



**Iniciação Científica**  
**Relatório Parcial**  
**Período: 01/03/2024 a 10/08/2024**

**Desenvolvimento de Simulador de Processador**  
***Many-Core* com Arquitetura de Rede *Network on Chip***  
**utilizando Abordagem Interativa Voltada ao Ensino**

**Autor: Breno Dias Arantes dos Santos**  
**Orientador: Prof. Dr. Emerson Carlos Pedrino**

São Carlos, 10 de Agosto de 2024

# 1 Resumo

De acordo com o cronograma e a legenda abaixo, pode-se ter uma visão geral do que foi feito até este momento do projeto:

Tabela 1: Cronograma da Primeira Etapa do Projeto.

Ano	2024					
Atividade	Mar	Abr	Mai	Jun	Jul	Ago
I	X	X				
II			X	X	X	
III					X	X
IV						
V						

**Vermelho:** Não iniciado

**Amarelo:** Parcialmente concluído

**Azul:** Quase totalmente concluído

**Verde:** Concluído

## I. Estudo de arquiteturas Nocs (Network-on-Chips)

Foi realizado um estudo completo sobre as arquiteturas Nocs, tanto no quesito de funcionalidade quanto em questões estruturais (roteadores, links, Network Interface, topologias, formas de roteamento dos dados etc.). Além disso, foi feito um estudo sobre os algoritmos de roteamento, tanto determinísticos quanto não determinísticos para complementar o entendimento.

## II. *Estudo de Algoritmos de Mapeamento e métricas de desempenho e eficiência*

Foi realizado um estudo completo de Algoritmos de mapeamento, passando por algoritmos determinísticos [1], que no geral são usados para construção de uma população inicial que serve como base para algoritmos evolutivos/genéticos, mas que também são utilizados para mapeamento. Também foram feitos estudos aprofundados sobre algoritmos evolutivos utilizando como base [2] que apresenta uma ideia de comportamento de aves perante algumas circunstâncias, as quais podem ser traduzidas/interpretadas como um algoritmo. Além disso, utilizou-se o artigo [3] para o estudo de mapeamento genético, que apresenta as bases de teoria da evolução aplicadas em uma população inicial, visando a melhoria da distribuição das tarefas. Vale ressaltar que, ambas as abordagens utilizam um *fitness* da energia gasta na transmissão dos dados como parâmetro de melhoria, que são apresentados no artigo [2], porém pode variar de acordo com os objetivos do mapeamento, o que leva ao estudo das métricas de desempenho.

Por ora, foi realizada um breve estudo sobre as métricas de desempenho e eficiência, apenas os conceitos foram introduzidos, como consumo de energia, latência do tráfego de dados, frequência, tolerância a falhas etc, porém ainda falta o estudo aprofundado de cada métrica, como ela é medida, quais fatores levar em consideração etc. Essas métricas serão valiosas quando o simulador estiver totalmente operacional e finalizado, assim o projeto direcionará o seu foco ao estudo comparativo de algoritmos de mapeamento, o que leva ao uso das métricas nos mapas de alocação produzidos pelos algoritmos.

### III. Desenvolvimento do simulador

O desenvolvimento do simulador foi realizado em duas partes principais: a interface gráfica de comunicação com o usuário com o objetivo de ser intuitiva e de fácil usabilidade, permitindo que os usuários possam interagir de maneira eficiente e amigável com o simulador e o simulador de NoC (Network-on-Chip). O simulador de NoC constitui o núcleo do projeto, sendo responsável pela modelagem e simulação detalhada das redes em *chip* com o objetivo principal de reproduzir o comportamento dessas redes, possibilitando a avaliação de diferentes topologias, protocolos de comunicação e algoritmos de roteamento. Para validar o simulador, foram implementados cenários de teste que incluem precisão do simulador verificada comparando os resultados obtidos com dados teóricos e experimentais disponíveis na literatura [4], utilizando métricas internas como eficiência na transferência dos pacotes, *Node throughput*, latência de pacote, *delay* extra etc. A interface gráfica no momento, foi feita de maneira simplificada, apenas com alguns campos para coleta de informações necessárias para o funcionamento do simulador, porém na segunda parte do projeto, pretende-se modificá-la, tornando-se possível passar a matriz de tarefas que o usuário deseja, assim como integralizar com recursos do Python para mostrar visualmente esta matriz, apresentar uma estrutura que seja possível visualizar o funcionamento do simulador de NoC passo-a-passo na transferência dos pacotes de núcleo para núcleo pelos links.

#### IV. *Revisão e melhoramento da etapa III*

Essa etapa ainda não foi iniciada, haja vista que será realizada na segunda parte do projeto.

#### V. *Escrita do relatório, confecção de um formulário de pesquisa e artigo científico pretendido*

O relatório, especialmente, devido à primeira entrega, já está sendo escrito ao longo das implementações. Além disso, em paralelo, todas essas implementações e resultados estão sendo detalhados em formato de artigo, com o objetivo de publicar posteriormente.

## 2 Introdução

Neste projeto, escolhemos como base a dissertação de mestrado de [4], que realiza um estudo sobre arquiteturas NoC (Network on Chip) e desenvolve um simulador para testar a performance de diferentes tamanhos de *grid*. Inicialmente, replicamos as métricas fornecidas na tese para fundamentar cientificamente e legitimar o funcionamento adequado do nosso simulador. Realizamos uma leitura detalhada da dissertação para capturar as ideias principais, incluindo o modelo de roteamento utilizado, a topologia da rede, o método de escolha do *buffer* e a abordagem da simulação.

Durante o desenvolvimento do simulador, identificamos que alguns aspectos do artigo base não foram explicitamente explicados, como o algoritmo de roteamento escolhido pelo autor e a forma de criação e distribuição de pacotes durante a simulação. Esses elementos ficaram implícitos ou a cargo do leitor, exigindo de nós interpretações e decisões adicionais.

Além disso, consultamos diversos artigos sobre mapeamento de tarefas, com foco nos algoritmos evolutivos [2] e genéticos [3]. Algoritmos evolutivos são inspirados nos processos de seleção natural e evolução biológica, enquanto algoritmos genéticos se baseiam na teoria genética de Darwin, utilizando operadores de seleção, crossover e mutação. Traduzimos esses algoritmos de forma fidedigna, exceto pelas funções de exploração e intensificação, que foram omitidas pelo autor [2] e, portanto, exigiram implementações próprias.

Esses algoritmos servirão de base para, em um futuro próximo, realizar testes estatísticos utilizando métricas de avaliação de mapeamento, com o objetivo de identificar a melhor disposição de tarefas no cenário desejado. Este estudo pretende contribuir para o avanço das pesquisas em arquiteturas NoC, oferecendo uma ferramenta robusta para análise de performance e mapeamento de tarefas.

## 3 Metodologia

### 3.1 Interface Gráfica

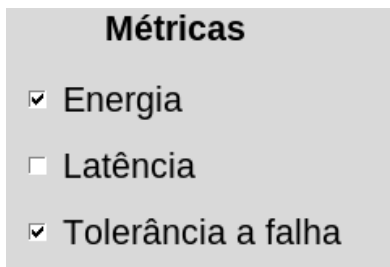
A criação de uma interface gráfica para o simulador de arquiteturas NoC (Network on Chip) representa um passo significativo na evolução do projeto. Inicialmente, o simulador foi desenvolvido de maneira simplificada, utilizando interfaces de linha de comando para executar simulações e exibir resultados (3.2). Embora funcional, essa abordagem apresenta limitações em termos de usabilidade e acessibilidade, especialmente para usuários não familiarizados com comandos de terminal ou programação.

Para superar essas limitações, foi decidido desenvolver uma interface gráfica amigável e intuitiva. A interface gráfica tem como objetivo principal facilitar a interação do usuário com o simulador, permitindo a configuração de parâmetros de simulação.

O desenvolvimento da interface gráfica foi realizado utilizando a biblioteca Tkinter [5], conhecida por sua flexibilidade e robustez na criação de interfaces gráficas interativas. O projeto foi dividido em várias etapas:

1. Definição dos Requisitos: Foram identificados os principais requisitos funcionais e não funcionais da interface. Isso incluiu a necessidade de permitir a entrada de parâmetros de simulação, a configuração de algoritmos de roteamento, o *grid* e a escolha de métricas de desempenho a serem avaliadas.

2. Implementação: A interface gráfica foi desenvolvida e integrada ao núcleo do simulador. Cada componente da interface foi cuidadosamente implementado para garantir a funcionalidade desejada. A interface permite ao usuário:
- Carregar e salvar configurações de simulação.
  - Executar simulações com um clique, sem a necessidade de comandos complexos.
3. inputs : Forma como as variáveis serão capturadas pela interface visando o funcionamento do simulador.
- Métricas : Foram escolhidos checkbox, pois futuramente, pretende-se utilizar métodos de otimização de múltiplas variáveis, não limitando a apenas 1 variável



**Métricas**

- ☒ Energia
- ☐ Latência
- ☒ Tolerância a falha

Figura 1: Seleção das Métricas

- *Grid* : Como os tamanhos podem variar a depender do uso, convém o uso de um label para o usuário escrever numericamente o tamanho que deseja.



**Grid (nxn)**

18

Figura 2: Seleção do *grid* de forma escrita



- Roteamento : Por apresentar poucas opções e somente uma pode ser escolhida para a simulação, foi utilizado o elemento da biblioteca chamado combobox, que apresenta os possíveis valores e o usuário escolhe somente 1 deles.

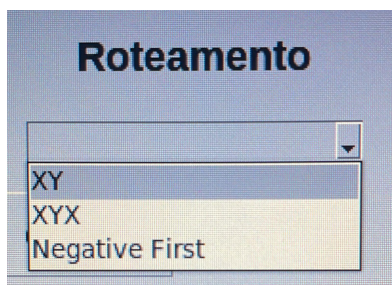


Figura 3: Seleção do roteamento

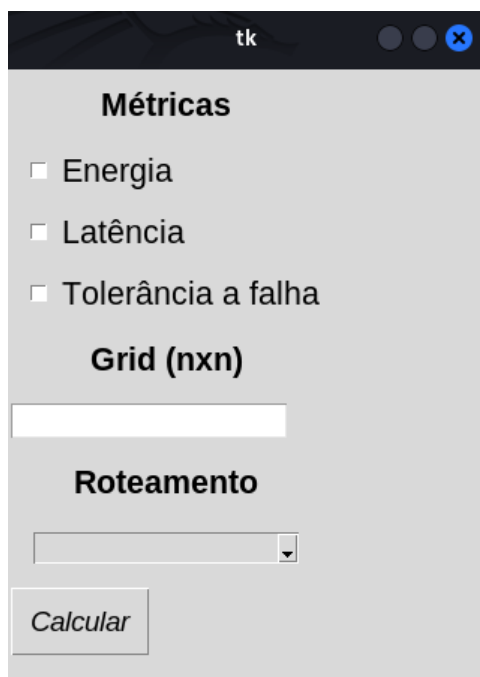


Figura 4 : Tela inicial simplificada do simulador

Inicialmente, a interface gráfica foi projetada para capturar e gerenciar os parâmetros de entrada do simulador, facilitando a configuração inicial das simulações. No entanto, apenas o *grid* é passado à instância do simulador, visto que ainda não foram implementadas as métricas para avaliação dos mapeamentos (Energia, Latência, Tolerância a falhas e outras a serem adicionadas). O roteamento, por ora, também não é passado, visto que o objetivo inicial dessa primeira fase era construir um simulador funcional seguindo o modelo do artigo base, que utiliza apenas o Roteamento XY.

Na segunda parte do projeto, pretende-se expandir essa interface para criar um ambiente mais completo que permita não apenas a execução dos testes, mas também a visualização dos resultados de maneira clara e eficiente. Pretende-se implementar um campo no qual o usuário poderá informar qual o grafo das tarefas de sua aplicação e visualizá-lo de forma gráfica utilizando a biblioteca NetworkX [6]. Essa evolução proporcionará aos usuários uma experiência mais integrada e intuitiva, permitindo que configurem, executem e analisem suas simulações dentro de um único ambiente gráfico.

## 3.2 Simulador

O simulador de NoC foi desenvolvido utilizando uma abordagem orientada a objetos (código completo nos apêndices), dividindo-o em classes específicas para organizar e gerenciar seus diferentes componentes. As três principais classes são:

### 3.2.1 Classe Pacote

A classe Pacote é responsável por representar as unidades de dados transmitidas através da NoC. Cada instância da classe Pacote contém informações essenciais, como o endereço de origem, endereço de destino, peso dados a serem transmitidos e um identificador único.

```
class Pacote():
    contador_global = 1

    def __init__(self, posicao_inicial, posicao_destino, tempo_criacao , peso = 1):
        self.id = Pacote.contador_global
        Pacote.contador_global += 1
        self.peso = peso
        self.tempo_criacao = tempo_criacao
        self.tempo_chegada = 0
        self.posicao_inicial = posicao_inicial
        self.posicao_destino = posicao_destino

    def __repr__(self):
        return f'Pacote {self.id}'
```

Figura 5: Imagem da classe Pacote

### 3.2.2 Classe Roteador

A classe Roteador é o cerne do simulador de Network on Chip (NoC), sendo a principal responsável pela recepção dos pacotes, cálculo do roteador de destino dos pacotes e seleção do *buffer* utilizando o algoritmo round-robin. Devido ao projeto estar sendo desenvolvido utilizando programação sequencial, e não paralela, o funcionamento correto da NoC requer a execução das funções de envio e recebimento de pacotes em etapas distintas. A seguir, apresentamos uma visão detalhada da implementação e funcionalidade da classe Roteador.

```
Noc.py > Roteador > enviar_pacote_RR
class Roteador:

    max_packets_buffer = 8 # variável global para definir a quantidade máxima de itens

    def __init__(self, posicao, noc):
        self.posicao = posicao # posicao na matriz de noc
        self.noc = noc # armazenando a instância de noc
        self.buffers = {
            "Norte": [],
            "Sul": [],
            "Leste": [],
            "Oeste": [],
            "Processador": [],
            "Pacotes_entregues": [], #pacotes que chegaram ao seu destino final
            "Pacotes_recebidos": [], # buffer de suporta da funcao de enviar/receber
        }
        self.buffer_atual = "Oeste"
        self.tempo = 0 # armazena o tempo
        self.total_pacotes_criados = 0
        self.total_pacotes_perdidos = 0
        self.total_pacotes_chegos = 0 # chegaram ao seu destino final
        self.total_pacotes_recebidos = 0 # quantos chegaram no roteador
```

Figura 6 : Atributos da classe Roteador

Dentre os atributos dessa classe, o dicionário “buffers” apresenta a maior importância, pois é nele que observamos os buffers, que nas implementações físicas, armazenam e entregam os pacotes que chegam no roteador.

O roteador, visto de forma macro, tem a função de receber um pacote da rede em um *buffer* de entrada, para então calcular para qual roteador o pacote irá e enviá-lo. Existem roteadores que são capazes de receber e entregar pacotes de forma paralela, aumentando significativamente a eficiência e a velocidade da rede. No entanto, nessa implementação, para seguir as diretrizes do artigo base, optamos por uma abordagem mais simples, onde a recepção e o envio de pacotes são realizados de forma sequencial.

De maneira simplificada, para o funcionamento do roteador, é necessário que ele selecione um pacote de um buffer no momento escolhido pelo algoritmo round robin. Posteriormente, por meio do destino indexado no pacote, o roteador calculará o próximo roteador com base no algoritmo de mapeamento (neste caso, o XY) utilizado em sua implementação, e então enviará o pacote ao próximo roteador. Nesta implementação, por ser sequencial, utilizou-se um *buffer* extra que armazena esses pacotes durante o ciclo de envio de toda a topologia. Isso ocorre porque, se enviássemos o pacote diretamente para o *buffer* de destino do roteador, haveria casos em que ele poderia ser enviado por mais de um roteador em um único ciclo de envio, ocasionando a perda de fidedignidade e possíveis erros nas estatísticas de desempenho do simulador. Também é necessário receber pacotes vindos de outros roteadores. Para isso, foi criada uma função que ajusta todos os pacotes que estão no buffer auxiliar e os encaminha para seus respectivos buffers. Em caso de *buffer* cheio, essa função descarta o pacote e incrementa o contador de pacotes perdidos.

```

def enviar_pacote_RR(self):
    buffer_pacote_para_enviar = self.selecionar_buffer() # seleciona qual buffer do roteador eu vou olhar no momento

    self.tempo = self.tempo + 1

    if self.buffers[buffer_pacote_para_enviar] == []:
        return
    else:
        if buffer_pacote_para_enviar == "Processador":
            self.total_pacotes_criados += 1
            pacote = self.buffers[buffer_pacote_para_enviar].pop(0) # seleciona o primeiro pacote do buffer selecionado

        roteador_destino, buffer_destino = self.seleciona_roteador_destino(pacote) #seleciona qual roteador de destino do pacote

        if roteador_destino == self:
            self.buffers["Pacotes_entregues"].append(pacote)
            pacote.tempo_chegada = self.tempo
            self.total_pacotes_chegos += 1
        else:
            roteador_destino.buffers["Pacotes_recebidos"].append((pacote,buffer_destino))

def ajustar_pacotes(self):
    tamanho = len(self.buffers["Pacotes_recebidos"])
    for i in range(tamanho):
        pacote , endereço = self.buffers["Pacotes_recebidos"].pop(0)
        qtd_dados = self.contar_dados(endereço)
        if pacote.peso <= (self.max_packets_buffer - qtd_dados):
            self.buffers[endereço].append(pacote)
            self.total_pacotes_recebidos += 1
        else:
            self.total_pacotes_perdidos += 1

```

Figura 7 : Funções de enviar e ajustar pacotes

A função `selecionar_buffer()` atua como árbitro nessa implementação, o qual utiliza a política Round Robin para garantir o acesso de todos os *buffers* de maneira equitativa, evitando o que chamamos de starvation, que é quando uma tarefa, pacote ou *buffer* é continuamente preterido em favor de outras, resultando em um atraso indefinido na sua execução ou entrega.

```
def selecionar_buffer(self):
    # pelo algoritmo round robin, inicialmente
    if self.buffer_atual == "Norte":
        self.buffer_atual = "Leste"
        return "Leste"

    if self.buffer_atual == "Leste":
        self.buffer_atual = "Sul"
        return "Sul"

    if self.buffer_atual == "Sul":
        self.buffer_atual = "Oeste"
        return "Oeste"

    if self.buffer_atual == "Oeste":
        self.buffer_atual = "Processador"
        return "Processador"

    if self.buffer_atual == "Processador":
        self.buffer_atual = "Norte"
        return "Norte"
```

Figura 8 : função selecionar buffer

A função `seleciona_rotador_destino()` consiste na seleção do roteador ao qual o pacote que está sendo analisado será transferido. Existem diversos algoritmos que são utilizados para calcular este destino. Para seguir o artigo base, esta função implementa o método XY, que consiste em um algoritmo de roteamento determinístico, porém na segunda parte do projeto será implementado o XYX [9] e Negative First [10]. Este método determina o caminho que o pacote deve seguir com base nas coordenadas X e Y. De início, a função verifica se o destino do pacote é o próprio roteador por meio das suas coordenadas de destino. Caso contrário, o pacote é roteado na direção X até que a coordenada X do roteador de destino seja alcançada. Em seguida, o pacote é roteado na direção Y até que a coordenada Y do roteador de destino seja alcançada. Este método é simples, porém

eficiente, evitando problemas como *deadlocks* (condição em que dois ou mais processos ficam bloqueados esperando recursos que estão sendo utilizados), congestionamentos, entre outros.

```
def seleciona_rotador_destino(self, pacote):
    x_destino, y_destino = pacote.posicao_destino #posicao destino
    x_atual, y_atual = self.posicao #posicao atual do roteador
    delta_x = x_destino - x_atual #deslocamento em x
    delta_y = y_destino - y_atual # deslocamento em y
    if delta_x == 0 and delta_y == 0:
        return self, "Nada"
    if delta_y > 0:
        return self.noc.matriz_rotadores[x_atual][y_atual+1], "Oeste" #.buffers["Oeste"]
    if delta_y < 0:
        return self.noc.matriz_rotadores[x_atual][y_atual-1], "Leste" #.buffers["Leste"]
    if delta_x > 0:
        return self.noc.matriz_rotadores[x_atual+1][y_atual], "Norte" #.buffers["Norte"]
    if delta_x < 0:
        return self.noc.matriz_rotadores[x_atual-1][y_atual], "Sul" #.buffers["Sul"]
```

Figura 9: Função seleciona roteador utilizando roteamento XY

### 3.2.2 Classe Noc

A classe NoC representa a rede como um todo, englobando uma matriz de instâncias da classe Roteador. Esta matriz define a topologia da rede (neste caso a Mesh) e a interconexão entre os roteadores. A classe NoC gerencia a criação da rede, a configuração dos roteadores, a criação de pacotes nos roteadores e a simulação do tráfego de pacotes.



```

class Noc:
    def __init__(self, dimensao):
        self.dimensao = dimensao
        self.matriz_rotadores = self.criar_matriz_rotadores(dimensao)
        self.total_pacotes = 0
        self.total_pacotes_recebidos = [0]
        self.total_pacotes_chegos = [0]
        self.total_pacotes_perdidos = [0]
        self.chegos = 0

    def criar_matriz_rotadores(self, dimensao):
        tamanho = dimensao

        matriz = [['' for _ in range(tamanho)] for _ in range(tamanho)] # matriz vazia

        for i in range(tamanho):
            for j in range(tamanho):
                matriz[i][j] = Rotador((i,j), self) # passando a instância de Noc para Rotador

        return matriz

    def alocar_pacotes(self, pacote):
        x,y = pacote.posicao_inicial
        self.total_pacotes += 1
        self.matriz_rotadores[x][y].buffers["Processador"].append(pacote)

```

Figura 10: Classe Noc

### 3.3 Algoritmos de mapeamento

Na literatura, todos os simuladores mencionam a presença de algoritmos de mapeamento, uma vez que sua função principal é simular sistemas de alto desempenho. Para alcançar esse objetivo, é crucial distribuir as tarefas de um processo de forma eficiente, garantindo a otimização de métricas de desempenho específicas para o problema em questão. Neste projeto de IC, além de focar no funcionamento do simulador em si, há também um interesse na busca pelo melhor algoritmo para uma métrica específica relacionada ao problema do usuário. Após a conclusão da construção do cerne do simulador, o foco foi direcionado para os algoritmos.

Inicialmente, foram selecionados algoritmos simples, os mapeamentos determinísticos apresentados em [1], que no artigo são chamados de Engineered

Mapping. Posteriormente, foram adicionados alguns algoritmos mais complexos, como algoritmos genéticos e evolutivos [2,3], que demonstraram melhorias significativas em comparação com uma alocação aleatória simples. Todas as implementações dos pseudocódigos apresentados a seguir estão disponíveis nos apêndices.

### 3.3.1 Engineered Mapping

De forma simplificada, esses algoritmos dividem o processo em dois passos. Primeiro, eles alocam as tarefas em um *array* com o mesmo tamanho do *grid* do NoC, existindo duas possibilidades: *clustered* ou *distributed*. Basicamente, na alocação *clustered*, as tarefas são colocadas uma após a outra, sem espaços entre elas. Já na alocação *distributed*, há espaços entre algumas tarefas, dependendo da quantidade de tarefas no problema. Posteriormente, há mais dois parâmetros que devem ser escolhidos: o formato de orientação que a matriz deve ser escaneada, seja horizontal ou diagonal, e a forma de transição, que pode ser *snake* ou *raster*. Com a junção dessas variáveis, é possível obter um mapeamento determinístico, conforme mostrado nas imagens abaixo retiradas do artigo.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	i	i	i	i

(b) 12 Clustered processes

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
P1	P2	P3	i	P4	P5	P6	i	P7	P8	P9	i	P10	P11	P12	i

(c) 12 Distributed Processes

Figura 11: Tipos de alocações dos processos

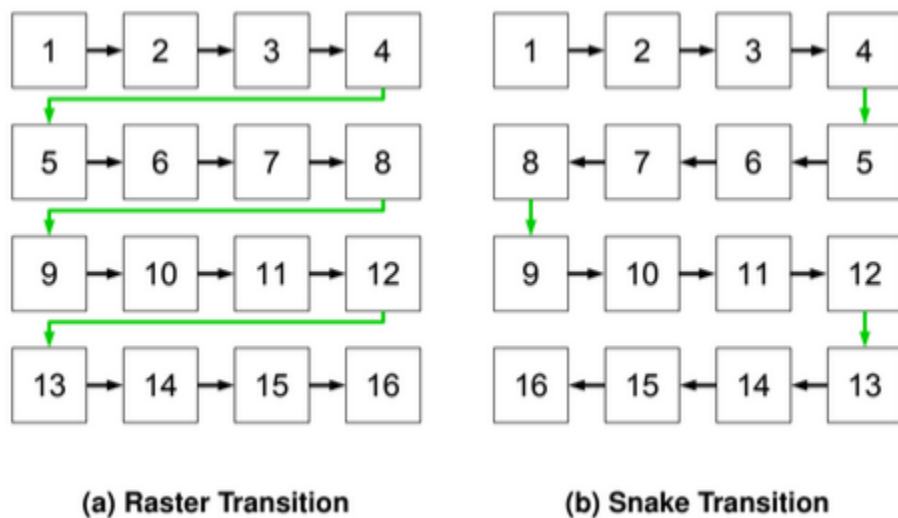


Figura 12: Tipos de orientação

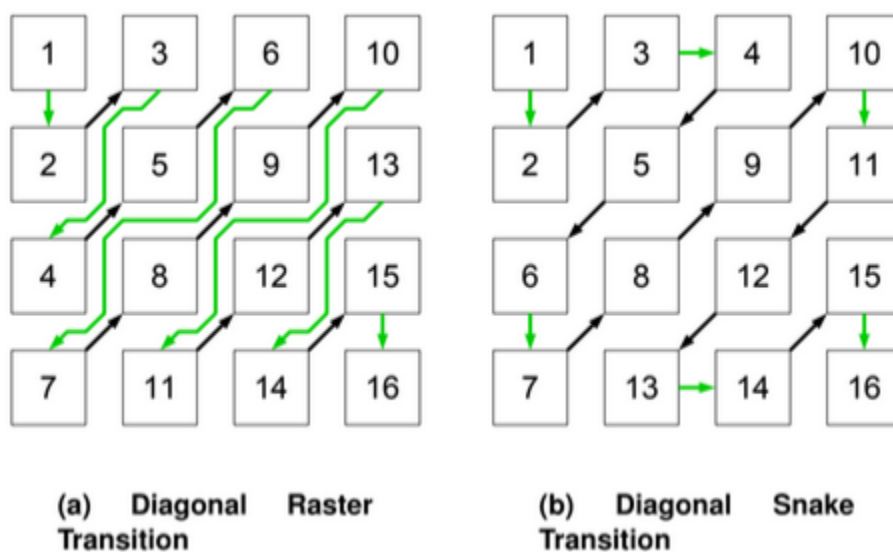


Figura 13: Tipos transição

Durante esta etapa, foram escolhidos apenas 3 pares de combinação de orientação/transição, que multiplicado ao formato alocação dos processos (2), resultam em 6 tipos de algoritmos de mapeamento determinísticos distintos.

### 3.3.2 População inicial

Nos algoritmos genético e dos condores implementados durante esta etapa do projeto, ambos necessitavam de uma maneira de gerar uma população inicial, contudo, apenas o artigo [2] apresenta uma maneira (*Cluster Based*), assim, para reutilizar código e facilitar a implementação, no algoritmo genético foi utilizado a mesma geração base, a qual o pseudocódigo está apresentada abaixo conforme especificado em [2].

#### Cluster Based initial population

##### Passo-1:

- Selecione uma única tarefa da ATG como tarefa principal.
- Identifique a conexão direta entre a tarefa principal e cada tarefa na ATG.

##### Passo-2:

- Agrupe as tarefas diretamente ligadas à tarefa principal em um único cluster.

##### Passo-3:

- Calcule a exigência total de comunicação do cluster usando a Eq. abaixo.

$$CW_i = \sum_{j=0}^m B_{ei,ej}$$

#### **Passo-4:**

- Classifique os clusters de acordo com a densidade de comunicação.

#### **Passo-5:**

- Selecione o *cluster* mais bem classificado e mapeie a tarefa principal do cluster selecionado em um tile escolhido aleatoriamente da arquitetura NoC.

#### **Passo-6:**

- Mapeie as tarefas restantes (tarefas filhas) do cluster selecionado nas proximidades da tarefa principal.

#### **Passo-7:**

- Selecione o cluster restante que compartilha um vizinho com o(s) *cluster(s)* já mapeado(s).
  - Se mais de um cluster compartilhar um vizinho, selecione o cluster com a maior densidade de comunicação.

#### **Passo-8:**

- Mapeie a(s) tarefa(s) do cluster selecionado, que tem uma ligação direta e maior exigência de comunicação, próximo às tarefas mapeadas.

#### **Passo-9:**

- Repita os Passos 7 e 8 até que todas as tarefas da ATG estejam mapeadas na plataforma NoC, NAG.

O código completo na linguagem python que foi utilizado está na parte dos apêndices.

### 3.3.3 Algoritmo Genético

Algoritmos Genéticos (AGs) são técnicas de otimização inspiradas nos processos biológicos de evolução e seleção natural, inseridos na área mais ampla dos algoritmos evolutivos. O objetivo dos Algoritmos Genéticos é encontrar soluções eficientes para problemas complexos, frequentemente em domínios onde abordagens tradicionais podem ser desafiadoras. No processo de Algoritmos Genéticos, as soluções candidatas são representadas como cromossomos, muitas vezes na forma de árvores de expressão. Esses cromossomos são então submetidos a operadores genéticos, como seleção, recombinação e mutação, para gerar novas gerações de programas. A aptidão de cada programa é avaliada com base em sua capacidade de resolver o problema em questão, determinada por uma função de *fitness*. Os programas mais aptos são selecionados para reprodução, resultando em uma população que evolui ao longo do tempo em busca de uma solução ou programa final capaz de transformar um input em um output desejado.

Neste caso, foi implementado o algoritmo do artigo [3] que utiliza a métrica de energia como função fitness. O pseudocódigo o qual o algoritmo foi traduzido e adaptado ao problema está apresentado abaixo conforme [3].

### Gerar uma população inicial:

- Gere uma população inicial de  $n$  cromossomos, que consiste em muitos posicionamentos de núcleos IP gerados aleatoriamente. O comprimento de cada cromossomo é igual ao número de vértices em um grafo de comunicação de núcleos (CCG), e o cromossomo é codificado em cadeias de inteiros. Cada gene (vértice no CCG) no cromossomo contém um inteiro que indica um vértice escolhido aleatoriamente no NAG, e os vértices não podem se sobrepor.

### Avaliar a aptidão de cada cromossomo na população:

- Para minimizar o consumo de energia da comunicação, a função de aptidão do nosso algoritmo é dada pela Eq. abaixo.

$$Cost = \sum_{i,j} [B_{ei,ej} * M_{dist}]$$

### Criar uma nova população:

- Aplique três operadores semelhantes aos operadores de seleção natural - seleção, cruzamento e mutação. A seleção escolhe dois cromossomos pais da população de acordo com sua aptidão (quanto melhor a aptidão, maior a chance de serem selecionados). Devemos prestar atenção ao fato de que o cruzamento e a mutação comuns podem criar respostas ilegais para o problema de mapeamento. Para o nosso problema de posicionamento, mapeamos cada núcleo IP como um número único não negativo como método de codificação. Ao realizar as operações comuns de cruzamento e mutação, os cromossomos dos descendentes são gerados a

partir do acasalamento dos cromossomos dos pais. Nesta fase, há possibilidades de que alguns núcleos IP possam ser atribuídos aos mesmos nós de recursos. Portanto, um cruzamento por ordem melhorado é usado para evitar este problema. Desta forma, dois pontos de cruzamento são selecionados aleatoriamente, os genes entre os dois pontos de cruzamento do primeiro pai são copiados para o segundo descendente, então o outro pai é escaneado e, se o número ainda não estiver no descendente, ele é adicionado ao descendente. Para a mutação, trocamos dois membros no cromossomo que são selecionados aleatoriamente.

**Repetir os passos (2) e (3):**

- Repita os passos (2) e (3) até que a menor distância média de saltos não tenha mudado por um número suficiente de iterações, então pare e reporte o melhor mapeamento como o posicionamento otimizado gerado de núcleos IP na NoC.

### **3.3.4 Andean condor algorithm**

Este algoritmo foi produzido por [2] e visa promover uma nova abordagem eficiente e de custo-benefício. O algoritmo utiliza os padrões de migração e o comportamento de voo dos condores-andinos para busca de alimento em diferentes épocas do ano analisados nesta espécie de ave. Foi observado que a distância que o condor-andino percorre em busca de alimento varia conforme a estação do ano. Esse comportamento resulta em uma taxa de exploração e intensificação baseada no valor médio de aptidão. Se esse valor for alto, o número de explorações diminui e o número de intensificações



aumenta. Portanto, a qualidade média da aptidão eventualmente controla o equilíbrio fino entre a taxa de exploração e intensificação. Em seguida, estará disponível o pseudocódigo que detalha a implementação desse algoritmo.

---

**Algorithm 1:** The ACA meta-heuristic.

---

**Input:**  $AC, N_c, DP, PC;$ 
**Output:**  $AC_b$  (the best mapping solution);

Calculate the fitness using Eq. (4) and sort the initial population from best to worst;

 Calculate average fitness  $AF$  using Eq. (9);

 Calculate quantities  $QE$  and  $QI$  using value of  $DP$ ;

 Update the states of condors in population for exploration and intensification using values  $QE$  and  $QI$ ;

**while stop criteria do**
**for each solution**  $i = 1 \dots N_c$  **do**
**if status of**  $AC_i$  **is equal to true then**

 | Exploration movement for  $AC_i$ ;

**else**

 | Intensification movement for  $AC_i$ ;

Sort the new population from best to worst;

Calculate and update the states of each condor using algo 2;

 Update best  $AC_b$ ;

 Get the best solution  $AC_b$ ;

---

Figura 14: Algoritmo 1 do Andean Condor Algorithm (ACA)

---

**Algorithm 2:** Calculate and Update States.

---

**Input:**  $AF, DP, PC;$ 
**Output:**  $DP$ ;

 Calculate the average fitness of newly generated population,  $AF_{new}$  using Eq. (9);

**if**  $AF_{new} \geq AF$  **then**

 |  $DP = DP + PC$ ;

**if**  $DP > 1$  **then**

 |  $DP = 1$ ;

**else**

 |  $DP = DP - PC$ ;

**if**  $DP < 0$  **then**

 |  $DP = 0$ ;

 $QE = N_c \times DP$ ;

 $QI = N_c - QE$ ;

**for**  $(i = 1; i \leq QI; i++)$  **do**

 |  $AC_i \leftarrow$  intensification status;

**for**  $(i = QI; i \leq QI + QE; i++)$  **do**

 |  $AC_i \leftarrow$  exploration status;

---

Figura 15: Algoritmo 2 Calculate and Update States

## 4 Experimentos

### 4.1 Simulador

Com o código do simulador pronto, realizamos uma série de testes para avaliar seu desempenho e precisão. Os experimentos foram conduzidos com o objetivo de validar as funcionalidades do simulador e verificar sua eficácia em replicar os padrões de comportamento observados nos NoCs. Para garantir a robustez dos resultados, comparamos as métricas obtidas com as apresentadas no artigo base. Neste artigo, foram utilizadas diversas métricas referentes ao desempenho do comportamento de um NoC (latência, *throughput* dos nós, eficiência, perda de pacotes etc.) frente a dois tipos de grid, 4x4 e 8x8. Foram introduzidos ao NoC, para a simulação, uma quantidade total de pacotes igual a  $\text{dimensão} \times \text{dimensão} \times \text{tempo\_de\_simulação}$ , distribuídos igualmente ao longo do tempo. Além disso, com o intuito de gerar gráficos com formato próximo ao do artigo, foi realizada uma regressão polinomial nos dados obtidos. Como não há disponibilidade de várias informações utilizadas pelo autor, como por exemplo a maneira com os quais os pacotes são alocados, o que pode ocasionar diversas distorções nos gráficos, foram feitos testes de forma que tentassem alcançar valores e métodos de forma empírica, tentando reproduzir os efeitos do artigo. A seguir, todas as métricas foram extraídas e traduzidas a partir do artigo [4] e os gráficos foram feitos utilizando [7].

#### 4.1.1 Total Network Throughput

De maneira simplificada, o artigo define esta métrica como sendo, quantos pacotes são recebidos pelo destino na rede durante o tempo de simulação. Ele é utilizado para medir o desempenho de toda a rede NoC.

Total *throughput*=(total arrived packets)/(simulation time)

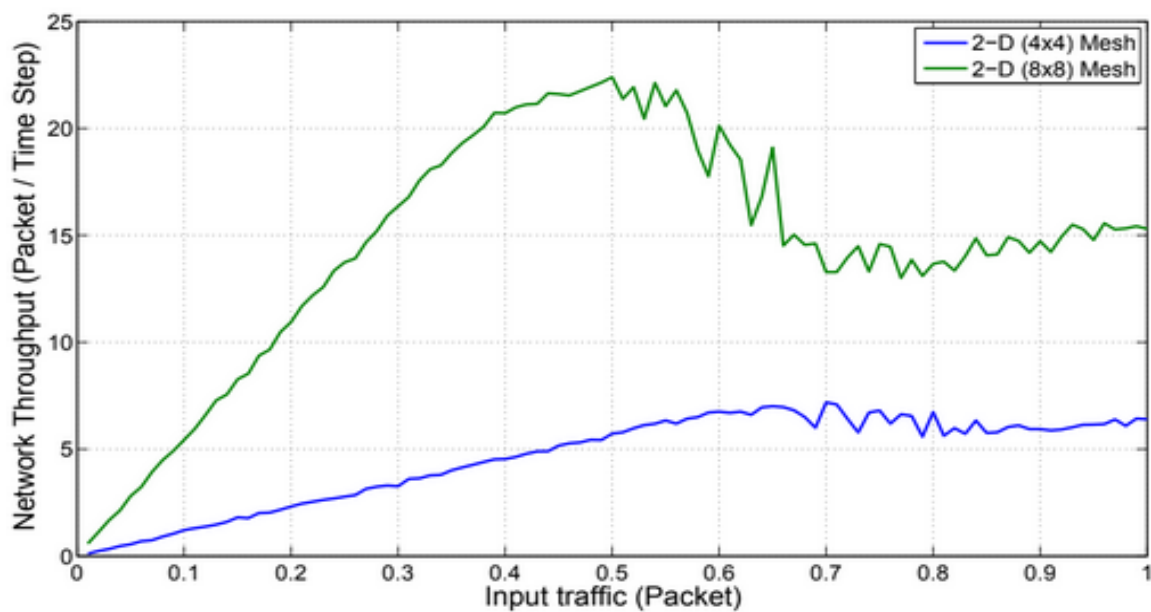


Figura 16: *Total throughput* retirado de [4]

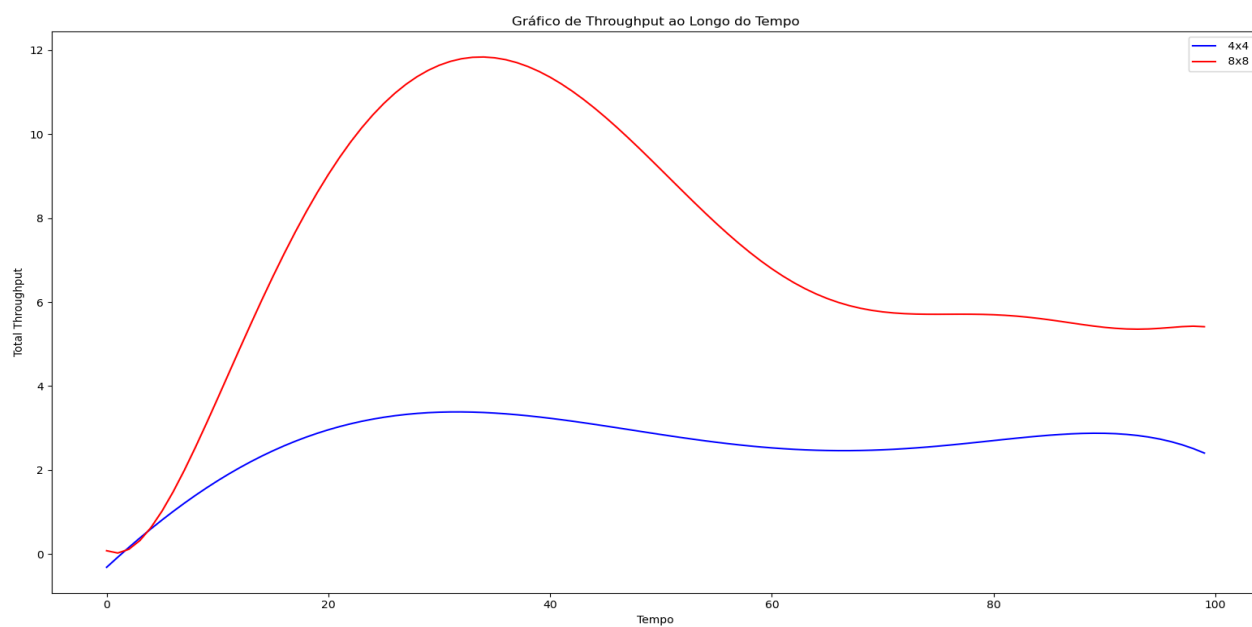


Figura 17: *Total throughput* obtido pela simulação

Como é possível perceber, ambas as simulações apresentam o mesmo comportamento. Nos *grids* 8x8, em ambos os casos, o crescimento é linear até certo ponto. Depois, devido ao acúmulo de pacotes trafegando na rede, os buffers dos roteadores começam a se encher, fazendo com que os pacotes não cheguem ao seu destino, o que causa uma queda nos valores. No *grid* 4x4, esse comportamento não ocorre ao atingir o limite dos buffers, a medida de *throughput* total dos nós se estabiliza.

#### 4.1.2 Node Throughput

Segundo [4], o *throughput* de nós pode ser obtido dividindo o *throughput* total pelo número total de nós presentes na rede NoC.

$$\text{Node Throughput} = (\text{pacotes totais recebidos}) / (\text{Linhas} \times \text{Colunas})$$

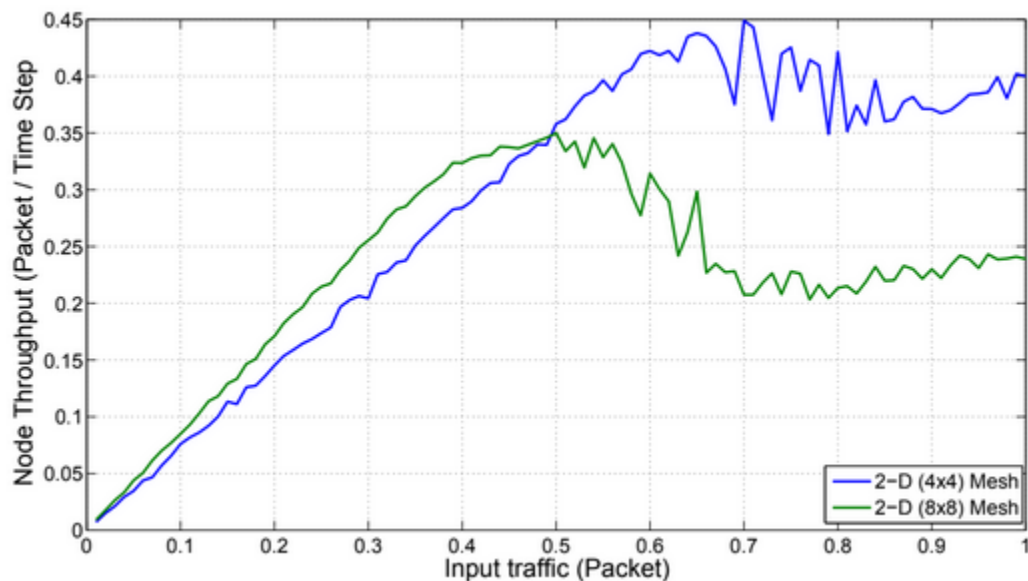


Figura 18: *Node Throughput* retirado de [4]

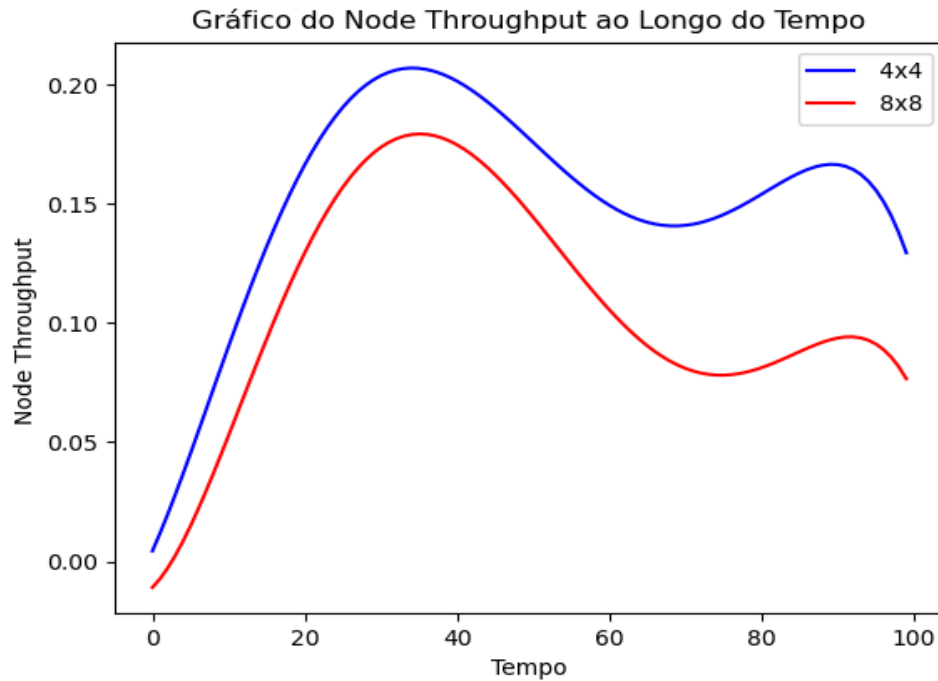


Figura 19: *Node Throughput* obtido pela simulação

Nesta métrica é possível perceber o movimento contínuo de subida de ambos os *grids* e posteriormente um decaimento, possivelmente devido ao mesmo fator do total *throughput*, que é a falta de espaço nos buffers, sendo que o *grid* 4x4 mantém uma média de *node throughput* maior que o 8x8

#### 4.1.3 Latência Média

A latência de pacote é definida, para cada pacote, como o tempo entre a geração do pacote e o tempo real de chegada.

$$\text{Latência de pacote} = (\text{tempo real de chegada} - \text{tempo de geração do pacote})$$

$$\text{Latência Média} = (\text{Soma de todas as latencias} / \text{quantidade de pacotes})$$

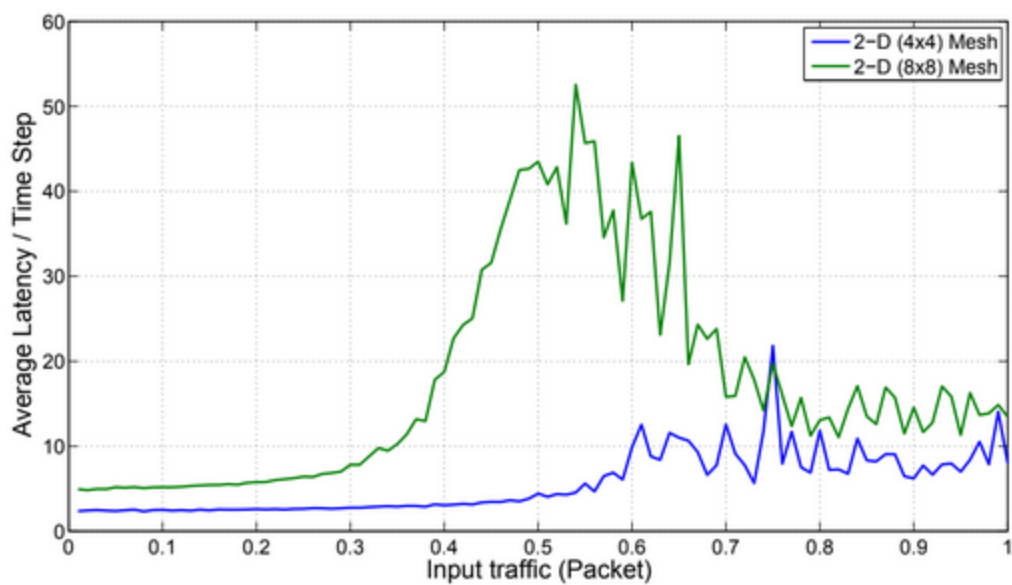


Figura 20: Latência Média retirado de [4]

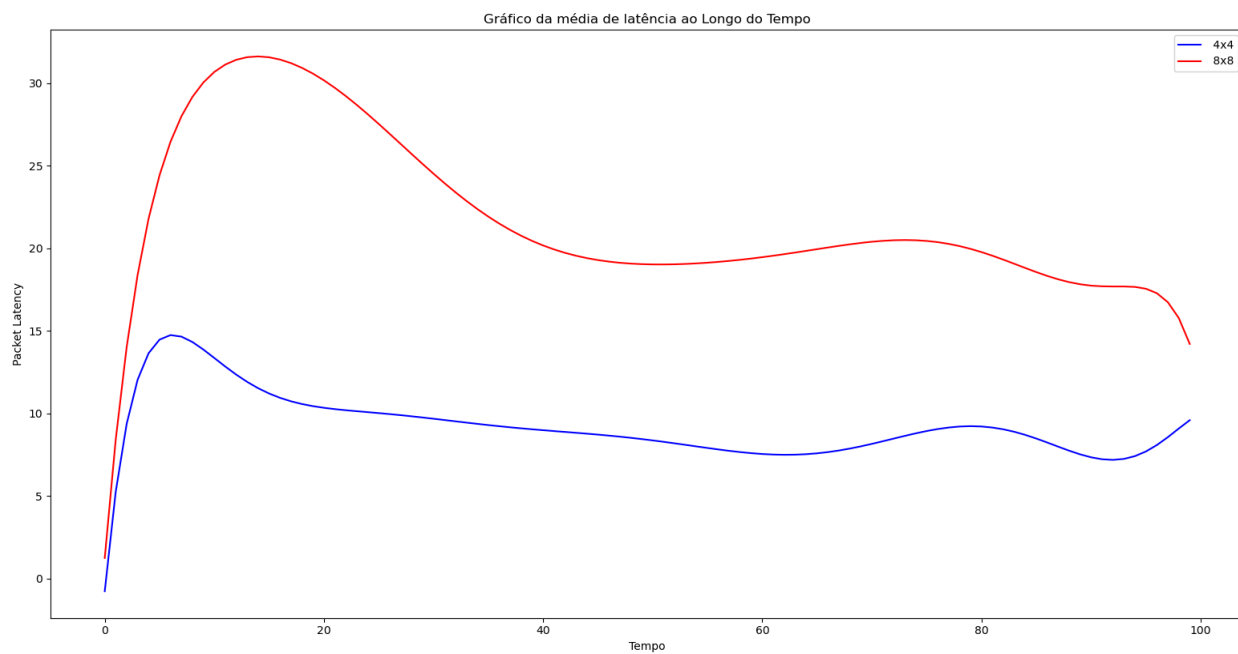


Figura 21: Latência Média obtido pela simulação

Neste caso, também é possível observar o pico de latência que o *grid* 8x8 apresenta e o leve pico do 4x4, posteriormente as duas medidas apresentam uma média estável. Provavelmente, no início da simulação os *buffers* estão vazios por isso a média acaba sendo menor, depois de um tempo os *buffers* se enchem e a média dos pacotes ficam constantes.

#### 4.1.4 Atraso extra médio

O atraso extra pode ser calculado subtraindo o tempo mínimo de chegada do tempo real de chegada.

Atraso extra = (tempo real de chegada do pacote – tempo mínimo de chegada do pacote)

Tempo mínimo = Distância manhattan =  $|x_i + x_j| + |y_i + y_j|$

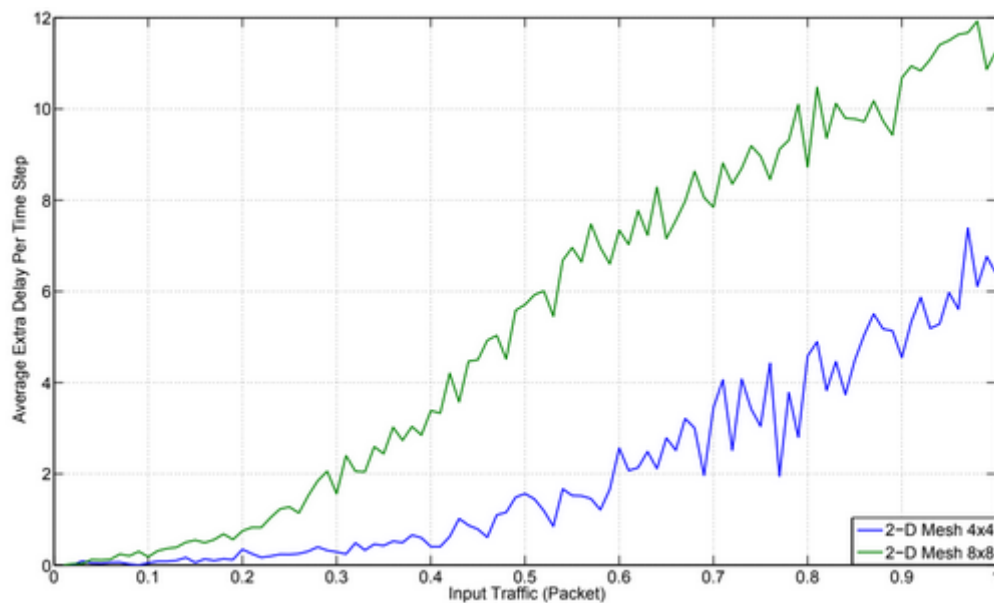


Figura 22: Atraso extra retirado de [4]

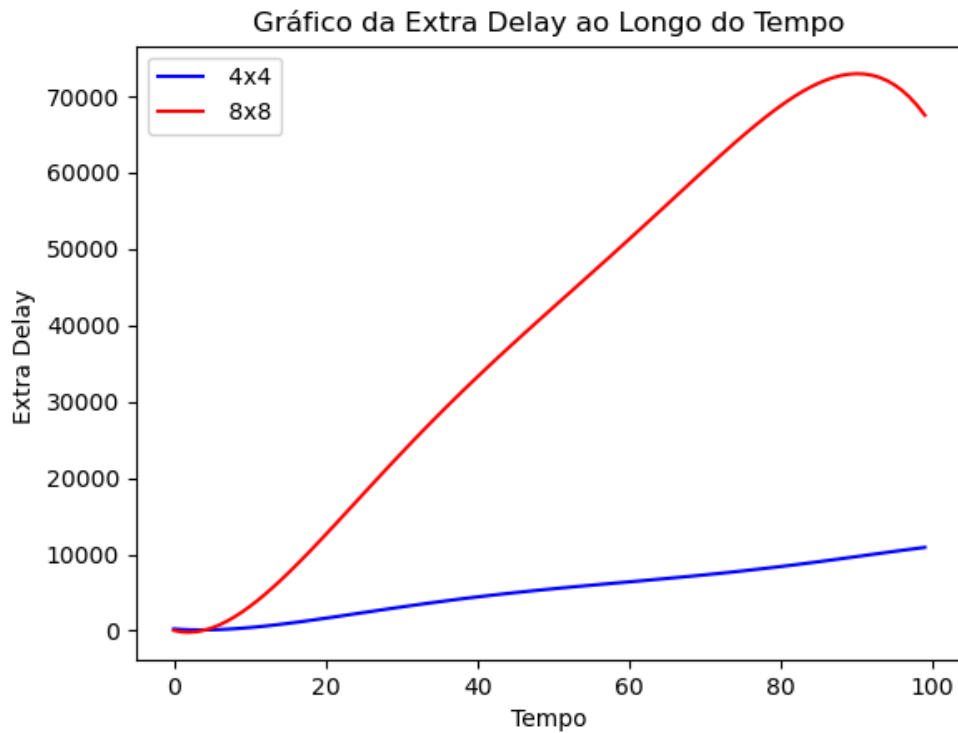


Figura 23: Atraso extra obtido pela simulação

Neste cenário fica nítido a semelhança, onde ambos os gráficos apresentam crescimento constantes nas suas métricas, com o *grid* 8x8 apresentando um maior valor absoluto no atraso extra dos pacotes, muito provavelmente devido ao tamanho do *grid* e a quantidade de roteadores que os pacotes tenham que passar até chegar em seu destino.

#### 4.1.5 Eficiência

A eficiência é a razão entre o total de pacotes e os pacotes recebidos, expressa em porcentagem.

$$\text{Eficiência} = (\text{total de pacotes recebidos} \times 100) / (\text{total de pacotes})$$



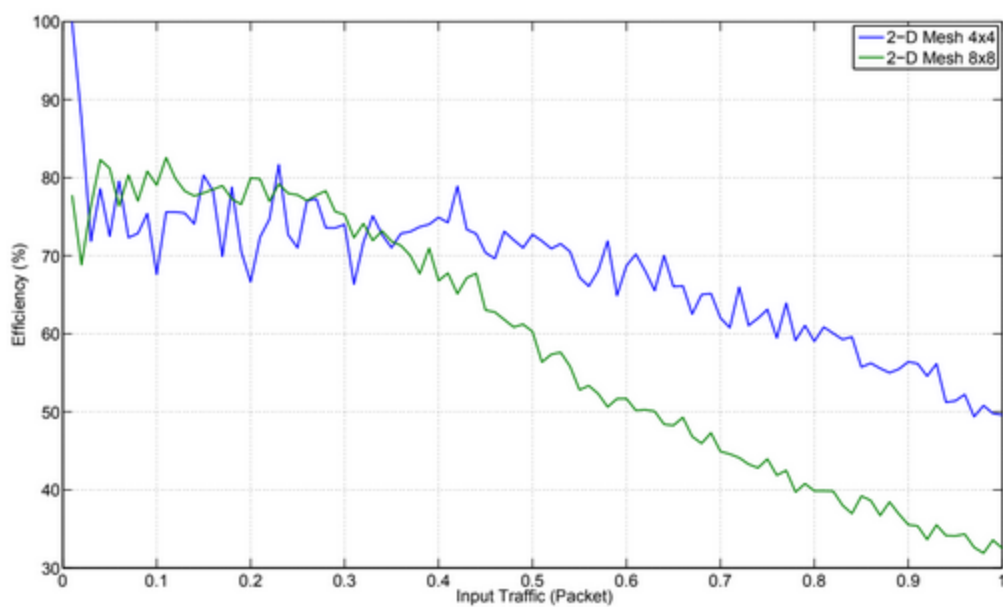


Figura 24: Eficiência retirado de [4]

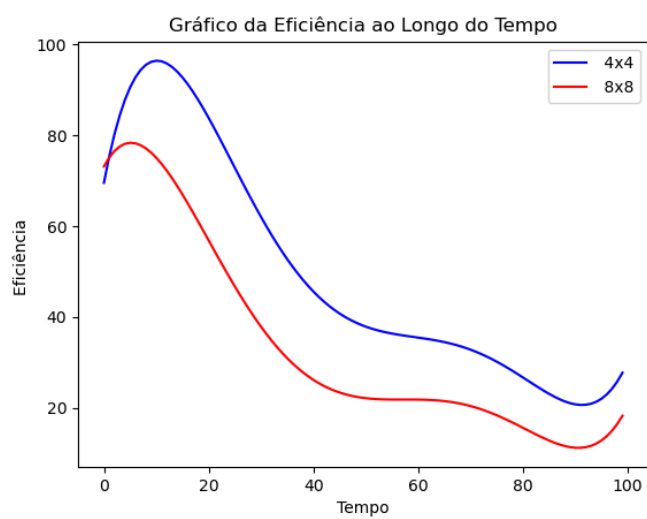


Figura 25: Eficiência obtido pela simulação

Neste cenário, inicialmente com os buffers vazios e poucos pacotes na rede, a maioria dos chegam a seu destino final com um tempo menor, conforme a quantidade de pacotes na rede vai aumentando, a eficiência cai e neste caso nesse quesito o *grid* 4x4 apresenta um melhor desempenho.

#### 4.1.6 Perda de Pacote

Se o *buffer* do núcleo estiver cheio e não houver espaço para adicionar o pacote na fila do *buffer*, este pacote é considerado perdido.

$$\text{Perda de Pacotes} = (\text{contagem de pacotes perdidos} \times 100) / (\text{total de pacotes})$$

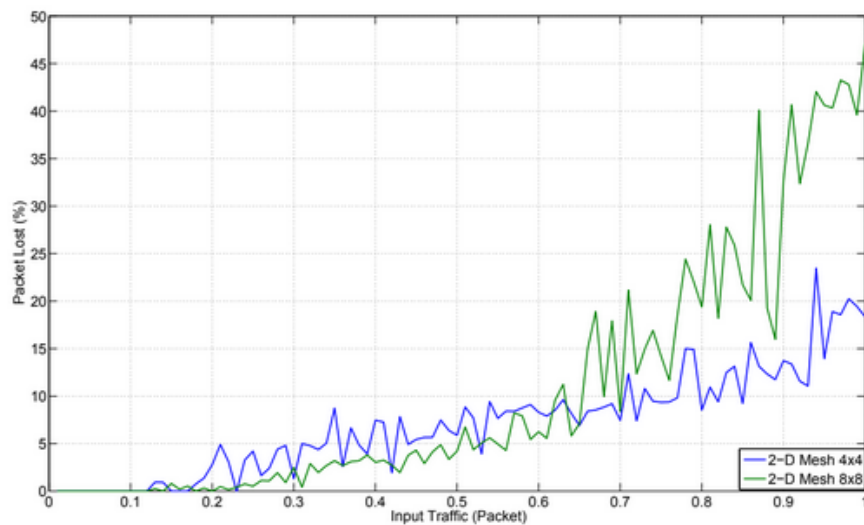


Figura 26: Perda de Pacotes retirado de [4]

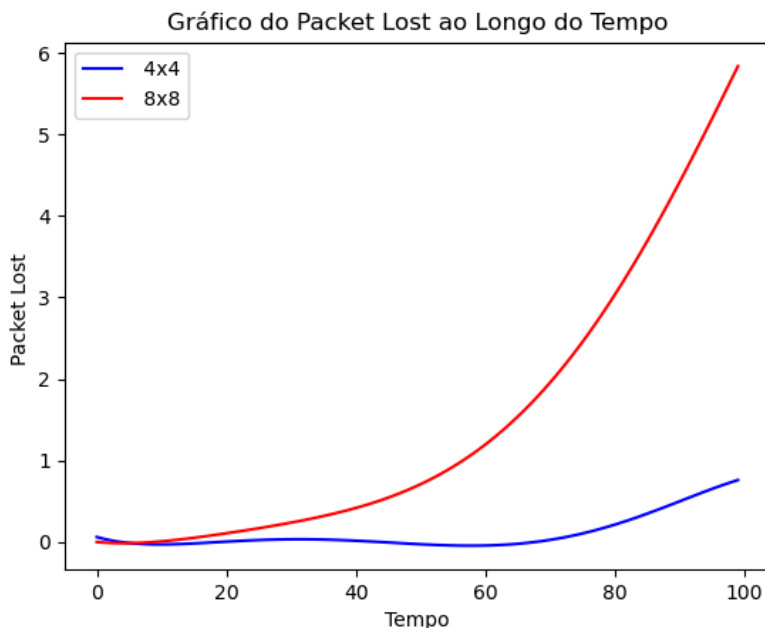


Figura 27: Perda de Pacotes obtido pela simulação

Inicialmente, ambos os *grid* apresentam uma perda de pacotes semelhante, porém com um *grid* maior, é esperado que em números absolutos este apresenta maior perda de pacotes, o que é mostrado em ambos os gráficos.

## 4.2 Algoritmos de mapeamento

Durante esta primeira parte da IC, não foram realizados testes comparativos entre os algoritmos. Nesta etapa, foram feitos apenas os testes necessários para verificar se o funcionamento dos algoritmos estava de acordo com o previsto. Ou seja, nos algoritmos determinísticos, verificou-se se o mapeamento seguia a premissa de seus parâmetros. Já para os algoritmos não determinísticos, por possuírem uma métrica intrínseca para seu funcionamento, foram monitoradas as evoluções nesses valores e se houve uma

evolução dentro das gerações, considerando um número pré-definido de gerações sem evolução.

#### 4.2.1 Algoritmos determinísticos

Foram utilizados um processo com 16 tarefas quaisquer e um grid de 4x4 para verificar se o algoritmo estava de correto funcionamento. Para isso, foram utilizadas as próprias imagens ilustrativas do artigo como base de comparação do resultado obtido.

##### 4.2.1.1 Clustered Horizontally e Raster

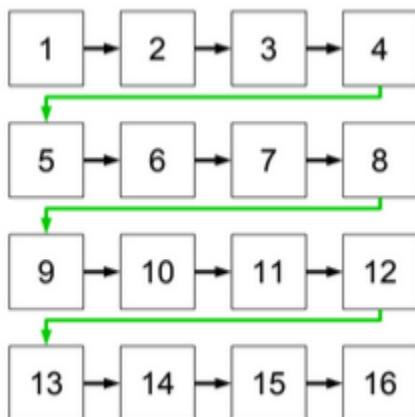


Figura 28: Exemplo retirado de [1]

```
[1, 2, 3, 4]
[5, 6, 7, 8]
[9, 10, 11, 12]
[13, 14, 15, 16]
```

Figura 29: Resultado obtido

#### 4.2.1.2 Clustered Horizontally e Snake

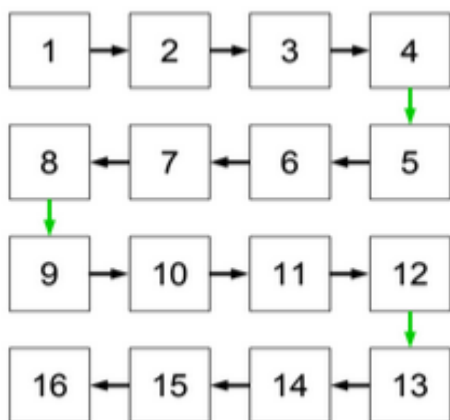


Figura 30: Exemplo retirado de [1]

```
[1, 2, 3, 4]
[8, 7, 6, 5]
[9, 10, 11, 12]
[16, 15, 14, 13]
```

Figura 31: Resultado obtido

#### 4.2.1.3 Clustered Diagonally e Raster

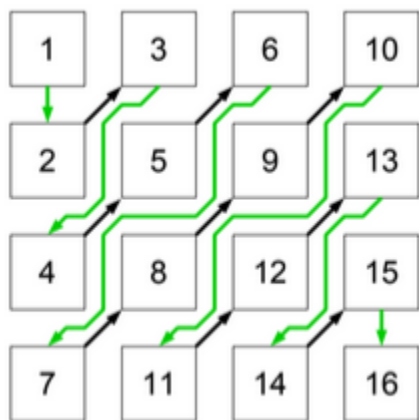


Figura 32: Exemplo retirado de [1]

```
[1, 3, 6, 10]
[2, 5, 9, 13]
[4, 8, 12, 15]
[7, 11, 14, 16]
```

Figura 33: Resultado obtido

#### 4.2.1.4 Demais mapeamentos

Para os próximos mapeamentos, a única diferença em relação aos apresentados anteriormente é o formato de distribuição das tarefas, que neste caso é distribuído. No artigo, não há ilustrações de como essas distribuições ficariam, nem um pseudocódigo explicando exatamente como ocorrem os padrões de distribuição, ficando a cargo do leitor interpretar e traduzir. Assim, as seguintes imagens se referem ao resultado obtido da tradução proposta pelo autor deste projeto, utilizando um processo com 13 tarefas e um noc 4x4.

```
[1, 2, 3, 4]
[None, 5, 6, 7]
[8, None, 9, 10]
[11, 12, None, 13]
```

Figura 34: mapeamento utilizando padrão *distributed,horizontally e raster*

```
[1, 2, 3, 4]
[7, 6, 5, None]
[8, None, 9, 10]
[13, None, 12, 11]
```

Figura 35 : mapeamento utilizando padrão *distributed,horizontally e snake*

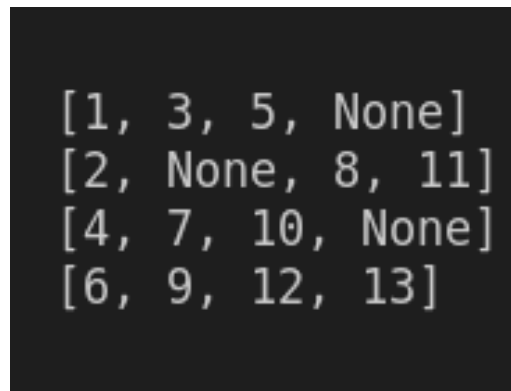


Figura 36: mapeamento utilizando padrão *distributed, diagonally e raster*

#### 4.2.2 Algoritmo Genético

Um algoritmo genético em funcionamento deve, assim como na vida real, apresentar melhoras em algum aspecto durante o passar das evoluções. Assim, foram monitoradas as evoluções conforme o passar das gerações, avaliando qual foi o *fitness* alcançado e, depois de várias gerações sem melhoras, qual foi o mapa de alocação ótima encontrado por este algoritmo. Alguns aspectos que podem influenciar na melhora da precisão do algoritmo poderão ser estudados adiante, como, por exemplo, mudar a geração das populações iniciais, ou ainda a forma com que ocorre a seleção dos pais que gerarão os filhos. Por ora, foi analisado apenas o funcionamento correto, conforme apresentado abaixo.

Para este mapeamento, foi utilizado um processo que contém 16 tarefas geradas aleatoriamente e um NoC de 4x4. Vale ressaltar que este mesmo padrão será também utilizado para a avaliação do algoritmo dos condores. Embora não esteja sendo feita

nenhuma comparação qualitativa entre os algoritmos, espera-se que ambos apresentem valores de *fitness* (ambos estão utilizando energia como *fitness*) próximos um do outro.

```
adj_matriz = [
    [0, 8, 5, 7, 6, 4, 8, 3, 9, 2, 1, 8, 9, 6, 7, 10],
    [9, 0, 3, 2, 5, 7, 6, 10, 1, 9, 4, 5, 8, 9, 1, 2],
    [7, 2, 0, 9, 2, 10, 4, 5, 6, 8, 7, 3, 9, 7, 6, 5],
    [4, 8, 6, 0, 3, 1, 5, 7, 2, 6, 10, 4, 7, 10, 9, 8],
    [10, 5, 8, 1, 0, 9, 7, 6, 3, 1, 5, 2, 8, 4, 6, 10],
    [2, 7, 1, 9, 6, 0, 8, 5, 10, 3, 7, 6, 4, 1, 9, 7],
    [8, 6, 10, 5, 2, 9, 0, 1, 7, 4, 6, 10, 9, 5, 7, 1],
    [5, 10, 4, 2, 9, 8, 1, 0, 6, 10, 3, 9, 5, 1, 8, 6],
    [6, 1, 7, 8, 10, 7, 6, 9, 0, 5, 9, 2, 10, 3, 5, 2],
    [3, 9, 10, 7, 1, 4, 10, 6, 5, 0, 8, 1, 6, 10, 2, 7],
    [9, 4, 7, 10, 5, 7, 6, 3, 9, 8, 0, 3, 2, 9, 10, 4],
    [8, 5, 3, 4, 2, 6, 10, 9, 2, 1, 3, 0, 7, 4, 1, 9],
    [5, 8, 9, 7, 8, 4, 9, 5, 10, 6, 2, 7, 0, 5, 6, 8],
    [10, 9, 7, 10, 6, 1, 5, 1, 3, 10, 9, 4, 5, 0, 2, 9],
    [7, 1, 6, 9, 10, 9, 7, 8, 5, 2, 10, 1, 6, 2, 0, 3],
    [1, 2, 5, 8, 10, 7, 1, 6, 2, 7, 4, 9, 8, 9, 3, 0]
```

Figura 37 : Matriz de adjacência 16x16 representando o processo e as ligações entre as tarefas utilizadas na avaliação.

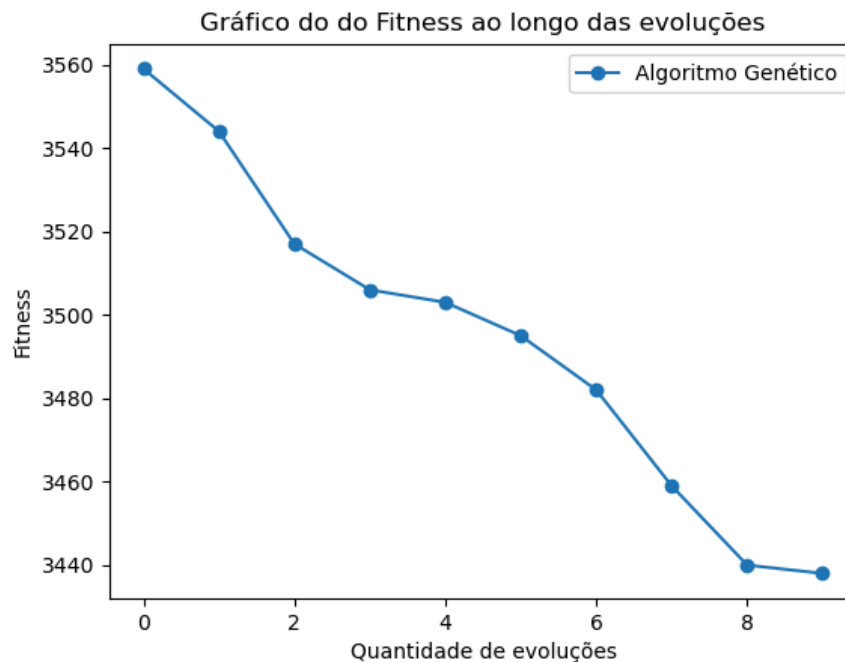


Figura 38 : Evolução do fitness



```
Melhor solução encontrada:  
[8, 5, 4, 7]  
[14, 10, 2, 15]  
[6, 3, 12, 9]  
[0, 11, 13, 1]  
Fitness da melhor solução: 3438
```

Figura 39 : Resultados obtidos do algoritmo genético

É possível concluir que, o algoritmo genético está de acordo com o previsto, apresentando evoluções com o passar do tempo que apresentam melhoras nas métricas de estudo.

#### 4.2.3 Algoritmo dos Condores

Assim como em um algoritmo genético, o funcionamento deve, assim como na vida real, apresentar melhoras em algum aspecto durante o passar das evoluções. Assim, foram monitoradas as evoluções conforme o passar das gerações, avaliando qual foi o fitness alcançado e, depois de várias gerações sem melhoras, qual foi o mapa de alocação ótima encontrado por este algoritmo. Foi utilizado a mesma matriz de adjacência e o mesmo grid utilizado no algoritmo genético.

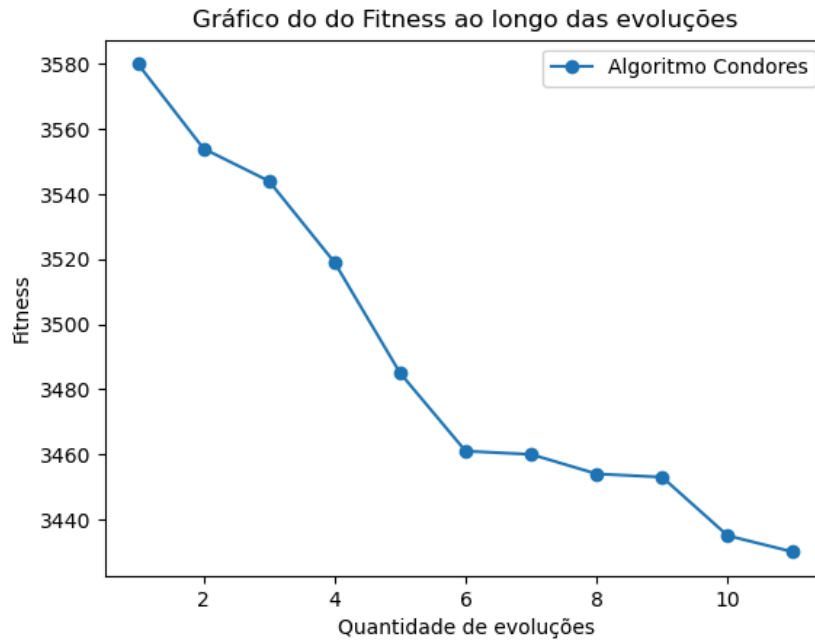


Figura 40 : Evolução do Fitness

```
Melhor solução encontrada:  
[15, 6, 12, 5]  
[0, 13, 7, 11]  
[1, 4, 8, 10]  
[2, 3, 14, 9]  
Fitness da melhor solução: 3430
```

Figura 41 : Resultados obtidos do algoritmo dos Condores

Portanto, assim como no algoritmo genético, é possível perceber a corretude do algoritmo, apresentando melhoras com o passar das gerações baseando-se no fitness.

## 5 Discussões e Resultados Parciais

Com os testes de comparação feitos, é possível afirmar que o simulador apresenta um alto grau de confiabilidade, visto que as métricas de desempenho atingiram resultados semelhantes. Os algoritmos de mapeamento traduzidos até o momento, apresentaram comportamento dentro do esperado.

A partir da análise do que já foi feito do cronograma primeiramente estabelecido e do que precisa ser concluído na segunda etapa no projeto, tem-se a seguinte reformulação do cronograma para o próximo semestre:

Tabela 2: Novo Cronograma da Segunda do Projeto.

Ano	2024				2025	
Atividade	Setembro	Outubro	Novembro	Dezembro	Janeiro	Fevereiro
I	X	X	X			
II			X			
III			X	X	X	
IV					X	X

- I. Melhoria na camada de interface com o usuário
- II. Implementação das outras formas de roteamento (XY e Negative First)
- III. Implementação de novos algoritmo de mapeamento e métodos de análise qualitativa
- IV. Terminar relatório e filtrar conteúdo para o artigo

Para a segunda parte deste projeto, temos como objetivo tornar o uso do simulador mais interativo, apresentando visualmente o melhor mapeamento dentre os existentes e uma forma de visualização do funcionamento de um NoC, permitindo observar o caminho percorrido pelos pacotes da tarefa desejada pelo usuário. Também, será realizada a implementação dos algoritmos de roteamentos propostos (XYX e Negative First). Além disso, será ampliado a gama de algoritmos de mapeamento, investigando algoritmos mais próximos do estado da arte, e realizar a análise qualitativa baseada na métrica escolhida pelo usuário para os algoritmos já existentes e para os novos algoritmos implementados.

É importante ressaltar que o objetivo do projeto inicial já foi bem desenvolvido e está em um estágio avançado. Assim, o projeto passa a ter um foco em tornar o programa mais interativo, visando a apresentação dos resultados de forma mais elucidativa e simples e, implementar algoritmos de mapeamento mais próximos do estado da arte. Essas etapas são parte fundamental para a conclusão do objetivo inicial desta IC.

Por fim, todos os códigos apresentados no desenvolvimento do relatório estão disponíveis no link do GitHub fornecido na seção de apêndices. O projeto está sendo desenvolvido por módulos, onde dentro da pasta "projeto" existe uma pasta com os algoritmos de mapeamento desenvolvidos em arquivos separados, assim como dois arquivos, main.py e Noc.py. No arquivo main.py, encontram-se os códigos da interface gráfica e o local onde, futuramente, após o término das implementações dos algoritmos e suas formas de comparação, será importado o simulador construído no arquivo Noc.py e os algoritmos de mapeamento. Isso permitirá criar toda a lógica para a coleta das

informações do usuário, cálculos qualitativos dos mapeamentos feitos a partir da tarefa fornecida pelo usuário, funcionamento do simulador com o melhor mapeamento encontrado e a visualização do mapeamento.

## 6 Apêndice

Link do github: <https://github.com/breninhoinho/lc>

## 7 Referências Bibliográficas

[1] Bonney, Colin Andrew. *Fault Tolerant Task Mapping in Many-Core Systems*. Diss. University of York, 2016.

[2] Mehmood, Farrukh, et al. "An efficient and cost effective application mapping for network-on-chip using Andean condor algorithm." *Journal of Network and Computer Applications* 200 (2022): 103319.

[3] Fen, G. E., and W. U. Ning. "Genetic algorithm based mapping and routing approach for network on chip architectures." *Chinese Journal of Electronics* 19.1 (2010): 91-96.

[4] Khan, Tahir. "Performance Analysis of XY Routing Algorithm using 2-D Mesh (M x N) Topology." (2017).

[5] TKINTER. Tkinter. Disponível em <<https://docs.python.org/3/library/tkinter.html>>

[6] NETWORKX. NetworkX. Disponível em <<https://networkx.org/>>

[7] MATPLOTLIB. Matplotlib: Python plotting — Matplotlib 3.1.1 documentation. Disponível em: <<https://matplotlib.org/>>.

[8] Ge, Fen, et al. "A Multi-Phase Based Multi-Application Mapping Approach for Many-Core Networks-on-Chip." *Micromachines* 12.6 (2021): 613.

[9] Patooghy, Ahmad, and Seyed Ghassem Miremadi. "XYX: A power & performance efficient fault-tolerant routing algorithm for network on chip." *2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*. IEEE, 2009.

[10] Gawish, Eman Kamel, M. Watheq El-Kharashi, and Mohamed Fathy Abu-Elyazeed. "Variability-tolerant routing algorithms for Networks-on-Chip." *Microprocessors and Microsystems* 38.8 (2014): 1037-1045.