UNIVERSIDADE CATÓLICA DE SANTOS
CIÊNCIA DA COMPUTAÇÃO

Arthur Henrique Silva
Eric Shiraishi de Freitas
Gabriel Carvalho Santana
Gabriel Ferreira Pereira
Vitor Cordeiro Paes Prieto

LEITURA E PROCESSAMENTO DE DADOS COM PARALELISMO EM CPU

Santos
2024

SUMÁRIO

1.0 INTRODUÇÃO

A análise de grandes conjuntos de dados é crucial em diversas áreas do conhecimento, como inteligência artificial, ciência de dados, bioinformática e logística, tornando-se, de maneira geral, indispensável também para o mundo corporativo, que baseia tomadas de decisões estratégicas na análise de dados (FORBES, 2023). No entanto, processar esses volumes massivos de informações de forma eficiente pode se transformar em um desafio computacional significativo. Diante disso, o paralelismo surge como uma solução promissora para acelerar a leitura e o processamento de dados, dividindo as tarefas em múltiplos núcleos e *threads*, que executam simultaneamente frações do trabalho a ser computado.

2.0 LEITURA DE DADOS E PARALELISMO

Uma abordagem tradicional para implementar o paralelismo na leitura de dados é utilizar *threads* convencionais fornecidas por APIs de sistemas operacionais. Cada *thread* é responsável por ler uma porção específica do conjunto de dados ou realizar uma ou mais tarefas do processo na totalidade, dividindo o trabalho entre os núcleos e subnúcleos do processador. Possui como vantagem a compatibilidade em diferentes hardwares e a flexibilidade para solucionar o problema. Em contrapartida, o gerenciamento de múltiplas tarefas sendo executadas ao mesmo tempo aumenta a dificuldade da codificação, pois é necessário garantir a sincronia das tarefas, juntamente da atomização e proteção de seções críticas e problemas derivados da multiprogramação como *Deadlocks ou Starvation* (TANENBAUM; HOS, 2015).

Uma alternativa para facilitar a implementação de um código paralelo (e também para adaptar códigos sequenciais) é a utilização da biblioteca *OpenMP - Open Multi-Processing* (CHANDRA *et al.,* 2001) - que possui como vantagem a simplicidade de poucas funções e macros simples de utilizar, somado a isso, a biblioteca em questão tem como pontos fortes a portabilidade entre diferentes plataformas e compiladores, além de possibilitar escalabilidade de forma automática.

A escolha entre *threads* convencionais e *OpenMP* (ou uma mescla dos dois) para leitura paralela de dados depende de diversos fatores, como a complexidade da tarefa, a portabilidade desejada, o desempenho e a familiaridade do desenvolvedor com as ferramentas. Em geral, o *OpenMP* oferece uma interface mais simples e portátil, porém os problemas precisam ser adaptados aos padrões esperados da biblioteca. Em contrapartida, *threads* convencionais podem oferecer maior flexibilidade e controle, apesar da necessidade de atenção extra no gerenciamento dos recursos disponíveis e na integridade das informações.

3.0 OBJETIVOS

Com o intuito de comparar as duas abordagens supracitadas, temos como objetivo processar um grande conjunto de dados tabular em diferentes *hardwares,* utilizando tanto a biblioteca *OpenMP* quanto a API de *threads* da linguagem de programação C++ e analisar as modificações de implementação e tempo de processamento.

4.0 ESTRUTURA DO CONJUNTO DE DADOS

O *dataset* possui informações sobre movimentações portuárias e agrega 14665111 linhas, sendo cada uma das linhas divididas em 26 colunas, das quais 14 são dados numéricos e 12 categóricos. O objetivo do processamento é separar as colunas categóricas e transformá-las em dados numéricos e reescrever tais informações processadas em um novo arquivo de texto (arquivo principal), além de um arquivo para cada uma das colunas com as informações de qual dado cada código representa em sua determinada coluna (arquivos secundários).

5.0 MODELAGEM DO PROBLEMA

Para solucionar o problema proposto, ele foi separado em duas partes: a leitura, geração das codificações e escrita do arquivo principal e, posteriormente, a ordenação das codificações e a escrita dos arquivos secundários.

A primeira parte foi tratada como um problema do tipo Produtor-Consumidor (CARISSIMI *et al*., 2010), em que uma *thread* é responsável por ler o conjunto de dados e adicionar os dados em um *buffer* - representado por uma fila circular com alocação estática (TENENBAUM *et al*., 1989) - enquanto outra *thread* retira dados do *buffer* e posteriormente os processa. Para cada uma das colunas categóricas há um *unordered map (*FURTADO*;* SANTOS*, 1979)* responsável por armazenar os pares de informação (um dado do tipo "*string*" como chave e um valor inteiro vinculado). A segunda parte, após o fim do processamento do arquivo principal, cada um dos *unordered maps* gerados é transferido para vetores estáticos e ordenados a partir do valor inteiro com o algoritmo Quicksort (FEOFILOFF, 2009) e posteriormente salvos em seus respectivos arquivos. Para a segunda parte, são

geradas *threads* que ficam responsáveis por cada um dos mapas existentes. Na implementação com *threads* convencionais foram de fato geradas 12 *threads* em todos os testes, na versão em *OpenMP* as *threads* foram geradas conforme a disponibilidade de recursos do sistema e particularidades da biblioteca.

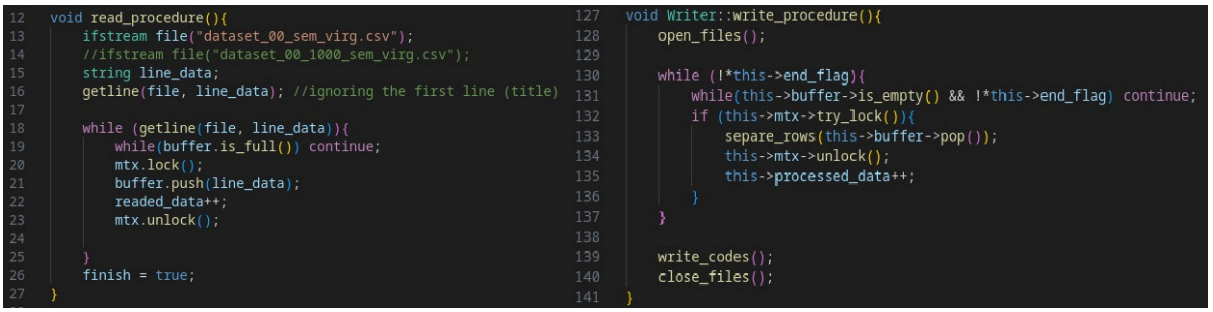
6.0 COMPARAÇÃO ENTRE AS IMPLEMENTAÇÕES

O funcionamento dos programas, apesar de manterem a lógica geral e chegar no mesmo resultado, tiveram algumas alterações para se adequarem ao uso da API de *threads* do C++ e ao *OpenMP.*

6.1 INTEGRIDADE E SINCRONISMO NO BUFFER

O *buffer* é utilizado pelo produtor e consumidor, portanto é necessário cuidados ao acessar o *buffer*. Primeiro, o consumidor deve verificar se há dados disponíveis para consumo, da mesma maneira que o produtor deve verificar se há espaço para armazenar novos valores. Após isso, deve-se garantir que o buffer não é acessado ao mesmo tempo pelas duas threads para não gerar condição de corrida - caso em que o resultado de uma rotina varia em decorrência da ordem e momento das alterações por ser acessado simultaneamente em diferentes partes de um programa paralelo (SILBERSCHATZ *et al.,* 2016). Para isso, na implementação com *threads* convencionais foi utilizado um "*Mutex*" - estrutura de dados que auxilia na gerência de sessões críticas, permitindo apenas um acesso por vez a determinadas partes do código (STALLINGS, 2012) - para a gerência e sincronismo do *buffer*, enquanto a versão com *OpenMP* utilizou-se da cláusula "pragma omp critical" ao incluir ou retirar dados do *buffer*. A figura 1 mostra a utilização de um *Mutex* no consumidor e produtor e a figura 2 mostra a utilização da diretiva do *OpenMP* nas duas situações supracitadas.

Figura 1 - Utilização de um *Mutex* na produção e consumo dos dados.

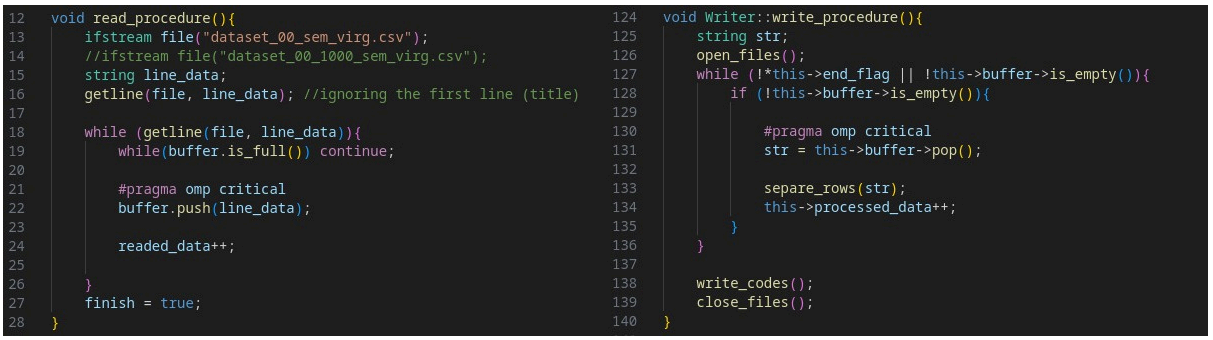


```
12  void read_procedure(){
13      ifstream file("dataset_00_sem_virg.csv");
14      //ifstream file("dataset_00_1000_sem_virg.csv");
15      string line_data;
16      getline(file, line_data); //ignoring the first line (title)
17
18      while (getline(file, line_data)){
19          while(buffer.is_full()) continue;
20          mtx.lock();
21          buffer.push(line_data);
22          readed_data++;
23          mtx.unlock();
24
25      }
26      finish = true;
27  }
```

```
127 void Writer::write_procedure(){
128     open_files();
129
130     while (!*this->end_flag){
131         while(this->buffer->is_empty() && !*this->end_flag) continue;
132         if (this->mtx->try_lock()){
133             separe_rows(this->buffer->pop());
134             this->mtx->unlock();
135             this->processed_data++;
136         }
137     }
138
139     write_codes();
140     close_files();
141 }
```

Fonte: Autores

Figura 2 - Utilização do macro “*critical*” na produção e consumo de dados.



```
void read_procedure(){
    ifstream file("dataset_00_sem_virg.csv");
    //ifstream file("dataset_00_1000_sem_virg.csv");
    string line_data;
    getline(file, line_data); //ignoring the first line (title)

    while (getline(file, line_data)){
        while(buffer.is_full()) continue;

        #pragma omp critical
        buffer.push(line_data);

        readed_data++;

    }
    finish = true;
}
```

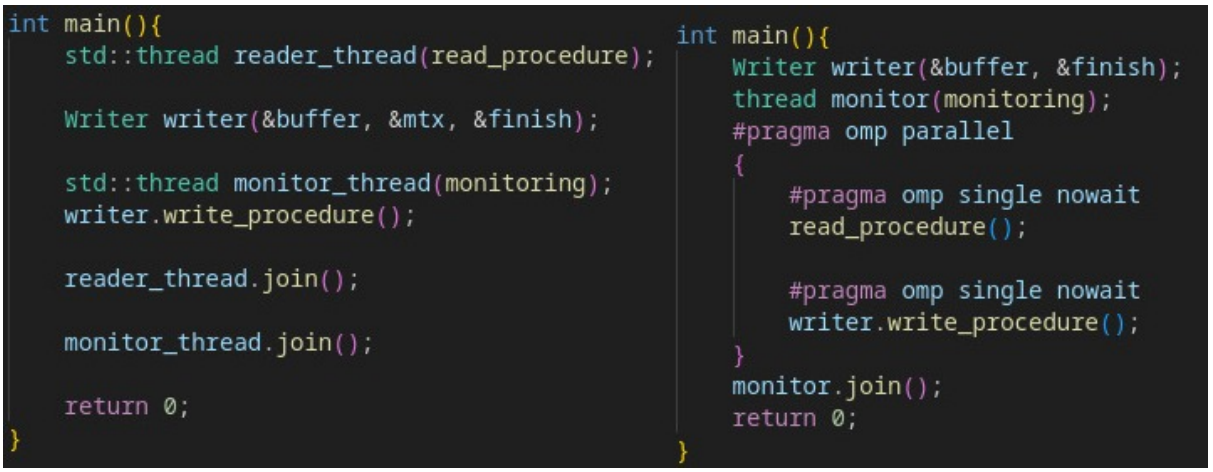
```
void Writer::write_procedure(){
    string str;
    open_files();
    while (!*this->end_flag || !this->buffer->is_empty()){
        if (!this->buffer->is_empty()){

            #pragma omp critical
            str = this->buffer->pop();

            separe_rows(str);
            this->processed_data++;
        }
    }

    write_codes();
    close_files();
}
```

Fonte: Autores

6.2 ALTERAÇÃO NA FUNÇÃO *MAIN*

Enquanto a primeira implementação cria duas *threads* (uma para ler os dados, e uma para acompanhar o andamento do programa e a chamada da função de escrita na *thread* ”*master*”), na segunda implementação foi criada uma thread de forma convencional para mostrar o andamento do programa, enquanto a thread de leitura e a de escrita estão em uma região paralela com a diretiva “*pragma omp parallel*” e cada uma recebe o macro “*pragma omp single nowait*” para serem executadas em threads únicas e fixas sem necessidade de aguardar algum evento. Além disso, a classe “Writer” não possui mais o atributo do tipo *Mutex*. A figura 3 mostra a comparação entre as duas implementações.

Figura 3 - Comparação entre as funções “*main*“



```
int main(){
    std::thread reader_thread(read_procedure);

    Writer writer(&buffer, &mtx, &finish);

    std::thread monitor_thread(monitoring);
    writer.write_procedure();

    reader_thread.join();

    monitor_thread.join();

    return 0;
}
```

```
int main(){
    Writer writer(&buffer, &finish);
    thread monitor(monitoring);
    #pragma omp parallel
    {
        #pragma omp single nowait
        read_procedure();

        #pragma omp single nowait
        writer.write_procedure();
    }
    monitor.join();
    return 0;
}
```

Fonte: Autores

6.3 LOOPS PARALELOS

Na segunda parte, há um *loop* que executa a rotina de ordenação e escrita das informações contidas nos *unordered maps.* Para a primeira implementação, cada iteração gera uma thread e posteriormente outro loop se encarrega de sincronizá-las com a função “*join*”. Na versão *OpenMP* é utilizado apenas um loop com a diretiva “*pragma omp parallel for*”. A figura 4 mostra os loops na primeira versão e a figura 5 na segunda versão.

Figura 4 - *Loops* com *threads* no C++

```
94   void Writer::write_codes(){
95       thread threads[12];
96       for (int i = 0; i < 12; i++){
97           threads[i] = thread(&Writer::write_codes_procedure, this, i);
98       }
99
100      for (int i = 0; i < 12; i++){
101          threads[i].join();
102      }
103  }
```

Fonte: Autores

Figura 5 - *Loops* com *OpenMP*

```
94   void Writer::write_codes(){ //adapted
95       #pragma omp parallel for
96       for (int i = 0; i < 12; i++){
97           write_codes_procedure(i);
98       }
99   }
```

Fonte: Autores

O restante dos códigos não sofreram alterações e podem ser visualizados no link a seguir:
https://drive.google.com/drive/u/1/folders/1-U4wSLNmJ4g8NLYNIjN1vd3bAj9gu-ds

7.0 COMPARAÇÃO DE VELOCIDADE EM DIFERENTES *HARDWARES*

Para comparar a velocidade entre as duas implementações foram utilizadas três máquinas com configurações diferentes. A tabela 1 traz as informações sobre os processadores, memórias e ambientes de execução de cada um dos computadores.

Tabela 1 - Comparação dos *hardwares* utilizados

	Modelo da CPU	**Velocidade do clock**	**Número de núcleos e *threads***	**Memória RAM**	**Velocidade de escrita e leitura do disco e tipo**	**Ambiente de execução**
***Hardware* 1**	Ryzen 5 5600G	3.60Ghz	6 e 12	24GB 3200Mhz DDR4	450 e 500 (MB/s) SSD	Debian 12
***Hardware* 2**	Intel Core i7 9700K	3.60Ghz	8 e 8	16GB 2666Mhz DDR4	156 e 210 (MB/s) HD	Windows Subsystem for Linux 2 (WSL2): Arch Linux
***Hardware* 3**	Intel(R) Xeon(R)	2.20Ghz	1 e 2	12GB (não possui informações sobre tipo e frequência)	Não especificado	Google Colab

Fonte: Autores

Para cada teste, o programa foi executado cinco vezes e posteriormente realizada a média aritmética entre os tempos verificados. A tabela 2 mostra os resultados.

Tabela 2 - Média de tempo nos processamentos

Implementação	**Configuração 1**	**Configuração 2**	**Configuração 3**
***Threads* convencionais**	1 minuto e 35 segundos	6 minutos e 45 segundos	4 minutos e 35 segundos
OpenMP	1 minuto e 5 segundos	1 minuto e 10 segundos	3 minutos e 50 segundos

Fonte: Autores

7.1 LEI DE AMDAHL

Segundo Gene Amdahl, a comparação entre versões paralelas e sequenciais de algoritmos deve levar em consideração tanto a parte do software que continuou sequencial quanto a parte que foi paralelizada, além do número de núcleos utilizados. Dessa forma, como o algoritmo proposto foi paralelizado da mesma forma nas duas versões (primeiramente o trabalho de produção e consumo executados de forma simultânea e posteriormente a ordenação e gravação de dados em paralelo), a variação teórica no tempo de execução seria praticamente igual, porém, na prática foi visto um melhor desempenho com o *OpenMP*, o que mostra uma gerência mais eficiente entre trocas de contexto e gerenciamento de recursos.

8.0 CONCLUSÃO

Após a comparação entre as implementações e os testes realizados, nota-se que a versão feita em *OpenMP* foi mais rápida nas três configurações de *hardware* e a velocidade de leitura e escrita dos discos aparenta ser a principal influência nos resultados. Nesse contexto, a versão escrita em *OpenMP* se mostrou superior. Além de possuir maior simplicidade na codificação (com a gerência, sincronia e construção das *threads* feita de forma automática por parte da biblioteca), ainda houve a utilização da diretiva *critical* que facilitou e simplificou os momentos de lidar com a sessão crítica do *buffer*. Apesar disso, o uso de *threads* convencionais não deve ser descartado no contexto de outros softwares por ter sido mais lento nesse teste, pois ao utilizar *openMP* é necessário adaptar o código para ser funcional com as diretivas, diminuindo a gama de possibilidades ao resolver um determinado problema, sendo assim, necessário uma avaliação prévia para decidir de qual forma paralelizar um software quando, de fato, for possível e necessário.

REFERÊNCIAS

AMDAHL, Gene. **Validity of the Single-Processor Approach to Achieving Large Scale Computing Capabilities. In AFIPS Conference Proceedings**. páginas 483–485. 1967.

CARISSIMI, Alexandre da Silva, OLIVEIRA Rômulo Silva e TOSCANI e Simão Sirineo. **Sistemas Operacionais - Vol. 11: Série Livros Didáticos Informática UFRGS**. Editora Bookman. 2010.

CHANDRA, Rohit, DAGUM, Leonardo, KOHR, Dave, MAYDAN, Dror, MCDONALD, Jeff, MENON, Ramesh. **Parallel Programming in OpenMP**. Morgan Kaufmann Publishers. 2001.

FEOFILOFF, Paulo. **Algoritmos em linguagem C**. Editora Elsevier. 2009.

FURTADO, Antonio Luz e SANTOS, Cleosmar Ribeiro. **Organização de Banco de Dados**. Editora Campus, Rio de Janeiro. 1979.

SILBERSCHATZ, Abraham, GALVIN, Peter Baer e GAGNE Greg. **Sistemas Operacionais com Java - 8º edição**. Editora LTC. 2016.

STALLINGS, William. **Operating Systems - Internals and Design Principles - Seventh Edition.** Prentice Hall. 2012.

TANENBAUM, Andrew e BOS, Herbert. **Sistemas Operacionais Modernos** - 5º edição. Editora Pearson. 2015.

TENENBAUM, Aaron Ai, LANGSAM, Langsam e AUGENSTEIN, Moshe. **Data Structures Using C**. McGraw-Hill, Inc. 1989.

WHITE, Jeff. **FORBES - Why High-Quality And Relevant Data Is Essential In Today's Business Landscape.** 2023. Disponível em: Why High-Quality And Relevant Data Is Essential In Today's Business Landscape (forbes.com)