Programa de Pós-Graduação em Computação PPGC

Programação Paralela: Uma Introdução

Cláudio Geyer

UFRGS
UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

Informática UFRGS

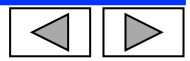
Autoria

- Autores
  - André (aluno pós, principal autor)
  - C. Geyer
- Versão
  - V18.1, maio de 2013

Informática UFRGS

Sumário

- Conceitos básicos de projeto de programação paralela
- MPI
  - Conceito
  - Características
  - Primitivas de controle
  - Primitivas básicas de send/receive
  - Primitivas de comunicação coletiva
  - Primitivas avançadas de comunicação
- Técnicas básicas para depuração, melhoria de desempenho, ...

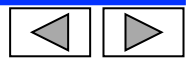


UFRGS
UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

Informática UFRGS

Bibliografia

- Pacheco 1
  - Pacheco, P. S. "Parallel Programming with MPI". Morgan Kaufmann Publishers, Inc., 1997.
- Wilkinson
  - Wilkinson, B. and Allen, M. "Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers". Prentice-Hall, 2004.
- Gropp
  - Gropp, W. et al. "Using MPI and Using MPI-2". The MIT Press. 1999.



UFRGS
UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

Bibliografia

- Pacheco 2
  - Pacheco, P. S. “An Introduction to Parallel Programming”. Morgan Kaufmann Publishers, Inc., 2011.

Objetivo

- O objetivo da paralelização de uma aplicação é
  - obter em menor tempo que o em seqüencial
  - resultados que sejam satisfatórios ao problema apresentado
- Em alguns problemas ou casos
  - Conseguir resolver um problema com *n* maior devido limite de memória em máquina seqüencial
  - *n*: tamanho da entrada
- Também
  - Poder calcular uma solução em tempo razoável para uma entrada maior

Projeto de Programas Paralelos

- Grande parte dos problemas possuem diversas soluções;
- Nem sempre a melhor solução seqüencial é a mais adequada a ser paralelizada;
- A solução escolhida deve se comportar bem com o crescimento do tamanho do problema e principalmente com o aumento do número de cpus (escalabilidade);
- O uso de uma metodologia permite maximizar o número de opções e fornece mecanismos para avaliação das alternativas;

Metodologia (Ian Foster)

- Particionamento: decomposição do problema em pequenas tarefas;
- Comunicação: determinação das estruturas e algoritmos necessários para a comunicação;
- Aglomeração: possível combinação de tarefas em tarefas maiores para o aumento de desempenho e diminuição de custos na comunicação;
- Mapeamento: distribuição de tarefas aos processadores.
- Obs: (mais) indicada para memória distribuída

Particionamento

- No processo de particionamento procura-se:
  - Obter uma grande quantidade de tarefas
  - Balancear a carga entre os processadores;
  - Cada processador deve ter uma carga equivalente ao seu poder computacional;
  - Em máquinas homogêneas essa tarefa é a mais simples;
  - Minimizar o número de comunicações e a quantidade de dados a serem comunicados;
  - Balancear carga e minimizar comunicação podem ser conflitantes.

Particionamento

- Dois grandes casos de técnicas de particionamento:
  - Domínio (dados):
    - divisão dos dados; tarefas idênticas
  - Funcional:
    - tarefas distintas, eventualmente sobre dados distintos

Comunicação

- Objetivo dessa fase é identificar o fluxo de informações entre os processos;
  - De forma precisa
- Esta fase é altamente dependente do particionamento do problema;
  - Ao particionar frequentemente se considera a comunicação
- Nesta fase deve-se analisar se cada processo realiza aproximadamente o mesmo número de comunicações e se as mensagens possuem tamanhos equivalentes;
  - Comunicação homogênea reduz esperas
  - Aumentando speedup

Comunicação

- Verificar a possibilidade de realizar sobreposição de comunicação com processamento;
  - Uma thread para cálculo e outra para receive
  - Ou 2 threads para cálculo/receive

Aglomeração

- Nesta etapa estuda-se a possibilidade de criar tarefas maiores a partir do agrupamento de tarefas identificadas na fase de particionamento;
  - Reduz-se a concorrência
- A aglomeração pode melhorar a razão entre o custo de comunicação e o processamento;
  - Tarefas maiores e mesma (ou quase) comunicação
  - Melhora o speedup e a eficiência

Aglomeração

- A aglomeração pode/deve ser aplicada em casos onde
  - o número de tarefas for maior que o número de processadores disponíveis (*p*).
  - #tasks >> p
- Mas em certos casos também quando número de tarefas for menor que *p*.

Mapeamento

- Nesta etapa é feita a distribuição das tarefas resultantes entre os processadores disponíveis;
  - # cpus é conhecida
- O mapeamento deve ser feito de forma a se adequar à arquitetura da máquina disponível;
  - Em arquiteturas com interconexão mais complexa o mapeamento é mais difícil e de grande importância;
  - Em arquiteturas onde todos os nodos possuem o mesmo custo de comunicação (clusters) o mapeamento é mais simples;
  - 2 tarefas com alta comunicação devem ficar em nós próximos
  - 2 tarefas com baixa comunicação podem ficar em nós distantes

Mapeamento

- Algumas bibliotecas possibilitam a criação de topologias de processos que permitem a adaptação das comunicações à arquitetura (MPI);

Passos para Paralelização de uma Aplicação

1. Desenvolver (ou obter) um programa seqüencial;
2. Analisar o programa seqüencial (profile) para identificar onde está o gargalo de processamento;
3. Efetuar a paralelização do programa;
4. Verificar resultados da versão paralela;
5. Identificação dos gargalos na versão paralela;
6. Otimização da versão paralela;
7. Voltar a 4 até que não seja mais necessário a otimização;
    1. Ou ganho não compense o esforço

Informática UFRGS

MPI
Message Passing Interface

UFRGS
UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

Informática UFRGS

O que é MPI ?

- Especificação de um padrão para
  - desenvolvimento de bibliotecas de troca de mensagens
  - em ambientes de memória distribuída;
- Criada no início de 1992 pelo MPI fórum (órgãos governamentais, universidades e empresas);
- Principais linguages utilizadas
  - C e Fortran
- Vantagens de um padrão
  - Programas portáveis entre diferentes máquinas (IBM, Cray, NEC, clusters, ...)

UFRGS
UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

Informática UFRGS

O que é MPI ?

- Define:
  - Rotinas de comunicação ponto a ponto.
  - Rotinas de comunicação coletiva;
  - Rotinas de gerenciamento de processos;
- Objetiva:
  - Portabilidade;
  - Eficiência;
  - Facilidade de programação.

UFRGS
UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

Distribuições do MPI

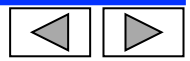
- Existem diferentes implementações do padrão MPI, entre as de domínio público as mais conhecidas são:
  - CHIMP/MPI – Edinburgh Parallel Computing Center (EPCC)
  - LAM – Ohio Supercomputer Center;
  - MPIAP – Australian National University;
  - MPICH – Argonne National Laboratory Mississipi State University;
  - MPI-FM – University Illinois;
  - W32MPI – Universidade de Coimbra - Portugal;
- Entre as implementações proprietárias estão as da IBM e da HP;

Distribuições do MPI

- Páginas com listas de implementações:
  - http://www.mcs.anl.gov/research/projects/mpi/implementations.html
  - http://en.wikipedia.org/wiki/Message_Passing_Interface#Implementations

Versões do MPI

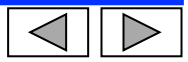
- Existem algumas versões do padrão MPI
- As mais importantes são:
  - 1.x:
    - Mais usada ainda hoje em termos de programação
    - Inicialmente não oferecia criação dinâmica de processos
  - 2.x:
    - Oferece criação dinâmica de processos
    - E outras funcionalidades
- Implementações se adaptam à evolução de formas distintas
- Slides não abordam a versão 2.x



Informática UFRGS

Tutoriais MPI

- Tutoriais MPI
  - Vários tutoriais:
    http://www.lam-mpi.org/tutorials/
    http://www-unix.mcs.anl.gov/mpi/tutorial/
  - Sobre ou incluindo MPI-2
    - https://computing.llnl.gov/tutorials/mpi/
    - http://www.mcs.anl.gov/research/projects/mpi/tutorial/advmpi/pvmmpi05-advmpi.pdf
    - http://sbel.wisc.edu/Courses/ME964/2008/LectureByLecture/me964Nov11.pdf



UFRGS
UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

Outros links MPI

- Open-MPI
  - http://www.open-mpi.org/
- Forum MPI
  - Página principal
    - http://www.mpi-forum.org/
  - Página com documentações (especificações)
    - http://www.mpi-forum.org/docs/docs.html
  - Página com especificação 2.2
    - http://www.mpi-forum.org/docs/mpi22-report/mpi22-report.htm#Node0

Informática UFRGS

Sumário

- Compilação;
- Execução;
- Rotinas Básicas;
- Rotinas de Comunicação Ponto a Ponto;
- Rotinas de Comunicação coletiva;
- Criação de Comunicadores;
- Dados Híbridos;
- Sincronização;
- ---*---
- Avaliação do desempenho;
- Problemas no Desenvolvimento;
- Referências Bibliográficas.

UFRGS
UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

Compilação

- Após um programa fazendo uso da biblioteca MPI ter sido escrito, este deve ser compilado com um script específico do MPI;
- mpicc -o <programa executável> <programa fonte>
- Este script permite utilizar todas as opções do compilador.

Execução

- Após a compilação o próximo passo é a execução do programa; para isso a implementação do MPI possui um script;
- mpirun <opção> <executável>
  - Opções:
    - - *h* : mostra todas opções disponíveis;
    - - *machinefile* : especifica o arquivo de máquinas;
    - - *np* : especifica o número de processos;
    - - *nolocal* : não executa na máquina servidora.
  - Observações:
    - *np* pode ser > que *p (#* de cpus)

Informática UFRGS

Rotinas Básicas

- Inicialização de um Processo;
- Identificação do Processo;
- Obtenção do Número de Processos;
- Envio de Mensagens;
- Recebimento de Mensagens;
- Tipos de Dados;
- Finalização de Processos;

UFRGS
UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

Inicialização de um processo

- Em um programa MPI, a rotina **int MPI_Init** é
  - responsável pela criação de um ambiente necessário para a execução do MPI
  - esta rotina sincroniza todos os processos na inicialização de uma aplicação.

int MPI_Init(int *argc, char **argv);

- *argc* - variável que indica o total de parâmetros, sendo que o nome do programa também é um parâmetro;
- *argv* - vetor onde estão os parâmetros.
- Obs: argumentos de retorno (saída)

Inicialização de um processo

- Código eventualmente existente antes do uso da MPI_INIT
  - Executado por todos os processos (!)

Identificação de um processo

- **MPI_Comm_rank**
  - é a primitiva responsável por atribuir um identificador a cada processo dentro de um grupo
  - este identificador deve ter valor entre **0** e **n-1**, onde **n** é o número total de processos.

int MPI_Comm_rank(MPI_Comm comm, int *rank);

- *comm* - grupo ao qual o processo pertence;
  - constante usual e geral: MPI_COMM_WORLD
  - Todos os processos do programa (execução)
- *rank* - variável que irá conter a identificação do processo.

Identificação de um processo

- Muito usado para identificar a tarefa de cada processo.

```
MPI_Comm_rank(MPI_COMM_WORLD, &id);
if (id==0)   {
    /*** Executa tarefa do processador 0 ***/
    ...
}
```

Identificação de um processo

- **MPI_Comm comm**
  - Identifica um "comunicador"
  - Um comunicador representa um sub-grupo de processos do programa
  - Útil em certos casos como
    - Programa usa um módulo de terceiros o qual pode usar os mesmos "tags" que o problema em desenvolvimento
    - Permite tratar de forma consistente mensagens distintas com o mesmo "tag"

Obtenção do total de processos

- **MPI_Comm_size**
  - é primitiva responsável por identificar o total de processos dentro de um grupo;

int MPI_Comm_size(MPI_Comm comm, int *size)

- *comm* - grupo ao qual o processo pertence;
- *size* - variável que ira conter o número total de processos.
- *size*: muito usada na partição do trabalho total
  - Por exemplo: divisão de matrizes

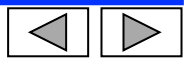
Obtenção do total de processos

- **MPI_Comm_size**
  - Muito usada para distribuir as tarefas entre os processos.
  - Também usada para distribuir os dados a processos idênticos.

```
MPI_Comm_size(MPI_COMM_WORLD, &tam);

for (i=1;i<tam;i++){
    /*** Escolhe e envia tarefa para o processador i
   ***/
        ...
}
```

Informática UFRGS

COMUNICAÇÃO PONTO A PONTO



UFRGS
UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

Envio de mensagens

- **MPI_Send**
  - é primitiva responsável por enviar uma mensagem de um processo para outro.

int MPI_Send(void *sndbuf, int count, MPI_Datatype datatype,int dest, int tag, MPI_Comm comm);

- *sndbuf* - identificação dos dados a serem enviados;
- *count* - quantidade de dados a serem enviados;
- datatype - tipo de dados a serem enviados;
- dest - destino da mensagem;
- tag - rótulo da mensagem;
- comm - grupo ao qual o processo - pertence.

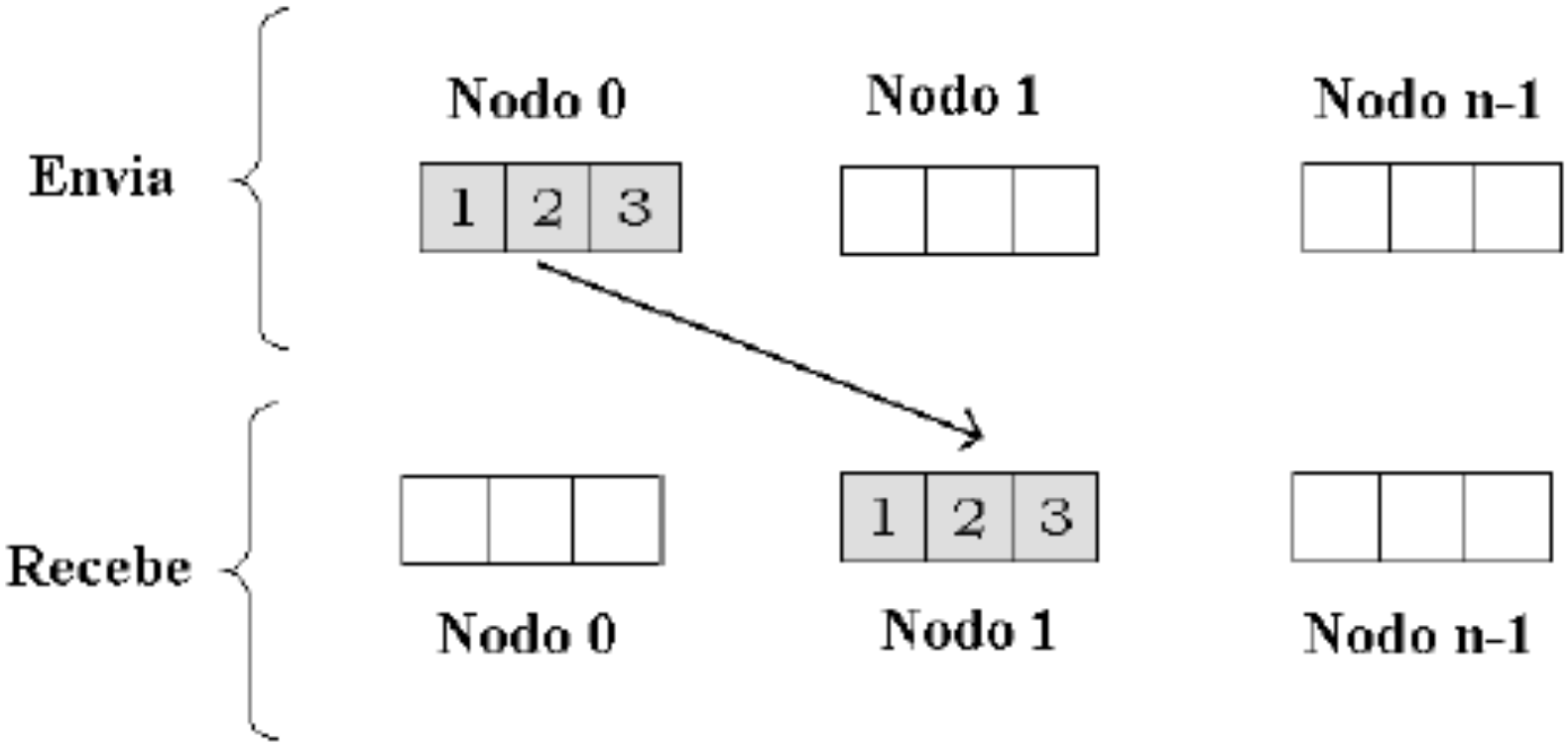
Recebimento de mensagens

- **MPI_Recv** é a primitiva responsável pelo recebimento de uma mensagem.

int MPI_Recv(void *recvbuf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status**);**

  - recvbuf - local onde os dados recebidos devem ser armazenados;
  - *count* – quantidade de dados a serem recebidos;
  - datatype - tipo de dados a serem recebidos;
  - source - origem da mensagem (MPI_ANY_SOURCE);
  - tag - rótulo da mensagem (MPI_ANY_TAG);
  - comm - grupo ao qual o processo pertence;
  - status - situação da operação.

Envio / Recebimento



Tipos de Dados

1	MPI_CHAR	signed char
2	MPI_SHORT	signed short int
3	MPI_INT	signed int
4	MPI_LONG	signed long int
5	MPI_UNSIGNED_CHAR	unsigned char
6	MPI_UNSIGNED_SHORT	unsigned short
7	MPI_UNSIGNED	unsigned int
8	MPI_UNSIGNED_LONG	unsigned long int
9	MPI_FLOAT	float
10	MPI_DOUBLE	double
11	MPI_LONG_DOUBLE	long double
12	MPI_BYTE	byte
13	MPI_PACKED	packed

Informática UFRGS

TAGs

- Tags
  - funcionam como tipos de mensagens
  - quando um *receive* indica um TAG específico
    - ficará bloqueado até que uma mensagem com o TAG específico seja recebida
  - Tag x *receive*
    - Pode usar "MPI_ANY_TAG" (wildcard): combina com qualquer tag na mensagem
  - Tag x *send*
    - *send* sempre deve especificar um tag

UFRGS
UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

Informática UFRGS

Bloqueios

- Bloqueios em receives
  - Uso de indicação de emissor
    - O receive (também) ficará bloqueado até que uma mensagem daquela origem seja recebida
  - Uso de "comunicadores especiais"
    - O receive ficará bloqueado até que uma mensagem com o mesmo comunicador seja recebida

UFRGS
UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

Casamento de argumentos

- Casamento de outros argumentos
  - Entre mensagem enviada e args do receive
  - No caso de não casamento
    - entre tipo de dado e tamanho
    - o efeito pode ser variado e dependente da implementação
  - Por exemplo, caso tamanhos diferentes
    - Alinhamento à esquerda com
      - Erro de overflow (eventualmente truncamento à direita)
      - Ou não preenchimento à direita

Status

- MPI_Status
  - Esse argumento de saída no *receive* oferece ao menos 3 campos
    - MPI_Source: ID (rank) do processo emissor
    - MPI_TAG: tag da mensagem recebida
    - MPI_ERROR: indicação de algum erro no processamento da mensagem

Tamanho da mensagem

- Tamanho da mensagem
  - O parâmetro status não contém o tamanho da mensagem
  - Nem sempre o tamanho é necessário e o custo para calcula-lo é considerável
  - MPI_Get_Count
    - Função que retorna o tamanho da mensagem
    - int MPI_Get_Count(MPI_Status* status,
      MPI_Datatype datatype,
      int* count_ptr)

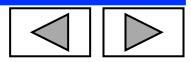
Retornos das Funções

- Retornos das funções
  - Usualmente retornam uma indicação de erro
  - Comportamento usual em MPI é não abortar a execução

Informática UFRGS

Finalização de processos

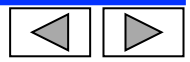
- **MPI_Finalize**
  - É a primitiva responsável por finalizar um processo MPI.
  - Essa primitiva sincroniza todos os processos na finalização de uma aplicação MPI.
  - Deve ser executada por todos os processos
  - Código eventualmente existente depois do uso da MPI_INIT
    - Executado por todos os processos
- **int MPI_Finalize()**



UFRGS
UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

Programa Básico: Olá mundo

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

int main(int argc, char *argv[]) {
    char mensagem[12];
    int id,tam,i;
    MPI_Status status;

    MPI_Init(&argc,&argv);          % inicializa
    /* quem sou? */
    MPI_Comm_rank(MPI_COMM_WORLD,&id);
    /* quantos são? */
    MPI_Comm_size(MPI_COMM_WORLD,&tam);
```

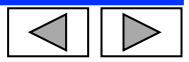


Programa Básico: Olá mundo

```
/* sou o zero? */
if (id==0) {
    strcpy(mensagem,"Ola Mundo");
    for (i=1;i<tam;i++)                % para todos os outros
        MPI_Send(&mensagem,12,MPI_CHAR,i,100,
                        MPI_COMM_WORLD);
/* sou um dos outros */
}else {
        MPI_Recv(&mensagem,12, MPI_CHAR,0,100,
                        MPI_COMM_WORLD, &status);
        }
```

Programa Básico: Olá mundo

```
    printf("Sou o processo %i: %s\n",id,mensagem);
        /* termina */
    MPI_Finalize();
}
```



Programa Básico: Olá mundo

- Questões:
  - Quantos processos recebem a saudação?
  - Estão em quais endereços de rede?

Bloqueante x Não-Bloqueante

- As Rotinas de Comunicação Ponto a Ponto (P2P) podem ser de dois tipos:
  - P2P Bloqueantes -
    - O processo emissor é bloqueado até o recebimento da mensagem pelo receptor
    - Receptor: entende-se o computador (nó, cpu) e não o processo MPI
      - Processo MPI não precisa executar o receive
      - Não é uma comunicação síncrona
    - Variações conforme implementação
      - Bloqueia somente até mover todos os dados para outro buffer

Bloqueante x Não-Bloqueante

- As Rotinas de Com. (P2P) podem ser de dois tipos:
  - P2P Não-Bloqueantes –
    - O processo emissor envia sua mensagem e continua sua execução normalmente
      - A especificação exige que o conteúdo seja ao menos transferido para outro buffer local, fora do programa
    - O receptor "pede" a recepção da mensagem e continua

Bloqueante x Não-Bloqueante

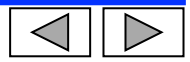
- As Rotinas de Com. (P2P) podem ser de dois tipos:
  - P2P Não-Bloqueantes –
    - O programador pode verificar se a mensagem foi enviada ou recebida corretamente através de outras primitivas como:
      - MPI_Wait(&request, &status);
    - Argumento &request:
      - indica qual send ou receive deve ser verificado
    - Por exemplo, o processo receptor poderia pedir a recepção de várias mensagens concorrentemente

Bloqueante x Não-Bloqueante

- As Rotinas de Com. P2P Não-Bloqueantes
  - O que significa "problema de consistência de dados"?
  - Analise o código abaixo
    - ...

```
strcpy(mensagem,"Ola Mundo");
loop infinito {
        MPI_Isend(&mensagem, ...);
        Strcpy(mensagem, "Ola POA");
}
...
```

  - O que o receptor vai receber em cada receive?



Bloqueante x Não-Bloqueante

- Implementações
  - Em algumas implementações de MPI não há diferença significativa entre as 2 semânticas
  - A implementação do Send (bloqueante) poderia incluir um ack implícito do nó receptor
    - Mas não obrigatoriamente

Bloqueante x Não-Bloqueante

- Variantes
  - MPI_Bsend
    - Um outro tipo de send cujo buffer pode ser gerenciado pelo programador
  - A especificação ainda oferece inúmeras possibilidades de implementação tanto na versão bloqueante quanto na não-bloqueante

Bloqueante x Não-Bloqueante

Bloqueante Send	MPI_Send(buffer,count,type,dest,tag,comm)
Não-Bloqueante Send	MPI_Isend(buffer,count,type,dest,tag,comm,request)
Bloqueante Receive	MPI_Recv(buffer,count,type,source,tag,comm,status)
Não Bloqueante Receive	MPI_Irecv(buffer,count,type,source,tag,comm,status,request)

Bloqueante x Não-Bloqueante

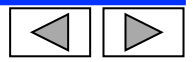
```
/* exemplo Olá Mundo com Não Bloqueante */
#include <mpi.h>
#include <stdio.h>
#include <string.h>
int main(int argc, char **argv) {
    char mensagem[12];
    int rank,size,i;
    MPI_Status status;
    MPI_Request request;

    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Comm_size(MPI_COMM_WORLD,&size);
```

Bloqueante x Não-Bloqueante

```
 if (rank==0) {
     strcpy(mensagem,”Ola Mundo”);
     for (i=0;i<size;i++)
       MPI_Isend(&mensagem,12, MPI_CHAR, i, 100,
                        MPI_COMM_WORLD,&request);
       MPI_Wait(&request,&status);
 }
else {
       MPI_Irecv(&mensagem,12, MPI_CHAR, 0, 100,
                        MPI_COMM_WORLD, &request);
       MPI_Wait(&request,&status);
       }
printf(“Sou o processo %i: %s\n”,rank,mensagem);
MPI_Finalize();
}
```

Informática UFRGS

COMUNICAÇÃO COLETIVA



UFRGS
UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

Comunicação Coletiva

- Permite a troca de mensagens entre todos os processos de um grupo;
- Programação mais simples e rápida
- Programas ficam mais concisos
- especialmente em paralelismo de dados

Comunicação Coletiva

- As rotinas de Comunicação Coletiva (CC) podem ser de dois tipos:

  - CC para Movimentação de Dados:

    Broadcast, Gather, Allgather, Scatter, Alltoall;

  - CC para Computação Global:

    Reduce, Allreduce, Scan;

CC - Broadcast

MPI_Bcast

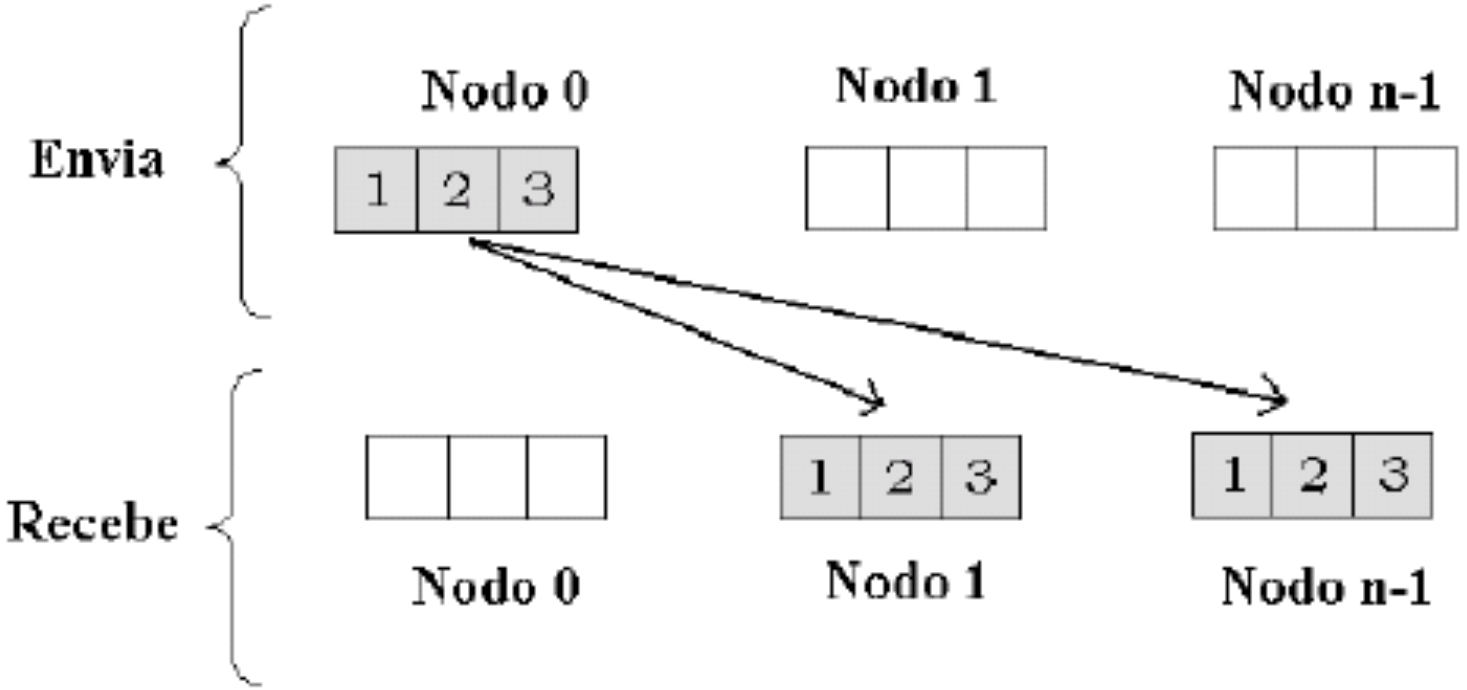
é primitiva responsável por enviar dados de um processo para os demais do seu grupo.

processo emissor e receptor executam a mesma primitiva, com os mesmos argumentos.

=> mesmo buffer para saída e entrada

processo emissor só emite, não recebe

CC - Broadcast



Informática UFRGS

CC - Broadcast

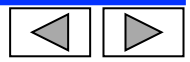
int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm);

- *buffer*: local onde os dados são armazenados;
- count: quantidade de dados a serem transmitidos;
- datatype: tipo de dados a serem transmitidos;
- root - identificação do processo emissor;
- comm - grupo onde os processos emissor e receptor estão.

UFRGS
UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

CC - Broadcast

```
/* Exemplo de Broadcast -  Olá Mundo */
/* mesma funcionalidade do exemplo anterior com S/R */
#include <stdio.h>
#include <mpi.h>

Int main(int argc, char **argv) {
    int size,rank;
    char nome[20];

    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Comm_size(MPI_COMM_WORLD,&size);
```



CC - Broadcast

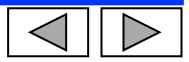
```
/* somente processo 0 inicializa nome */
if (rank==0)
     strcpy(nome, “Olá Mundo!”);

/* todos executam o comando Bcast */
/*   argumento origem: processo 0 */
MPI_Bcast(&nome, 20, MPI_CHAR, 0,
                  MPI_COMM_WORLD);

printf(“%s Sou o processo %d\n”, nome, rank);
MPI_Finalize();

return(0);
}
```

CC - Broadcast

MPI_Bcast

- Exercício:

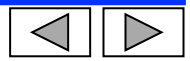
  o problema pode ser resolvido sem comunicação?



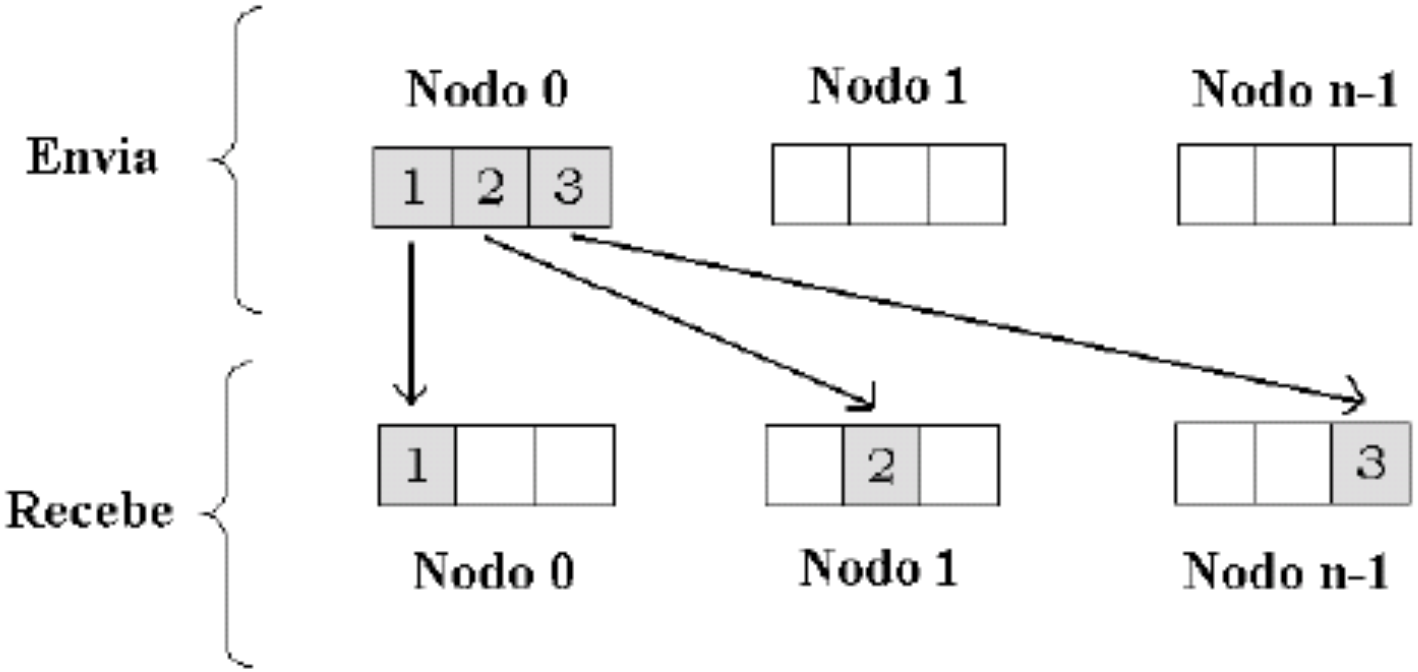
CC - Scatter

MPI_Scatter

- é a primitiva responsável por distribuir uma estrutura de dados de um processo para os demais processos do grupo;
  - incluindo o emissor
- o processo emissor e os receptores executam a mesma primitiva, com os mesmos argumentos.



CC - Scatter



CC - Scatter

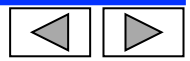
int MPI_Scatter(void *sbuf, int scnt, MPI_Datatype stype, void *rbuf, int rcnt, MPI_Datatype rtype, int root, MPI_Comm comm);

- *sbuf* - local onde estão armazenados os dados a serem enviados;
- *scnt* - quantidade de dados a serem enviados para cada processo;
- *stype* - tipo de dados a serem enviados;
- *rbuf* - local onde serão armazenados os dados recebidos;
- *rcnt* – quant. de dados a serem recebidos, por proc.;
- *rtype* - tipo de dados a serem recebidos;
- *root* - identificação do processo emissor;
- *comm* - grupo onde os processos emissor e receptor estão.

CC - Scatter

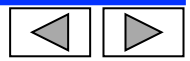
MPI_Scatter

- Notar que em muitos programas (caso comum)
  - *scnt = rcnt*;
  - *stype = rtype*
- É possível escrever programas com parâmetros distintos mas a consistência da semântica fica a cargo do programador



CC - Scatter

```
/* processo zero envia a todos uma parte */
/* processo zero também recebe uma parte */
#define N 1000
int main(int argc, char **argv){
        int size,rank,i;
        int *comp;          /* área origem */
        int *parc;          /* área destino */
        MPI_Init(&argc,&argv);
        MPI_Comm_rank(MPI_COMM_WORLD,&rank);
        MPI_Comm_size(MPI_COMM_WORLD,&size);
```



CC - Scatter

```
/* calcula dados a enviar; somente o zero */
/* envia N inteiros */
  if (rank==0){
      comp=(int  *) malloc(N*sizeof(int));
       for(i=0;i<N;i++)
           comp[i]=i;
 }
```

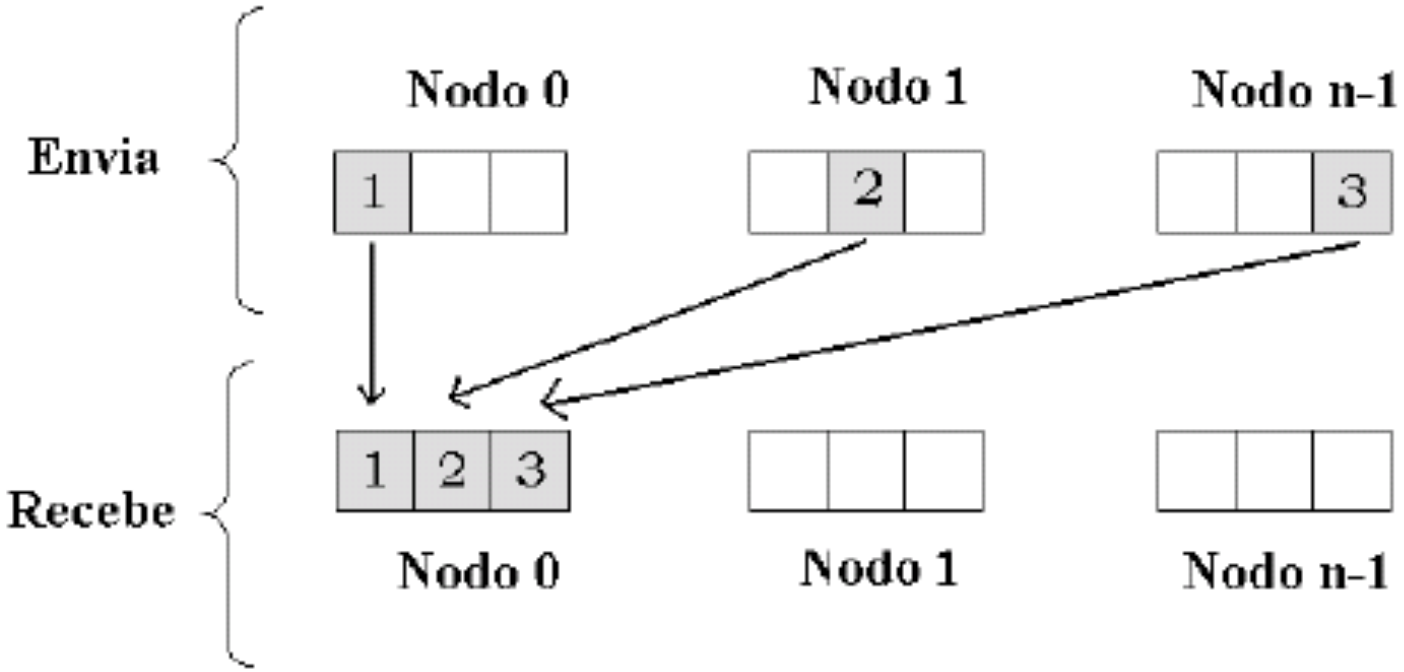
CC - Scatter

```
/* aloca área de recepção por processo */
/*      (total_de_dados / total_de_processos) * tam_dado */
/*      obs.: tamanho de int = tamanho de MPI_INT */
    parc = (int  *) malloc(N/size*sizeof(int));

    MPI_Scatter(comp, N/size, MPI_INT, parc, N/size,
                        MPI_INT, 0, MPI_COMM_WORLD);

    for (i=0;i<N/size;i++)
          printf(" %d\n ",parc[i]);

    MPI_Finalize();
    return(0);
}
```

CC - Gather

- **MPI_Gather**
  - é primitiva responsável por agrupar em apenas um processo dados que estão distribuídos em **n** processos;
  - faz o inverso da Scatter;
  - o processo emissor e os receptores executam a mesma primitiva, com os mesmos argumentos.

CC - Gather



CC - Gather

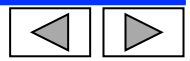
int MPI_Gather(void *sbuf, int scnt, MPI_Datatype stype, void *rbuf, int rcnt, MPI_Datatype rtype, int root, MPI_Comm comm);

- *sbuf* - local onde estão armazenados os dados a serem enviados;
- *scnt* - quantidade de dados a serem enviados;
- *stype* - tipo de dados a serem enviados;
- *rbuf* - local onde serão armazenados os dados recebidos;
- *rcnt* - quantidade de dados a serem recebidos;
- *rtype* - tipo de dados a serem recebidos;
- *root* - identificação do processo receptor;
- *comm* - grupo onde os processos emissor e receptor estão.

CC - Gather

MPI_Gather

- Notar que em muitos programas (caso comum)
  - *scnt = rcnt*;
  - *stype = rtype*
- É possível escrever programas com parâmetros distintos mas a consistência da semântica fica a cargo do programador



CC - Gather

MPI_Gather

- Se o root não é zero
  - A posição dos valores no array resultado é dada sempre pelo ID de cada processo

CC - Gather

```
#define N 1000
/* todos enviam para o processo zero */
int main(int argc, char **argv){

        int size,rank,i;
        int *num;                    /* área origem e destino */
        MPI_Init(&argc,&argv);
        MPI_Comm_rank(MPI_COMM_WORLD,&rank);
        MPI_Comm_size(MPI_COMM_WORLD,&size);
```

CC - Gather

```
/* processo zero precisa de área com N dados */
        if (rank==0)
            num=(int  *) malloc(N*sizeof(int));
        else
            num=(int  *) malloc(N/size*sizeof(int));
```

CC - Gather

```
/* todos inicializam área de send, cada um com seu ID */
for (i=0;i<N/size;i++)
      num[i]=rank;

/* zero envia e recebe da/na mesma área */
 MPI_Gather(num, N/size, MPI_INT, num, N/size,
                  MPI_INT, 0, MPI_COMM_WORLD);

 if (rank==0)
       for (i=0;i<N;i++)
            printf("%d",num[i]);

  MPI_Finalize();
  return(0);
}
```

CC - Allgather

- **MPI_Allgather**
  - faz com que todos os processos coletem dados de cada processo.
  - Sua execução é a similar que ocorreria se cada processo de um grupo efetuasse um Broadcast.

CC - Reduce

- **MPI_Reduce**
  - Definição
    - é primitiva responsável por realizar uma operação pré-definida
    - sobre dados localizados em todos os processos do grupo
    - e retorna o resultado desta operação em um único processo;
  - Inclui processamento e comunicação

CC - Reduce

int MPI_Reduce(void *sbuf, void *rbuf, int count, MPI_Datatype stype, MPI_Op op, int root, MPI_Comm comm);

- *sbuf* - local onde estão armazenados os dados a serem enviados;
- *rbuf* - local onde serão armazenados os dados recebidos;
- *count* - quantidade de dados a serem enviados;
- *stype* - tipo de dados a serem enviados;
- *op* - operação a ser efetuada entre os dados;
- *root* - identificação do processo receptor;
- *comm* - grupo onde os processos emissor e receptor estão.

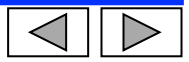
CC - Reduce

As operações processadas pela primitiva **MPI_Reduce**.

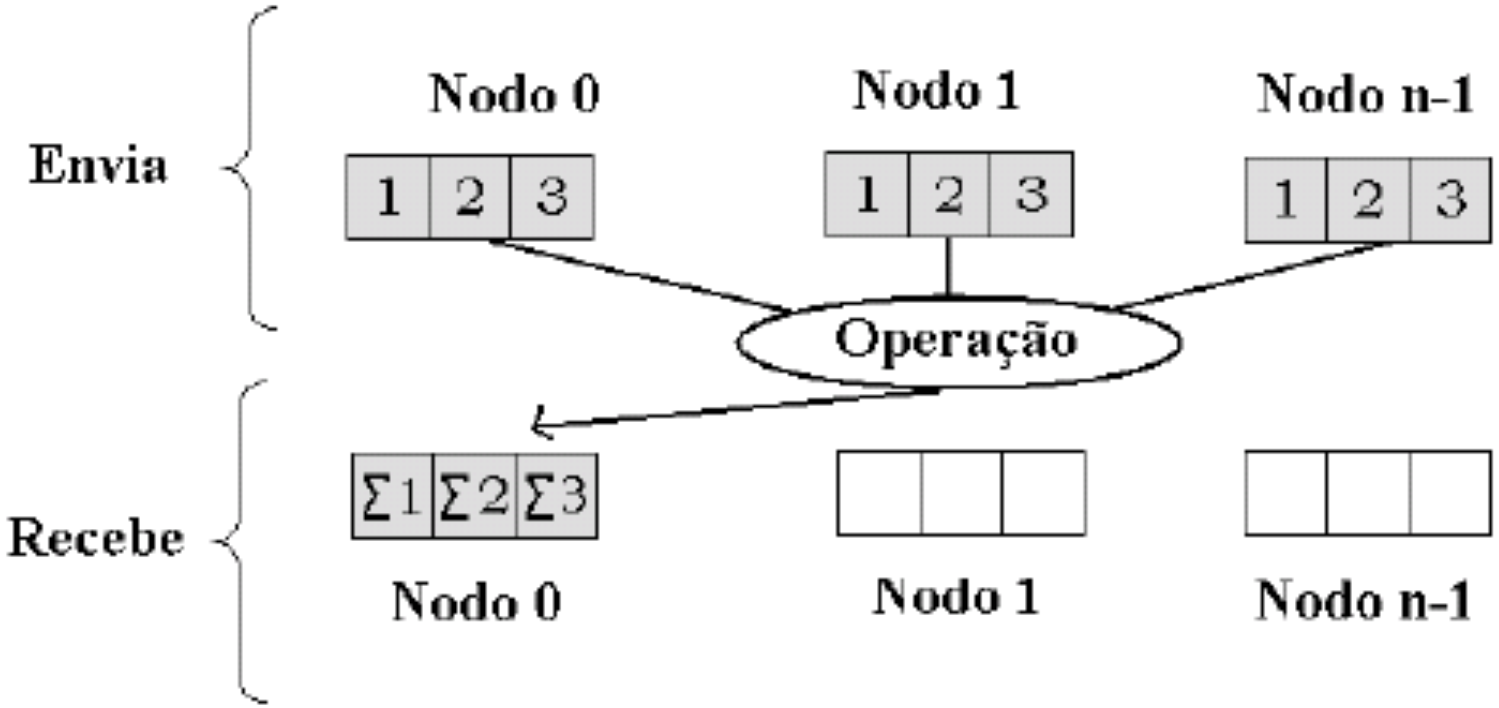
1	MPI_MAX	Valor Máximo
2	MPI_MIN	Valor Mínimo
3	MPI_SUM	Somatório
4	MPI_PROD	Produto
5	MPI_LAND	Valor Máximo
6	MPI_BAND	Valor Mínimo
7	MPI_LOR	Operação Lógica OR
8	MPI_BOR	Operação Lógica OR bit a bit
9	MPI_LXOR	Operação Lógica XOR
10	MPI_BXOR	Operação Lógica XOR bit a bit
11	MPI_MAXLOC	Máximo valor e localização
12	MPI_MINLOC	Mínimo valor e localização

CC - Reduce

- **MPI_Reduce**
  - Operações dependem do tipo de dados



CC - Reduce


Envia
Nodo 0
Nodo 1
Nodo n-1
1 2 3
1 2 3
1 2 3
Operação
Recebe
Σ1 Σ2 Σ3
Nodo 0
Nodo 1
Nodo n-1

CC - Reduce

```
/* calcula ? (exercício) */
#define N 1000

int main(int argc, char **argv) {
        int rank,size,i *vet, restotal, resparcial=0;

      MPI_Init(&argc,&argv);
      MPI_Comm_rank(MPI_COMM_WORLD,&rank);
      MPI_Comm_size(MPI_COMM_WORLD,&size);

      /* cada processo precisa de parte dos dados: N/size */
      vet = (int *)malloc(N/size*sizeof(int));
```

CC - Reduce

```
/* todos inicializam seu vetor */
for(i=0;i<N/size;i++)
    vet[i] = rank;

/* cada um calcula sua parte */
for(i=0;i<N/size;i++)
    resparcial += vet[i] * vet[i];
```

CC - Reduce

```
    /* soma geral; tamanho por proc: 1 */
    MPI_Reduce(&resparcial, &restotal, 1, MPI_INT,
                        MPI_SUM, 0, MPI_COMM_WORLD);

    /* imprime resultado final */
    if (rank==0)
        printf("Produto Escalar %d\n", restotal);

     MPI_Finalize();
     return(0);
}
```

Informática UFRGS

CC - Reduce

- Exercício sobre exemplo Reduce (anterior)
  - Qual é a operação geral resolvida pelo programa?
  - Qual o resultado para 4 processos?
  - É possível aumentar o tamanho dos dados no Reduce? Efeito?

UFRGS
UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

CC - Allreduce

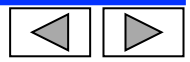
- **MPI_Allreduce**
  - A primitiva **MPI_Allreduce** é semelhante a **MPI_Reduce**, sendo que difere somente pelo fato de que o resultado desta é enviado para todos os processos do grupo.

Exemplo: algoritmo?

```
#define N 100

int main(int argc, char **argv){
    int size,rank,j,i;
    int *mat,*matparc, *vet, *res;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Comm_size(MPI_COMM_WORLD,&size);

    if (rank == 0) {
        mat = (int *)malloc(N*N*sizeof(int));
        res = (int *)malloc(N*sizeof(int));
    } else res = (int *)malloc(N/size*sizeof(int));
```

Exemplo : algoritmo?

```
matparc = (int *)malloc(N/size*N*sizeof(int));
vet = (int *)malloc(N*sizeof(int));

/* preenche com valores iniciais */
if (rank == 0){
    for(i=0;i<N;i++)
        for (j=0;j<N;j++)
            mat[i*N+j] = rand()% 100;
    for (i=0;i<N;i++)
        vet[i] = rand()%100;
}
```



Exemplo : algoritmo?

```
MPI_Scatter(mat, N/size*N, MPI_INT, matparc,
            N/size*N, MPI_INT, 0, MPI_COMM_WORLD);

MPI_Bcast(vet,N,MPI_INT,0,MPI_COMM_WORLD);
```

Exemplo : algoritmo?

```
for (i=0;i<N/size;i++){
      res[i]=0;
      for (j=0;j<N;j++)
           res[i] += matparc[i*N+j] * vet[j];
 }

 MPI_Gather(res, N/size, MPI_INT, res, N/size,
                    MPI_INT, 0, MPI_COMM_WORLD);
 if (rank==0)
    for (i=0;i<N;i++)
          printf("%d\n",res[i]);

 MPI_Finalize();
 return(0);
}
```

Exemplo : (Matriz x Vetor)

- Exercício exemplo all-reduce
  - Descreva (explique) o programa indicando
    - Quais os principais passos?
    - Qual a comunicação?
    - Quem faz o que?

Dados Híbridos

- Um dos tipos de dados existentes no MPI é o **PACKED**
- Este tipo de dado vem a ser uma estrutura de pacote onde diferentes dados foram empacotados.
- O **PACKED** possibilita que em apenas uma primitiva de envio ou de recebimento vários tipos de dados sejam enviados;

- O MPI possui duas primitivas para o uso de pacotes:
  - **MPI_Pack** - Responsável pela inclusão de um dado em um pacote;
  - **MPI_Unpack** - Responsável pela retirada de um dado em um pacote;

Dados Híbridos

- Para garantir a consistência dos dados a ordem de retirada deve ser a mesma aplicada na inclusão dos dados.
- Ou deve-se fazer um controle com o ponteiro que indica o posicionamento dos dados no pacote.

- Contras
  - Código é menos legível
  - Facilita a introdução de bugs

Empacotamento de Dados

- **MPI_Pack**
  - é primitiva responsável por empacotar qualquer tipo de dados em apenas um pacote;

Empacotamento de Dados

int MPI_Pack(void *buff, int count, MPI_Datatype datatype,void *outbuff, int outcount, int posi, MPI_Comm comm);

- *buff* - local onde estão os dados a serem empacotados;
- *count* - número de dados a serem empacotados;
- *datatype* - tipo de dados a serem empacotados;
- *outbuff* - local onde estarão os dados sendo empacotados;
- *outcount* - número máximo de dados do pacote;
- *posi* - posição dos dados empacotados até o momento;
- *comm* - grupo onde os processos emissor e receptor estão.

Desempacotamento de Dados

- **MPI_Unpack**
  - é primitiva responsável por desempacotar quaisquer tipo de dados de apenas um pacote;

Desempacotamento de Dados

int MPI_Unpack(void *buff, int count,int posi,void *outbuff, int outcount, MPI_Datatype datatype, MPI_Comm comm);

- *buff* - local onde estão os dados a serem desempacotados;
- *count* - número de dados a serem desempacotados;
- *posi* - posição dos dados desempacotados até o momento;
- *outbuff* - local onde estarão os dados desempacotados;
- *count* - número (total) máximo de dados do pacote;
- *datatype* - tipo de dados a ser desempacotado;
- *comm* - grupo onde os processos emissor e receptor estão.

Empacota/Desempacota

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

int main(int argc, char *argv[]){
    int id, num_proc;       int b, posicao = 0;
    char a, buffer[100];    float c;

    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&id);
    MPI_Comm_size(MPI_COMM_WORLD,&num_proc);
```

Empacota/Desempacota

```
if (id==0) {
    a='a';b=2;c=2.15;
    MPI_Pack(&a, 1, MPI_CHAR, buffer, 100,
                &posicao, MPI_COMM_WORLD);
    MPI_Pack(&b, 1, MPI_INT, buffer, 100, &posicao,
                MPI_COMM_WORLD);
    MPI_Pack(&c, 1, MPI_FLOAT, buffer, 100,
                &posicao, MPI_COMM_WORLD);
    MPI_Bcast(buffer, 100, MPI_PACKED, 0,
                MPI_COMM_WORLD);
}
```

Empacota/Desempacota

```
    else {
            MPI_Bcast(buffer, 100, MPI_PACKED, 0,
                            MPI_COMM_WORLD);
            MPI_Unpack(buffer, 100, &posicao, &a, 1,
                            MPI_CHAR,MPI_COMM_WORLD);
            MPI_Unpack(buffer, 100, &posicao, &b, 1,
                            MPI_INT, MPI_COMM_WORLD);
            MPI_Unpack(buffer, 100, &posicao, &c, 1,
                            MPI_FLOAT, MPI_COMM_WORLD);
          }

     printf("processo %i a=%c b=%d c=%f\n",id,a,b,c);
     MPI_Finalize();
}
```

Empacota/Desempacota

- Exercício sobre exemplo anterior
  - Quem envia e o que?
  - Quem recebe e o que?

Sincronização

- Algumas operações que envolvem todos os processos causam uma sincronia global.
- Mas dependendo do aplicação, pode ser necessário um ponto de sincronismo explícito sem nenhuma operação
- Este ponto pode ser obtido através de uma barreira.
- Vale ressaltar que a inclusão de uma barreira tende a diminuir o desempenho de uma aplicação.

MPI_Barrier(MPI_Comm comm);

- *comm* - grupo onde os processos emissor e receptor estão.

Sincronização

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

int main(int argc, char *argv[]){
    int id, num_proc;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&id);
    MPI_Comm_size(MPI_COMM_WORLD,&num_proc);
        ...
    MPI_Barrier(MPI_COMM_WORLD);
        ...
    MPI_Finalize();
}
```

Avaliação do desempenho

- As métricas mais utilizadas para avaliar o desempenho de uma aplicação paralela são:
  - **Tempo de Execução** –
    - o MPI disponibiliza a primitiva MPI_Wtime que retorna o tempo do relógio em precisão dupla;
  - ***Speedup*** –
    - é calculado dividindo-se o tempo de execução seqüencial pelo tempo de execução paralelo;

Avaliação do desempenho

- **Escalabilidade** –
  - é uma métrica baseada na eficiência.
  - Um programa é escalável se sua eficiência não diminui (muito) com o aumento do número de processos.

- **Eficiência** –
  - é calculada dividindo-se o speedup pelo número de processadores utilizados;

Avaliação do desempenho

```
int main(int argc, char *argv[]){
    int id, num_proc;
    double inicio,fim;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&id);
    MPI_Comm_size(MPI_COMM_WORLD,&num_proc);
    inicio = MPI_Wtime();
        ...
    fim = MPI_Wtime();
    if (id==0)
        printf("tempo = %e\n", fim - inicio);
    MPI_Finalize();
}
```

Problemas no desenvolvimento

- O desenvolvimento de aplicações paralelas pode apresentar problemas que não existiam no desenvolvimento seqüencial;
- O **principal fator**
  - que pode causar problema em uma aplicação paralela é o não determinismo da execução em conjunto de todos os processos em execução.
- Os principais problemas que podem ocorrer são:
  - *Deadlock;*
  - Inconsistência de dados;

Problemas no desenvolvimento

- O *deadlock*
  - ocorre quando um ou mais processos ficam esperando por um evento que não vai ocorrer;
  - Por exemplo, em *receive* de mensagem não enviada

- Na programação por troca de mensagens um exemplo de evento que pode causar *deadlock* é
  - o recebimento de mensagens
  - basta que dois processos queiram trocar dados ambos querendo receber antes;

Problemas no desenvolvimento

- Deadlock x buffer
  - Embora seja mais raro de ocorrer, o envio de mensagens também pode causar deadlock
  - Basta que o buffer existente não seja suficiente para todas as comunicações;
- Erros na comunicação
  - Além do deadlock, existe a possibilidade de ocorrer inconsistência dos dados
  - Na maioria dos casos é resultante de comunicações incorretas.

Informática UFRGS

Depuração

- O processo de **depuração**
  - consiste em verificar passo a passo quais as ações que estão sendo efetuadas por cada processo e sobre quais dados;

- Ferramentas x técnicas
  - Existem algumas ferramentas para depuração
  - Mas a estrategia mais usada é a geração de *trace* de ações
    - fazendo-se uso comandos de escrita como por exemplo **printf**.

UFRGS
UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

Informática UFRGS

Depuração

Entre as ferramentas podemos citar:

- GDB (Sequencial);
- Upshot (Troca de mensagens);
- Jumpshot (Troca de mensagens).

UFRGS
UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

Informática UFRGS

Resumo

- MPI
  - Padrão (especificação) de bibliotecas para programação paralela
  - Em plataformas distribuídas
  - Uso de troca de mensagens
  - Diversas implementações
  - Mais usado em clusters, máquinas paralelas proprietárias (IBM, NEC, Hitachi, ...)

UFRGS
UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

Informática UFRGS

Resumo

- MPI
  - Modelo de programa SPMD
  - 3 grupos de primitivas
    - Controle
    - Send/receive ponto-a-ponto
    - Comunicação coletiva

UFRGS
UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

Exercícios

- Exercícios:
  - A) explique como passar um algoritmo PRAM (Jaja) para MPI?
    - global -> local (entrada)
    - local -> global (resultado)
    - local -> local (dependência de dados)
  - B) reavalie o desempenho de algum dos algoritmos PRAM em MPI: complexidade (tempo), speedup, eficiência

Informática UFRGS

Revisão

- Revisão
  - Porque é importante uma metodologia para PP?
    - Porque somente o estudo de algoritmos paralelos não é suficiente?
  - Quais os principais passos da metodologia proposta por Foster?

UFRGS
UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

Revisão

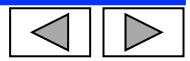
- Revisão
  - O que é feito no passo particionamento?
  - O que é feito no passo comunicação?
  - O que é feito no passo aglomeração?
  - O que é feito no passo mapeamento?
  - Porque comunicação e particionamento (e aglomeração) freqüentemente se opõem?
  - Porque é difícil fazer aglomeração e mapeamento durante a projeto e codificação do programa?

Revisão

- Revisão:
  - o que é MPI?
  - quem apoiou a criação de MPI?
  - qual o modelo de concorrência e código?
  - quais os grupos de primitivas?
  - como um programa MPI é gerado?
  - como um programa MPI é executado?
    - em que máquinas?

Revisão

- Revisão:
  - como um programa MPI é iniciado?
  - e finalizado?
  - que processos trabalham antes/depois do início/ fim MPI?
  - como é feita a nomeação dos processos?
  - como o programador sabe a quantidade de processos?

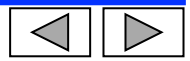


Revisão

- Revisão:
  - quais os principais argumentos das primitivas send/receive?
  - quais as opções de sincronização em send/ receive?
  - o que ocorre se tamanhos (args.) de send/receive diferem?
  - o que ocorre se tamanho da área de receive é menor que arg tam do receive?
  - idem se maior?

Revisão

- Revisão:
  - o que são primitivas de comunicação coletiva (CC)MPI?
  - quais são os dois principais grupos de CC?
  - qual primitiva se usa para receber uma comunicação coletiva?



Revisão

- Revisão:
  - qual a função da primitiva broadcast?
  - quais os principais argumentos?
  - qual a função da scatter?
  - qual a função da gather?
  - quais os principais argumentos dessas 2?
  - qual a função da algather?
  - qual a função geral da reduce?
  - como é possível enviar dados de vários tipos na mesma mensagem?

Informática UFRGS

Referências

[1] PACHECO, P. S. **Parallel Programming with MPI**. San Francisco: Morgan Kaufmann Publisher, p. 419, 1997.

[2] SNIR, M.; OTTO, S.; HUSS-LEDERMAN, S.; WALKER, D.; DONGARRA, J. **MPI: The Complete Reference**. Boston: The MIT Press, 1996.

UFRGS
UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL