Programação Paralela – OPRP001

Programação com MPI

Prof. Guilherme Koslovski e Prof. Maurício Pillon

Referências

- Cursos da ERAD
  - http://www2.sbc.org.br/erad/doku.php?id=start
- MPI fórum
  - http://www.mpi-forum.org
  - http://www.mpi-forum.org/docs/docs.html

Agenda

- **Programação Paralela com Troca de Mensagens**
- MPI
- Exemplos
- Considerações finais

Programação com Troca de Mensagens

- Opções de programação
  - Linguagem de programação paralela (específica)
    - Occam (Transputer)
  - Extensão de linguagens de programação existentes
    - CC++ (extensão de C++)
    - Fortran M
    - Geração automática usando anotações em código e compilação (FORTRAN)
  - Linguagem padrão com biblioteca para troca de mensagens
    - MPI (Message Passing Interface)
    - PVM (Parallel Virtual Machine)
    - ProActive

Linguagem padrão com biblioteca para troca de mensagens

- Descrição explícita do paralelismo e troca de mensagens entre processos
- Métodos principais
  - Criação de processos para execução em diferentes computadores
  - Troca de mensagens (envio e recebimento) entre processos
  - Sincronização entre processos

SPMD e MPMD

- SPMD (*Single Program Multiple Data*)
  - Existe somente um programa
  - O mesmo programa é executado em diversas máquinas sobre um conjunto de dados distinto
- MPMD (*Multiple Program Multiple Data*)
  - Existem diversos programas
  - Programas diferentes são executados em máquinas distintas
  - Cada máquina possui um programa e conjunto de dados distinto

Criação de processos

- Criação estática de processos
  - Processos especificados antes da execução
  - Número fixo de processos
  - Mais comum com o modelo SPMD
- Criação dinâmica de processos
  - Processos criados durante a execução da aplicação (spawn)
  - Destruição também é dinâmica
  - Número de processos variável durante execução
  - Mais comum com o modelo MPMD

Troca de mensagens

- Primitivas *send* e *receive*
  - Comunicação síncrona (bloqueante)
    - Send bloqueia emissor até receptor executar receive
    - Receive bloqueia receptor até emissor enviar mensagem
  - Comunicação assíncrona (não bloqueante)
    - Send não bloqueia emissor
    - Receive pode ser realizado durante execução
      - Chamada é realizada antes da necessidade da mensagem a ser recebida, quando o processo precisa da mensagem, ele verifica se já foi armazenada no buffer local indicado

Troca de mensagens

- Seleção de mensagens
  - Filtro para receber uma mensagem de um determinado tipo *(message tag)*, ou ainda de um emissor específico
- Comunicação em grupo
  - **Broadcast**
    - Envio de mensagem a todos os processos do grupo
  - **Gather/scatter**
    - Envio de partes de uma mensagem de um processo para diferentes processos de um grupo (distribuir), e recebimento de partes de mensagens de diversos processos de um grupo por um processo (coletar)

Sincronização

- Barreiras
  - Permitem especificar pontos de sincronismo entre diversos processos
  - Um processo que chega a uma barreira só continua quando todos os outros processos do seu grupo também chegam a barreira
  - O último processo libera todos os demais bloqueados

Agenda

- Programação Paralela com Troca de Mensagens
- **MPI**
- **Exemplos**
- Considerações finais

Message Passing Interface (MPI)

- MPI fórum definiu o primeiro padrão em 1994
- Troca de mensagens entre processos
- Versão atual: MPI-2.2 e especificação MPI-3
- Implementações mais usadas
  - mpich
  - lammpi
  - Java-mpi
- SPMD -> Single Program Multiple Data

Message Passing Interface (MPI)

- Cenário
  - 1 processo inicia a execução
  - Execução ocorre em um conjunto de computadores
- Mais de 125 funções!!
- Compiladores
  - mpicc
  - mpif77
- mpirun -> dispatcher de execução
- Exemplo
  - mpicc o- teste teste.c
  - mpirun –np 4 teste

Diretivas Básicas

```
int MPI_Init(int *argc, char *argv[])
```

- Inicializa um processo MPI, e estabelece o ambiente necessário para sua execução. Sincroniza os processos para o início da aplicação paralela.

```
int MPI_Finalize()
```

- Finaliza um processo MPI. Sincroniza os processos para o término da aplicação paralela.

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

int main(int argc, char **argv)
{
	MPI_Init(&argc, &argv);

	printf("Hello World!\n");

	MPI_Finalize();
	return 0;
}
```

Diretivas Básicas

*int MPI_Comm_size(MPI_Comm comm, int *size)*

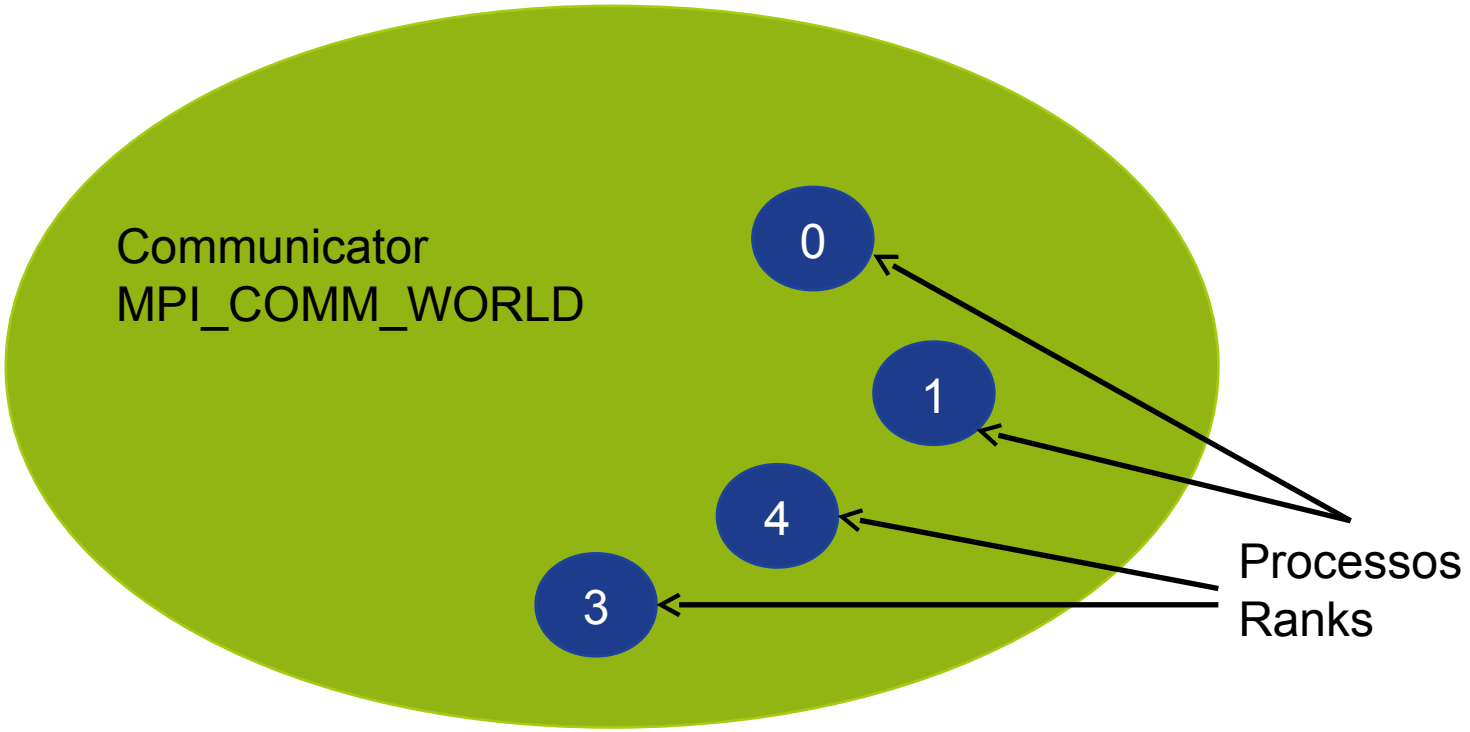
- Retorna o número de processos dentro do grupo.

*int MPI_Comm_rank(MPI_Comm comm, int *rank)*

- Identifica um processo MPI dentro de um determinado grupo. O valor de retorno está compreendido entre 0 e (número de processos)-1.

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

int main(int argc, char **argv)
{
    int rank, size;

    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    printf("Hello World! I'm %d of %d\n",rank,size);

    MPI_Finalize();
    return 0;
}
```

Componentes

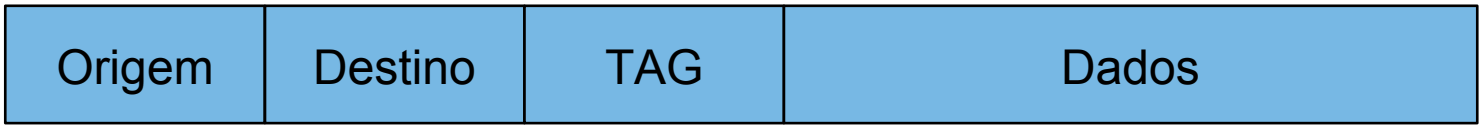


Comunicação

- A comunicação pode ser bloqueante ou não bloqueante.
- Funções bloqueantes:
  - MPI_Send(), MPI_Recv()
- Funções não bloqueantes;
  - MPI_Isend(), MPI_Irecv(), MPI_Wait(), MPI_Test()

Mensagens

Basicamente….



Comunicação

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

- sndbuf: dados a serem enviados
- count: número de dados
- datatype: tipo dos dados
- dest: rank do processo destino
- tag: identificador
- comm: comunicador

Comunicação

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

- recvbuf: área de memória para receber dados
- count: número de dados a serem recebidos
- datatype: tipo dos dados
- source: processo que envio mensagem
- tag: identificador
- comm: comunicador
- status: informação de controle

Comunicação

Tipos de dados (MPI_Datatype):

- MPI_CHAR
- MPI_DOUBLE
- MPI_FLOAT
- MPI_INT
- MPI_LONG
- MPI_LONG_DOUBLE
- MPI_SHORT
- MPI_UNSIGNED_CHAR
- MPI_UNSIGNED
- MPI_UNSIGNED_LONG
- MPI_UNSIGNED_SHORT

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

int main(int argc, char **argv){
        int rank,size,i;
        int tag=0;
        MPI_Status status;
        char msg[20];
        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Comm_size(MPI_COMM_WORLD, &size);
        if(rank == 0) {
                strcpy(msg,"Hello World!\n");
                for(i=1;i<size;i++)
                        MPI_Send(msg,13,MPI_CHAR,i,tag, MPI_COMM_WORLD);
        } else {
                MPI_Recv(msg,20,MPI_CHAR,0,tag, MPI_COMM_WORLD, &status);
                printf("Message received: %s\n", msg);
        }
        MPI_Finalize();
        return 0;
}
```

Comunicação

*int MPI_Isend(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)*

- buf: dados a serem enviados
- count: número de dados
- datatype: tipo dos dados
- dest: rank do processo destino
- tag: identificador
- comm: comunicador
- request: identificador da transmissão

Comunicação

*int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request)*

- buf: área de dados para receber
- count: nro de dados
- datatype: tipo dos dados
- source: rank do processo que enviou
- tag: identificador
- comm: comunicador
- request: identificador da transmissão

Exercício 01

- Construa um anel usando MPI:
  - cada processo deve receber uma mensagem do processo anterior (um inteiro);
  - somar o valor recebido com seu rank;
  - enviar a mensagem para o processo seguinte.
- O rank 0 inicia o token com 0;
- Utilize MPI_Send() e MPI_Recv()
- Prepare o ambiente de execução: [4, 8] computadores

Comunicação

*int MPI_Wait(MPI_Request *request, MPI_Status *status)*

- request: identificador da transmissão
- status: informação de controle

*int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)*

- request: identificador da transmissão
- flag: resultado do teste
- status: informação de controle

Sincronização

int MPI_Barrier(MPI_Comm comm)

- comm: comunicador

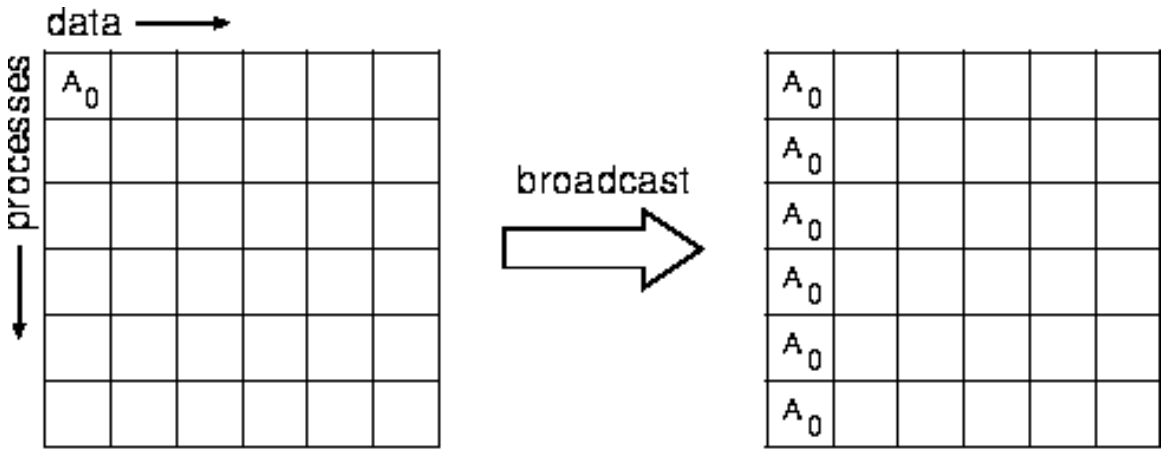
- Outras formas de sincronização:
  - Utilizando as funções de troca de mensagens bloqueantes

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char **argv) {
	int rank,size;

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

	printf("I'm %d of %d\n",rank,size);

	if(rank == 0) {
		printf("(%d) -> Primeiro a escrever!\n",rank);
		MPI_Barrier(MPI_COMM_WORLD);
	}else  {
		MPI_Barrier(MPI_COMM_WORLD);
		printf("(%d) -> Agora posso escrever!\n",rank);
	}

	MPI_Finalize();
	return 0;
}
```

Comunicação em grupo

*int MPI_BCast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm com)*

- buffer: área de memória
- count: nro de dados
- datatype: tipo dos dados
- root: rank do processo mestre
- com: comunicador

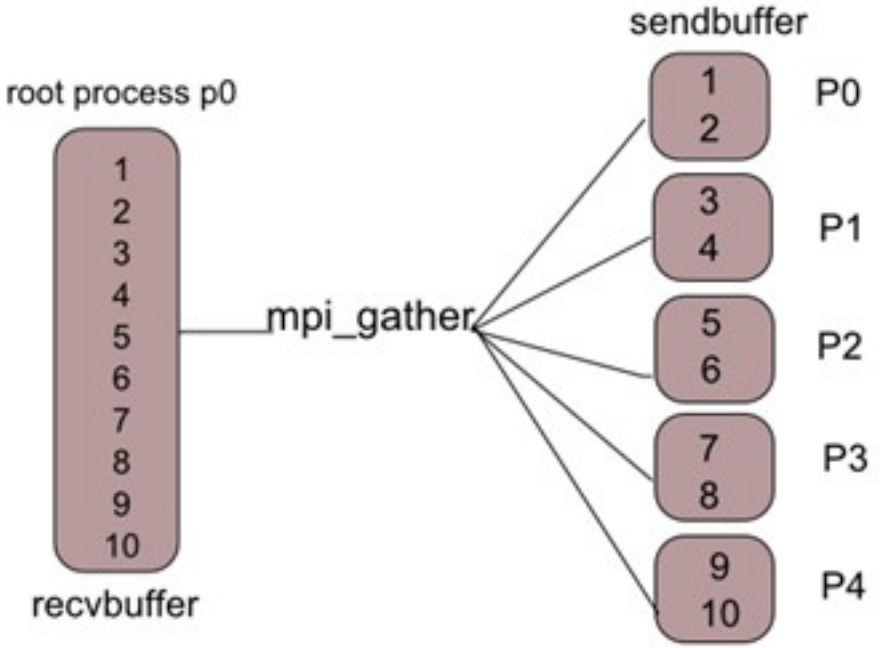


Comunicação em grupo

*int MPI_Gather(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm com)*

- sendbuf: buffer para envio
- sendcount: nro de dados a serem enviados
- sendtype: tipo dos dados
- recvbuf: buffer para recebimento
- recvcount: nro de dados para recebimento
- recvtype: tipo dos dados para recebimento
- root: processo mestre
- com: comunicador

Comunicação em grupo



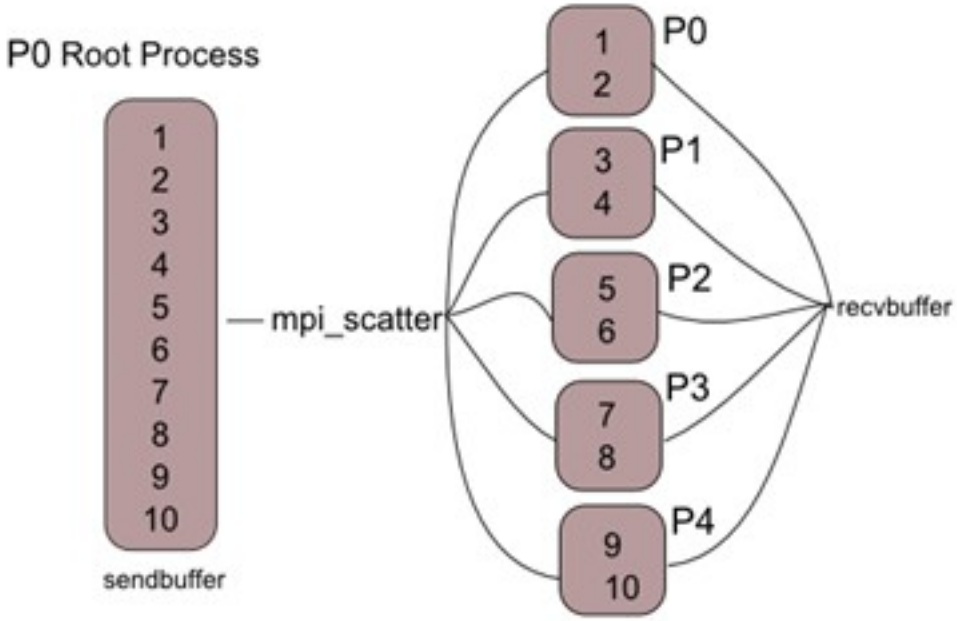
```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {
        int sndbuffer, *recvbuffer;
        int rank, size, i;

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

        recvbuffer = (int *)malloc(size*sizeof(int));
        sndbuffer = rank*rank;

        MPI_Gather(&sndbuffer, 1, MPI_INT, recvbuffer, 1,
                        MPI_INT, 0, MPI_COMM_WORLD);

        if(rank == 0) {
                printf("(%d) – Recebi vetor: ", rank);
                for(i=0; i<size; i++)
                        printf("%d ", recvbuffer[i]);
        }

        MPI_Finalize();
        return 0;
}
```

Comunicação em grupo

*int MPI_Scatter(void *sendbuf, int sendcount,*
*MPI_Datatype sendtype, void *recvbuf, int recvcount,*
MPI_Datatype recvtype, int root, MPI_Comm com)

- sendbuf: buffer para envio
- sendcount: nro de dados a serem enviados
- sendtype: tipo dos dados
- recvbuf: buffer para recebimento
- recvcount: número de dados para recebimento
- recvtype: tipo dos dados para recebimento
- root: processo mestre
- com: comunicador

Comunicação em grupo



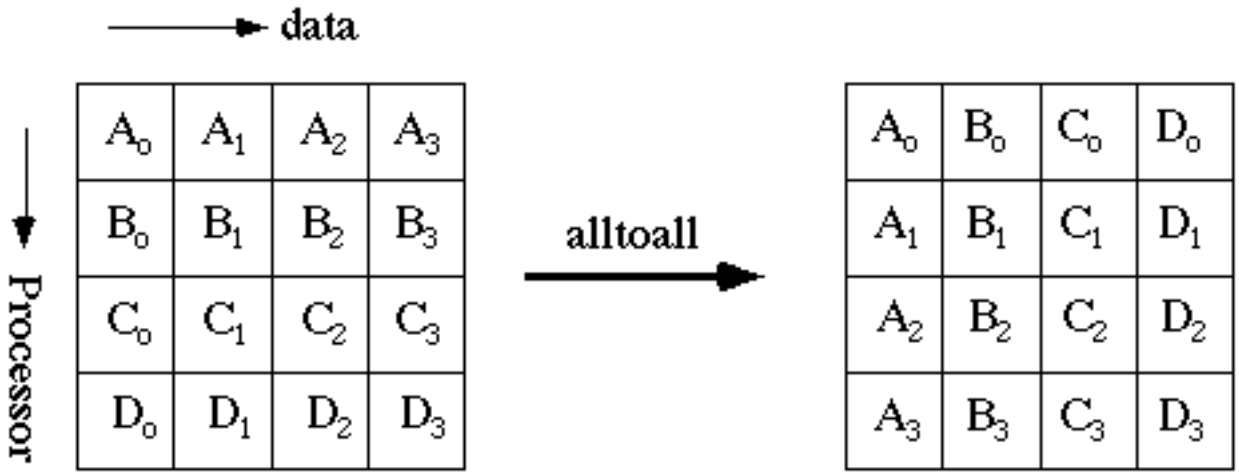
```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {
        int *sndbuffer, recvbuffer;
        int rank, size, i;

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

        sndbuffer = (int *)malloc(size*sizeof(int));

        if(rank == 0)
                for(i=0; i<size; i++) sndbuffer[i] = i*i;

        MPI_Scatter(sndbuffer, 1, MPI_INT, &recvbuffer, 1,
                                MPI_INT, 0, MPI_COMM_WORLD);
        if(rank != 0)
                printf("(%d) – Received %d\n", rank, recvbuffer);

        MPI_Finalize();
        return 0;
}
```

Comunicação em grupo

*MPI_Alltoall(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm com)*

- sendbuf: buffer para envio
- sendcount: número de dados a serem enviados
- sendtype: tipo dos dados
- recvbuf: buffer para recebimento
- recvcount: número de dados para recebimento
- recvtype: tipo dos dados para recebimento
- com: comunicador

Comunicação em grupo



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

int main(int argc, char **argv) {
        int *sndbuffer, *recvbuffer;
        int rank, size, i;

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

        sndbuffer = (int *)malloc(size*sizeof(int));
        recvbuffer = (int *)malloc(size*sizeof(int));

        for(i=0; i<size; i++) sndbuffer[i] = i*i+rank;
        printvector(rank, sndbuffer);

        MPI_Alltoall(sndbuffer, 1, MPI_INT, recvbuffer, 1, MPI_INT,
                                                MPI_COMM_WORLD);
        printvector(rank, sndbuffer);

        MPI_Finalize();
        return 0;
}
```

Comunicação em grupo

MPI_Reduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm com)

- Operações
  - MPI_MAX, MPI_MIN
  - MPI_SUM, MPI_PROD
  - MPI_LAND, MPI_BAND
  - MPI_LOR, MPI_BOR
  - MPI_LXOR, MPI_BXOR
    - etc

Agenda

- Modelos de aplicação
- Programação Paralela com Troca de Mensagens
- MPI
- **Exemplos**
- Considerações finais

Exemplos

Os arquivos estão no moodle

Considerações finais

- Ptherads, OpenMP e MPI são amplamente utilizados
- MPI define uma interface padrão para troca de mensagens entre processos distribuídos
- Diversas bibliotecas implementam esta API