UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS DE QUIXADÁ

Programando para Memória Distribuída com MPI

Prof. João Marcelo Uchôa de Alencar

joao.marcelo@ufc.br

Introdução

- Computadores distribuídos não podem se comunicar por memória compartilhada:
  - Mensagens, pela rede, são usadas para coordenar tarefas paralelas em computadores geograficamente distribuídos.
  - Cada máquina terá um ou mais processos parte da aplicação paralela.
- *Message Passing Interface* (MPI)
  - Especificação que trata da criação, gerência e comunicação entre processos distribuídos de uma aplicação paralela.
  - Operações básicas que podem ser combinadas de maneira complexa.

Aplicações Paralelas em Computadores de Memória Distribuída

- O preço da comunicação:
  - O intervalo de tempo necessário para trocar dados entre processos.
  - Na memória compartilhada, é a latência da memória principal no melhor caso, ou a sobrecarga da sincronização (amenizada pela redução, etc).
  - Na memória distribuída, latência da rede, ordens de grandeza maior, porém a sincronização é **explícita**.
- A topologia da rede tem impacto no tempo exigido pela troca de mensagens.

Message Passing Interface (MPI)

- Padrão bem estabelecido e aceito (até mais que o OpenMP):
  - Funções para criação e gestão de processos.
  - Suporte a C/C++/Fortran.
  - Comunicação ponto-a-ponto e coletivas.
  - Comunicação por grupo.
- Em constante evolução, adições recentes:
  - RDMA.
  - Topologia de processos.
  - Operações coletivas estendidas.

Message Passing Interface (MPI)

- Questões que o MPI auxilia o programador a tratar:
  - Interdependência dos dados.
  - Tipo e frequência da comunicação.
  - Balanceamento de carga entre os processos.
  - Intercalação de comunicação e computação.
  - Fluxo de programa síncrono ou assíncrono.
  - Critérios de parada.
- Entretanto, cabe ao desenvolver maturidade em cada funcionalidade do MPI para poder fazer o uso correto.

Message Passing Interface (MPI)

- A padronização começou na década de 90.
- É uma **especificação**, não uma **implementação**.
  - As operações são especificadas como funções, sub-rotinas ou métodos.
  - Linguagens C/C++/Fortran (oficialmente).
  - Sintática e semântica das operações.
- Programa MPI:
  - Processos autônomos que executam no paradigma MIMD (*Multiple Instruction Multiple Data*).
  - Cada processo pode ter código diferente.
  - Na prática, todos usam o mesmo código, mas cada um execução de forma independente, dificilmente estão na mesma instrução ou sentença em um dado momento.

Message Passing Interface (MPI)

- Os processos MPI se comunicam através das operações de troca de mensagens.
- Essas operações independem do sistema operacional.
- Existem implementações do MPI para os mais diversos sistemas.
- Código feito para uma implementação (SO) pode ser executado em outra, desde que seja compilado novamente.
- Para *execução* em si, todas as máquinas devem usar a mesma implementação.

Message Passing Interface (MPI)

- Estrutura básica do programa MPI:
  - Operações para inicializar o ambiente de execução.
  - Operações de comunicação intercaladas pela execução de computação.
  - Operações para finalizar processos.
- Organização dos processos distribuídos:
  - Processos organizados em grupos de tamanho fixo específico.
  - A cada grupo, um objeto comunicador permite a troca de mensagens.
  - Cada processo em um *rank* dentro do grupo.
  - MPI_COMM_WORLD: comunicador padrão que engloba todos os processos.

Message Passing Interface (MPI)

- Quantos processos há em uma aplicação? Quais são seus identificadores?
- Operações básicas:
  - MPI_SEND.
  - MPI_RECV.
- Além da troca de dados, essas operações “forçam” a sincronização.
- Versões do MPI:
  - MPI-1: 120 operações.
  - MPI-2: adicionou operações para gerência dinâmica de processos.
  - MPI-3: operações coletivas não bloqueantes.
- Vamos começar vendo as básicas.

Olá Mundo em MPI

```
#include "mpi.h"
#include <stdio.h>

int main(int argc, char *argv[]) {
  int  numtasks, rank, len, rc;
  char hostname[MPI_MAX_PROCESSOR_NAME];

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  MPI_Get_processor_name(hostname, &len);

  printf ("Número de processos: %d Meu rank: %d Executando em %s\n",
        numtasks, rank, hostname);

  MPI_Finalize();
  return 0;
}
```

- O prefixo *MPI_* é faz parte do nome de toda operação.
- Cabeçalho “mpi.h”.
- Operações:
  - Inicializa o ambiente.
  - Recupera a quantidade de processos (*numtasks*).
  - Recupera o identificador do processo (*rank*).
  - Finaliza a execução.

Sintaxe das Operações do MPI

- As operações MPI costumam receber vários argumentos:
  - Argumentos de leitura (IN) não são alterados, apenas lidos. Vamos representa-los no texto com escrita normal: buf, sendbuf, MPI_COMM_WORLD, etc.
  - Argumentos de escrita (OUT), são apenas atualizados. Serão sublinhados: rank, recbuf, etc.
  - Argumentos de leitura e escrita, podem ser lidos ou atualizados. Sublinhados e em itálico: *inbuf*, *request*, etc.
- Os exemplos em C tem as seguintes convenções:
  - Prefixo *MPI_* e primeira letra em maiúsculo: MPI_Init, MPI_Send, etc.
  - Cada operação retorna MPI_SUCCESS (definida em mpi.h) ou outro valor para erro.
  - Argumentos IN são passados por valor. Argumentos OUT e INOUT são passados por referência.
- MPI_Comm_size(MPI_COMM_WORLD, &size);

Tipos de Dados do MPI

- As operações do MPI especificam o tamanho da mensagem em unidades de elementos:
  - “Vou transferir 100 inteiros”.
  - O tamanho do inteiro é dependente da arquitetura.
  - Para poder ser compilado em várias plataformas, código em MPI deve referenciar os tipos definidos em “mpi.h”.
- Portabilidade de código.

Tipo de dado MPI	Tipo de Dado em C
MPI_INT	*int*
MPI_SHORT	*short*
MPI_LONG	*long int*
MPI_FLOAT	*float*
MPI_DOUBLE	*double*
MPI_CHAR	*char*
MPI_BYTE	valores binários
MPI_PACKED	vários tipos em fluxo

Tratamento de Erros no MPI

- O MPI pressupõe uma rede funcional, portanto não oferece nada para lidar com falhas na rede:
  - Erros de transmissão.
  - *Timeouts*.
- O MPI também pressupõe que processadores e memórias funcionam corretamente.
- Os erros que podem ser tratados são:
  - Argumentos mal formulados.
  - Destinatário não existente.
  - Estouros de *buffers* (nível de *software*).
- Por padrão, um valor de erro é retornado e todos os processos são finalizados.

Ambiente para Execução de Aplicações MPI

- Em apenas um computador:
  - $ sudo apt install libopenmpi-dev
  - Você poderá compilar e executar em apenas uma máquina.
- Em vários computadores:
  - É necessário ter um mesmo usuário em cada máquina, com permissão de *login* automático.
  - O executável precisa se copiado para o mesmo caminho em cada máquina.
  - Programar, compilar e copiar é trabalhoso:
    - Usar NFS ou outro protocolo para compartilhar diretórios na rede.
    - Usar NIS/LDAP para replicar usuários.
  - Para facilitar e não ter que fazer essas configurações sempre, vamos utilizar um modelo preparado na nuvem.

Executando e Configurando Processos MPI

- O compilador é o **mpicc** (mpic++, mpif90, etc).
  - É um *wrapper* no *gcc*.
  - Trata de incluir as bibliotecas e cabeçalhos.
- Para executar, usamos o **mpirun** (ou mpiexec).
  - mpirun –np 4 ./OlaMundo
    - Executa o binário *OlaMundo*.
    - Cria 4 processos.
  - mpirun –np 8 --hostfile machines.hostfile ./OlaMundo
    - Cria 8 processos.
    - Usa as máquinas listadas em *machines.hostfile* para distribuir os processos.

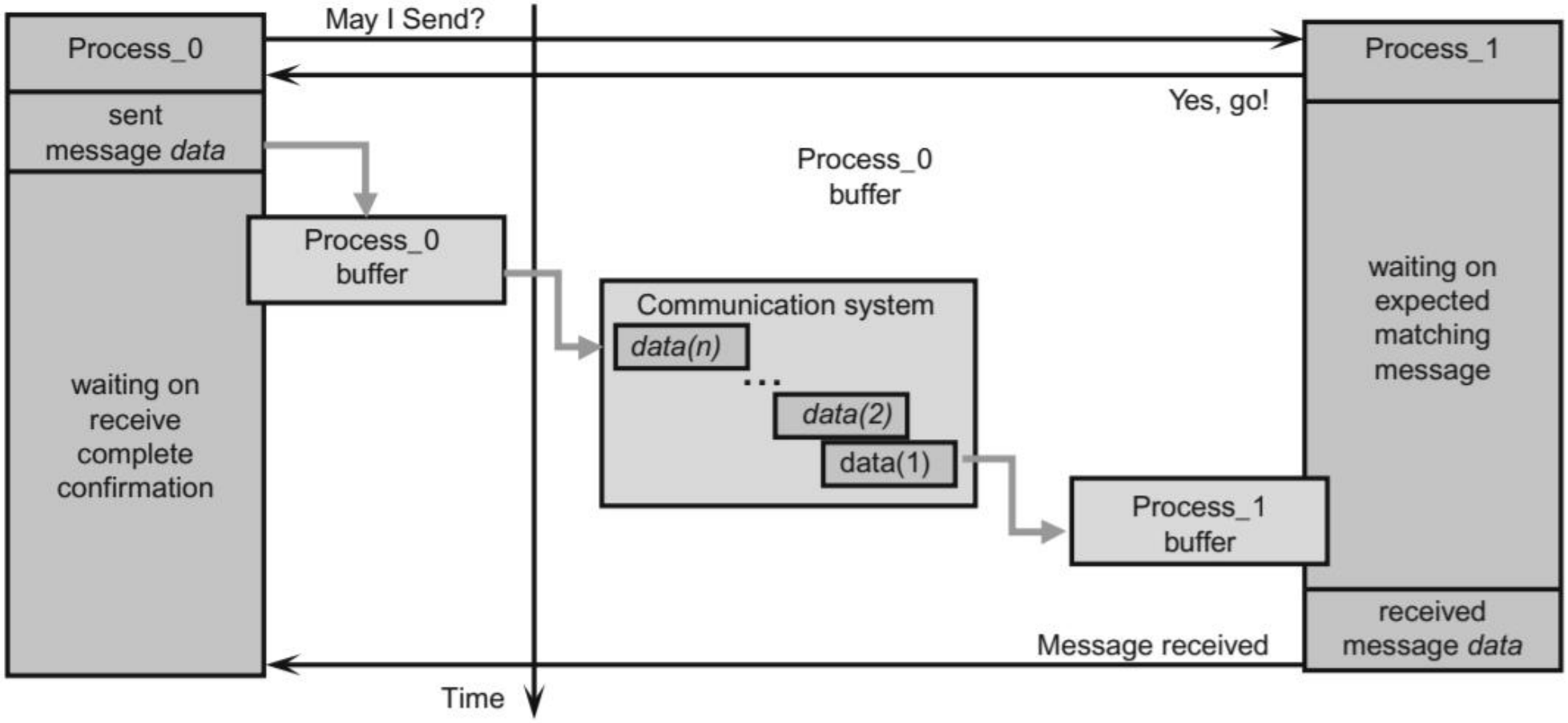
Operações Básicas do MPI

- MPI_Init (int *argc, char *argv): inicializa o ambiente da biblioteca MPI. Os argumentos do programa estão disponíveis para todos os processos.
- MPI_Finalize(): termina o ambiente. Todas as outras chamadas devem ocorrer entre o MPI_Init() e o MPI_Finalize().
- MPI_Comm_size(comm, size): número de processos no comunicador.
- MPI_Comm_rank(comm, rank): o identificador do processo dentro do comunicador, entre 0 e 1.

Comunicação entre Processos

- O modelo de troca de mensagens formaliza a comunicação entre processos que tem espaço de endereçamentos distintos.
  - Movimento de dados.
  - Sincronização: toda operação de envio exige uma operação de recebimento correspondente.
- O uso de *buffers*, tanto para o emissor quanto para o receptor, auxilia no controle do fluxo.
- Em uma mesma máquina, os dados são copiados entre *buffers* na mesma memória.
- Em várias máquinas, os dados dos *buffers* são copiados pela rede.

Comunicação entre Processos



Comunicação entre Processos

- O que acontece se Processo 0 tentar enviar mas Processo 1 não estiver pronto para receber?
  - **Bloqueio** do Processo 0.
- O que acontece se Processo 1 tentar receber mas Processo 0 não estiver pronto para enviar?
  - **Bloqueio** do Processo 1.
- Este é o comportamento padrão. Mas os *buffers* e mensagens de confirmação podem ser configurados para permitir comunicação assíncrona (mais adiante...).
- Qual problema pode surgir com chamadas bloqueantes?

Comunicação entre Processos – MPI_Send

- MPI_Send (buf, count, datatype, dest, tag, comm)
  - Não completa até uma MPI_Recv ser invocada no processo com *rank* igual a *dest*.
  - Somente após a transmissão *buf* pode ser reutilizado com segurança.
  - *buf* é um ponteiro para o início de uma região de memória.
    - *count* informa quantos itens de dados são transmitidos a partir de *buf*.
    - *datatype* vai determinar o tamanho de cada item.
  - A *tag* serve para organizar as mensagens entre um mesmo par emissor e receptor. Caso a ordem não importe, existe o valor MPI_ANY_TAG.

Comunicação entre Processos – MPI_Recv

- MPI_Recv (buf, count, datatype, source, tag, comm, status)
  - *source* é o *rank* do processo emissor, ou MPI_ANY_SOURCE para qualquer envio.
  - *count* diz o limite superior de elementos de dados que podem ser armazenados no endereço definido por *buf*.
  - *status* guarda informações sobre a transferência, inclusive sobre erros.

MPI_Send e MPI_Recv – *Buffers*

- *count, datatype, tag* e *comm* devem casar nas duas chamadas MPI_Send e MPI_Recv correspondentes.
- O endereço indicado em *buf* pertence a memória do processo em espaço de usuário.
  - Enquanto a transmissão não ocorre, não é seguro acessar esse endereço.
  - A implementação do MPI pode manter, em espaço de memória do SO, ***buffers* de sistema** para recuperar os dados dos *buffers* de usuário e liberar o acesso.
- Então o bloqueio pode existir só até a cópia [*buffer* sistema -> *buffer* usuário] ocorrer, não necessariamente a cópia completa pela rede.
- Se o *buffer* de sistema não suportar todo o conteúdo do *buffer* de usuário, o **bloqueio irá ocorrer** de qualquer forma.

Comunicação entre Processos

- Arquivo *ping_pong.c*
  - Dois processos enviando mensagens cada vez mais maiores.
  - Observem a ordem das chamadas:
    - Há algum problema?
    - Qual seria a solução?
- Qual é a razão dos bloqueios das chamadas?

Comunicação entre Processos – MPI_Sendrecv

- MPI_Sendrecv(sendbuf, sendcount, sendtype, dest, sendtag, recvbug, recvcount, recvtype, source, recvtag, comm, status)
  - Permite enviar e receber ao mesmo tempo, em *buffers* distintos.
  - Útil para processos em topologia linear ou anel.
  - Os *buffers* podem ser tamanhos e tipos diferentes.
- Resumindo, as operações básicas são:
  - MPI_Init
  - MPI_Finalize
  - MPI_Comm_size
  - MPI_Comm_rank
  - MPI_Send
  - MPI_Recv
- Com elas, já é possível implementar a maioria dos algoritmos paralelos.
- Outras operações facilitam a legibilidade do código e aprimoram o desempenho.

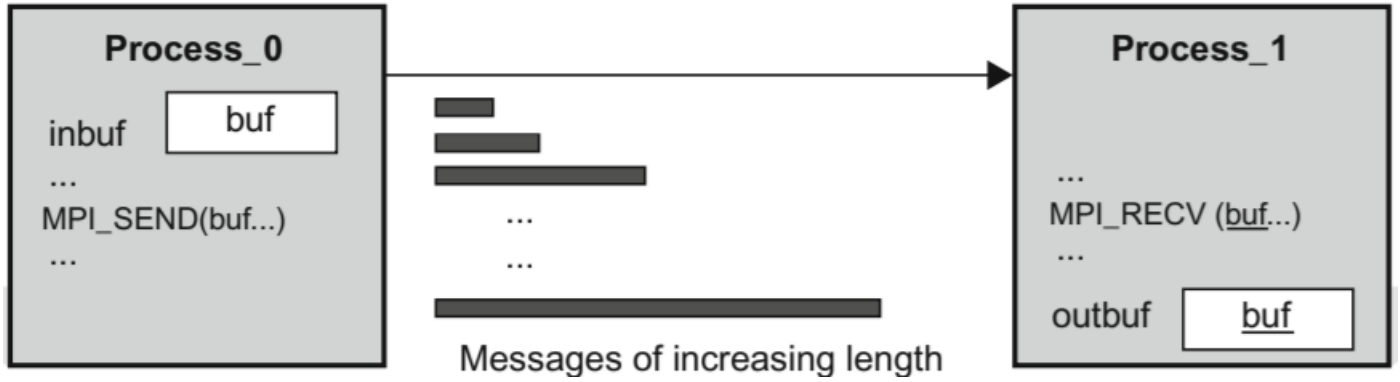
Medindo Desempenho

- MPI_Wtime()
  - Captura o tempo desde o início do programa até o ponto da invocação.

```
double start, finish;
start = MPI_Wtime();
... // faz algum trabalho
finish = MPI_Wtime();
printf(“Tempo gasto: %f\n”, finish – start);
```

Exemplo: Medindo Largura de Banda

- Arquivo *desempenho_rede.c*



  - Um processo envia mensagens cada vez maiores.
  - Enquanto o *buffer* de sistema suporta a mensagem, o bloqueio só ocorre até ele receber os dados.
  - Mas quando os dados não cabem mais no *buffer* de sistema?

Exemplo: Soma de Vetores

- Arquivo *vector_sum/vector_sum.c*:
  - Um processo irá gerar dois vetores.
  - Irá enviar metade dos dois vetores para que sejam somados.
  - Após o envio, faz a soma da sua parte.
  - Recebe a contribuição do outro processo e anexa a sua.
  - O vetor resultado da soma é o apresentado.
- Essa solução usa dois processos.
- Como seria, para utilizar um número arbitrário de processos?
  - Podemos considerar que o tamanho dos vetores é divisível pela quantidade de processos.

Comunicação Coletiva

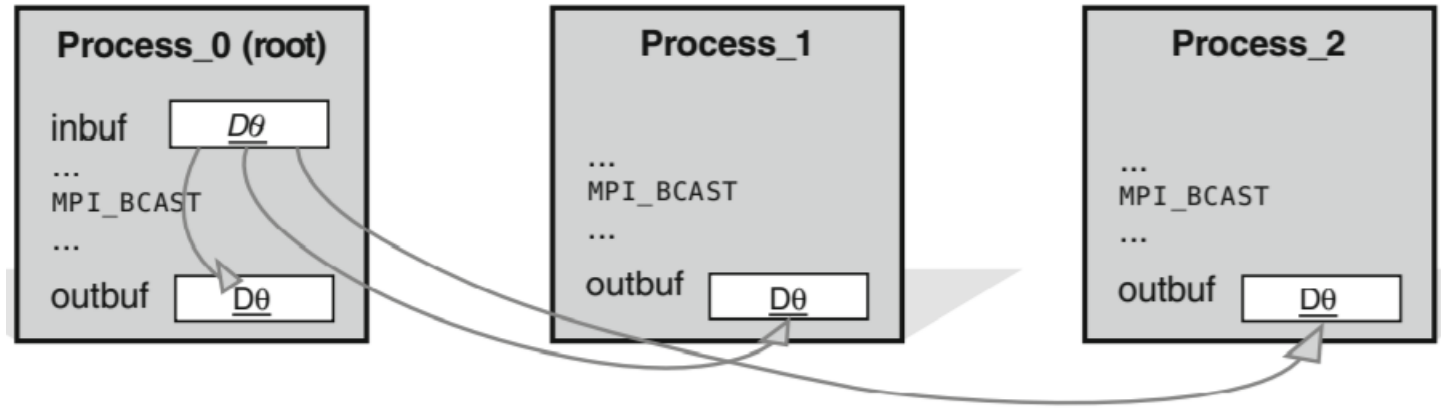
- É muito complicado sempre ter que tratar o *rank* para decidir o que cada processo tem que fazer.
- Usando comunicação coletiva, todos os processos em um comunicador/grupo invocam uma mesma operação de comunicação.
- Em geral, essa operação cuida da distribuição de uma estrutura de dados, que já está pronta para ser tratada pelo processo correspondente.
- Há sincronização de todos os processos envolvidos na operação coletiva no comunicador.

Comunicação Coletiva - Barreiras

- MPI_Barrier (comm)
  - Sincronizar a execução de processo no comunicador informado.
  - Ao atingir essa operação, o processo bloqueia até todos os outros atingirem.
  - Se o programador esquecer de incluir a operação no fluxo de algum processo, a aplicação pode bloquear indefinidamente.
  - É uma maneira simples de separar duas fases de uma computação.
- Devido ao atraso acarretado, barreiras devem ser evitadas, usadas apenas na sincronização de fases mais amplas da aplicação.

Comunicação Coletiva - Difusão

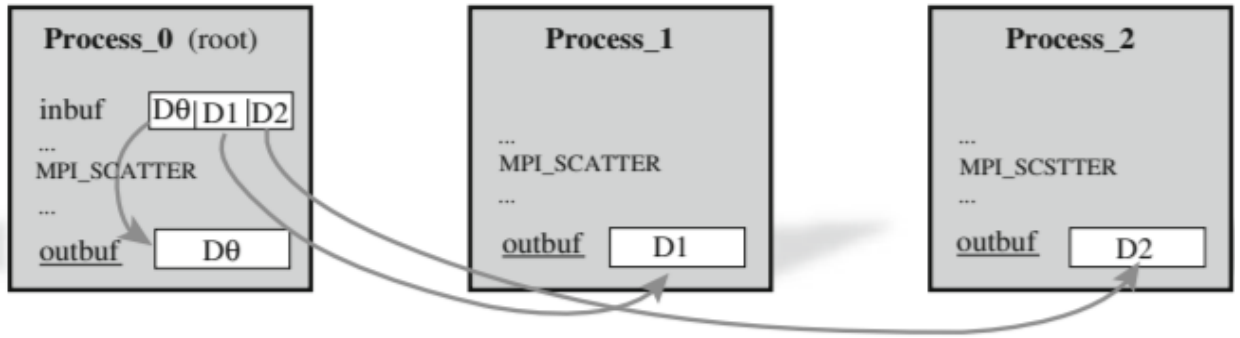
- MPI_Bcast (*inbuf*, incnt, intype, root, comm) – arquivo *broadcast/broadcast.c*



  - O processo *root* envia dados para todos outros processos.
  - *inbuf* é usado como saída no processo raiz, e entrada em todos os outros.
  - *incnt* é a quantidade de elementos do tipo *intype* que são enviados.
- Como seria a implementação de um M_Bcast usando apenas MPI_Send/MPI_Recv?

Comunicação Coletiva - Distribuição

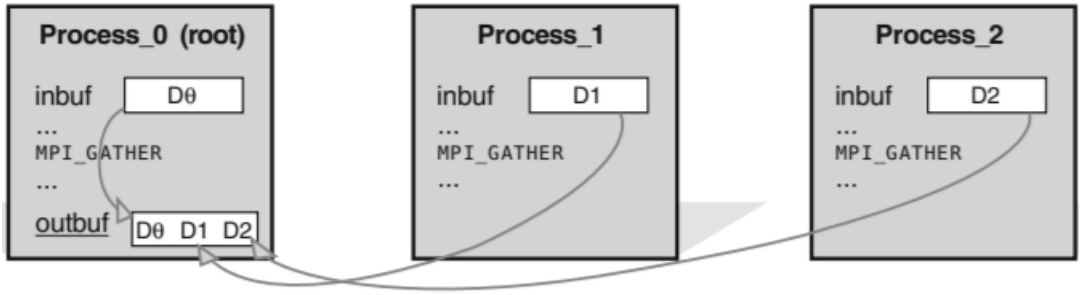
- MPI_Scatter (inbuf, incnt, intype, outbuf, outcnt, outtype, root, comm) – arquivo *scatter/scatter.c*



  - Distribui dados de *inbuf* do processo *root* para *outbuf* nos outros.
  - O conteúdo de *inbuf* é dividido pelo número de processos
    - Vários segmentos são formados.
    - O primeiro segmento vai para o processo 0, o segundo para o processo 1, etc .
  - MPI_Bcast envia a mesma informação, enquanto MPI_Scatter envia segmentos diferentes de um vetor.

Comunicação Coletiva - Coleta

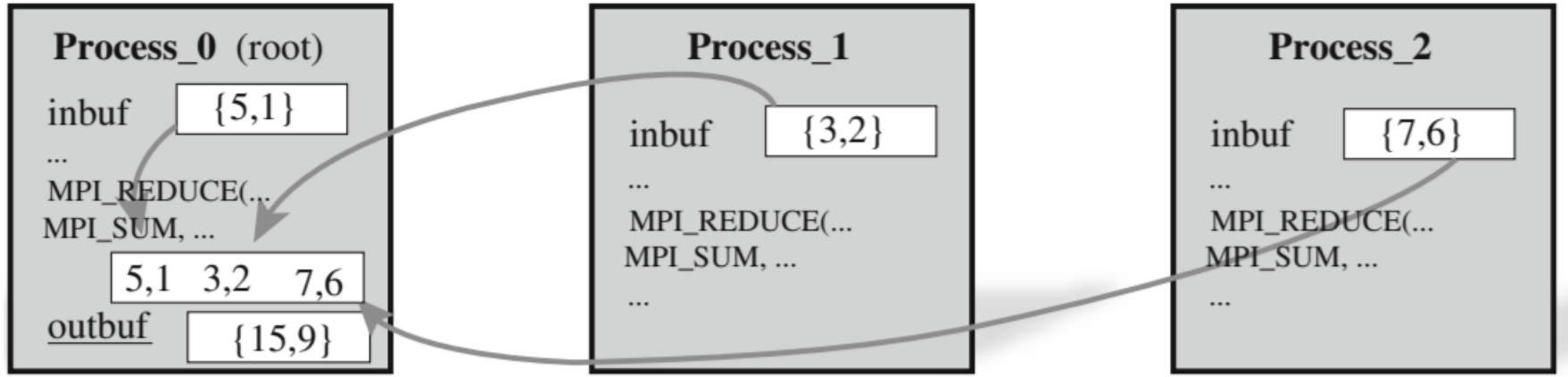
- MPI_Gather (inbuf, incnt, intype, outbuf, outcnt, outtype, root, comm) – arquivo *gather/gather.c*



  - Todos os processos, raiz inclusive, enviam dados de *inbuf* para consolidação no processo raiz em *outbuf*.
  - Cada segmento é armazenado na ordem do *rank* do processo emissor.
  - MPI_Gather e MPI_Scatter são operações inversas.

Comunicação Coletiva - Redução

- MPI_Reduce(inbuf, outbuf, incnt, inttype, op, root, comm)
  - Arquivo *reduce/reduce.c*



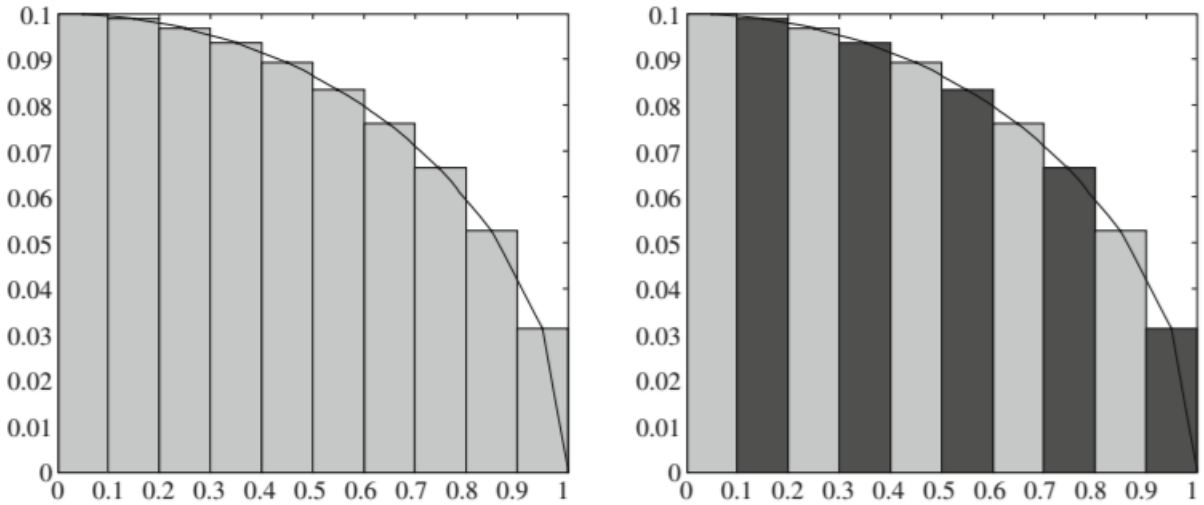
  - Combina os valores armazenados em *inbuf* usando a operação op, armazenando o resultado em *outbuf* no processo *root*.
  - O i-ésimo elemento em *outbuf* é resultado da combinação dos i-ésimos elementos em *inbuf*.
  - As operações típicas são adição, máximo, mínimo, AND, OR, etc.
  - Mesmo que apenas o processo root faça uso de outbuf, todos os processos devem fornecer o parâmetro.

Comunicação Coletiva - Redução

- Redução com Difusão.
  - MPI_Allreduce.
- Coleta com Difusão.
  - MPI_Allgather.
- Transposição.
  - MPI_Alltoall.
- São padrões mais avançados, mas podem ser vistos como combinações das operações anteriores.

Exemplo - Cálculo do Valor de π com Operações Coletivas

- Uma primeira versão, *mpi_pi_p2p.c*, utiliza operações MPI_Send/MPI_Recv e o método de Monte Carlo.
- Uma segunda versão, *mpi_pi_col.c,* usa MPI_Bcast e MPI_Reduce, também no método de Monte Carlo.
- Uma terceira versão (*mpi_pi_integral_col.c*), usa a integral da área do círculo.



Modos de Comunicação

- A comunicação coletiva facilita a criação de programas paralelos com inúmeros processos.
- Entretanto, em alguns casos a comunicação **ponto a ponto** permite controle maior sobre a comunicação.
- Desempenho através da sobreposição de comunicação e computação.
- Modos de comunicação e Bloqueio.

Modos de Comunicação

- Em relação ao **momento de retorno** da operação:
  - **Bloqueante**: a função só retorna quando determinado *evento* é concluído.
  - **Não-bloqueante**: a função retorna imediatamente, mas permite verificar se determinado *evento* concluiu ou não.
- Em relação ao evento que dispara o bloqueio ou pode ser verificado por conclusão: síncrono, pronto, bufferizado e padrão.

Modo de Comunicação	Rotinas Bloqueantes	Rotinas Não-Bloqueantes
Síncrono	MPI_Ssend	MPI_Issend
Pronto	MPI_Rsend	MPI_Irsend
Bufferizado	MPI_Bsend	MPI_Ibsend
Padrão	MPI_Send MPI_Recv	MPI_Isend MPI_Irecv

Modo Padrão – Operações Bloqueantes

- O evento que controla o bloqueio não é definido pela norma técnica do MPI, fica a cargo da implementação definir, mas em geral:
  - Mensagens pequenas: no MPI_Send, o evento é a cópia da mensagem do espaço do usuário para o *buffer* do sistema.
  - Mensagens grandes: no MPI_Recv, o evento é a recepção da mensagem pelo outro processo.
  - No MPI_Recv, o evento são os dados chegarem, independente do tamanho da mensagem.
- De qualquer forma, no envio, a operação retorna quando é seguro reutilizar o *buffer* de usuário.
- O comportamento não é confiável (ver *ping_pong_blocking.c*).
- Pode levar a *deadlocks*.

Modo Padrão – Operações Não-Bloqueantes

- Operações MPI_Isend e MPI_Irecv.
- Assim como nas bloqueantes, o evento a ser verificado varia de acordo com a implementação.
- As operações MPI_Wait e MPI_Test verificam se o evento que permite reutilizar o *buffer* já ocorreu:
  - Mensagens pequenas: o evento é a mensagem já estar no buffer do sistema.
  - Mensagens grandes: o evento é o recebimento por outro processo.
- Ver *ping_pong_non_blocking.c*

Modo Síncrono

- Força que o evento de controle do bloqueio ou conclusão seja o início da operação correspondente no outro processo.
- MPI_Ssend (**bloqueante**) só retorna quando um MPI_Recv começar a receber os dados do outro lado.
- MPI_Issend (**não-bloqueante**) retorna imediatamente, mas permite verificar com MPI_Wait se o outro lado iniciou o recebimento, independente do tamanho da mensagem.
- São operações mais seguras e previsíveis, mas podem acarretar perda de desempenho.

Modo *Bufferizado*

- Permite que o desenvolvedor configure um *buffer* do sistema intermediário.
- MPI_Buffer_attach(buffer, tam_buffer) estabelece um espaço na memória utilizado para guardar as mensagens enviadas.
- MPI_Bsend **bloqueia** até a mensagem ser copiada ao *buffer*.
- MPI_Ibsend **não bloqueia**, mas permite verificar se a mensagem já está no *buffer*.
- Ao contrário do modo Padrão, o uso do *buffer* de sistema é assegurado neste modo. É mais seguro, bom desempenho, mas consome memória.
- O tamanho do buffer precisa ser calculado com MPI_Pack_size(cnt, type, comm, tam_buffer).

Modo Pronto

- O envio só pode ser iniciado se houver uma rotina de recebimento já iniciada.
- Caso contrário ocorre erro!
- Como fazer funcionar?
  - Vários processos emite MPI_Irecv, enquanto um processo raiz nada faz.
  - Todos os processos atingem uma MPI_Barrier.
  - O processo raiz realiza uma MPI_Rsend correspondente a cada MPI_Irecv.
- É necessário maior atenção no projeto de aplicações no modo Pronto para evitar falhas desnecessárias.

Modos de Comunicação - Resumo

Modo	Vantagens	Desvantagens
Síncrono	Mais seguro e, portanto, mais portátil. A ordem Send/Recv não é crítica. Total de espaço gasto é irrelevante.	Pode haver substancial *overhead* de sincronização.
Pronto	O *overhead* total é mais baixo. O protocolo Send/Recv não é necessário.	O Recv deve preceder o Send.
Bufferizado	Desacopla o Send o Recv. O programador pode controlar o tamanho do *buffer.*	O *overhead* adicional do sistema para copiar o *buffer*. A ordem do Send/Recv é irrelevante.
Padrão	Bom na maioria das vezes.	Seu programa pode não funcionar corretamente em algumas implementações.

Deadlocks

Possível *Deadlock*

Processo 0	**Processo 1**
MPI_Send(1)	MPI_Send(0)
MPI_Recv(1)	MPI_Recv(0)

Solução usando *Buffers*

Processo 0	**Processo 1**
MPI_Bsend(1)	MPI_Bsend(0)
MPI_Recv(1)	MPI_Recv(0)

Solução usando MPI_Sendrecv

Processo 0	**Processo 1**
MPI_Sendrecv(1)	MPI_Sendrecv(0)

Solução no modo Padrão

Processo 0	**Processo 1**
MPI_Send(1)	MPI_Recv(0)
MPI_Recv(1)	MPI_Send(0)

Solução usando rotinas não bloqueantes

Processo 0	**Processo 1**
MPI_Isend(1) MPI_Irecv(1)	MPI_Isend(0) MPI_Irecv(0)
MPI_Waitall	MPI_Waitall(0)

Processos em Topologia Anel

- Exemplo do modo padrão.
- Arquivos na pasta *simple_ring*.
- Processos organizados em um anel, cada processo troca mensagens com seus vizinhos diretos.
- Vamos analisar:
  - Modo padrão (*simple_ring.c*)
  - Modo padrão com garantia de ausência de deadlock (*simple_ring_nodeadlock.c*)
  - Modo padrão com operações não bloqueantes (*simple_ring_asyn.c)*
  - Usando MPI_Sendrecv (*simple_ring_sendreceive.c*)

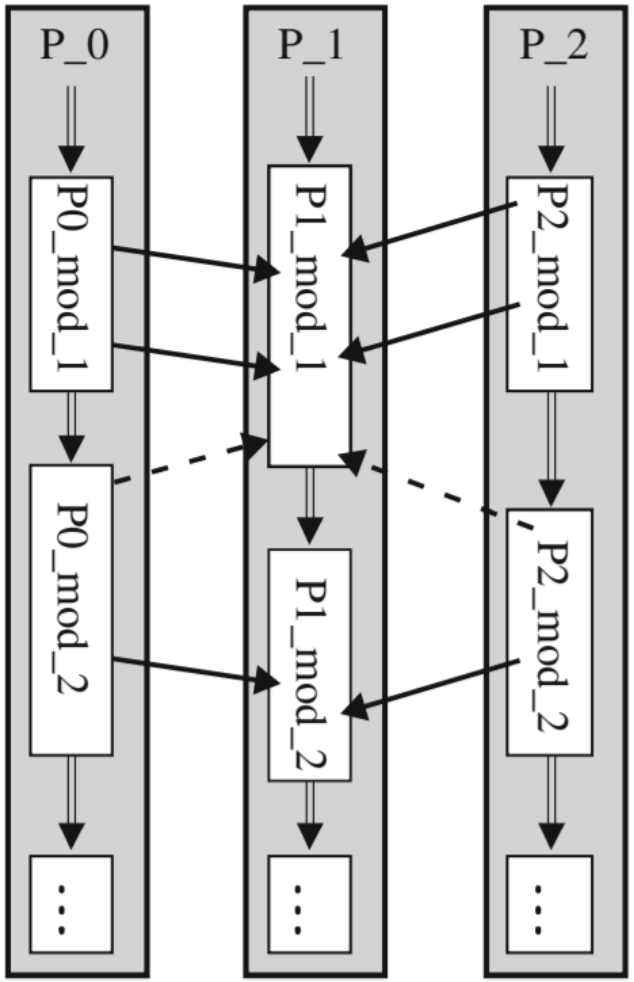
Sobreposição de Comunicação e Computação

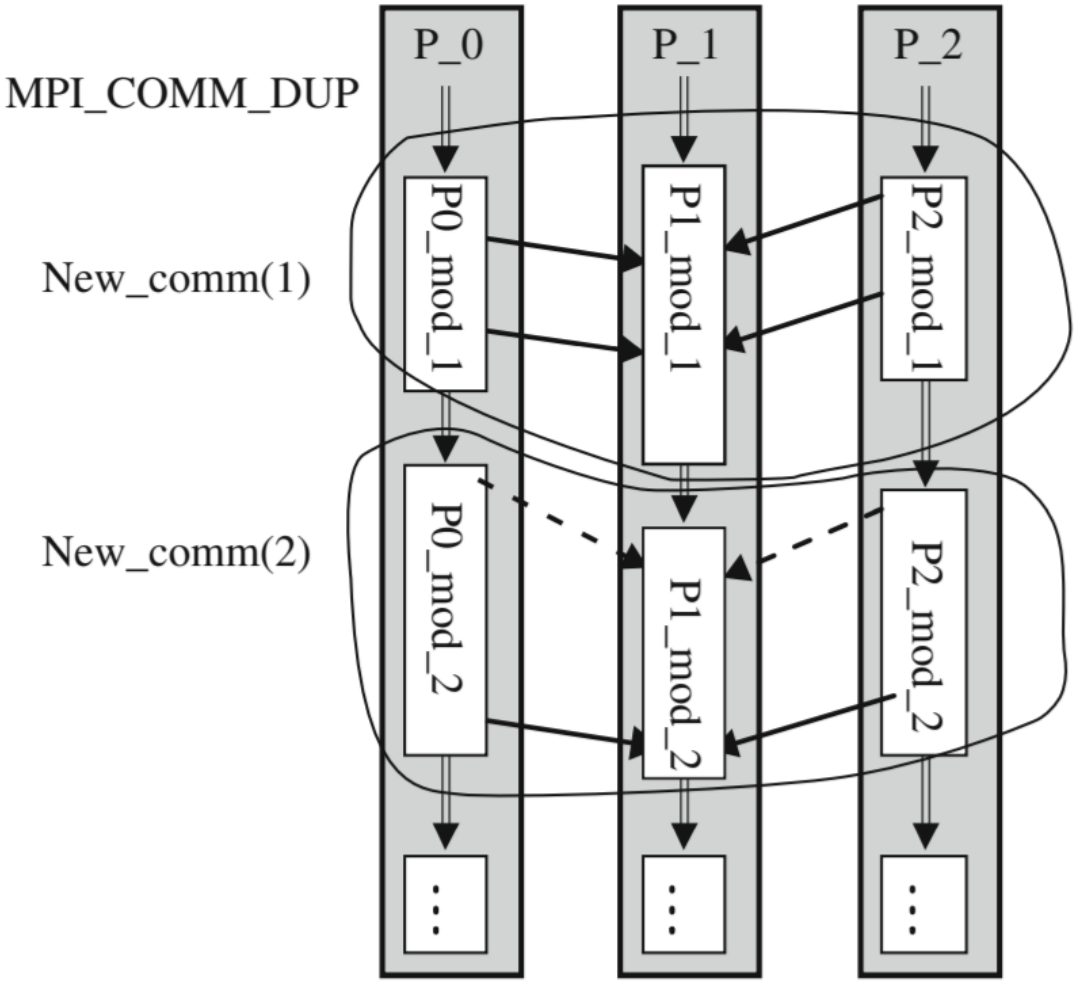
- Programa *overlapping.c*
- Definida uma função arbitrária que executa cálculos.
- Um processo mestre espera mensagens dos escravos e depois realiza trabalho local.
- São dois casos:
  - Recebendo bloqueado no modo padrão.
  - Recebendo não bloqueado também no modo padrão.
- O programa permite analisar o tempo gasto em computação e comunicação, além da duração total.

Comunicadores MPI

- Até agora utilizamos o comunicador MPI_COMM_WORLD, que engloba todos os processos.
- Mas às vezes é interessante separar os processos em **grupos**, de acordo com o **contexto** da execução.
  - A aplicação pode ter várias fases. Queremos garantir que mensagens de uma determinada fase não sejam confundidas com mensagens de outra etapa.
  - Várias bibliotecas podem utilizar MPI. É preciso garantir que mensagens de bibliotecas distintas fiquem restritas aos seus trechos de código.
- O MPI permite criar novos comunicadores “dividindo” o MPI_COMM_WORLD.

Comunicadores MPI





Comunicadores MPI

- MPI_Comm_dup(comm, new_comm)
  - Permite criar um novo comunicador com os mesmos processos do antigo.
  - É útil para separar a comunicação entre fases de uma aplicação.
- MPI_Comm_split(comm, color, key, new_comm)
  - É invocada por todos os processos do comunicador original.
  - Processos que oferecerem o mesmo valor de *color* ficaram no mesmo novo comunicador.
  - O valor de *key* indica como determinar o novo *rank*.
    - Se um processo informa um valor de *key* menor do que o fornecido por outro processo, o primeiro processo terá um *rank* menor que o segundo.
    - Valores iguais preservam a ordem.
- MPI_Comm_free.

Comunicadores MPI

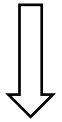
communicators.c

P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7
c=0	c=1	c=0	c=1	c=0	c=1	c=0	c=1
k=0	k=0	k=0	k=0	k=0	k=0	k=0	k=0

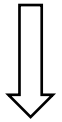
rank do processo em MPI_COMM_WORLD

color = *rank* % 2

key = 0



MPI_Comm_split (MPI_COMM_WORLD, color, key, &new_comm) ;



P_0 r_0	P_2 r_1	P_4 r_2	P_6 r_3

P_1 r_0	P_3 r_1	P_5 r_2	P_7 r_3

Dois comunicadores, *ranks* dos processos em cada um.