

Universidade Federal do Rio
de Janeiro
Instituto de Matemática
Departamento de Ciência da
Computação



MPI I/O

Parte 2

Vinicius Silva
viniciussilva@nce.ufrj.br



Organização

- **MPI I/O – Parte II**
 - **Vistas**
 - **Acesso aos dados**
 - **Resumo das funções**
 - **Offsets explícitos**
 - **Ponteiros individuais**
 - **Ponteiro compartilhado**
 - **Split collective**



Vistas

```
int MPI_File_set_view(MPI_File fh, MPI_Offset disp,  
MPI_Datatype etype, MPI_Datatype filetype, char  
*datarep, MPI_Info info)
```

- A função `MPI_File_set_view` modifica a vista do processo. O início da vista é definido por *disp*. O tipo de dados é definido como *etype*. A distribuição de dados para os processos é definida por *filetype*. A representação dos dados é definida por *datarep*.



Vistas

- Toda chamada a `MPI_File_set_view` reseta os ponteiros individuais e o ponteiro compartilhado para zero.
- `MPI_File_set_view` é uma função coletiva. Os valores para *datarep* e *etype* precisam ser os mesmos para todos os processos no grupo. Os valores de *disp*, *filetype* e *info* podem variar.
- Se o modo de acesso (*amode*) utilizado na abertura do arquivo tiver sido `MPI_MODE_SEQUENTIAL`, o deslocamento `MPI_DISPLACEMENT_CURRENT` deve ser passado em *disp*. Isto define o deslocamento para a posição atual do ponteiro compartilhado.



Vistas

- O deslocamento *disp* pode ser utilizado para saltar cabeçalhos ou quando o arquivo inclui uma sequência de segmentos de dados que devem ser acessados em diferentes padrões. Vistas separadas, cada uma utilizando valores distintos de *disp* e *filetype*, podem ser utilizadas para acessar cada um dos segmentos.

first view 

second view 

file structure:




first displacement


second displacement

Vistas

- Um *filetype* pode ser tanto um único *etype* como um tipo derivado construído através de múltiplas instâncias do mesmo *etype*.
- O tamanho dos *holes* em um *filetype* deve ser um múltiplo do *etype*.
- Esses deslocamentos não necessariamente devem ser distintos, mas não podem ser negativos.
- Se um arquivo for aberto para escrita, nem o *etype* nem o *filetype* poderão conter regiões de sobreposição no mesmo processo. *Filetypes* de processos diferentes ainda podem se sobrepor.



Vistas

- Se o *filetype* possui *holes*, os dados que estiverem nos *holes* não são acessíveis ao processo associado ao *filetype*. No entanto, os valores de *disp*, *etype* e *filetype* podem ser alterados por uma nova chamada a `MPI_File_set_view` para permitir acesso a diferentes partes do arquivo.
- É incorreto utilizar endereços absolutos na construção de um *etype* ou de um *filetype*.



Vistas

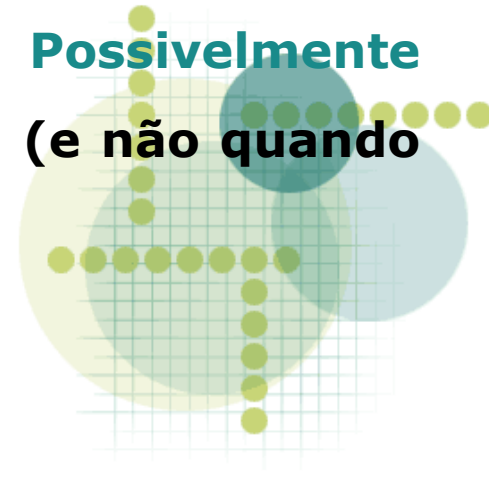
- O programador é responsável por garantir que todas as operações não-bloqueantes e operações split collectives de um determinado file handle serão completadas antes de uma nova chamada a `MPI_File_set_view`.



Acesso aos Dados

- **Posicionamento**

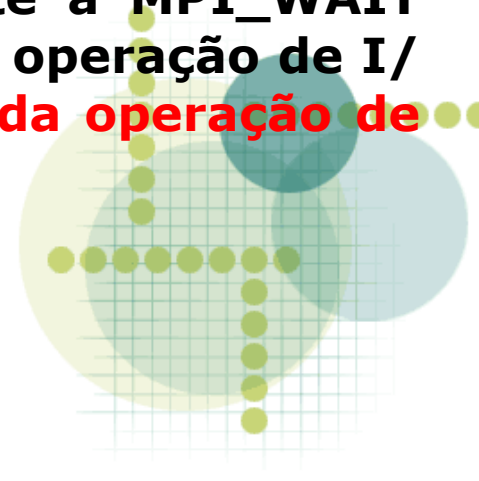
- O acesso pode ser feito através de offsets explícitos, ponteiros individuais ou ponteiro compartilhado.
- Cada operação de I/O deixa o ponteiro apontando para o próximo *etype* depois do último que foi acessado pela operação. Em uma operação não-bloqueante, o ponteiro é atualizado pela função que inicia o I/O. Possivelmente isso ocorre **assim que a função é chamada** (e não quando ela termina).



Acesso aos Dados

- Sincronismo

- As operações podem ser bloqueantes, não-bloqueantes ou *split* (uma forma de acesso não bloqueante em duas fases).
- Uma função de I/O bloqueante retorna apenas quando a requisição de I/O termina. Isso significa que os dados foram enviados para o buffer do SO.
- Uma função não-bloquante retorna imediatamente após ser chamada. Uma chamada independente a `MPI_WAIT` ou `MPI_TEST` é necessária para terminar a operação de I/O. **É um erro utilizar o buffer local antes da operação de I/O não-bloqueante terminar.**



Acesso aos Dados

- **Coordenação**

- **As operações podem ser coletivas ou não-coletivas.**
- **O término de uma operação não-coletiva depende apenas do processo que a chamou.**
- **O término de uma operação coletiva pode depender da atividade dos outros processos participantes.**



Funções Básicas

- A tabela a seguir resume as funções do MPI I/O, separando-as pelo tipo de posicionamento de ponteiro (offset explícito, ponteiros individuais e ponteiro compartilhado), sincronismo (bloqueante e não-bloqueante) e coordenação (operações não-coletivas e coletivas).
- No total são 30 funções, sendo 15 de leitura e 15 de escrita.



Funções Básicas

Posicionamento	Sincronismo	Coordenação	
		não-coletivas	coletivas
offsets explícitos	bloqueante	MPI_FILE_READ_AT MPI_FILE_WRITE_AT	MPI_FILE_READ_AT_ALL MPI_FILE_WRITE_AT_ALL
	não-bloqueante e split	MPI_FILE_IREAD_AT MPI_FILE_IWRITE_AT	MPI_FILE_READ_AT_ALL_BEGIN MPI_FILE_READ_AT_ALL_END MPI_FILE_WRITE_AT_ALL_BEGIN MPI_FILE_WRITE_AT_ALL_END
ponteiros individuais	bloqueante	MPI_FILE_READ MPI_FILE_WRITE	MPI_FILE_READ_ALL MPI_FILE_WRITE_ALL
	não-bloqueante e split	MPI_FILE_IREAD MPI_FILE_IWRITE	MPI_FILE_READ_ALL_BEGIN MPI_FILE_READ_ALL_END MPI_FILE_WRITE_ALL_BEGIN MPI_FILE_WRITE_ALL_END
ponteiros compartilhados	bloqueante	MPI_FILE_READ_SHARED MPI_FILE_WRITE_SHARED	MPI_FILE_READ_ORDERED MPI_FILE_WRITE_ORDERED
	não-bloqueante e split	MPI_FILE_IREAD_SHARED MPI_FILE_IWRITE_SHARED	MPI_FILE_READ_ORDERED_BEGIN MPI_FILE_READ_ORDERED_END MPI_FILE_WRITE_ORDERED_BEGIN MPI_FILE_WRITE_ORDERED_END

Offsets Explícitos

- Se o arquivo for aberto com o modo de acesso (*amode*) **MPI_MODE_SEQUENTIAL**, as chamadas as funções que utilizam offsets explícitos retornarão erro.



Offsets Explícitos

- **MPI_File_read_at()**

```
int MPI_File_read_at(MPI_File fh, MPI_Offset offset,  
void *buf, int count, MPI_Datatype datatype,  
MPI_Status *status);
```

- Lê **count** valores, a partir de **offset**, do tipo **datatype** do arquivo associado a **fh** para **buf**. O buffer **buf** deve poder armazenar pelo menos tantos valores quanto **count*sizeof(type)**.
- **status** pode ser utilizado para monitorar quantos bytes foram lidos até o momento.



Offsets Explícitos

- **MPI_File_iread_at()**

```
int MPI_File_iread_at(MPI_File fh, MPI_Offset offset,  
void *buf, int count, MPI_Datatype datatype,  
MPI_Request *request);
```

- Versão não-bloqueante de MPI_File_read_at().
- *request* deve ser utilizado com MPI_WAIT() ou MPI_TEST() para esperar ou testar o término do I/O.

Offsets Explícitos

- **MPI_File_read_at_all()**

```
int MPI_File_read_at_all(MPI_File fh, MPI_Offset  
offset, void *buf, int count, MPI_Datatype datatype,  
MPI_Status *status);
```

- Versão coletiva de MPI_File_read_at().



Offsets Explícitos

- `MPI_File_write_at()`

```
int MPI_File_write_at(MPI_File fh, MPI_Offset offset,  
void *buf, int count, MPI_Datatype datatype,  
MPI_Status *status);
```

- Lê *count* valores, a partir de *offset*, do tipo *datatype* do buffer *buf* para o arquivo associado a *fh*. O buffer *buf* deve poder armazenar pelo menos tantos valores quanto *count*sizeof(type)*.
- *status* pode ser utilizado para monitorar quantos bytes foram escritos até o momento.

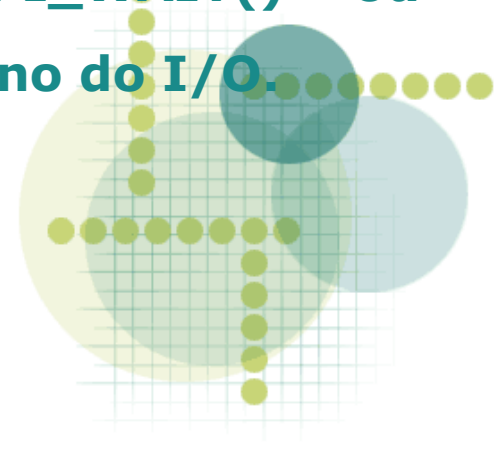


Offsets Explícitos

- **MPI_File_iwrite_at()**

```
int MPI_File_iwrite_at(MPI_File fh, MPI_Offset offset,  
void *buf, int count, MPI_Datatype datatype,  
MPI_Request *request);
```

- Versão não-bloqueante de MPI_File_write_at().
- *request* deve ser utilizado com MPI_WAIT() ou MPI_TEST() para esperar ou testar o término do I/O.



Offsets Explícitos

- Exemplo de uma operação de I/O não-bloqueante:

```
MPI_Request request;
```

```
MPI_Status status;
```

```
MPI_File fh;
```

```
float data[100];
```

```
/* assumes an open file and set view */
```

```
MPI_File_iread (fh, data, 100, MPI_FLOAT, &request);
```

```
/* do work */
```

```
MPI_Wait (&request, &status);
```

```
/* now safe to use data[100] */
```



Offsets Explícitos

- **MPI_File_write_at_all()**

```
int MPI_File_write_at_all(MPI_File fh, MPI_Offset  
offset, void *buf, int count, MPI_Datatype datatype,  
MPI_Status *status);
```

- Versão coletiva de MPI_File_write_at().



Ponteiros Individuais

- O MPI mantém um ponteiro individual por processo por *file handle*.
- As funções que utilizam ponteiros individuais são semanticamente equivalentes as que usam offsets explícitos, com apenas uma modificação:
 - ***O offset é definido como a posição atual do ponteiro.***
- Após uma operação de I/O ser concluída (se for bloqueante, ao seu término; se for não-bloqueante, pode ocorrer antes do término), o ponteiro é atualizado para apontar para o próximo *etype* após o último acessado.

Ponteiros Individuais

- O ponteiro compartilhado, se existir, não é utilizado nem alterado pelas chamadas a estas funções.
- Assim como no caso das funções que utilizam offsets explícitos, se o arquivo for aberto com o modo de acesso (*amode*) **MPI_MODE_SEQUENTIAL**, as funções retornarão erro.
- O ponteiro é relativo a vista corrente do arquivo.

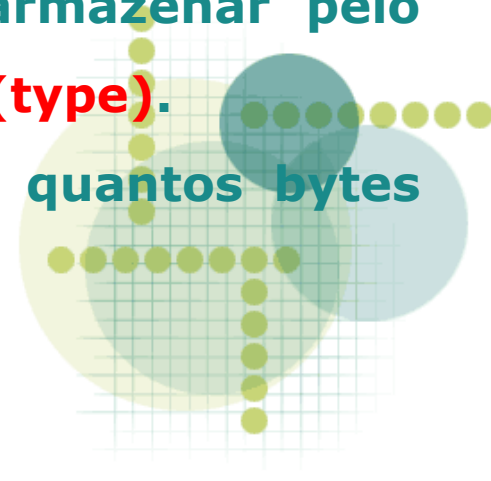


Ponteiros Individuais

- `MPI_File_read()`

```
int MPI_File_read(MPI_File fh, void *buf, int count,  
MPI_Datatype type, MPI_Status *status);
```

- Lê *count* valores do tipo *datatype* do arquivo associado a *fh* para *buf*. O buffer *buf* deve poder armazenar pelo menos tantos valores quanto $\text{count} * \text{sizeof}(\text{type})$.
- *status* pode ser utilizado para monitorar quantos bytes foram lidos até o momento.

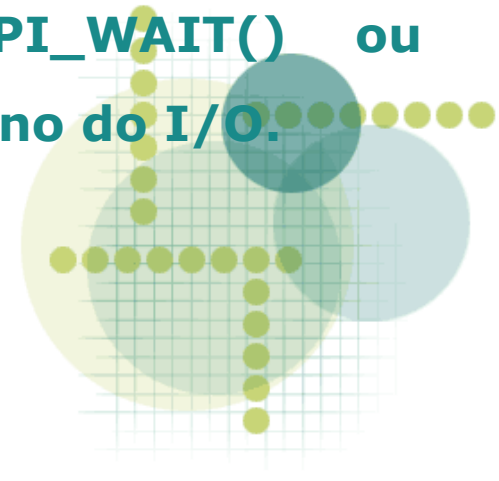


Ponteiros Individuais Não Bloqueantes

- `MPI_File_iread()`

```
int MPI_File_iread(MPI_File fh, void *buf, int count,  
MPI_Datatype datatype, MPI_Request *request);
```

- Versão não-bloqueante de `MPI_File_read()`.
- ***request*** deve ser utilizado com `MPI_WAIT()` ou `MPI_TEST()` para esperar ou testar o término do I/O.



Ponteiros Individuais Coletivas

- **MPI_File_read_all()**

```
int MPI_File_read_all(MPI_File fh, void *buf, int count,  
MPI_Datatype datatype, MPI_Status *status);
```

- Versão coletiva de MPI_File_read().



Ponteiros Individuais

- `MPI_File_write()`

```
int MPI_File_write(MPI_File fh, void *buf, int count,  
MPI_Datatype type, MPI_Status *status);
```

- Lê *count* valores do tipo *datatype* de *buf* para arquivo associado a *fh*. O buffer *buf* deve poder armazenar pelo menos tantos valores quanto *count*sizeof(type)*.
- *status* pode ser utilizado para monitorar quantos bytes foram escritos até o momento.

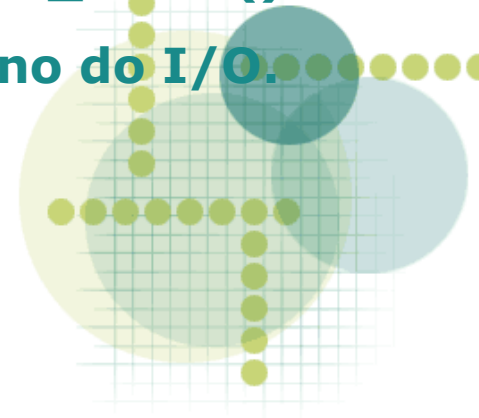


Ponteiros Individuais Não Bloqueantes

- **MPI_File_iwrite()**

```
int MPI_File_iwrite(MPI_File fh, void *buf, int count,  
MPI_Datatype datatype, MPI_Request *request);
```

- Versão não-bloqueante de MPI_File_write().
- *request* deve ser utilizado com MPI_WAIT() ou MPI_TEST() para esperar ou testar o término do I/O.



Ponteiros Individuais Coletivas

- **MPI_File_write_all()**

```
int MPI_File_write_all(MPI_File fh, void *buf, int count,  
MPI_Datatype datatype, MPI_Status *status);
```

- Versão coletiva de MPI_File_write().



Ponteiros Individuais

- **MPI_File_seek()**

```
int MPI_File_seek(MPI_File fh, MPI_Offset offset, int  
whence);
```

- **MPI_File_seek()** atualiza o valor do ponteiro individual.
- **offset** determina o deslocamento do ponteiro a partir da posição atual. Esse valor pode ser negativo, embora deslocar o ponteiro para posições anteriores ao início do arquivo (ou da vista) gere um erro.



Ponteiros Individuais

- **whence** determina como o ponteiro será atualizado:
 - **MPI SEEK SET** – o ponteiro é atualizado para offset.
 - **MPI SEEK CUR** – o ponteiro é atualizado para a posição atual + offset.
 - **MPI SEEK END** – o ponteiro é atualizado para o final do arquivo + offset.
- Os deslocamentos são realizados em termos do tipo de dados atual (**MPI_BYTE** por padrão).



Ponteiros Individuais

- **MPI_File_get_position()**

int MPI_File_get_position(MPI_File fh, MPI_Offset *offset)

- MPI_File_get_position() retorna, em **offset**, a posição atual do ponteiro individual (relativo à vista) em unidades de *etypes*.
- Para definir o deslocamento (*displacement*) na posição atual do ponteiro, converta o offset para um valor absoluto utilizando MPI_File_get_byte_offset() e então utilize MPI_File_set_view para atualizar o deslocamento.

Ponteiros Individuais

- **MPI_File_get_byte_offset()**

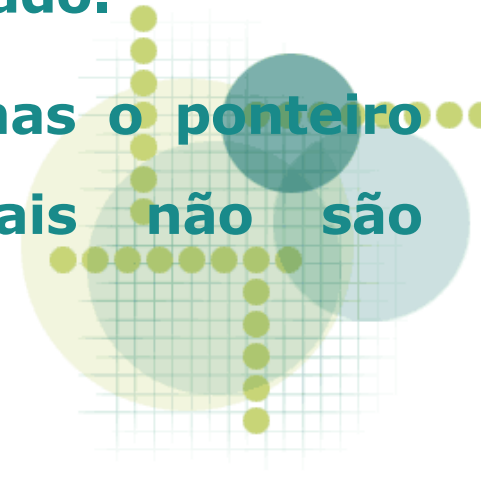
```
int MPI_File_get_byte_offset(MPI_File fh, MPI_Offset  
offset, MPI_Offset *disp)
```

- **MPI_File_get_byte_offset()** converte o valor do offset relativo à vista em uma posição absoluta medida em bytes.

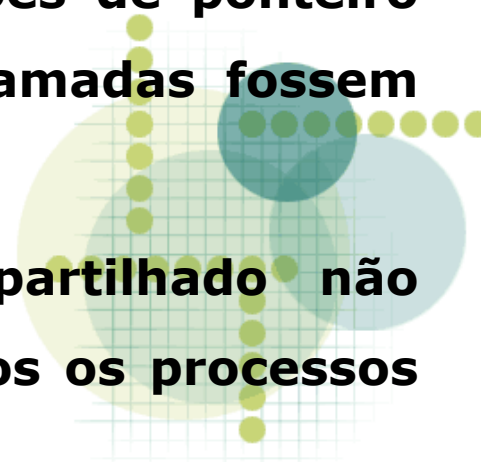


Ponteiros Compartilhados

- O MPI mantém um único ponteiro compartilhado por chamada coletiva a `MPI_File_open()` que é compartilhado entre todos os processos do comunicador.
- O valor deste ponteiro determina implicitamente o *offset* das funções de ponteiro compartilhado.
- Essas funções utilizam e atualizam apenas o ponteiro compartilhado. Os ponteiros individuais não são afetados.



Ponteiros Compartilhados

- **As funções que utilizam ponteiro compartilhado são semanticamente equivalentes as que usam offsets explícitos, com as seguintes modificações:**
 - **O offset é definido implicitamente pelo valor atual do ponteiro compartilhado.**
 - **O efeito de múltiplas chamadas às funções de ponteiro compartilhado é definido como se as chamadas fossem serializadas.**
 - **O uso das funções de ponteiro compartilhado não funciona corretamente a não ser que todos os processos utilizem a mesma vista.**
- 

Ponteiros Compartilhados

- Para funções de ponteiro compartilhado não-coletivas a ordem da serialização é não-determinística. O programador precisa prover outros meios de sincronização para especificar uma ordem.
- Para funções coletivas compartilhadas, o acesso ao arquivo irá ocorrer na ordem determinada pelo rank dos processos no grupo. Para cada processo, a localização no arquivo é a posição na qual o ponteiro estará depois de todos os ranks com valores menores do que seu rank terem acessado o arquivo.



Ponteiros Compartilhados

- Após uma operação de I/O ser concluída (se for bloqueante, ao seu término; se for não-bloqueante, pode ocorrer antes do término), o ponteiro é atualizado para apontar para o próximo etype após o último acessado.



Ponteiros Compartilhados

- **MPI_File_read_shared()**

int MPI_File_read_shared(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)

- MPI_File_read_shared() é a operação de leitura não-coletiva utilizando ponteiro compartilhado.
- Equivalente a MPI_File_read, i.e., lê **count** valores do tipo **datatype** do arquivo associado a **fh** para **buf**. O buffer **buf** deve poder armazenar pelo menos tantos valores quanto **count*sizeof(type)**.



Ponteiros Compartilhados

- **MPI_File_iread_shared()**

int MPI_File_iread_shared(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Request *request)

- **MPI_File_iread_shared()** é a operação não-bloqueante de leitura não-coletiva utilizando ponteiro compartilhado.



Ponteiros Compartilhados

- **MPI_File_write_shared()**

int MPI_File_write_shared(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)

- MPI_File_write_shared() é a operação de escrita não-coletiva utilizando ponteiro compartilhado.
- Equivalente a MPI_File_write, i.e., lê **count** valores do tipo **datatype** do buffer **buf** para arquivo associado a **fh**. O buffer **buf** deve poder armazenar pelo menos tantos valores quanto **count*sizeof(type)**.



Ponteiros Compartilhados

- **MPI_File_ fwrite_shared()**

int MPI_File_ fwrite_shared(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Request *request)

- **MPI_File_ fwrite_shared()** é a operação não-bloqueante de escrita não-coletiva utilizando ponteiro compartilhado.



Ponteiros Compartilhados

- **MPI_File_read_ordered()**

int MPI_File_read_ordered(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)

- **MPI_File_read_ordered()** é a operação de leitura coletiva utilizando ponteiro compartilhado.



Ponteiros Compartilhados

- **MPI_File_write_ordered()**

int MPI_File_write_ordered(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)

- **MPI_File_write_ordered()** é a operação de escrita coletiva utilizando ponteiro compartilhado.



Operações Coletivas Particionadas

- O MPI provê uma forma restrita de operações de I/O coletivas não-bloqueantes. Essas rotinas são denominadas "split collective" pois uma única operação coletiva é dividida em duas: uma função de início (*begin*) e uma função de fim (*end*).
- A função de início começa a operação, como um acesso não-bloqueante. A função de fim termina a operação, como uma rotina de *test* ou *wait*.
- Como qualquer outra operação não-bloqueante, o usuário não deve utilizar o buffer associado a ela enquanto a operação não terminar.



Operações Coletivas Particionadas

- Operações "split collective" sobre um file handle *fh* estão sujeitas as seguintes regras:
 - Em um processo MPI, cada file handle pode estar associado a, no máximo, 1 operação split ao mesmo tempo.
 - Funções de início e fim operam sobre o grupo de processos que abriu coletivamente o arquivo através de MPI_File_open e seguem as regras de ordenação para chamadas coletivas.



Operações Coletivas Particionadas

- Cada função de término (*end*) está associada a uma função de início (*begin*) precedente para a mesma operação coletiva.
- Quando uma operação de término é feita, exatamente 1 operação de início para a mesma operação deve estar ativa.
- Operações split não estão associadas com as suas versões correspondentes não-split. Por exemplo, uma chamada a `MPI_File_read_all` em um processo não está associada a alguma chamada a `MPI_File_read_all_begin` ou `MPI_File_read_all_end`.

Operações Coletivas

Particionadas

- Operações split devem especificar um buffer tanto nas funções de início quanto de fim.
- Nenhuma operação coletiva de I/O é permitida em um file handle concorrentemente a uma operação split (isto é, entre o begin e o end). Ou seja:

```
MPI_File_read_all_begin(fh, ...);
```

```
...
```

```
MPI_File_read_all(fh, ...);
```

```
...
```

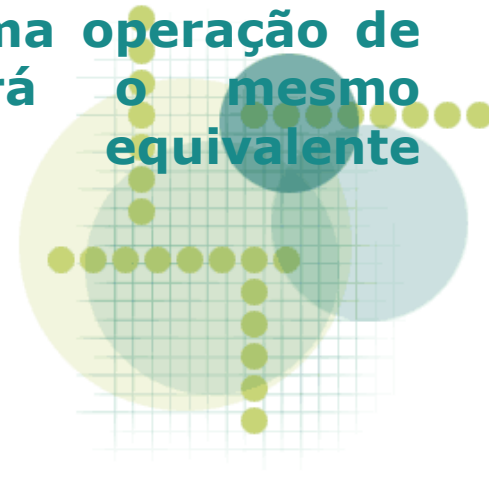
```
MPI_File_read_all_end(fh, ...);
```

está incorreto.



Operações Coletivas Particionadas

- Em ambientes *multithreaded*, as operações de início e fim devem ser chamadas pela mesma *thread*.
- Note que não é possível que duas ou mais *threads* iniciem uma função coletiva no mesmo file handle, já que somente uma operação split pode estar ativa no mesmo file handle.
- Os argumentos para essas funções têm o mesmo significado nas versões coletivas equivalentes. Uma rotina de início (`MPI_File_read_all_begin`) começa uma operação que, quando completada por uma operação de fim (`MPI_File_read_all_end`), produzirá o mesmo resultado da versão coletiva equivalente (`MPI_File_read_all`).



Operações Coletivas Particionadas

- **MPI_File_read_at_all_begin()**

```
int      MPI_File_read_at_all_begin(MPI_File      fh,  
MPI_Offset offset, void *buf, int count, MPI_Datatype  
datatype);
```

- Função de início equivalente a MPI_File_read_at_all().
- Inicia a leitura coletiva utilizando offsets explícitos.



Operações Coletivas Particionadas

- **MPI_File_read_at_all_end()**

```
int MPI_File_read_at_all_end(MPI_File fh, void *buf,  
MPI_Status *status);
```

- Função de término equivalente a MPI_File_read_at_all().
- Finaliza a leitura.

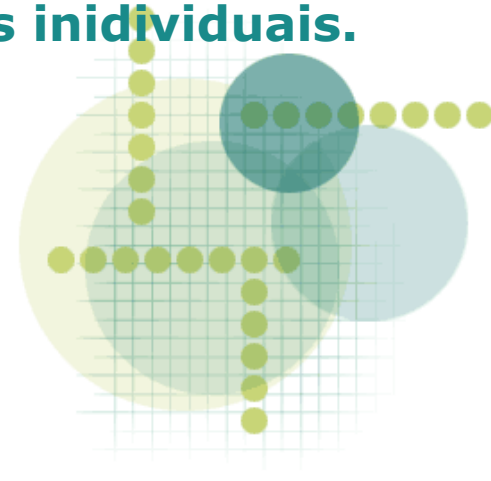


Operações Coletivas Particionadas

- **MPI_File_read_all_begin()**

int MPI_File_read_all_begin(MPI_File fh, void *buf, int count, MPI_Datatype datatype);

- Função de início equivalente a MPI_File_read_all().
- Inicia a leitura coletiva utilizando ponteiros individuais.



Operações Coletivas Particionadas

- **MPI_File_read_all_end()**

```
int MPI_File_read_all_end(MPI_File fh, void *buf,  
MPI_Status *status);
```

- Função de término equivalente a MPI_File_read_all().
- Finaliza a leitura coletiva utilizando ponteiros individuais.



Operações Coletivas Particionadas

- **MPI_File_read_ordered_begin()**

```
int MPI_File_read_ordered_begin(MPI_File fh, void  
*buf, int count, MPI_Datatype datatype);
```

- Função de início equivalente a MPI_File_read_ordered().
- Inicia a leitura coletiva utilizando ponteiro compartilhado.



Operações Coletivas Particionadas

- **MPI_File_read_ordered_end()**

```
int MPI_File_read_ordered_end(MPI_File fh, void *buf,  
MPI_Status *status);
```

- Função de término equivalente a
MPI_File_read_ordered().

- Finaliza a leitura coletiva utilizando ponteiro
compartilhado.



Operações Coletivas Particionadas

- **MPI_File_write_at_all_begin()**

int MPI_File_write_all_begin(MPI_File fh, void *buf, int count, MPI_Datatype datatype);

- Função de início equivalente a MPI_File_write_at_all().
- Inicia a escrita coletiva utilizando offsets explícitos.



Operações Coletivas Particionadas

- **MPI_File_write_at_all_end()**

**int MPI_File_write_at_all_end(MPI_File fh, void *buf,
MPI_Status *status)**

- Função de término equivalente a MPI_File_write_at_all().
- Finaliza a escrita.



Operações Coletivas Particionadas

- **MPI_File_write_all_begin()**

int MPI_File_write_all_begin(MPI_File fh, void *buf, int count, MPI_Datatype datatype);

- Função de início equivalente a MPI_File_write_all().
- Inicia a escrita coletiva utilizando ponteiros individuais.



Operações Coletivas Particionadas

- **MPI_File_write_all_end()**

```
int MPI_File_write_all_end(MPI_File fh, void *buf,  
MPI_Status *status);
```

- Função de término equivalente a MPI_File_write_all().
- Finaliza a escrita coletiva utilizando ponteiros individuais.



Operações Coletivas Particionadas

- **MPI_File_write_ordered_begin()**

```
int MPI_File_write_ordered_begin(MPI_File fh, void  
*buf, int count, MPI_Datatype datatype);
```

- Função de início equivalente a MPI_File_write_ordered().
- Inicia a escrita coletiva utilizando ponteiro compartilhado.



Operações Coletivas Particionadas

- **MPI_File_write_ordered_end()**

```
int MPI_File_write_ordered_end(MPI_File fh, void *buf,  
MPI_Status *status);
```

- Função de término equivalente a
MPI_File_write_ordered().

- Finaliza a escrita coletiva utilizando ponteiro
compartilhado.



Operações Coletivas Particionadas

- Exemplo de uma operação de I/O não-bloqueante:

```
MPI_File fh;
```

```
MPI_Status status;
```

```
float data[100];
```

```
/* assumes an open file and set view */
```

```
MPI_File_write_all_begin (fh, data, 100, MPI_FLOAT);
```

```
/* do work */
```

```
/* no other collective operations on this file */
```

```
/* no use of data[100] */
```

```
MPI_File_write_all_end (fh, data, &status);
```

```
/* now safe to use data[100] */
```

