



Relatório 8 - Arquitetura de Alto Desempenho

Conceitos de Arquiteturas Avançadas - Multicomputadores

Alunos: **Lucas Mantovani Gomes**

Jonas Gabriel dos Santos Costa Fagundes

Lucas Gabriel Valenti

Professor: **Emerson Carlos Pedrino**

São Carlos, 24 de julho de 2025

Resumo

O presente relatório descreve o que foi desenvolvido na oitava atividade experimental da disciplina de arquitetura de alto desempenho. Na qual o objetivo era estudar o conceito de Multicomputadores e como estes podem contribuir para os ganhos de desempenho, em um nível de paralelismo de granularidade mais grossa. Para isso, utilizamos a biblioteca mpi4py, para implementar o uma aproximação de pi pelo método de Monte Carlo e avaliar o *speedup*.

1 Exercícios para Avaliação

Exercícios propostos no material utilizado em aula pelo professor Dr. Emerson Pedrino (PEDRINO, 2025), na disciplina de Arquitetura de Alto Desempenho 2025/1.

1.1 Que tipo de problema não temos agora em relação aos multiprocessadores? E quais temos?

Nos multicomputadores, cada nó é um computador completo, com sua própria memória, se comunicando através de um sistema de interconexão, ou seja, não existe mais o problema de concorrência no acesso à memória compartilhada ou coerência de cache. No entanto, surgem novos desafios, como latência e sobrecarga na comunicação entre os nós, complexidade na sincronização das trocas de mensagens e distribuição de tarefas.

1.2 Caso a rede de interconexão seja por meio compartilhado (Barramento), o que ocorrerá à fase de roteamento? Neste caso, como ocorre a comunicação entre dois nós?

Em um meio compartilhado, como todos os nós compartilham o mesmo meio físico a fase de roteamento não é elaborada, não há necessidade de escolha de rota, pois todos os dispositivos vão receber as mensagens e apenas o receptor responderá, o que resulta em problemas de colisões, travamento de acesso (só um pode transmitir por vez) e baixa escalabilidade (quanto mais dispositivos no barramento pior o desempenho).

1.3 Estratégias de roteamento: Wormhole

Nessa estratégia as mensagens são divididas em pedaços menores (flits) o primeiro flit de endereçamento chega no roteador e reserva a rota para o próximo nó, e os outros flits de dados e controle, seguem o caminho, como um verme rastejando. Apesar de ser uma estratégia eficiente para mensagens curtas (baixa latência e pouco uso de buffer), é extremamente sensível a bloqueios e deadlocks.

1.4 Tórus 2D: Comente sobre sua Tolerância a falhas e largura de banda!

A topologia tórus 2D é semelhante a uma malha 2D simples, mas com suas bordas interconectadas, formando um toroide, o que permite menores caminhos entre os nós – o que permite suportar maior largura de banda – e maior redundância de rotas, que consequentemente traz maior tolerância a falhas.

1.5 Anel de Anéis: E sua tolerância a falhas e latência?

Superior ao anel simples, com vários anéis permite vários desvios no caso de falhas.

1.6 Compare a técnica de comutação por circuito com a de comutação por pacotes.

A comutação por circuito tem a vantagem de possuir uma menor latência e não ter disputas de caminhos, pois o caminho é previamente reservado. Mas em comparação, a comutação por pacotes apresenta um maior aproveitamento da topologia de interconexão, suportando múltiplas comunicações simultâneas e maior escalabilidade, apesar da baixa latência e possibilidade de pacotes disputarem o mesmo caminho.

1.7 Quais as vantagens da Rede Torus 2D em relação à Malha 2D?

A Torus 2D elimina o problema das bordas, reduz consideravelmente a distância entre os nós, consequentemente apresentando maior largura de banda e tolerância a falhas.

1.8 Quais as vantagens do Anel com Cordas? Para $p=16$ nós, qual é a relação entre cordas e nós?

Apresenta menor latência e escalabilidade muito simples sem aumentar a complexidade da rede. Se uma corda conecta dois nós, temos então uma relação de, para ' p ' nós temos $\log_2(p)$ cordas. Para 16 nós, precisamos de 4 cordas.

1.9 Aborde sobre as vantagens do Anel de Anéis:

Com múltiplos caminhos, apresenta desempenho e tráfego superiores ao anel simples, evitando gargalos e falhas.

1.10 Pesquisa sobre topologias baseadas em árvores, suas vantagens e exemplos comerciais (Trees e Fat Trees)!

As topologias baseadas em árvores organizam os nós hierarquicamente, com largura de banda proporcional ao nível da árvore. Essa característica, permite suporte a grande número de nós e comunicação paralela intensiva, excelente para redes comerciais como InfiniBand e interconexão de supercomputadores.

1.11 Rede Omega: descreva o impacto do diâmetro da rede e da largura de bisseção.

O diâmetro da rede ômega se trata de uma métrica fixa de quantos saltos entre dois nós ocorrem do início até o destino. Essa é uma das propriedades que conferem à rede ômega sua característica determinística, latência previsível. Já a largura da bisseção, trata-se do número mínimo de conexões que precisa ser cortada para dividir a rede em duas metades(uma vez que a localidade em que ocorrem os saltos mais críticos é justamente nessa bisseção), tal característica determina a quantidade de dados que podem passar de um lado para o outro sem haver congestionamento ou gargalos.

1.12 Vantagens da Rede de Barramentos Hierárquicos

Reduz a carga sobre um único barramento, dividindo em níveis e subníveis, organizando a comunicação e melhorando o desempenho, conferindo modularidade e escalabilidade.

1.13 Compare Store-And-Forward, Virtual-Cut-Through, Wormhole e Deflection Routing

Tabela 1: Comparação entre técnicas de roteamento

Técnica	Vantagens	Desvantagens
Store-and-Forward	Simples, fácil de implementar	Alta latência e buffers grandes
Virtual Cut-Through	Menor latência que Store-and-Forward	Ainda precisa de buffers consideráveis
Wormhole	Latência baixa, buffer mínimo	Sensível a bloqueios (deadlock)
Deflection Routing	Não usa buffer, desvia pacotes	Aumenta o número de saltos, imprevisível
Comutação por circuito	Baixa latência e sem risco de contenção	Baixa flexibilidade, uso ineficiente da rede

1.14 Pesquisa sobre a Classificação de Algoritmos de Roteamento

- Determinísticos: caminho fixo (Ex: XY).
- Aleatórios: escolhe aleatoriamente qualquer caminho válido para o destino.
- Adaptativo: reage ao tráfego, escolhendo caminhos livres de acordo com diretrizes previamente estabelecidas.

1.15 Pesquisa sobre Técnicas de Arbitragem

- FIFO (first in first out): primeiro a chegar é o primeiro a ser atendido.
fixa: sempre atende portas de maior prioridade.
- Age-based: pacotes mais antigos do sistema têm prioridade (evita starvation).
- Grant-Based (baseada em concessão): uma técnica que descreve o protocolo a ser seguido pelo árbitro, como um semáforo mais complexo, o árbitro opera em três fases: Request (cada entrada envia pedidos para as saídas que deseja usar), Grant(a

saída recebe o pedido e escolhe quem vai usá-la, e envia a resposta), Acknowledge (a entrada confirma que vai usar a porta concedida).

1.16 Dê exemplos de Deadlock e Livelock:

Em um deadlock, vários pacotes esperam recursos uns dos outros, formando um ciclo de intertravamento: A espera recursos de B, que espera recursos de C, que por sua vez espera recursos de A. E o ciclo trava; Em um livelock, o pacote não fica travado, mas todas as saídas que precisa estão bloqueadas ou indisponíveis, e o pacote continua se movendo, mas sendo desviado constantemente, sem nunca chegar ao destino.(como em sistemas adaptativos, que os pacotes escolhem “caminhos livres” mas entram em ciclos);

2 Desenvolvimento Prático: Estimando Pi por Monte Carlo

O método de monte carlo consiste em utilizar uma amostragem aleatória muito grande para estimar resultados numéricos. Na nossa prática, geramos valores aleatórios para um par ordenado (x,y) num intervalo de $2r$, variando de $[-1,1]$ e verificamos se o par está dentro de um círculo inscrito, de raio 1 – através da equação geral do círculo.

2.1 Biblioteca MPI4py(MPI for python)

De acordo com a documentação oficial da biblioteca mpi4py (DALCIN, LISANDRO, 2024) o sistema MPI (Message Passing Interface) é um padrão de troca de mensagens de alto desempenho, amplamente utilizado em supercomputadores e computadores paralelos, para fazer o gerenciamento de memória de forma eficiente. Com as vantagens de ser modularizado, portátil de uma plataforma para outra e oferecer controle total sobre a comunicação. Pois, fundamentalmente, no MPI o sistema opera com memória distribuída, na qual cada nó (computador, núcleo ou processo) possui sua própria memória local e sua própria cópia do código. A comunicação entre os nós — ou seja, a troca de dados — é realizada por meio de mensagens. Nesse padrão, encontramos operações ponto a ponto (send e receive), operações coletivas (broadcast, scatter, gather, reduce) e operações de sincronização (barrier, que funciona como um *checkpoint* em que todos os processos devem aguardar). Em nossa prática, vamos utilizar a biblioteca mpi4py que faz uma ponte entre o python e o OpenMPI em C, uma das especificações mais populares de MPI.

2.2 Código utilizado:

```
#montecarlopi_mpi.py

from mpi4py import MPI

import numpy as np

def ponto_dentro_do_circulo(x, y):
```



```
return x**2 + y**2 <= 1.0
```

```
def monte_carlo_pi(local_n):
```

```
    np.random.seed() # garante aleatoriedade por processo
```

```
    x = np.random.uniform(-1, 1, local_n)
```

```
    y = np.random.uniform(-1, 1, local_n)
```

```
    dentro = np.sum(x**2 + y**2 <= 1.0)
```

```
    return dentro
```

```
def main():
```

```
    comm = MPI.COMM_WORLD #interface MPI, agrupamento de #processos
```

```
    rank = comm.Get_rank()#ID do processo
```

```
    size = comm.Get_size()#numero de processos
```

```
    n = 10**7 # total de pontos
```

```
    local_n = n // size
```

```
    # Início da contagem de tempo (apenas no processo 0)
```

```
    if rank == 0:
```

```
        tempo_inicio = MPI.Wtime()
```

```
    # Cada processo executa sua parte
```

```
    local_contagem = monte_carlo_pi(local_n)
```

```
    # Redução da contagem total de pontos dentro do círculo, processo(root) 0 recebe
```

```
    total_dentro = comm.reduce(local_contagem, op=MPI.SUM, root=0)
```

```
    if rank == 0:
```

```
        pi_estimado = 4.0 * total_dentro / n
```

```
        tempo_fim = MPI.Wtime()
```

```

        print(f"Estimativa de pi com {n} pontos e {size} processos: {pi_estimado}")
        print(f"Tempo total de execucao: {tempo_fim - tempo_inicio:.6f} segundos")

if __name__ == "__main__":
    main()

```

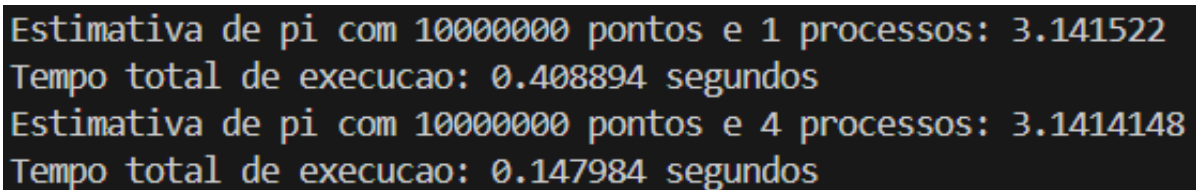
Para executar a biblioteca MPI é necessário especificar a quantidade de núcleos que vamos utilizar em paralelo através do prompt:

```
mpiexec -n # python montecarlopi_mpi.py
```

Com # variando conforme a quantidade de processos desejada. Vamos calcular para 1 processo e 4 processos para verificar o speedup:

$$SpeedUp = TempoSerial / TempoParalelo = 0.251097 / 0.146600 = 1.71x .$$

O que foge ao SpeedUp ideal de 4. Pela lei de Amdahl, podemos calcular que nosso código é 58% paralelizável.



```

Estimativa de pi com 10000000 pontos e 1 processos: 3.141522
Tempo total de execucao: 0.408894 segundos
Estimativa de pi com 10000000 pontos e 4 processos: 3.1414148
Tempo total de execucao: 0.147984 segundos

```

Figura 1: Estimando pi

2.3 Visualização e comparação

Para visualizar em um gráfico utilizamos a biblioteca Multiprocessing e pyplot:

```

#montecarlopi.py

import time

import numpy as np

import multiprocessing

import matplotlib.pyplot as plt

#gera pontos x,y com valores aleatórios entre -1 e 1

#verifica se o ponto gerado está dentro de um circulo de raio 1

```

```

def ponto_dentro_do_circulo(_):
    x, y = np.random.uniform(-1, 1), np.random.uniform(-1, 1) #valores aleatórios e
    return x**2 + y**2 <= 1.0 #verifica se o valor está dentro do circulo

def monte_carlo_pi_paralelo(n_pontos, n_processos):
    #a biblioteca multiprocessing separa automaticamente a função ponto_dentro_do_c
    #em n_processos paralelos
    with multiprocessing.Pool(processes=n_processos) as pool:
        resultados = pool.map(ponto_dentro_do_circulo, range(n_pontos))
    total_dentro = sum(resultados)
    #pi é estimado pela razão entre o número dentro do círculo pelo total de pontos
    pi_estimado = 4.0 * total_dentro / n_pontos
    return pi_estimado

#mede o tempo de execução para calcular speedup
def medir_tempo_e_pi(n_pontos, n_processos):
    inicio = time.time()
    pi = monte_carlo_pi_paralelo(n_pontos, n_processos)
    fim = time.time()
    return fim - inicio, pi

#gera os valores de base 2, para a qntd de núcleos por iteração de acordo com sua c
def gerar_potencias_de_2(max_cores):
    potencias = []
    n = 0
    while (valor := 2**n) <= max_cores:
        potencias.append(valor)
        n += 1
    return potencias

```

```

def main():
    n_pontos = 10_000_000
    max_cores = multiprocessing.cpu_count() #nucleos da sua cpu ou trocar por valor
    cores_para_testar = gerar_potencias_de_2(max_cores) #conjunto[1,2,4,8,...,max]

    print(f"Número de núcleos disponíveis: {max_cores}")
    print(f"Testando para: {cores_para_testar}\n")

    tempos = []
    estimativas_pi = []

    for n in cores_para_testar:
        print(f" Executando com {n} processo(s)...")
        tempo, pi = medir_tempo_e_pi(n_pontos, n)
        tempos.append(tempo)
        estimativas_pi.append(pi)
        print(f"  {pi:.8f} | Tempo: {tempo:.4f} segundos\n")

    # Speedup real e ideal
    tempo_base = tempos[0] #cores = 1
    speedups = [tempo_base / t for t in tempos]
    speedup_ideal = cores_para_testar

    # Plot
    plt.figure(figsize=(8, 5))
    plt.plot(cores_para_testar, speedups, marker='o', label='Speedup real')
    plt.plot(cores_para_testar, speedup_ideal, 'k--', label='Speedup ideal (linear)')
    plt.title('Speedup vs Número de Núcleos')
    plt.xlabel('Número de núcleos')
    plt.ylabel('Speedup')

```

```

plt.xticks(cores_para_testar)

plt.legend()

plt.grid(True)

plt.tight_layout()

plt.show()

if __name__ == "__main__":
    main()

```

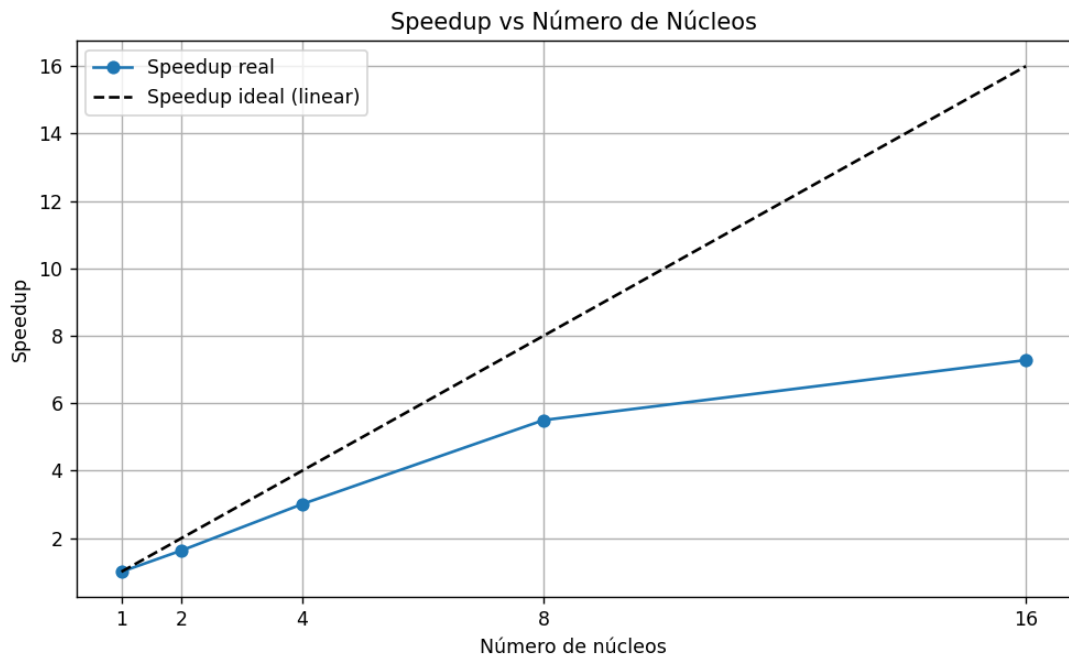


Figura 2: Gráfico - Speedup em Multiprocessing

Neste gráfico, podemos observar os rendimentos decrescentes, decorrentes da tentativa de paralelizar tarefas simples e com baixo nível de paralelização.

Outro ponto que notamos é o desempenho superior da biblioteca multiprocessing em relação à MPI. Isso se deve ao modelo de paralelismo empregado em cada uma:

- A biblioteca MPI replica todo o código em vários nós independentes (processos para diferentes cores no nosso caso) e precisa fazer a comunicação entre esses nós mesmo em um ambiente local, e isso gera um custo de inicialização que para tarefas simples

e com poucos nós como a nossa, não traz tanto benefício. É uma estratégia melhor aproveitada em supercomputadores.

- A biblioteca multiprocessing, apenas faz um `fork()` dividindo os processos em subprocessos e copiando o espaço de memória ou usando memória compartilhada, fazendo com que localmente para tarefas simples e poucos nós, entregue um desempenho adequado.

3 Conclusão

A atividade permitiu compreender, na prática, como o modelo de memória distribuída dos multicomputadores exige estratégias específicas de comunicação, como as oferecidas pelo padrão MPI. Enquanto o MPI mostrou-se robusto e adequado para ambientes distribuídos reais e de larga escala, sua sobrecarga de inicialização o torna menos vantajoso em tarefas simples executadas localmente. Por outro lado, o multiprocessing, por aproveitar a memória compartilhada local e exigir menos coordenação entre processos, apresentou melhor desempenho nesse contexto.

Complementando a análise prática, os exercícios teóricos reforçaram o entendimento sobre topologias de interconexão, estratégias de roteamento, técnicas de arbitragem e tolerância a falhas — elementos fundamentais para o projeto de arquiteturas de alto desempenho.

Referências

DALCIN, LISANDRO. **Overview — mpi4py 3.1.6 documentation**. Acesso em: 23 jul. 2025. 2024. Disponível em:

`https://mpi4py.readthedocs.io/en/stable/overview.html`.

PEDRINO, Emerson Carlos. **Material didático de Arquitetura de Alto**

Desempenho. [S.l.: s.n.], 2025. Apostila em formato PDF disponibilizada no AVA2 da disciplina, Universidade Federal de São Carlos.