+1] == 1){currentCost = currentCost + table[5][1];}
                                                if(curr_gen[organism][
    gene+1] == 2){currentCost = currentCost + table[5][2];}
                                                if(curr_gen[organism][
    gene+1] == 3){currentCost = currentCost + table[5][3];}
                                                if(curr_gen[organism][
    gene+1] == 4){currentCost = currentCost + table[5][4];}
                                                if(curr_gen[organism][
    gene+1] == 9){currentCost = currentCost + table[5][9];}
                                                if(curr_gen[organism][
    gene+1] == 10){currentCost = currentCost + table[5][10];}
                                                if(curr_gen[organism][
    gene+1] == 11){currentCost = currentCost + table[5][11];}
                                                if(curr_gen[organism][
    gene+1] == 12){currentCost = currentCost + table[5][12];}
                                        }
                                        else{
                                                currentCost = currentCost
     + max_cost;
                                        }
                                }

                                //if the bin is equal to 6
                                if(curr_gen[organism][gene] == 6){
                                        if(curr_gen[organism][gene+1] ==
    1 || curr_gen[organism][gene+1] == 2 || curr_gen[organism][gene+1] ==
    3 || curr_gen[organism][gene+1] == 4  || curr_gen[organism][gene+1] ==
     9 || curr_gen[organism][gene+1] == 10 || curr_gen[organism][gene+1]
    == 11 || curr_gen[organism][gene+1] == 12){
                                                if(curr_gen[organism][
    gene+1] == 1){currentCost = currentCost + table[6][1];}
                                                if(curr_gen[organism][
    gene+1] == 2){currentCost = currentCost + table[6][2];}
                                                if(curr_gen[organism][
    gene+1] == 3){currentCost = currentCost + table[6][3];}
                                                if(curr_gen[organism][
    gene+1] == 4){currentCost = currentCost + table[6][4];}
                                                if(curr_gen[organism][
    gene+1] == 9){currentCost = currentCost + table[6][9];}
                                                if(curr_gen[organism][
    gene+1] == 10){currentCost = currentCost + table[6][10];}
```

```
                                              if(curr_gen[organism][
gene+1] == 11){currentCost = currentCost + table[6][11];}
                                              if(curr_gen[organism][
gene+1] == 12){currentCost = currentCost + table[6][12];}
                                      }
                                      else{
                                              currentCost = currentCost
 + max_cost;
                                      }
                              }

                              //if the bin is equal to 7
                              if(curr_gen[organism][gene] == 7){
                                      if(curr_gen[organism][gene+1] ==
1 || curr_gen[organism][gene+1] == 2 || curr_gen[organism][gene+1] ==
3 || curr_gen[organism][gene+1] == 4  || curr_gen[organism][gene+1] ==
 9 || curr_gen[organism][gene+1] == 10 || curr_gen[organism][gene+1]
== 11 || curr_gen[organism][gene+1] == 12){
                                              if(curr_gen[organism][
gene+1] == 1){currentCost = currentCost + table[7][1];}
                                              if(curr_gen[organism][
gene+1] == 2){currentCost = currentCost + table[7][2];}
                                              if(curr_gen[organism][
gene+1] == 3){currentCost = currentCost + table[7][3];}
                                              if(curr_gen[organism][
gene+1] == 4){currentCost = currentCost + table[7][4];}
                                              if(curr_gen[organism][
gene+1] == 9){currentCost = currentCost + table[7][9];}
                                              if(curr_gen[organism][
gene+1] == 10){currentCost = currentCost + table[7][10];}
                                              if(curr_gen[organism][
gene+1] == 11){currentCost = currentCost + table[7][11];}
                                              if(curr_gen[organism][
gene+1] == 12){currentCost = currentCost + table[7][12];}
                                      }
                                      else{
                                              currentCost = currentCost
 + max_cost;
                                      }
                              }

                              //if the bin is equal to 8
                              if(curr_gen[organism][gene] == 8){
                                      if(curr_gen[organism][gene+1] ==
1 || curr_gen[organism][gene+1] == 2 || curr_gen[organism][gene+1] ==
3 || curr_gen[organism][gene+1] == 4  || curr_gen[organism][gene+1] ==
 9 || curr_gen[organism][gene+1] == 10 || curr_gen[organism][gene+1]
== 11 || curr_gen[organism][gene+1] == 12){
```

```
                                                    if(curr_gen[organism][
gene+1] == 1){currentCost = currentCost + table[8][1];}
                                                    if(curr_gen[organism][
gene+1] == 2){currentCost = currentCost + table[8][2];}
                                                    if(curr_gen[organism][
gene+1] == 3){currentCost = currentCost + table[8][3];}
                                                    if(curr_gen[organism][
gene+1] == 4){currentCost = currentCost + table[8][4];}
                                                    if(curr_gen[organism][
gene+1] == 9){currentCost = currentCost + table[8][9];}
                                                    if(curr_gen[organism][
gene+1] == 10){currentCost = currentCost + table[8][10];}
                                                    if(curr_gen[organism][
gene+1] == 11){currentCost = currentCost + table[8][11];}
                                                    if(curr_gen[organism][
gene+1] == 12){currentCost = currentCost + table[8][12];}
                                            }
                                            else{
                                                    currentCost = currentCost
 + max_cost;
                                            }
                                    }

                                    //if the bin is equal to 9
                                    if(curr_gen[organism][gene] == 9){
                                            if(curr_gen[organism][gene+1] ==
5 || curr_gen[organism][gene+1] == 6 || curr_gen[organism][gene+1] ==
7 || curr_gen[organism][gene+1] == 8  || curr_gen[organism][gene+1] ==
 13 || curr_gen[organism][gene+1] == 14 || curr_gen[organism][gene+1]
== 15 || curr_gen[organism][gene+1] == 16){
                                                    if(curr_gen[organism][
gene+1] == 5){currentCost = currentCost + table[9][5];}
                                                    if(curr_gen[organism][
gene+1] == 6){currentCost = currentCost + table[9][6];}
                                                    if(curr_gen[organism][
gene+1] == 7){currentCost = currentCost + table[9][7];}
                                                    if(curr_gen[organism][
gene+1] == 8){currentCost = currentCost + table[9][8];}
                                                    if(curr_gen[organism][
gene+1] == 13){currentCost = currentCost + table[9][13];}
                                                    if(curr_gen[organism][
gene+1] == 14){currentCost = currentCost + table[9][14];}
                                                    if(curr_gen[organism][
gene+1] == 15){currentCost = currentCost + table[9][15];}
                                                    if(curr_gen[organism][
gene+1] == 16){currentCost = currentCost + table[9][16];}
                                            }
                                            else{
                                                    currentCost = currentCost
 + max_cost;
                                              }
                                        }

                                        //if the bin is equal to 10
                                        if(curr_gen[organism][gene] == 10){
                                              if(curr_gen[organism][gene+1] ==
5 || curr_gen[organism][gene+1] == 6 || curr_gen[organism][gene+1] ==
7 || curr_gen[organism][gene+1] == 8  || curr_gen[organism][gene+1] ==
 13 || curr_gen[organism][gene+1] == 14 || curr_gen[organism][gene+1]
== 15 || curr_gen[organism][gene+1] == 16){
                                                    if(curr_gen[organism][
gene+1] == 5){currentCost = currentCost + table[10][5];}
                                                    if(curr_gen[organism][
gene+1] == 6){currentCost = currentCost + table[10][6];}
                                                    if(curr_gen[organism][
gene+1] == 7){currentCost = currentCost + table[10][7];}
                                                    if(curr_gen[organism][
gene+1] == 8){currentCost = currentCost + table[10][8];}
                                                    if(curr_gen[organism][
gene+1] == 13){currentCost = currentCost + table[10][13];}
                                                    if(curr_gen[organism][
gene+1] == 14){currentCost = currentCost + table[10][14];}
                                                    if(curr_gen[organism][
gene+1] == 15){currentCost = currentCost + table[10][15];}
                                                    if(curr_gen[organism][
gene+1] == 16){currentCost = currentCost + table[10][16];}
                                              }
                                              else{
                                                    currentCost = currentCost
 + max_cost;
                                              }
                                        }

                                        //if the bin is equal to 11
                                        if(curr_gen[organism][gene] == 11){
                                              if(curr_gen[organism][gene+1] ==
5 || curr_gen[organism][gene+1] == 6 || curr_gen[organism][gene+1] ==
7 || curr_gen[organism][gene+1] == 8  || curr_gen[organism][gene+1] ==
 13 || curr_gen[organism][gene+1] == 14 || curr_gen[organism][gene+1]
== 15 || curr_gen[organism][gene+1] == 16){
                                                    if(curr_gen[organism][
gene+1] == 5){currentCost = currentCost + table[11][5];}
                                                    if(curr_gen[organism][
gene+1] == 6){currentCost = currentCost + table[11][6];}
                                                    if(curr_gen[organism][
gene+1] == 7){currentCost = currentCost + table[11][7];}
```

```
                                                if(curr_gen[organism][
gene+1] == 8){currentCost = currentCost + table[11][8];}
                                                if(curr_gen[organism][
gene+1] == 13){currentCost = currentCost + table[11][13];}
                                                if(curr_gen[organism][
gene+1] == 14){currentCost = currentCost + table[11][14];}
                                                if(curr_gen[organism][
gene+1] == 15){currentCost = currentCost + table[11][15];}
                                                if(curr_gen[organism][
gene+1] == 16){currentCost = currentCost + table[11][16];}
                                        }
                                        else{
                                                currentCost = currentCost
 + max_cost;
                                        }
                                }

                                //if the bin is equal to 12
                                if(curr_gen[organism][gene] == 12){
                                        if(curr_gen[organism][gene+1] ==
5 || curr_gen[organism][gene+1] == 6 || curr_gen[organism][gene+1] ==
7 || curr_gen[organism][gene+1] == 8  || curr_gen[organism][gene+1] ==
 13 || curr_gen[organism][gene+1] == 14 || curr_gen[organism][gene+1]
== 15 || curr_gen[organism][gene+1] == 16){
                                                if(curr_gen[organism][
gene+1] == 5){currentCost = currentCost + table[12][5];}
                                                if(curr_gen[organism][
gene+1] == 6){currentCost = currentCost + table[12][6];}
                                                if(curr_gen[organism][
gene+1] == 7){currentCost = currentCost + table[12][7];}
                                                if(curr_gen[organism][
gene+1] == 8){currentCost = currentCost + table[12][8];}
                                                if(curr_gen[organism][
gene+1] == 13){currentCost = currentCost + table[12][13];}
                                                if(curr_gen[organism][
gene+1] == 14){currentCost = currentCost + table[12][14];}
                                                if(curr_gen[organism][
gene+1] == 15){currentCost = currentCost + table[12][15];}
                                                if(curr_gen[organism][
gene+1] == 16){currentCost = currentCost + table[12][16];}
                                        }
                                        else{
                                                currentCost = currentCost
 + max_cost;
                                        }
                                }

                                //if the bin is equal to 13
                                if(curr_gen[organism][gene] == 13){
                                        if(curr_gen[organism][gene+1] ==
9 || curr_gen[organism][gene+1] == 10 || curr_gen[organism][gene+1] ==
 11 || curr_gen[organism][gene+1] == 12  || curr_gen[organism][gene+1]
 == 17 || curr_gen[organism][gene+1] == 18){
                                                if(curr_gen[organism][
gene+1] == 9){currentCost = currentCost + table[13][9];}
                                                if(curr_gen[organism][
gene+1] == 10){currentCost = currentCost + table[13][10];}
                                                if(curr_gen[organism][
gene+1] == 11){currentCost = currentCost + table[13][11];}
                                                if(curr_gen[organism][
gene+1] == 12){currentCost = currentCost + table[13][12];}
                                                if(curr_gen[organism][
gene+1] == 17){currentCost = currentCost + table[13][17];}
                                                if(curr_gen[organism][
gene+1] == 18){currentCost = currentCost + table[13][18];}
                                        }
                                        else{
                                                currentCost = currentCost
 + max_cost;
                                        }
                                }

                                //if the bin is equal to 14
                                if(curr_gen[organism][gene] == 14){
                                        if(curr_gen[organism][gene+1] ==
9 || curr_gen[organism][gene+1] == 10 || curr_gen[organism][gene+1] ==
 11 || curr_gen[organism][gene+1] == 12  || curr_gen[organism][gene+1]
 == 17 || curr_gen[organism][gene+1] == 18){
                                                if(curr_gen[organism][
gene+1] == 9){currentCost = currentCost + table[14][9];}
                                                if(curr_gen[organism][
gene+1] == 10){currentCost = currentCost + table[14][10];}
                                                if(curr_gen[organism][
gene+1] == 11){currentCost = currentCost + table[14][11];}
                                                if(curr_gen[organism][
gene+1] == 12){currentCost = currentCost + table[14][12];}
                                                if(curr_gen[organism][
gene+1] == 17){currentCost = currentCost + table[14][17];}
                                                if(curr_gen[organism][
gene+1] == 18){currentCost = currentCost + table[14][18];}
                                        }
                                        else{
                                                currentCost = currentCost
 + max_cost;
                                        }
                                }
```

```
                              //if the bin is equal to 15
                              if(curr_gen[organism][gene] == 15){
                                      if(curr_gen[organism][gene+1] ==
9 || curr_gen[organism][gene+1] == 10 || curr_gen[organism][gene+1] ==
 11 || curr_gen[organism][gene+1] == 12  || curr_gen[organism][gene+1]
 == 17 || curr_gen[organism][gene+1] == 18){
                                              if(curr_gen[organism][
gene+1] == 9){currentCost = currentCost + table[15][9];}
                                              if(curr_gen[organism][
gene+1] == 10){currentCost = currentCost + table[15][10];}
                                              if(curr_gen[organism][
gene+1] == 11){currentCost = currentCost + table[15][11];}
                                              if(curr_gen[organism][
gene+1] == 12){currentCost = currentCost + table[15][12];}
                                              if(curr_gen[organism][
gene+1] == 17){currentCost = currentCost + table[15][17];}
                                              if(curr_gen[organism][
gene+1] == 18){currentCost = currentCost + table[15][18];}
                                      }
                                      else{
                                              currentCost = currentCost
 + max_cost;
                                      }
                              }

                              //if the bin is equal to 16
                              if(curr_gen[organism][gene] == 16){
                                      if(curr_gen[organism][gene+1] ==
9 || curr_gen[organism][gene+1] == 10 || curr_gen[organism][gene+1] ==
 11 || curr_gen[organism][gene+1] == 12  || curr_gen[organism][gene+1]
 == 17 || curr_gen[organism][gene+1] == 18){
                                              if(curr_gen[organism][
gene+1] == 9){currentCost = currentCost + table[16][9];}
                                              if(curr_gen[organism][
gene+1] == 10){currentCost = currentCost + table[16][10];}
                                              if(curr_gen[organism][
gene+1] == 11){currentCost = currentCost + table[16][11];}
                                              if(curr_gen[organism][
gene+1] == 12){currentCost = currentCost + table[16][12];}
                                              if(curr_gen[organism][
gene+1] == 17){currentCost = currentCost + table[16][17];}
                                              if(curr_gen[organism][
gene+1] == 18){currentCost = currentCost + table[16][18];}
                                      }
                                      else{
                                              currentCost = currentCost
 + max_cost;
```

```
                                        }
                                }

                                //if the bin is equal to 17
                                if(curr_gen[organism][gene] == 17){
                                        if(curr_gen[organism][gene+1] ==
13 || curr_gen[organism][gene+1] == 14 || curr_gen[organism][gene+1]
== 15 || curr_gen[organism][gene+1] == 16  || curr_gen[organism][gene
+1] == 19){
                                                if(curr_gen[organism][
gene+1] == 13){currentCost = currentCost + table[17][13];}
                                                if(curr_gen[organism][
gene+1] == 14){currentCost = currentCost + table[17][14];}
                                                if(curr_gen[organism][
gene+1] == 15){currentCost = currentCost + table[17][15];}
                                                if(curr_gen[organism][
gene+1] == 16){currentCost = currentCost + table[17][16];}
                                                if(curr_gen[organism][
gene+1] == 19){currentCost = currentCost + table[17][19];}
                                        }
                                        else{
                                                currentCost = currentCost
 + max_cost;
                                        }
                                }

                                //if the bin is equal to 18
                                if(curr_gen[organism][gene] == 18){
                                        if(curr_gen[organism][gene+1] ==
13 || curr_gen[organism][gene+1] == 14 || curr_gen[organism][gene+1]
== 15 || curr_gen[organism][gene+1] == 16  || curr_gen[organism][gene
+1] == 19){
                                                if(curr_gen[organism][
gene+1] == 13){currentCost = currentCost + table[18][13];}
                                                if(curr_gen[organism][
gene+1] == 14){currentCost = currentCost + table[18][14];}
                                                if(curr_gen[organism][
gene+1] == 15){currentCost = currentCost + table[18][15];}
                                                if(curr_gen[organism][
gene+1] == 16){currentCost = currentCost + table[18][16];}
                                                if(curr_gen[organism][
gene+1] == 19){currentCost = currentCost + table[18][19];}
                                        }
                                        else{
                                                currentCost = currentCost
 + max_cost;
                                        }
                                }
                                //if the bin is equal to 19
                                if(curr_gen[organism][gene] == 19){
                                        if(curr_gen[organism][gene+1] ==
    17 || curr_gen[organism][gene+1] == 18){
                                                if(curr_gen[organism][
    gene+1] == 17){currentCost = currentCost + table[19][17];}
                                                if(curr_gen[organism][
    gene+1] == 18){currentCost = currentCost + table[19][18];}
                                        }
                                        else{
                                                currentCost = currentCost
     + max_cost;
                                        }
                                }

                        }
                        else{
                                break;
                        }

                }
                org_cost[organism] = currentCost;
                sort_cost[organism] = currentCost;
                tot_cost = tot_cost + currentCost;
        }

        //sort the sort_cost array
        for(organism = 0; organism < N_org-1; organism++){
                for(i = organism+1; i < N_org; i++){
                        if(sort_cost[organism] > sort_cost[i]){
                                temp = sort_cost[organism];
                                sort_cost[organism] = sort_cost[i];
                                sort_cost[i] = temp;
                        }
                }
        }


        //save the min cost and the position of the fittest organism
        min_cost = org_cost[0];
        for(organism = 1; organism < N_org; organism++){
                if(org_cost[organism] < min_cost){
                        min_cost = org_cost[organism];
                        clon = organism;
                }
        }
}
```

```
/*

*/

void prod_next_gen(void){
        int organism;
        int gene, i, count;
        int ParentOne;
        int ParentTwo;
        int intersection[N_gen];
        int crossoverPoint, point;
        int mutate;

        for(organism = 0; organism < N_org; organism++){
                for(gene = 0; gene < N_gen; gene++){
                        next_gen[organism][gene] = N_gen;
                }
        }

        for(organism = 0; organism < N_org; organism = organism+2){
                //select the two parents
                ParentOne = select_one();
                ParentTwo = select_one();

                //initialize the arrays that will contain the
    intersections points between the parents
                for(gene = 0; gene < N_gen; gene++){
                        intersection[gene] = N_gen;
                }

                //search for the intersection points and saves it on the
    intersection array
                i = 0;
                for(gene = 1; gene < N_gen; gene++){
                        if(curr_gen[ParentOne][gene] != N_gen && curr_gen
    [ParentTwo][gene] != N_gen && curr_gen[ParentOne][gene] != destination
     && curr_gen[ParentTwo][gene] != destination){

                                if(curr_gen[ParentOne][gene] == curr_gen[
    ParentTwo][gene]){
                                        intersection[i] = gene;
                                        i++;
                                }
                        }
                }
                //scan the intersection array and save in count the
number of the intersection points
                //initialize count
                count = 0;
                for(gene = 0; gene < N_gen; ++gene){
                        if(intersection[gene] != N_gen){count++;}
                }

                //if count = 0; there is no intersection point. Copy in
next_gen the ParentOne and ParentTwo
                if(count == 0){
                        for(gene = 0; gene < N_gen; ++gene){
                                next_gen[organism][gene] = curr_gen[
ParentOne][gene];
                                next_gen[organism+1][gene] = curr_gen[
ParentTwo][gene];
                        }
                }
                //if count != 0
                else{
                        //if count = 1, use the singular intersection
point as crossover point
                        if(count == 1){
                                crossoverPoint = intersection[0];
                        }
                        //if count != [0,1], choose a random crossover
point between the intersections
                        else if(count != 0 && count != 1){
                                point = rand() % count;
                                crossoverPoint = intersection[point];
                        }

                        //once obtained the crossover point, start the
mating between the parents
                        for(gene = 0; gene < N_gen; ++gene){
                                //apply the mutation
                                mutate = rand() % (int)(1.0/Mut_rate);
                                if(mutate == 0){
                                        next_gen[organism][gene] = rand()
 % N_gen;
                                }
                                //if not mutated, make the crossover of
genes between the parents
                                else{
                                        if(gene < crossoverPoint){
                                                next_gen[organism][gene]
= curr_gen[ParentOne][gene];
```

```
                                                        next_gen[organism+1][gene
    ] = curr_gen[ParentTwo][gene];
                                                }
                                                else{
                                                        next_gen[organism][gene]
    = curr_gen[ParentTwo][gene];
                                                        next_gen[organism+1][gene
    ] = curr_gen[ParentOne][gene];
                                                }
                                        }
                                }

                                //valutate the fitness of the new organisms
                                eval_org(organism);
                                eval_org(organism+1);
                        }
                }

                //copy the offsprings in the curr_gen
                for(organism = 0; organism < N_org; organism++){
                        for(gene = 0; gene < N_gen; gene++){
                                curr_gen[organism][gene] = next_gen[organism][
    gene];
                        }
                }

}

/*

*/

void eval_org(int organism){

        int gene;
        int currentCost = 0;

        for(gene = 0; gene < N_gen; ++gene){

                //verificates if the bin is not 20 or the destination
    node (end of the info), else break the analysis
                if(next_gen[organism][gene] != 20 && next_gen[organism][
    gene] != destination){

                        //if the bin is equal to 0
                        if(next_gen[organism][gene] == 0){
                                if(next_gen[organism][gene+1] == 1 ||
next_gen[organism][gene+1] == 2 || next_gen[organism][gene+1] == 3  ||
 next_gen[organism][gene+1] == 4){
                                        if(next_gen[organism][gene+1] ==
1){currentCost = currentCost + table[0][1];}
                                        if(next_gen[organism][gene+1] ==
2){currentCost = currentCost + table[0][2];}
                                        if(next_gen[organism][gene+1] ==
3){currentCost = currentCost + table[0][3];}
                                        if(next_gen[organism][gene+1] ==
4){currentCost = currentCost + table[0][4];}
                                }
                                else{
                                        currentCost = currentCost +
max_cost;
                                }
                        }

                        //if the bin is equal to 1
                        if(next_gen[organism][gene] == 1){
                                if(next_gen[organism][gene+1] == 0 ||
next_gen[organism][gene+1] == 5 || next_gen[organism][gene+1] == 6 ||
next_gen[organism][gene+1] == 7  || next_gen[organism][gene+1] == 8){
                                        if(next_gen[organism][gene+1] ==
0){currentCost = currentCost + table[1][0];}
                                        if(next_gen[organism][gene+1] ==
5){currentCost = currentCost + table[1][5];}
                                        if(next_gen[organism][gene+1] ==
6){currentCost = currentCost + table[1][6];}
                                        if(next_gen[organism][gene+1] ==
7){currentCost = currentCost + table[1][7];}
                                        if(next_gen[organism][gene+1] ==
8){currentCost = currentCost + table[1][8];}
                                }
                                else{
                                        currentCost = currentCost +
max_cost;
                                }
                        }

                        //if the bin is equal to 2
                        if(next_gen[organism][gene] == 2){
                                if(next_gen[organism][gene+1] == 0 ||
next_gen[organism][gene+1] == 5 || next_gen[organism][gene+1] == 6 ||
next_gen[organism][gene+1] == 7  || next_gen[organism][gene+1] == 8){
                                        if(next_gen[organism][gene+1] ==
0){currentCost = currentCost + table[2][0];}
```

```
                                                if(next_gen[organism][gene+1] ==
5){currentCost = currentCost + table[2][5];}
                                                if(next_gen[organism][gene+1] ==
6){currentCost = currentCost + table[2][6];}
                                                if(next_gen[organism][gene+1] ==
7){currentCost = currentCost + table[2][7];}
                                                if(next_gen[organism][gene+1] ==
8){currentCost = currentCost + table[2][8];}
                                        }
                                        else{
                                                currentCost = currentCost +
max_cost;
                                        }
                                }

                                //if the bin is equal to 3
                                if(next_gen[organism][gene] == 3){
                                        if(next_gen[organism][gene+1] == 0 ||
next_gen[organism][gene+1] == 5 || next_gen[organism][gene+1] == 6 ||
next_gen[organism][gene+1] == 7  || next_gen[organism][gene+1] == 8){
                                                if(next_gen[organism][gene+1] ==
0){currentCost = currentCost + table[3][0];}
                                                if(next_gen[organism][gene+1] ==
5){currentCost = currentCost + table[3][5];}
                                                if(next_gen[organism][gene+1] ==
6){currentCost = currentCost + table[3][6];}
                                                if(next_gen[organism][gene+1] ==
7){currentCost = currentCost + table[3][7];}
                                                if(next_gen[organism][gene+1] ==
8){currentCost = currentCost + table[3][8];}
                                        }
                                        else{
                                                currentCost = currentCost +
max_cost;
                                        }
                                }

                                //if the bin is equal to 4
                                if(next_gen[organism][gene] == 4){
                                        if(next_gen[organism][gene+1] == 0 ||
next_gen[organism][gene+1] == 5 || next_gen[organism][gene+1] == 6 ||
next_gen[organism][gene+1] == 7  || next_gen[organism][gene+1] == 8){
                                                if(next_gen[organism][gene+1] ==
0){currentCost = currentCost + table[4][0];}
                                                if(next_gen[organism][gene+1] ==
5){currentCost = currentCost + table[4][5];}
                                                if(next_gen[organism][gene+1] ==
6){currentCost = currentCost + table[4][6];}
```

```
                                        if(next_gen[organism][gene+1] ==
7){currentCost = currentCost + table[4][7];}
                                        if(next_gen[organism][gene+1] ==
8){currentCost = currentCost + table[4][8];}
                                }
                                else{
                                        currentCost = currentCost +
max_cost;
                                }
                        }

                        //if the bin is equal to 5
                        if(next_gen[organism][gene] == 5){
                                if(next_gen[organism][gene+1] == 1 ||
next_gen[organism][gene+1] == 2 || next_gen[organism][gene+1] == 3 ||
next_gen[organism][gene+1] == 4  || next_gen[organism][gene+1] == 9 ||
 next_gen[organism][gene+1] == 10 || next_gen[organism][gene+1] == 11
|| next_gen[organism][gene+1] == 12){
                                        if(next_gen[organism][gene+1] ==
1){currentCost = currentCost + table[5][1];}
                                        if(next_gen[organism][gene+1] ==
2){currentCost = currentCost + table[5][2];}
                                        if(next_gen[organism][gene+1] ==
3){currentCost = currentCost + table[5][3];}
                                        if(next_gen[organism][gene+1] ==
4){currentCost = currentCost + table[5][4];}
                                        if(next_gen[organism][gene+1] ==
9){currentCost = currentCost + table[5][9];}
                                        if(next_gen[organism][gene+1] ==
10){currentCost = currentCost + table[5][10];}
                                        if(next_gen[organism][gene+1] ==
11){currentCost = currentCost + table[5][11];}
                                        if(next_gen[organism][gene+1] ==
12){currentCost = currentCost + table[5][12];}
                                }
                                else{
                                        currentCost = currentCost +
max_cost;
                                }
                        }

                        //if the bin is equal to 6
                        if(next_gen[organism][gene] == 6){
                                if(next_gen[organism][gene+1] == 1 ||
next_gen[organism][gene+1] == 2 || next_gen[organism][gene+1] == 3 ||
next_gen[organism][gene+1] == 4  || next_gen[organism][gene+1] == 9 ||
 next_gen[organism][gene+1] == 10 || next_gen[organism][gene+1] == 11
|| next_gen[organism][gene+1] == 12){
```

```
                                        if(next_gen[organism][gene+1] ==
    1){currentCost = currentCost + table[6][1];}
                                        if(next_gen[organism][gene+1] ==
    2){currentCost = currentCost + table[6][2];}
                                        if(next_gen[organism][gene+1] ==
    3){currentCost = currentCost + table[6][3];}
                                        if(next_gen[organism][gene+1] ==
    4){currentCost = currentCost + table[6][4];}
                                        if(next_gen[organism][gene+1] ==
    9){currentCost = currentCost + table[6][9];}
                                        if(next_gen[organism][gene+1] ==
    10){currentCost = currentCost + table[6][10];}
                                        if(next_gen[organism][gene+1] ==
    11){currentCost = currentCost + table[6][11];}
                                        if(next_gen[organism][gene+1] ==
    12){currentCost = currentCost + table[6][12];}
                                }
                                else{
                                        currentCost = currentCost +
    max_cost;
                                }
                        }

                        //if the bin is equal to 7
                        if(next_gen[organism][gene] == 7){
                                if(next_gen[organism][gene+1] == 1 ||
    next_gen[organism][gene+1] == 2 || next_gen[organism][gene+1] == 3 ||
    next_gen[organism][gene+1] == 4  || next_gen[organism][gene+1] == 9 ||
     next_gen[organism][gene+1] == 10 || next_gen[organism][gene+1] == 11
    || next_gen[organism][gene+1] == 12){
                                        if(next_gen[organism][gene+1] ==
    1){currentCost = currentCost + table[7][1];}
                                        if(next_gen[organism][gene+1] ==
    2){currentCost = currentCost + table[7][2];}
                                        if(next_gen[organism][gene+1] ==
    3){currentCost = currentCost + table[7][3];}
                                        if(next_gen[organism][gene+1] ==
    4){currentCost = currentCost + table[7][4];}
                                        if(next_gen[organism][gene+1] ==
    9){currentCost = currentCost + table[7][9];}
                                        if(next_gen[organism][gene+1] ==
    10){currentCost = currentCost + table[7][10];}
                                        if(next_gen[organism][gene+1] ==
    11){currentCost = currentCost + table[7][11];}
                                        if(next_gen[organism][gene+1] ==
    12){currentCost = currentCost + table[7][12];}
                                }
                                else{
                                        currentCost = currentCost +
max_cost;
                                }
                        }

                        //if the bin is equal to 8
                        if(next_gen[organism][gene] == 8){
                                if(next_gen[organism][gene+1] == 1 ||
next_gen[organism][gene+1] == 2 || next_gen[organism][gene+1] == 3 ||
next_gen[organism][gene+1] == 4  || next_gen[organism][gene+1] == 9 ||
 next_gen[organism][gene+1] == 10 || next_gen[organism][gene+1] == 11
|| next_gen[organism][gene+1] == 12){
                                        if(next_gen[organism][gene+1] ==
1){currentCost = currentCost + table[8][1];}
                                        if(next_gen[organism][gene+1] ==
2){currentCost = currentCost + table[8][2];}
                                        if(next_gen[organism][gene+1] ==
3){currentCost = currentCost + table[8][3];}
                                        if(next_gen[organism][gene+1] ==
4){currentCost = currentCost + table[8][4];}
                                        if(next_gen[organism][gene+1] ==
9){currentCost = currentCost + table[8][9];}
                                        if(next_gen[organism][gene+1] ==
10){currentCost = currentCost + table[8][10];}
                                        if(next_gen[organism][gene+1] ==
11){currentCost = currentCost + table[8][11];}
                                        if(next_gen[organism][gene+1] ==
12){currentCost = currentCost + table[8][12];}
                                }
                                else{
                                        currentCost = currentCost +
max_cost;
                                }
                        }

                        //if the bin is equal to 9
                        if(next_gen[organism][gene] == 9){
                                if(next_gen[organism][gene+1] == 5 ||
next_gen[organism][gene+1] == 6 || next_gen[organism][gene+1] == 7 ||
next_gen[organism][gene+1] == 8  || next_gen[organism][gene+1] == 13
|| next_gen[organism][gene+1] == 14 || next_gen[organism][gene+1] ==
15 || next_gen[organism][gene+1] == 16){
                                        if(next_gen[organism][gene+1] ==
5){currentCost = currentCost + table[9][5];}
                                        if(next_gen[organism][gene+1] ==
6){currentCost = currentCost + table[9][6];}
                                        if(next_gen[organism][gene+1] ==
7){currentCost = currentCost + table[9][7];}
```

```
                                        if(next_gen[organism][gene+1] ==
8){currentCost = currentCost + table[9][8];}
                                        if(next_gen[organism][gene+1] ==
13){currentCost = currentCost + table[9][13];}
                                        if(next_gen[organism][gene+1] ==
14){currentCost = currentCost + table[9][14];}
                                        if(next_gen[organism][gene+1] ==
15){currentCost = currentCost + table[9][15];}
                                        if(next_gen[organism][gene+1] ==
16){currentCost = currentCost + table[9][16];}
                                }
                                else{
                                        currentCost = currentCost +
max_cost;
                                }
                        }


                        //if the bin is equal to 10
                        if(next_gen[organism][gene] == 10){
                                if(next_gen[organism][gene+1] == 5 ||
next_gen[organism][gene+1] == 6 || next_gen[organism][gene+1] == 7 ||
next_gen[organism][gene+1] == 8  || next_gen[organism][gene+1] == 13
|| next_gen[organism][gene+1] == 14 || next_gen[organism][gene+1] ==
15 || next_gen[organism][gene+1] == 16){
                                        if(next_gen[organism][gene+1] ==
5){currentCost = currentCost + table[10][5];}
                                        if(next_gen[organism][gene+1] ==
6){currentCost = currentCost + table[10][6];}
                                        if(next_gen[organism][gene+1] ==
7){currentCost = currentCost + table[10][7];}
                                        if(next_gen[organism][gene+1] ==
8){currentCost = currentCost + table[10][8];}
                                        if(next_gen[organism][gene+1] ==
13){currentCost = currentCost + table[10][13];}
                                        if(next_gen[organism][gene+1] ==
14){currentCost = currentCost + table[10][14];}
                                        if(next_gen[organism][gene+1] ==
15){currentCost = currentCost + table[10][15];}
                                        if(next_gen[organism][gene+1] ==
16){currentCost = currentCost + table[10][16];}
                                }
                                else{
                                        currentCost = currentCost +
max_cost;
                                }
                        }

                        //if the bin is equal to 11
```

```
                    if(next_gen[organism][gene] == 11){
                            if(next_gen[organism][gene+1] == 5 ||
next_gen[organism][gene+1] == 6 || next_gen[organism][gene+1] == 7 ||
next_gen[organism][gene+1] == 8  || next_gen[organism][gene+1] == 13
|| next_gen[organism][gene+1] == 14 || next_gen[organism][gene+1] ==
15 || next_gen[organism][gene+1] == 16){
                                    if(next_gen[organism][gene+1] ==
5){currentCost = currentCost + table[11][5];}
                                    if(next_gen[organism][gene+1] ==
6){currentCost = currentCost + table[11][6];}
                                    if(next_gen[organism][gene+1] ==
7){currentCost = currentCost + table[11][7];}
                                    if(next_gen[organism][gene+1] ==
8){currentCost = currentCost + table[11][8];}
                                    if(next_gen[organism][gene+1] ==
13){currentCost = currentCost + table[11][13];}
                                    if(next_gen[organism][gene+1] ==
14){currentCost = currentCost + table[11][14];}
                                    if(next_gen[organism][gene+1] ==
15){currentCost = currentCost + table[11][15];}
                                    if(next_gen[organism][gene+1] ==
16){currentCost = currentCost + table[11][16];}
                            }
                            else{
                                    currentCost = currentCost +
max_cost;
                            }
                    }

                    //if the bin is equal to 12
                    if(next_gen[organism][gene] == 12){
                            if(next_gen[organism][gene+1] == 5 ||
next_gen[organism][gene+1] == 6 || next_gen[organism][gene+1] == 7 ||
next_gen[organism][gene+1] == 8  || next_gen[organism][gene+1] == 13
|| next_gen[organism][gene+1] == 14 || next_gen[organism][gene+1] ==
15 || next_gen[organism][gene+1] == 16){
                                    if(next_gen[organism][gene+1] ==
5){currentCost = currentCost + table[12][5];}
                                    if(next_gen[organism][gene+1] ==
6){currentCost = currentCost + table[12][6];}
                                    if(next_gen[organism][gene+1] ==
7){currentCost = currentCost + table[12][7];}
                                    if(next_gen[organism][gene+1] ==
8){currentCost = currentCost + table[12][8];}
                                    if(next_gen[organism][gene+1] ==
13){currentCost = currentCost + table[12][13];}
                                    if(next_gen[organism][gene+1] ==
14){currentCost = currentCost + table[12][14];}
```

```
                                            if(next_gen[organism][gene+1] ==
15){currentCost = currentCost + table[12][15];}
                                            if(next_gen[organism][gene+1] ==
16){currentCost = currentCost + table[12][16];}
                                    }
                                    else{
                                            currentCost = currentCost +
max_cost;
                                    }
                            }

                            //if the bin is equal to 13
                            if(next_gen[organism][gene] == 13){
                                    if(next_gen[organism][gene+1] == 9 ||
next_gen[organism][gene+1] == 10 || next_gen[organism][gene+1] == 11
|| next_gen[organism][gene+1] == 12  || next_gen[organism][gene+1] ==
17 || next_gen[organism][gene+1] == 18){
                                            if(next_gen[organism][gene+1] ==
9){currentCost = currentCost + table[13][9];}
                                            if(next_gen[organism][gene+1] ==
10){currentCost = currentCost + table[13][10];}
                                            if(next_gen[organism][gene+1] ==
11){currentCost = currentCost + table[13][11];}
                                            if(next_gen[organism][gene+1] ==
12){currentCost = currentCost + table[13][12];}
                                            if(next_gen[organism][gene+1] ==
17){currentCost = currentCost + table[13][17];}
                                            if(next_gen[organism][gene+1] ==
18){currentCost = currentCost + table[13][18];}
                                    }
                                    else{
                                            currentCost = currentCost +
max_cost;
                                    }
                            }

                            //if the bin is equal to 14
                            if(next_gen[organism][gene] == 14){
                                    if(next_gen[organism][gene+1] == 9 ||
next_gen[organism][gene+1] == 10 || next_gen[organism][gene+1] == 11
|| next_gen[organism][gene+1] == 12  || next_gen[organism][gene+1] ==
17 || next_gen[organism][gene+1] == 18){
                                            if(next_gen[organism][gene+1] ==
9){currentCost = currentCost + table[14][9];}
                                            if(next_gen[organism][gene+1] ==
10){currentCost = currentCost + table[14][10];}
                                            if(next_gen[organism][gene+1] ==
11){currentCost = currentCost + table[14][11];}
```

```
                                            if(next_gen[organism][gene+1] ==
12){currentCost = currentCost + table[14][12];}
                                            if(next_gen[organism][gene+1] ==
17){currentCost = currentCost + table[14][17];}
                                            if(next_gen[organism][gene+1] ==
18){currentCost = currentCost + table[14][18];}
                                    }
                                    else{
                                            currentCost = currentCost +
max_cost;
                                    }
                            }

                            //if the bin is equal to 15
                            if(next_gen[organism][gene] == 15){
                                    if(next_gen[organism][gene+1] == 9 ||
next_gen[organism][gene+1] == 10 || next_gen[organism][gene+1] == 11
|| next_gen[organism][gene+1] == 12  || next_gen[organism][gene+1] ==
17 || next_gen[organism][gene+1] == 18){
                                            if(next_gen[organism][gene+1] ==
9){currentCost = currentCost + table[15][9];}
                                            if(next_gen[organism][gene+1] ==
10){currentCost = currentCost + table[15][10];}
                                            if(next_gen[organism][gene+1] ==
11){currentCost = currentCost + table[15][11];}
                                            if(next_gen[organism][gene+1] ==
12){currentCost = currentCost + table[15][12];}
                                            if(next_gen[organism][gene+1] ==
17){currentCost = currentCost + table[15][17];}
                                            if(next_gen[organism][gene+1] ==
18){currentCost = currentCost + table[15][18];}
                                    }
                                    else{
                                            currentCost = currentCost +
max_cost;
                                    }
                            }

                            //if the bin is equal to 16
                            if(next_gen[organism][gene] == 16){
                                    if(next_gen[organism][gene+1] == 9 ||
next_gen[organism][gene+1] == 10 || next_gen[organism][gene+1] == 11
|| next_gen[organism][gene+1] == 12  || next_gen[organism][gene+1] ==
17 || next_gen[organism][gene+1] == 18){
                                            if(next_gen[organism][gene+1] ==
9){currentCost = currentCost + table[16][9];}
                                            if(next_gen[organism][gene+1] ==
10){currentCost = currentCost + table[16][10];}
```

```
                                                if(next_gen[organism][gene+1] ==
11){currentCost = currentCost + table[16][11];}
                                                if(next_gen[organism][gene+1] ==
12){currentCost = currentCost + table[16][12];}
                                                if(next_gen[organism][gene+1] ==
17){currentCost = currentCost + table[16][17];}
                                                if(next_gen[organism][gene+1] ==
18){currentCost = currentCost + table[16][18];}
                                        }
                                        else{
                                                currentCost = currentCost +
max_cost;
                                        }
                                }

                                //if the bin is equal to 17
                                if(next_gen[organism][gene] == 17){
                                        if(next_gen[organism][gene+1] == 13 ||
next_gen[organism][gene+1] == 14 || next_gen[organism][gene+1] == 15
|| next_gen[organism][gene+1] == 16  || next_gen[organism][gene+1] ==
19){
                                                if(next_gen[organism][gene+1] ==
13){currentCost = currentCost + table[17][13];}
                                                if(next_gen[organism][gene+1] ==
14){currentCost = currentCost + table[17][14];}
                                                if(next_gen[organism][gene+1] ==
15){currentCost = currentCost + table[17][15];}
                                                if(next_gen[organism][gene+1] ==
16){currentCost = currentCost + table[17][16];}
                                                if(next_gen[organism][gene+1] ==
19){currentCost = currentCost + table[17][19];}
                                        }
                                        else{
                                                currentCost = currentCost +
max_cost;
                                        }
                                }

                                //if the bin is equal to 18
                                if(next_gen[organism][gene] == 18){
                                        if(next_gen[organism][gene+1] == 13 ||
next_gen[organism][gene+1] == 14 || next_gen[organism][gene+1] == 15
|| next_gen[organism][gene+1] == 16  || next_gen[organism][gene+1] ==
19){
                                                if(next_gen[organism][gene+1] ==
13){currentCost = currentCost + table[18][13];}
                                                if(next_gen[organism][gene+1] ==
14){currentCost = currentCost + table[18][14];}
```

```
                                                if(next_gen[organism][gene+1] ==
15){currentCost = currentCost + table[18][15];}
                                                if(next_gen[organism][gene+1] ==
16){currentCost = currentCost + table[18][16];}
                                                if(next_gen[organism][gene+1] ==
19){currentCost = currentCost + table[18][19];}
                                        }
                                        else{
                                                currentCost = currentCost +
max_cost;
                                        }
                                }

                                //if the bin is equal to 19
                                if(next_gen[organism][gene] == 19){
                                        if(next_gen[organism][gene+1] == 17 ||
next_gen[organism][gene+1] == 18){
                                                if(next_gen[organism][gene+1] ==
17){currentCost = currentCost + table[19][17];}
                                                if(next_gen[organism][gene+1] ==
18){currentCost = currentCost + table[19][18];}
                                        }
                                        else{
                                                currentCost = currentCost +
max_cost;
                                        }
                                }

                        }
                        else{
                                break;
                        }

                }

                int a, b, c;
                a = bound_rand((int)(N_org/2), (N_org-1));
                b = sort_cost[a];
                //if the offspring is less fitter than the selected, select
another one
                if(currentCost > b){
                        c = select_best();
                        for(gene = 0; gene < N_gen; gene++){
                                next_gen[organism][gene] = curr_gen[c][gene];
                        }
                }

}
```

```
/*

*/

int select_one(void){
        int organism;
        int randomSelectPoint;
        int runningTotal;

        runningTotal = tot_cost;

        randomSelectPoint = rand() % (tot_cost+1);

        for(organism = 0; organism < N_org; ++organism){
                runningTotal = runningTotal - org_cost[organism];
                if(runningTotal <= randomSelectPoint){
                        return organism;
                }
        }

}

int select_best(void){
        int organism;
        int mean_value;
        int runningTotal;
        int randomSelectPoint;

        //the select point is the mean value of the organism's cost
        mean_value = (int)(tot_cost/N_org);

        //
        randomSelectPoint = bound_rand(min_cost, mean_value);

        for(organism = 0; organism < N_org; ++organism){
                if(org_cost[organism] <= randomSelectPoint){
                        return organism;
                }
        }
}

/*
Auxiliar functions to choose between the neighbors during the creation of
     the first generation
*/
```

```
int bound_rand(int min, int max){
        int ans;
        ans = (rand() % ((max+1)-min)) + min;

        return ans;
}

int num_rand(int num, int min, int max){
        int aux;
        int ans;
        aux = rand() % 2;
        if(aux == 0){ans = num;}
        else{ans = bound_rand(min,max);}

        return ans;
}

int rand_rand(int min1, int max1, int min2, int max2){
        int aux;
        int ans;
        aux = rand() % 2;
        if(aux == 0){ans = bound_rand(min1,max1);}
        else{ans = bound_rand(min2,max2);}

        return ans;
}
```

Bibliography

[1] A. Inc., *Epiphany SDK Reference.* Adapteva Inc., 2013.

[2] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[3] E. F. Moore, *The shortest path through a maze.* Bell Telephone System., 1959.

[4] M. Sniedovich, "Dijkstra's algorithm revisited: the dynamic programming connexion," *Control and cybernetics*, vol. 35, no. 3, p. 599, 2006.

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein *et al.*, *Introduction to algorithms.* MIT press Cambridge, 2001, vol. 2.

[6] J. Inagaki, M. Haseyama, and H. Kitajima, "A genetic algorithm for determining multiple routes and its applications," in *Circuits and Systems, 1999. ISCAS'99. Proceedings of the 1999 IEEE International Symposium on*, vol. 6. IEEE, 1999, pp. 137–140.

[7] M. Noto and H. Sato, "A method for the shortest path search by extended dijkstra algorithm," in *Systems, Man, and Cybernetics, 2000 IEEE International Conference on*, vol. 3. IEEE, 2000, pp. 2316–2320.

[8] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*, 1st ed. Addison-Wesley Professional, 1989.

[9] F. B. Zhan, "Three fastest shortest path algorithms on real road networks: Data structures and procedures," *Journal of geographic information and decision analysis*, vol. 1, no. 1, pp. 69–82, 1997.

[10] A. Chaudhary and N. K. Pandey, "Genetic algorithm for shortest path in packet switching networks," *Journal of Theoretical and Applied Information Technology*, vol. 29, no. 2, 2011.

[11] A. Inc., *Epiphany 16 core Microprocessor datasheet.* Adapteva Inc., 2013.

[12] ——, *Epiphany Architecture Reference.* Adapteva Inc., 2013.