# Relatório do Trabalho Prático

## INE5645 – Trabalho II: Programação Distribuída

Pedro Santi Binotto [20200634][*1], Gabriel Lemos da Silva [18200628][†1] e Bruno Alexandre Schmitt Costa [17105532][‡1]

[1]Departamento de Informática e Estatística, Universidade Federal de Santa Catarina

11 de julho de 2025

**Resumo**

Este documento busca registrar os detalhes da implementação técnica e casos de teste referentes ao trabalho proposto em "Trabalho 2: Programação Distribuída".

[*]pedro.binotto@grad.ufsc.br
[†]glemoss.dev@gmail.com
[‡]bruno_alexandre.s@hotmail.com

# Sumário



# 1 Introdução

Este relatório apresenta o desenvolvimento do protótipo de um modelo de memória compartilhada distribuída entre múltiplas instâncias de um programa (processos).

## 1.1 Proposta

Conforme especificado no enunciado do projeto, o objetivo central é criar uma abstração inter–processos onde um espaço de endereçamento de memória é logicamente unificado e acessível por múltiplos processos, embora seus blocos constituintes estejam fisicamente distribuídos entre eles (podendo ser em uma mesma máquina física ou não). Além disso, um requisito central da proposta é também a implementação de um mecanismo de *caching* local a cada processo, de forma que seja mantida a coerência de *caches* no nível da rede de processos:

> Operações tanto de leitura quanto de escrita podem acessar blocos locais ou remotos, ou, ainda, locais e remotos. Para acessos remotos, o processo solicitante deve fazer uma cópia do(s) bloco(s) de interesse e mantê-la em uma cache local. Dessa forma, leituras sobre o bloco podem ser feitas pela cache, caso haja uma cópia do bloco de interesse disponível. No caso de leituras e leituras sucessivas, se o processo tiver uma cópia válida do bloco, basta ler da sua cache. No caso de escritas, o processo deve atualizar o valor do bloco no conteúdo do dono do bloco e gerar uma invalidação das cópias daquele bloco em caches de outro processos. Este procedimento é comum na implementação de mecanismos para coerência de cache e chama-se *invalidação na escrita.*

# 2 Solução

Para a elaboração da implementação do projeto, foram adotadas as seguintes ferramentas e tecnologias:

**C++:** A linguagem de programação escolhida para a elaboração do projeto foi o C++, como apresenta algumas facilidades a mais quando comparado com a especificação pura do C (orientação a objetos, etc...) mas mantém a granularidade de controle, da qual nos aproveitamos para modelar a alocação de memória, além de ser facilmente integrado com o sistema de comunicação MPI;

**OpenMPI:** O OpenMPI é uma das mais populares e difundidas implementações da especificação MPI, sendo facilmente instalado na maioria dos sistemas operacionais UNIX-like.

Durante o desenvolvimento, foi realizada pesquisa principalmente nos recursos [2], [3] e [1].

## 2.1 Arquitetura

A arquitetura do protótipo adota um modelo SIMD (Single Instruction, Multiple Data), onde cada processo MPI executa o mesmo código, mas assume papéis distintos com base em seu rank. O sistema é construído sobre a API MPI, utilizando um modelo de passagem de mensagens e *threads* para gerenciar a comunicação e a coerência de cache.

### 2.1.1 Visão Geral

O projeto implementa dois tipos distintos de processo para realizar as soluções de comunicação e compartilhamento de memória:

***Worker Processes*:** São os processos de *rank* $0..N$ (em que $N$ é o parâmetro de `N_PROCS` informado pelo usuário), que tem o papel de gerenciar blocos de memória:

- Mantém localmente os blocos de memória que foram atribuídos à instância;
- Mantém um índice de todos os *maintainers* para os blocos que estão alocados remotamente – ou seja, em outras instâncias – de forma que pode consultar estes blocos através de uma mensagem MPI ao bloco responsável;
- Mantém uma *cache* local dos blocos remotos que já foram acessados, invalidando-a quando recebe uma notificação de atualização de bloco (via *broadcast*);
- Realiza o controle de leitura e escrita sobre os blocos locais, prevenindo condições de corrida e notificando o *broadcaster* quando ocorre uma atualização a um dos blocos locais;

***Broadcaster Process*:** É o processo identificado pelo *rank* $N + 1$, e desempenha o papel de repassar as mensagens de notificação em *broadcast* para todos os processos *worker*:

- Não tem conhecimento do estado da aplicação (blocos alocados), apenas recebe as mensagens de notificação dos processos *worker* (contendo o ID do bloco atualizado e o *timestamp* da operação) e realiza o *broadcast* para os demais processos;

A determinação da relação de *maintainers* para os blocos alocados é dada no *startup* da aplicação, produzindo um mapa estático que perdurará até o fim da execução. O tamanho e número de blocos, assim como o número de *workers* é, da mesma forma, imutável, sendo informado ao processo através dos parâmetros da linha de comando.

A organização pode ser visualizada através das seguintes representações gráficas:

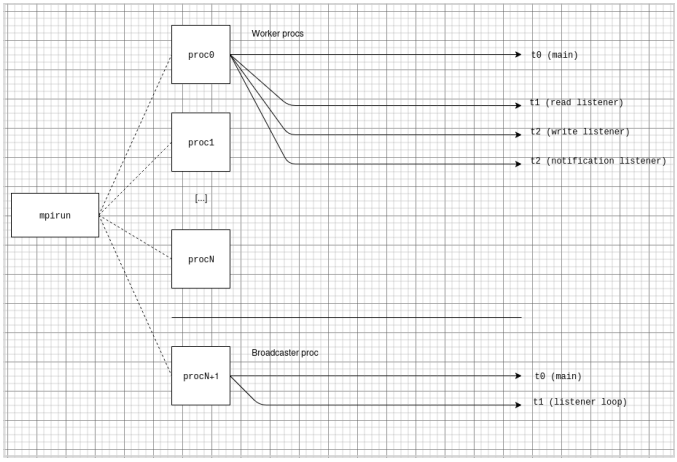


Figura 1: Visão geral da organização dos processos

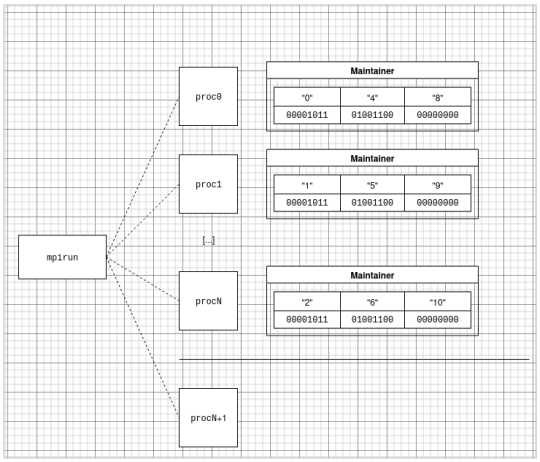


Figura 2: Representação da alocação de blocos

## 2.2 Implementação

### 2.2.1 Acesso aos dados

A implementação interna dos processos descritos até então é realizada através de uma série de componentes e interfaces de forma que, da perspectiva da *thread* principal do processo *worker*, a única API disponível oferece as operações de `read` e `write` da classe de acesso de dados (no caso concreto, através de um *wrapper* desenvolvido para adequar-se à especificação de `le` e `escreve`, descrito na proposta do trabalho), de forma que não é necessário "preocupar-se"com a diferença entre um acesso local e remoto, invalidação de *cache* e demais detalhes que são abstraídos pela classe de repositório e automatizado pelas *threads* auxiliares.

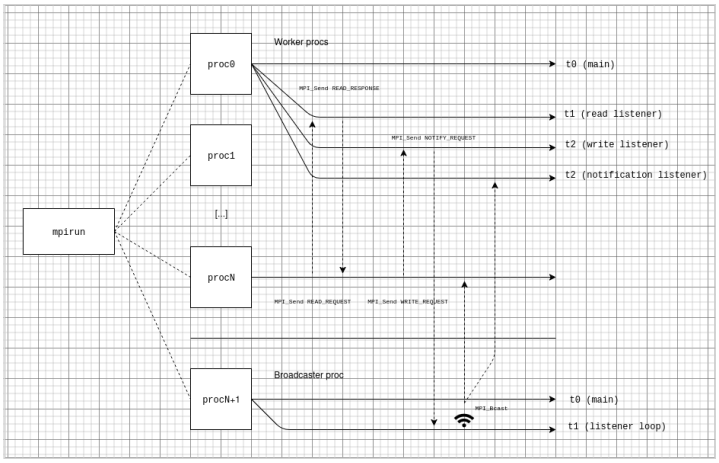


Figura 3: Representação do ciclo de vida dos processos

Listing 1: (`main.cpp`) A classe `UnifiedRepositoryFacade` oferece simples acesso de leitura e escrita à memória compartilhada, implementando os detalhes da lógica de acesso através do MPI e dos blocos locais:

```
int escreve(int posicao, std::shared_ptr<uint8_t[]> buffer, int tamanho) {
  int num_blocks = registry_get(GlobalRegistryIndex::NumBlocks);
  int block_size = registry_get(GlobalRegistryIndex::BlockSize);
  int scoped_blocks = std::ceil(static_cast<double>(tamanho) / block_size);
  int final_pos = posicao + scoped_blocks;

  if (final_pos > num_blocks)
    return 1;

  for (int i = 0; i < scoped_blocks; i++) {
    block new_buf = std::make_shared<uint8_t[]>(block_size);
    std::memcpy(new_buf.get(), buffer.get() + (i * block_size), block_size);
    repository->write(posicao + i, new_buf);
    thread_safe_log_with_id(std::format(
        "Performing␣WRITE␣operation␣to␣block␣{0}␣at␣`main`␣level", posicao));
  }

  return 0;
}

int le(int posicao, std::shared_ptr<uint8_t[]> buffer, int tamanho) {
  int num_blocks = registry_get(GlobalRegistryIndex::NumBlocks);
  int block_size = registry_get(GlobalRegistryIndex::BlockSize);
  int scoped_blocks = std::ceil(static_cast<double>(tamanho) / block_size);
  int final_pos = posicao + scoped_blocks;

  if (final_pos > num_blocks)
    return 1;

  for (int i = 0; i < scoped_blocks; i++) {
    block result = repository->read(posicao + i);
    std::memcpy(buffer.get() + (i * block_size), result.get(), block_size);
```

```
    thread_safe_log_with_id(std::format(
        "Performing␣READ␣operation␣to␣block␣{0}␣at␣`main`␣level", posicao));
  }

  return 0;
}
```

Isso é possível por que a classe utilizada pelo *worker* (`UnifiedRepositoryFacade`) implementa um *proxy* para as classes especializadas `LocalRepository` (mantém os registros locais e realiza o gerenciamento de acessos simultâneos) e `RemoteRepository` (oferece acesso remoto aos demais processos e mantém *caches* locais para operações repetidas):

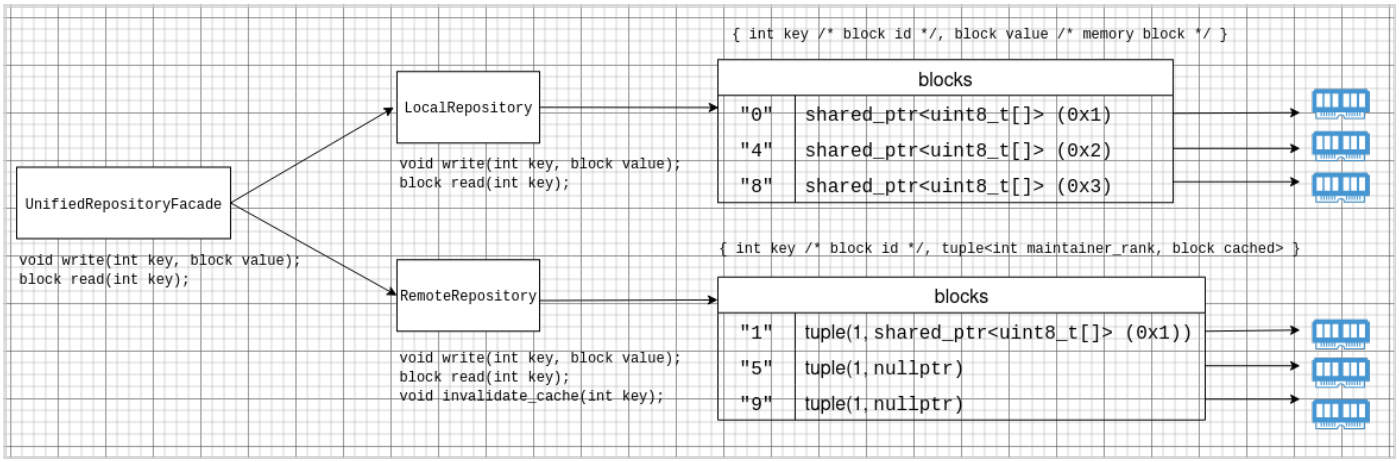


Figura 4: Implementação dos *repositories* (`lib.hpp`)

Assim, a *thread* `main` realiza as operações através desta interface, enquanto as *threads* `read` e `write` recebem e processam requisições dos outros processos (para *leitura* e *escrita* nos blocos mantidos, respectivamente) e a *thread* `notification` escuta por eventos de atualização, que serão recebidos via *broadcast* do processo *broadcaster*. Ao receber um evento de notificação, uma chamada é realizada à instância de `RemoteRepository::invalidate_cache` para que o registro local (*cached*) seja *invalidado* (`shared_ptr<uint_8[]>` (*bytearray*) ⇒ `nullptr` (NULL)).

### 2.2.2 Comunicação e codificação de mensagens

A comunicação entre processos é dividida em dois tipos principais (`lib.hpp`, `servers.hpp`, `servers.cpp`):

**Comunicação Individual (*point–to–point*):** Engloba as mensagens enviadas entre um processo e outro através do MPI (via `MPI_Send` e `MPI_Recv`). Na implementação do sistema, este tipo de comunicação foi empregado nos seguintes cenários:

**Operações de leitura:** As operações de *leitura* remota (de um processo para o bloco mantido por outro) são realizadas através de dois pares send/receive: o processo solicitante envia uma requisição (`MPI_Send` com a *tag* `MESSAGE_TAG_BLOCK_READ_` e o ID do bloco desejado) que será interpretada pela *thread* `read` do processo

remoto, e recebe o *receive* de confirmação. Após isso, o processo *maintainer* do bloco solicitado realiza um `MPI_Send` com a *tag* `MESSAGE_TAG_BLOCK_READ_RESPONSE` e o conteúdo do bloco solicitado;

**Operações de escrita:** As operações de *escrita* remota (de um processo para o bloco mantido por outro) são realizadas através de dois pares send/receive: o processo solicitante envia uma requisição (`MPI_Send` com a *tag* `MESSAGE_TAG_BLOCK_WRITE_REQUEST` e o ID e *bytes* para escrever no bloco desejado) que será interpretada pela *thread* `read` do processo remoto, e recebe o *receive* de confirmação. Após isso, o processo *maintainer* do bloco solicitado realiza um a escrita dos dados e solicita uma notificação de atualização ao processo *broadcaster*;

**Solicitação de notificação:** Quando o processo *maintainer* realiza a escrita a um bloco, este realiza uma chamada de `MPI_Send` ao processo *broadcaster* com a *tag* `MESSAGE_TAG_BLOCK_UPDATE_NOTIFICATION` e o ID do bloco, acompanhado de um *timestamp* UNIX indicativo do momento da escrita do bloco;

**Comunicação Coletiva:** Engloba as mensagens enviadas entre vários processos de uma só vez. Na implementação do sistema, a única primitiva de comunicação coletiva do MPI empregada foi o `MPI_Bcast`, que representa uma comunicação de *um–para–muitos* de uma mesma mensagem. Esta primitiva foi utilizada no seguinte cenário:

**Transmissão de notificação:** Quando o processo *broadcaster* recebe uma requisição de notificação de um processo *worker* (ocorre logo em seguida de uma operação de escrita bem-sucedida), este se encarrega de repassar para todos os *workers* que o bloco informado foi atualizado, para que estes possam invalidar suas *caches*. Isto é implementado através de uma chamada à `MPI_Bcast`, repassando a mensagem (composta do ID e *timestamp* da escrita) em um *broadcast* que será recebido pela *thread* `notification` de cada processo (e ignorado pelo processo que enviou a requisição, por motivos óbvios).

# 3 Casos de teste e verificação

## 3.1 Coerência de cache

O mecanismo de coerência de *cache* e prevenção de acesso inválido (*race–condition*) pode ser verificado ao realizar uma análise cronológica das operações registradas nos arquivos de *log* e comparar os estados em diferentes processos no mesmo ponto no tempo (identificável pela *timestamp* no prefixo de cada linha do log). Assim, é possível realizar cruzamento dos dados ao concatenar (via `cat`) os arquivos de *log* e aplicar uma ordenação pelo *timestamp*, produzindo um documento que representa a ordem dos acontecimentos registrados por diferentes processos conforme a execução do código progride.

Listing 2: Neste trecho de *log* é possível observar a ordem com que os eventos ocorrem entre os diferentes processos em execução (no exemplo, uma requisição de *escrita* para o bloco `2` é enviada do processo `0` para o processo `2`, que realiza a operação e solicita a transmissão da notificação ao processo `4 (broadcaster)`):

```
[1752262034] (@proc 0) Performing WRITE operation to block 2 at `main` level
[1752262034] (@proc 0) Read listener probing...
```

```
[1752262034] (@proc 0) Read thread started
[1752262034] (@proc 0) Received method call to execute remote WRITE operation
   to block 2 with value 01100111 11000110 01101001 01110011 01010001 11111111
    01001010 00000000 on process ID 2
[1752262034] (@proc 0) Sending WRITE request of key: 2, value: 01100111
   11000110 01101001 01110011 01010001 11111111 01001010 00000000 serialized
   as 00000010 00000000 00000000 00000000 01100111 11000110 01101001 01110011
   01010001 11111111 01001010 00000000 over MPI
[1752262034] (@proc 0) Started as worker process
[1752262034] (@proc 0) Started helper threads
[1752262034] (@proc 0) WRITE operation to block 2 called at remote repository
   level
...
[1752262035] (@proc 2) Allocated memory for buffer
[1752262035] (@proc 2) Cached data not available for block 0. Performing remote
    access request...
[1752262035] (@proc 2) Completed READ request from process of ID 1 successfully
   . Sending out response for block 2...
[1752262035] (@proc 2) Completed WRITE request from process of ID {0}
   successfully.
[1752262035] (@proc 2) Constructed object: key 2, value 01100111 11000110
   01101001 01110011 01010001 11111111 01001010 00000000 from raw message
   buffer 00000010 00000000 00000000 00000000 01100111 11000110 01101001
   01110011 01010001 11111111 01001010 00000000
[1752262035] (@proc 2) Copied buffer to memory
[1752262035] (@proc 2) DEBUG: Current local allocated block configuration:
[1752262035] (@proc 2) Decoding write message from bytearray buffer: 00000010
   00000000 00000000 00000000 01100111 11000110 01101001 01110011 01010001
   11111111 01001010 00000000
[1752262035] (@proc 2) Detected READ operation request at `listener` level
[1752262035] (@proc 2) Detected WRITE operation request at `listener` level
[1752262035] (@proc 2) Encoding notification message from of key: 2, timestamp:
    1752262035 as bytearray buffer 00000010 00000000 00000000 00000000
   10010011 01100101 01110001 01101000 00000000 00000000 00000000 00000000
[1752262035] (@proc 2) Processing READ operation request at `handler` level...
[1752262035] (@proc 2) Processing WRITE operation request at `handler` level...
[1752262035] (@proc 2) READ operation to block 0 called at remote repository
   level
[1752262035] (@proc 2) READ operation to block 2 called at local repository
   level
[1752262035] (@proc 2) Read listener probing...
[1752262035] (@proc 2) Sending out update notification request for block 2...
```

## 3.2 Cenários de teste e casos críticos

### 3.2.1 Testes de funcionalidade básica

**Descrição:** Verificar se o sistema inicializa corretamente com diferentes configurações de número de processos, tamanho de blocos e número total de blocos. A distribuição round-robin dos blocos para os processos worker deve ser observada.

**Procedimento:**

- Compilar o projeto: `make build`;
- Executar o sistema com parâmetros padrão (4 processos *worker* + 1 processo *broadcaster*);
- Verificar a distribuição dos blocos entre os processos *worker* nos *logs* iniciais;
- Repetir com diferentes configurações:
  - `make run N_PROCS=4 ARGS="16 8"` (4 workers, 1 broadcaster, blocos de 16 bytes, 8 blocos totais);
  - `make run N_PROCS=6 ARGS="8 16"` (6 workers, 1 broadcaster, blocos de 8 bytes, 16 blocos totais);

**Resultados Esperados:**

- O sistema deve inicializar sem erros em todas as configurações;
- Os *logs* de cada processo (proc-X_output.log) devem mostrar mensagens de inicialização bem-sucedidas;
- O *dump* de memória (se logado) deve indicar que os blocos foram distribuídos corretamente via round-robin entre os processos *worker*;
- `LocalRepository` de cada *worker* deve logar a inicialização de seus blocos proprietários, e `RemoteRepository` deve inicializar suas entradas de *cache* iniciais como `nullptr`;

**Descrição:** Verificar se as operações de `le` e `escreve` funcionam corretamente quando acessando blocos mantidos pelo próprio processo (*worker*).

**Procedimento:**

- Configurar o `Makefile` para `N_PROCS=4`;
- Executar o sistema: `make run ARGS="8 8"` (4 processos *worker*, 8 blocos de 8 `bytes`).
- No `main.cpp` (dentro de `worker_proc`), garantir que haja operações de escrita e leitura em blocos dos quais o processo é proprietário, assim como operações de remotas (leitura e escrita, também; ex: processo rank 0 escreve e lê em bloco 0, bloco 1... bloco N);
- Observar os logs gerados (nível de log 2 para verbosidade máxima).

**Resultados Esperados:**

- As operações de leitura e escrita em blocos locais (`LocalRepository::read`, `LocalRepository:` devem ser bem-sucedidas.
- As operações de leitura e escrita em blocos remotos (`RemoteRepository::read`, `RemoteRepository::write`) devem ser bem-sucedidas.

# 4 Bibliografia

# Referências

[1] cppreference.com contributors. C++ reference. https://cppreference.com/, 2024. Accessed: 2025-07-11.

[2] Wes Kendall et al. Mpi tutorial — parallel programming guide. https://mpitutorial.com/tutorials/. Accessed: 2025-07-11.

[3] Open MPI Project. Open mpi documentation. https://www.open-mpi.org/doc/, 2024. Accessed: 2025-07-11.