



**UNIVERSIDADE ESTADUAL DO PARANÁ - *CAMPUS* APUCARANA**

**Paloma de Castro Leite**

## **RELATÓRIO TÉCNICO - AOC**

APUCARANA – PR  
2024

**Paloma de Castro Leite**

## **RELATÓRIO TÉCNICO – AOC**

Trabalho apresentado à disciplina de Arquitetura e Organização de Computadores do curso de Bacharelado em Ciência da Computação.

**Professor:** Guilherme Henrique de Souza Nakahata;

**APUCARANA – PR  
2024**

## SUMÁRIO

<b>INTRODUÇÃO .....</b>	<b>03</b>
<b>CAPÍTULO 1: OBJETIVOS .....</b>	<b>04</b>
<b>CAPÍTULO 2: MOTIVAÇÃO E RECURSOS UTILIZADOS .....</b>	<b>05</b>
<b>2.1 Motivação.....</b>	<b>05</b>
<b>2.2 Estrutura de Dados .....</b>	<b>06</b>
<b>2.3 Linguagem de programação e demais informações.....</b>	<b>07</b>
<b>CAPÍTULO 3: RESULTADOS .....</b>	<b>09</b>
<b>CONCLUSÃO .....</b>	<b>14</b>
<b>REFERÊNCIAS .....</b>	<b>15</b>

## INTRODUÇÃO

A Arquitetura e Organização de Computadores (AOC) é o estudo do funcionamento interno, da estrutura e da implementação de sistemas de computador porque as organizações definem como o sistema é estruturado, enquanto a arquitetura se concentra nas propriedades que afetam diretamente a execução do programa e são visíveis para o programador. Simplificando, AOC envolve o projeto de computadores, dispositivos de armazenamento de dados e componentes de rede que armazenam e executam programas, transferem dados e interagem entre computadores, redes e usuários

Neste trabalho, iremos analisar especificamente um benchmark, testes que geralmente são realizados contra um sistema para determinar o desempenho atual e podem ser usados para detectar tipos específicos de informação de acordo com o tipo de teste aplicado e as condições do ambiente. Dessa maneira, testes são feitos de maneira repetitiva e sob as mesmas condições para que gere resultados autênticos, facilitando assim a comparação e garantindo a melhor eficiência possível de um código, para que após o resultado final, possa-se fazer a implementação de melhorias e garantir que o sistema seja executado da maneira mais eficiente possível.

## **CAPÍTULO 1**

### **OBJETIVOS**

Este trabalho tem por objetivo apresentar uma proposta de um benchmark - termo que se refere a um conjunto de testes expressivos e específicos para avaliar o desempenho de um componente de hardware -, a finalidade do código fonte é medir e analisar o tempo de alocação e manipulação de memória de maneira aleatória e sequencial utilizando diferentes tamanhos de blocos, proporcionando assim uma compreensão do desempenho da memória em um sistema. A análise dos resultados obtidos por meio deste código pode ajudar a identificar padrões de desempenho, como diferenças de tempo em relação ao tamanho do bloco e variações entre iterações, sendo útil para otimização de sistemas, testes de desempenho e análise de eficiência.

## **CAPÍTULO 2**

### **MOTIVAÇÃO E RECURSOS UTILIZADOS**

Tendo em vista o que foi descrito anteriormente, devemos explicitar os motivos para a realização do trabalho e seu objetivo final, além dos recursos utilizados para que o trabalho seja executado de maneira eficiente e concisa.

#### **2.1 Motivação**

Conforme mencionado no capítulo que trata acerca dos objetivos do trabalho em pauta, a motivação seria a execução de um código fonte funcional em que possamos demonstrar e exemplificar o funcionamento de um benchmark de alocação e manipulação de memória, em que o código realiza múltiplas iterações de alocação para obter uma média confiável do tempo necessário para alocar blocos de memória de tamanhos específicos, além de preencher a memória alocada com valores aleatórios e sequenciais para simular a manipulação de dados, demonstrando como a alocação e o uso de memória afetam o desempenho.

Este tipo de benchmark é uma ferramenta importante para várias áreas da ciência da computação e sistemas de software, sendo essencial para o desempenho e a estabilidade de sistemas computacionais, especialmente em ambientes com recursos limitados ou em aplicações que lidam com grandes volumes de dados, pois ao medir o tempo de alocação e manipulação de memória em diferentes tamanhos de blocos, é possível identificar potenciais problemas e ineficiências em um sistema, especialmente quando se trata de gerenciamento de memória em aplicações críticas ou de alto desempenho.

Portanto, faz-se necessário, com as informações pertinentes acerca dos objetivos e motivações, detalhar os dados mais relevantes à Estrutura de Dados, Linguagem de Programação e demais questões acerca da implementação do código fonte em questão para melhor análise de suas funcionalidades, a fim de obter uma conclusão geral.

## 2.2 Estrutura de Dados

Este código em C implementa um benchmark para avaliar o desempenho da alocação e manipulação de memória em diferentes tamanhos de bloco. Inicialmente, são incluídas as bibliotecas padrões como *stdio.h*, *stdlib.h* e *time.h*, para entrada e saída de dados, alocação da memória dinamicamente e medição do tempo respectivamente, além de uma constante definida como *ITERATIONS*, usada para especificar o número de vezes (1000) que cada execução será repetida e obter uma média dos tempos.

Para medir o tempo de alocação e manipulação da memória em cada um dos blocos, foi criada a função *medir\_tempos*, no qual usamos o relógio do sistema denominado como *clock()* para medir o tempo antes e depois da alocação de memória através da função *malloc* - retorna NULL se não houver memória suficiente disponível, exibindo uma mensagem de erro -, em seguida inicializamos a semente *srand* para o gerador de números aleatórios usando o tempo atual apenas uma vez, a fim de garantir uma sequência de números única em todo o programa, e depois incrementamos cada byte de forma sequencial, medindo o tempo de ambas as manipulações e os armazenando em ponteiros passados como os parâmetros *tempo\_alocacao* e *tempo\_manipulacao*. Já a função *calcular\_tempo\_medio* recebe como parâmetro o array *tempo[ ]* e o número de iterações, utilizando os tempos fornecidos como entrada para somá-los através de um loop e calcular a média entre eles.

A medição ocorre apenas após a liberação da memória através do *free(bloco)* para incluir todas as operações e garantir que uma alocação de memória não permaneça ocupada até o fim do programa, evitando assim que a função *medir\_tempos*, que é chamada repetidamente em um loop, acumule memória não utilizada ao longo do tempo, o que pode levar a problemas de desempenho e falhas no programa.

Agora, tendo em foco a função principal *main*, que administra todo o processo de testes, temos inicialmente duas arrays principais:

a) *tamanho [ ]* : Armazena os diferentes tamanhos de alocação que serão testados, sendo eles: 1 KB, 8 KB, 64 KB, 512 KB e 4 MB, selecionados para abranger uma variedade de cenários comuns de alocação de memória em programas de computador.

b) *tempos\_alocacao[ ]* e *tempos\_manipulacao[ ]*: Armazenam os tempos de alocação e manipulação medidos durante os testes para cada tamanho de alocação. Cada array tem um tamanho igual ao número de iterações definido pela constante `ITERATIONS`.

Em seguida, decretamos um *for* que itera sobre os tamanhos de alocação definidos no array *tamanho[ ]*, medindo os tempos de alocação e manipulação para cada um deles para finalmente calcular os tempos médios de alocação e manipulação para cada um dos tamanhos de blocos descritos anteriormente. Por fim, os resultados são então exibidos na saída padrão, mostrando o tempo médio para a manipulação aleatória e sequencial da memória em segundos.

A execução do programa finaliza ao usuário inserir um caractere qualquer, que é lida pela variável *char* definida como *c* e recebida através de um *scanf*, uma função utilizada para garantir que o programa feche apenas após termos obtido e visualizado todos os testes. Algumas outras maneiras de organizar e gerir o código fonte foram utilizadas, porém apenas com o objetivo de facilitar e possibilitar o desempenho geral do mesmo.

## 2.3 Linguagem de Programação e demais informações

A linguagem de programação C foi escolhida para o desenvolvimento do código, ela foi abordada principalmente durante o primeiro ano do curso, possibilitando assim um maior domínio para o desenvolvimento do programa. Esta linguagem fornece controle de baixo nível sobre o hardware do computador e é altamente otimizada para desempenho, além de possibilitar manipular ponteiros e gerenciar a memória manualmente, o que pode levar a um código mais eficiente, ao mesmo tempo que também requer cuidado para evitar vazamentos de memória e erros de acesso.



Dessa maneira, a linguagem de programação C se tornou ideal para este caso específico, em que o benchmarking foi feito para avaliar o desempenho da alocação de memória em diferentes tamanhos e os tempos associados à manipulação aleatória e sequencial desses blocos de memória.

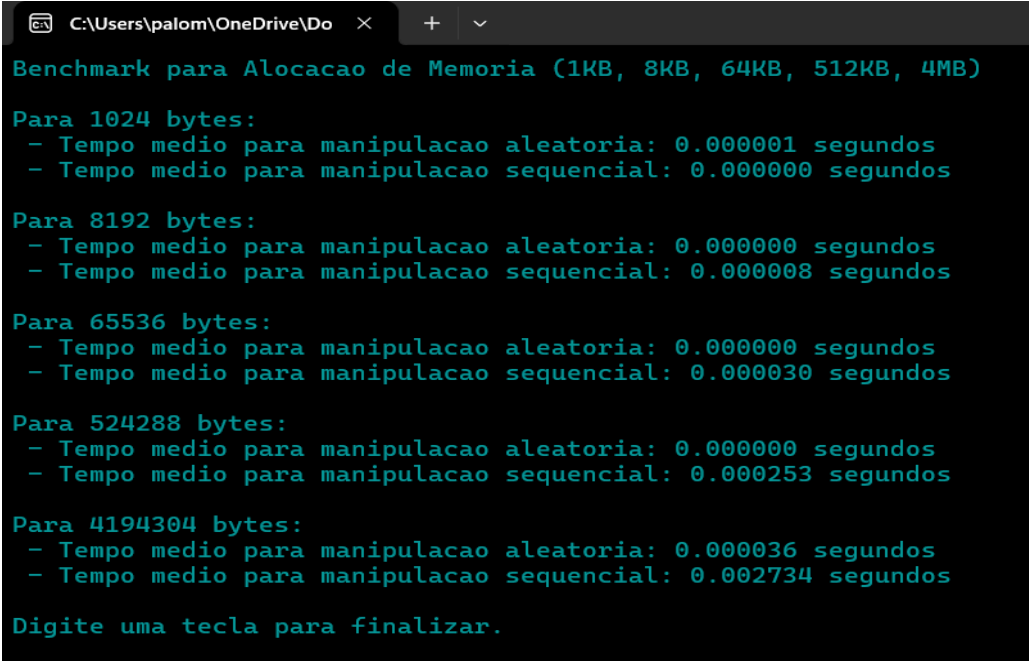
## CAPÍTULO 3

### RESULTADOS

Diante dos expostos apresentados ao longo do trabalho, a finalidade esperada seria o pleno funcionamento de um código que exemplifica como um benchmark destinado a alocação e manipulação de memória funciona de maneira prática, demonstrando a média obtida em cada teste de acordo com o bloco de memória especificado, usando diferentes ambientes e tamanhos de dados para assegurar um melhor desempenho e evitar possíveis falhas.

Por consequência, com a implementação total e sua revisão, o objetivo principal do código fonte foi atingido, resultando em uma aplicação funcional no qual os tempos de alocação e manipulação variam conforme os blocos aumentam sua capacidade. Entretanto, manipular esses dados se torna mais complexo à medida que o tamanho dos blocos aumenta, o que pode influenciar na eficiência das operações de manipulação sequencial ao lidar com uma maior quantidade de dados.

Abaixo, nas figuras 1, 2, 3, 4 e 5, podemos ver o funcionamento do benchmark em diferentes máquinas e algumas especificações das mesmas.



```
C:\Users\palom\OneDrive\Do  X  +  v
Benchmark para Alocação de Memória (1KB, 8KB, 64KB, 512KB, 4MB)
Para 1024 bytes:
- Tempo medio para manipulacao aleatoria: 0.000001 segundos
- Tempo medio para manipulacao sequencial: 0.000000 segundos
Para 8192 bytes:
- Tempo medio para manipulacao aleatoria: 0.000000 segundos
- Tempo medio para manipulacao sequencial: 0.000008 segundos
Para 65536 bytes:
- Tempo medio para manipulacao aleatoria: 0.000000 segundos
- Tempo medio para manipulacao sequencial: 0.000030 segundos
Para 524288 bytes:
- Tempo medio para manipulacao aleatoria: 0.000000 segundos
- Tempo medio para manipulacao sequencial: 0.000253 segundos
Para 4194304 bytes:
- Tempo medio para manipulacao aleatoria: 0.000036 segundos
- Tempo medio para manipulacao sequencial: 0.002734 segundos
Digite uma tecla para finalizar.
```

*Figura 1 - Asus I5 - 1025G1, 8GB.*

```
C:\Users\rayss\OneDrive\Area de Trabalho\BenchmarkTeste.exe
Benchmark para Alocação de Memória (1KB, 8KB, 64KB, 512KB, 4MB)

Para 1024 bytes:
- Tempo médio para manipulação aleatória: 0.000000 segundos
- Tempo médio para manipulação sequencial: 0.000004 segundos

Para 8192 bytes:
- Tempo médio para manipulação aleatória: 0.000000 segundos
- Tempo médio para manipulação sequencial: 0.000024 segundos

Para 65536 bytes:
- Tempo médio para manipulação aleatória: 0.000000 segundos
- Tempo médio para manipulação sequencial: 0.000195 segundos

Para 524288 bytes:
- Tempo médio para manipulação aleatória: 0.000010 segundos
- Tempo médio para manipulação sequencial: 0.001285 segundos

Para 4194304 bytes:
- Tempo médio para manipulação aleatória: 0.000019 segundos
- Tempo médio para manipulação sequencial: 0.010037 segundos

Digite uma tecla para finalizar.■
```

*Figura 2 - Acer Nitro AMD Ryzen 5, 24GB*

```
C:\Users\Eduar\OneDrive\Áre x + v
Benchmark para Alocação de Memória (1KB, 8KB, 64KB, 512KB, 4MB)

Para 1024 bytes:
- Tempo médio para manipulação aleatória: 0.000000 segundos
- Tempo médio para manipulação sequencial: 0.000000 segundos

Para 8192 bytes:
- Tempo médio para manipulação aleatória: 0.000000 segundos
- Tempo médio para manipulação sequencial: 0.000008 segundos

Para 65536 bytes:
- Tempo médio para manipulação aleatória: 0.000000 segundos
- Tempo médio para manipulação sequencial: 0.000101 segundos

Para 524288 bytes:
- Tempo médio para manipulação aleatória: 0.000009 segundos
- Tempo médio para manipulação sequencial: 0.000666 segundos

Para 4194304 bytes:
- Tempo médio para manipulação aleatória: 0.000008 segundos
- Tempo médio para manipulação sequencial: 0.006416 segundos

Digite uma tecla para finalizar.Good Game
```

*Figura 3 - Acer Nitro AMD Ryzen 5, 16GB*

```

Benchmark para Alocação de Memória (1KB, 8KB, 64KB, 512KB, 4MB)

Para 1024 bytes:
- Tempo médio para manipulação aleatória: 0.000000 segundos
- Tempo médio para manipulação sequencial: 0.000001 segundos

Para 8192 bytes:
- Tempo médio para manipulação aleatória: 0.000001 segundos
- Tempo médio para manipulação sequencial: 0.000004 segundos

Para 65536 bytes:
- Tempo médio para manipulação aleatória: 0.000000 segundos
- Tempo médio para manipulação sequencial: 0.000044 segundos

Para 524288 bytes:
- Tempo médio para manipulação aleatória: 0.000007 segundos
- Tempo médio para manipulação sequencial: 0.000408 segundos

Para 4194304 bytes:
- Tempo médio para manipulação aleatória: 0.000010 segundos
- Tempo médio para manipulação sequencial: 0.003237 segundos

Digite uma tecla para finalizar.|

```

*Figura 4 - Acer intel core i7, 8GB*

```

C:\Users\fortu\AppData\Local x + v
Benchmark para Alocação de Memória (1KB, 8KB, 64KB, 512KB, 4MB)

Para 1024 bytes:
- Tempo médio para manipulação aleatória: 0.000000 segundos
- Tempo médio para manipulação sequencial: 0.000000 segundos

Para 8192 bytes:
- Tempo médio para manipulação aleatória: 0.000000 segundos
- Tempo médio para manipulação sequencial: 0.000008 segundos

Para 65536 bytes:
- Tempo médio para manipulação aleatória: 0.000000 segundos
- Tempo médio para manipulação sequencial: 0.000082 segundos

Para 524288 bytes:
- Tempo médio para manipulação aleatória: 0.000003 segundos
- Tempo médio para manipulação sequencial: 0.000480 segundos

Para 4194304 bytes:
- Tempo médio para manipulação aleatória: 0.000007 segundos
- Tempo médio para manipulação sequencial: 0.004324 segundos

Digite uma tecla para finalizar.|

```

*Figura 5 - Acer Nitro AMD Ryzen 5, 8GB*

Com base nos resultados obtidos nos testes do código, podemos analisar e tirar algumas conclusões sobre o desempenho da alocação e manipulação de memória para cada tamanho de bloco utilizado:

- a) bloco de 1024 bytes: tanto os tempos de alocação quanto de manipulação (aleatória e sequencial) foram extremamente baixos, na ordem de microssegundos ( $10^{-6}$  segundos) ou nanossegundos ( $10^{-9}$  segundos). Isso indica que as operações realizadas são mais eficientes e rápidas para blocos pequenos;
- b) bloco de 8192 bytes: os tempos deste bloco semelhantes aos observados para o bloco de 1024 bytes. No entanto, os tempos de manipulação sequencial foram significativamente maiores do que os tempos de manipulação aleatória, o que insinua que a manipulação sequencial terá mais dificuldade em termos de tempo conforme os blocos aumentam;
- c) bloco de 65536 bytes: os tempos deste bloco aumentaram, mas ainda permaneceram relativamente baixos em comparação com os anteriores. Porém, os tempos de manipulação sequencial foram significativamente maiores do que os tempos de manipulação aleatória pelo mesmo motivo citado anteriormente;
- d) bloco de 524288 bytes: os tempos de alocação e manipulação deste bloco aumentaram consideravelmente em relação aos anteriores. Porém, os tempos de manipulação sequencial foram notavelmente maiores do que os tempos de manipulação aleatória, ou seja, a manipulação sequencial se torna consideravelmente mais lenta para blocos de tamanho médio a grande;

- e) bloco de 4194304 bytes: os tempos de alocação e manipulação foram os mais altos em comparação com os outros blocos. Os tempos de manipulação sequencial foram bem maiores do que os tempos de manipulação aleatória, indicando que a manipulação sequencial se torna extremamente custosa em termos de tempo para blocos grandes.

Em resumo, à medida que o tamanho do bloco aumenta, os tempos de alocação e manipulação tendem a aumentar, além disso, a manipulação sequencial se torna progressivamente mais lenta em comparação com a manipulação aleatória à medida que o tamanho do bloco aumenta. Entretanto, para todos os tamanhos de bloco testados, os tempos de alocação e manipulação aleatória foram relativamente muito baixos, permanecendo na faixa de microssegundos ( $10^{-6}$  segundos) e nanossegundos ( $10^{-9}$  segundos), ou seja, a alocação de memória e o preenchimento aleatório dos blocos são operações extremamente eficientes, independentemente do tamanho do bloco.

## CONCLUSÃO

Com base no que foi descrito ao longo do trabalho, podemos ver que essa aplicação prática se torna essencial para demonstrar como os benchmarks, que são frequentemente usados para avaliar o desempenho de diferentes componentes de hardware, como processadores, unidades de armazenamento e memória, estão associados a matéria de organização e arquitetura de computadores. Eles fornecem métricas padronizadas que permitem comparar o desempenho de diferentes dispositivos em termos de velocidade, eficiência, desempenho, custo e outros aspectos que são apresentados durante as aulas.

Dessa maneira, tendo como foco o benchmark desenvolvido para este trabalho, podemos concluir que a manipulação da memória sequencial tende a ser mais complexa em termos de tempo do que a aleatória e varia de acordo com o tamanho do bloco que está sendo utilizado. Além disso, os diferentes tipos de sistema e componentes do computador influenciam diretamente nos resultados obtidos em cada teste, visto que um tipo de arquitetura mais eficiente trará melhores resultados.

Portanto, os resultados apurados durante a execução desse programa podem ser usados para futuras otimizações no código, a fim de melhorar o tempo de execução e realizar diferentes tipos de teste para obter resultados mais específicos e que se adaptem para cada situação, ademais, mudanças no software e hardware do sistema também podem ser levadas em consideração. Também é importante ressaltar que os teste são consistentes, o que leva a resultados mais estáveis, mesmo com diferentes execuções do benchmark de alocação e manipulação de memória, já que as condições do ambiente são mantidas para manter uma padronização e alcançar o objetivo desejado.

## REFERÊNCIAS

**Db2 11.1.** Disponível em:

<<https://www.ibm.com/docs/en/db2/11.1?topic=fundamentals-performance-tuning>>.

RODRIGUES, R. A.; PEREIRA JÚNIOR, L. A. BENCHMARK PARA ANÁLISE COMPORTAMENTAL DO SISTEMA DE MEMÓRIA VIRTUAL DO LINUX. Em: **Ciência da Computação: Tecnologias Emergentes em Computação**. [s.l.] Editora Científica Digital, 2020. p. 44–61. Disponível em:

<<https://downloads.editoracientifica.org/articles/201102053.pdf>>.

TYLERMSFT. **malloc**. Disponível em:

<<https://learn.microsoft.com/pt-br/cpp/c-runtime-library/reference/malloc?view=msvc-170>>.