ÍNDICE

Capítulo 1 - CONCEITOS BÁSICOS 3

1.1 - O QUE É PARALELISMO? 3
1.2 - PRA QUE PARALELIZAR? 3
1.3 - TIPOS DE PARALELISMO..... 3
1.4 - AMBIENTE PARALELO 4
1.5 – PROGRAMAÇÃO PARALELA..... 4
1.6 - COMO PROGRAMAR EM PARALELO COM MPI? 5
1.7 - OBSTÁCULOS NO PARALELISMO 5
1.8 - CLASSIFICAÇÃO DE FLYNN 7
1.9 - ACESSO A MEMÓRIA 8
1.10 - CONSIDERAÇÕES DE PERFORMANCE 9
1.11 – GRANULOSIDADE 10
1.12 - BALANCEAMENTO DE CARGA 11
1.13 – ESCALABILIDADE 11

Capítulo 2 - MESSAGE PASSING 12

2.1 - O QUE É O MODELO MESSAGE PASSING? 12
2.2 - MPI FORUM 12
2.3 - PADRÃO MPI 13
2.4 - ROTINAS DE MESSAGE PASSING 13
2.5 – IMPLEMENTAÇÕES 14

Capítulo 3 - INTRODUÇÃO AO MPI 15

3.1 - O QUE É O MPI? 15
3.2 - CONCEITOS BÁSICOS DE MPI 15
3.3 - COMUNICAÇÃO ENTRE OS PROCESSOS EM MPI 18
3.4 - PROGRAMANDO EM PARALELO UTILIZANDO MPI 20
3.5 - PRIMEIROS PASSOS PARA UTILIZAÇÃO DO MPI 21
3.6 - COMO UTILIZAR O MPI? 21
3.7 - NOMENCLATURA DAS ROTINAS MPI 22

Capítulo 4 - MPI BÁSICO 24

4.1 - MPI_INIT 24
4.2 - MPI_COMM_RANK 25
4.3 - MPI_COMM_SIZE 26
4.4 - MPI_SEND 27
4.5 - MPI_RECV 28
4.6 - MPI_FINALIZE 29
4.7 - PROGRAMAS EXEMPLOS 30

Capítulo 5 - MPI AVANÇADO 31

5.1 - MPI_SSEND 33
5.2 - MPI_RSEND 34
5.3 - MPI_BSEND 35
5.4 - MPI_ISEND 36
5.5 - MPI_ISSEND 37
5.6 - MPI_IRSEND 38
5.7 - MPI_IBSEND 39

5.8 - MPI_IRECV 41
5.9 - MPI_BUFFER_ATTACH 42
5.10 - MPI_BUFFER_DETACH 43
5.11 - MPI_PACK 44
5.12 - MPI_PACK_SIZE 46
5.13 - MPI_UNPACK 48
5.14 - MPI_WAIT 50
5.15 - MPI_WAITANY 51
5.16 - MPI_WAITSOME 52
5.17 - MPI_WAITALL 53
5.18 - MPI_BARRIER 54
5.19 - MPI_BCAST 55
5.20 - MPI_SCATTER 56
5.21 - MPI_GATHER 58
5.22 - MPI_ALLGATHER 60
5.23 - MPI_ALLTOALL 62
5.24 - MPI_REDUCE 64
5.25 - MPI_WTIME 66
5.26 - MPI_WTICK 67
5.27 - MPI_GET_PROCESSOR_NAME 68

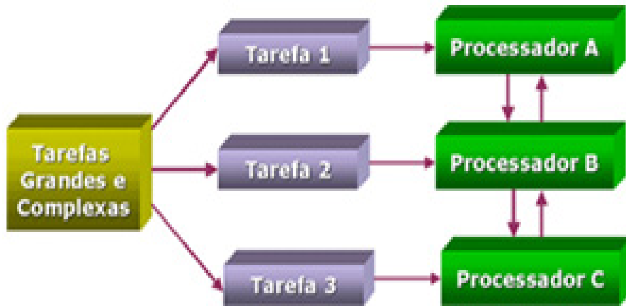
Capítulo 6 – CONCLUSÃO 69
6.1 - UM POUCO DE PVM 69
6.2 - PVM VERSUS MPI 70
6.3 – PVMPI 71
6.4 - JAVA E MPP 71
6.5 - MPI NA INTERNET 72

Capítulo 1

CONCEITOS BÁSICOS

1.1 - O QUE É PARALELISMO?

Paralelismo é uma técnica usada em tarefas grandes e complexas para obter resultados mais rápidos, dividindo-as em tarefas pequenas que serão distribuídas em vários processadores para serem executadas simultaneamente.



Esses processadores se comunicam entre si para que haja coordenação (sincronização) na execução das diversas tarefas em paralelo.

1.2 - PRA QUE PARALELIZAR?

- Os principais objetivos do paralelismo são:
  - Aumentar o desempenho (reduzindo o tempo) no processamento;
  - Resolver grandes desafios computacionais;
  - Fazer uso de um sistema distribuído para resolução de tarefas;
  - Finalmente, obter ganhos de performance.

É o paralelismo "*a solução*"?

Existem complexidades pertinentes ao próprio paralelismo;
Os resultados (performance) podem não corresponder ao esforço (programação) empregados;
O programador é diretamente responsável (em ferramentas não automáticas) pelo paralelismo.

1.3 - TIPOS DE PARALELISMO

Dentre as várias formas de classificar o paralelismo, levamos em consideração o objeto paralelizado, como segue abaixo:

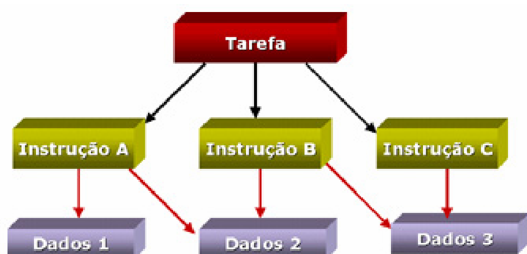
- ***Paralelismo de Dados***

O processador executa as mesmas instruções sobre dados diferentes. É aplicado, por exemplo, em programas que utilizam matrizes imensas e para cálculos de elementos finitos.

<u>*Exemplos:*</u>

- Resolução de sistemas de equações;
- Multiplicação de matrizes;
- Integração numérica.

- ***Paralelismo Funcional***

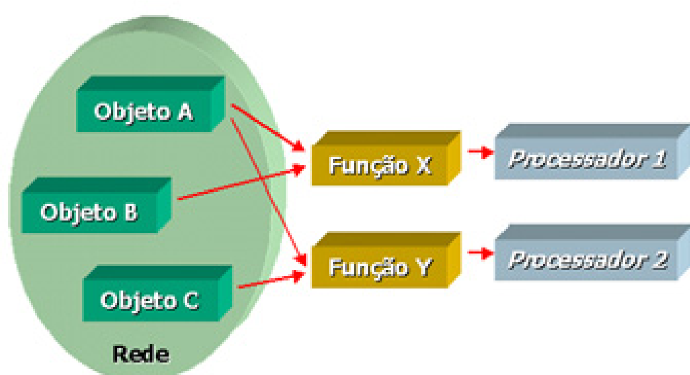


O processador executa instruções diferentes que podem ou não operar sobre o mesmo conjunto de dados. É aplicado em programas dinâmicos e modulares onde cada tarefa será um programa diferente.

Exemplos:

- Paradigma Produtor-Consumidor;
- Simulação;
- Rotinas específicas para tratamento de dados (imagens).

- ***Paralelismo de Objetos***

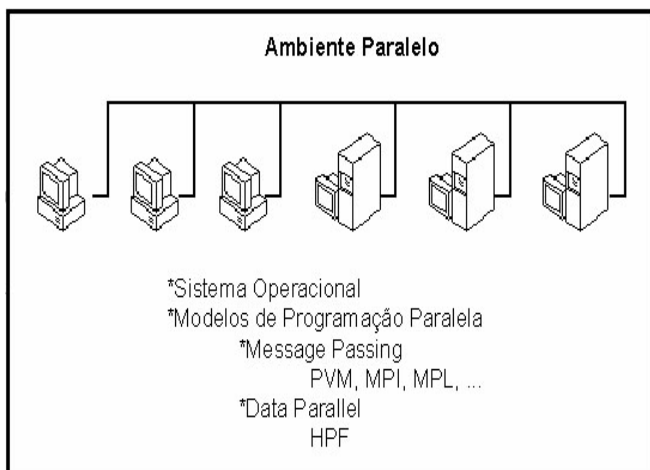


Este é o modelo mais recente, que se utiliza do conceito de objetos distribuídos por uma rede (como a Internet), capazes de serem acessados por métodos (funções) em diferentes processadores para uma determinada finalidade.

Exemplo:

- MPP - Massive Parallel Processing (Processamento Massivamente Paralelo).

1.4 - Ambiente Paralelo



Como se define um ambiente paralelo?

- Vários processadores interligados em rede
- Plataforma para manipulação de processos paralelos
- Sistema Operacional
- Linguagem de Programação
- Modelos de Programação Paralela

- Tipos de Ambientes:

Message Passing: é o método de comunicação baseada no envio e recebimento de mensagens através da rede seguindo as regras do protocolo de comunicação entre vários processadores que possuam memória própria. O programador é responsável pela sincronização das tarefas. Exemplos:

PVM - Parallel Virtual Machine
MPI - Message Passing Interface
MPL - Message Passing Library

Data Parallel: é a técnica de paralelismo de dados, normalmente automática ou semi-automática, ou seja, é o método que se encarrega de efetuar toda a comunicação necessária entre os processos de forma que o programador não necessita entender os métodos de comunicação. Exemplo:

HPF - High Performance Fortran

1.5 - Programação Paralela

Objetivo:

Algoritmos grandes e complexos são divididos em pequenas tarefas executadas simultaneamente por vários processadores que se comunicam entre si para que haja sincronização entre as tarefas em execução.

Existem as seguintes alternativas para programação paralela:

- **Bibliotecas:**
  - Construídas para classes de aplicações especiais como Álgebra Linear
  - Reduz flexibilidade
  - Manipula muitos conjuntos de entrada de dados que impedem possíveis otimizações para certos tipos de dado

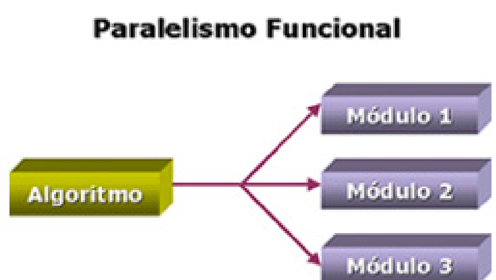
- **Ferramentas Automáticas (Computadores Paralelos):**
  - Visam auxiliar o progamador, já que, muitas vezes é difícil deduzir o algoritmo de um programa serial mesmo com análise detalhada.
  - Fazem uma análise e transformação de dados de forma autônoma.
  - As técnicas utilizadas não se adequam a todos os tipos de programas

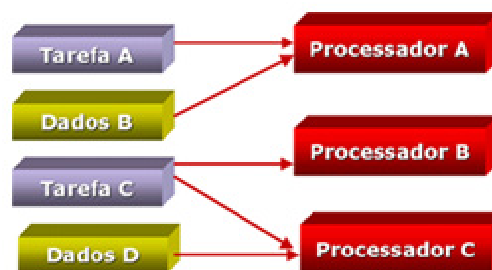
Por quê não Ferramentas Automáticas?

As ferramentas automáticas não são a "solução definitiva" para o processamento paralelo, pois:

- Podem não resolver problemas de dependências;
- Tendem a atuar sobre partes restritas do código;
- São específicas e pouco padronizadas;
- Finalmente, não superam a atividade humana.

1.6 - Como Programar em Paralelo com MPI?





Inicialmente devemos:

- Decompor o algoritmo (paralelismo funcional) e/ou decompor os dados (paralelismo de dados)
- Distribuir as tarefas ou dados entre vários processadores que trabalham simultaneamente
- Coordenar os processos em execução e a comunicação entre os processadores (sincronização)

1.7 - Obstáculos no Paralelismo

Sincronização entre os processos

As tarefas executadas em paralelo, num determinado instante, aguardam a finalização mútua para coordenar os resultados ou trocar dados e reiniciar novas tarefas em paralelo.

É necessário que haja a coordenação dos processos e da comunicação entre eles para evitar que a comunicação seja maior do que o processamento e que consequentemente haja uma queda no desempenho dos processos em execução.

Compiladores e ferramentas em amadurecimento

Existem poucos compiladores e ferramentas prontas para paralelização automática que resolvam definitivamente o problema do paralelismo. Exemplo: ForgeExplorer para linguagem Fortran.

A conversão de algoritmos sequenciais em paralelos

Há um considerável tempo gasto do programador em analisar o código fonte para paralelizar e recodificar.

Basicamente é necessário rever o programa sequencial, avaliar como será particionado, quais os pontos de sincronização, quais as partições que poderão ser executadas em paralelo e qual a forma de comunicação que será empregada entre as tarefas paralelas.

Nem tudo é paralelizável

É necessário que haja uma análise do algoritmo a ser paralelizado para que seja possível a paralelização nos dados ou nas funções do mesmo, levando sempre em consideração a quantidade de comunicação em relação ao processamento.

Tempo de processamento

No processamento paralelo, o tempo de CPU quase sempre aumenta, no entanto pode-se reduzir o tempo total de processamento.

Tecnologia Nova

Por ser uma tecnologia recente, há uma dificuldade em se desenvolver aplicações usando paralelismo.

Perda de Portabilidade

A programação paralela utiliza multiprocessadores com arquiteturas de comunicação entre os processadores baseados em memória compartilhada (shared memory) e memória distribuída (distributed memory).

Os programas adaptados a arquitetura de comunicação entre os processadores baseado em memória compartilhada não podem ser executados em uma máquina com memória distribuída, pois não utilizam os mecanismos de troca de mensagens, impossibilitando assim a portabilidade.

No caso inverso, memória distribuída para memória compartilhada, seria possível a portabilidade; porém o mecanismo de troca de mensagem seria ignorado, pois todos os dados já poderiam ser acessados pelos processadores através da memória compartilhada, isto apenas traria um fluxo desnecessário de comunicação.

Depuração (Debug)

Os processos são distribuídos e executados em vários processadores simultaneamente, entretanto não existe uma forma eficiente de acompanhar passo-a-passo a alteração das variáveis durante a execução do processamento das diversas tarefas paralelas.

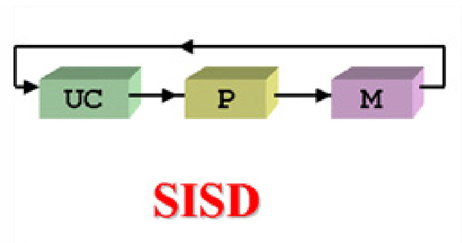
Um alternativa que está sendo empregada é a utilização de arquivos de rastro de execução, chamados tracefiles, que contem um conjunto de dados gerados durante a execução para que sejam posteriormente analisados pelo usuário após a execução do programa.

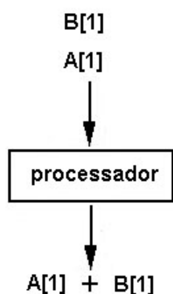
Existem programas com interface gráfica que permitem ao usuário acompanhar a evolução do desempenho durante o processamento das tarefas, como exemplo o Xmpi para MPI LAM, Paragraph, Pablo, Paradyn, PE e VT.

1.8 - Classificação de Flynn

Michael Flynn (1966) classificou as máquinas computacionais em 4 categorias, de acordo com o fluxo de dados e instruções a serem processadas simultâneamente. A seguir, damos as características desta classificação.

SISD - Single Instruction Single Data

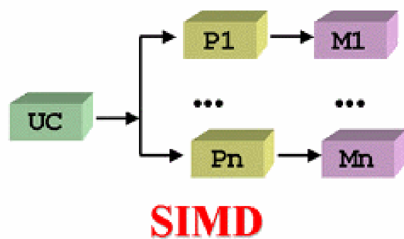




Características:

- Uma instrução é executada a cada ciclo de clock
- Cada instrução manipula um dado por vez
- Segue o modelo de Von Neumann (Entrada->Processamento->Saida)
- Possui um único processador
- A performance é medida em MIPS (milhão de instruções por segundo)

SIMD – Single Instruction Multiple Data



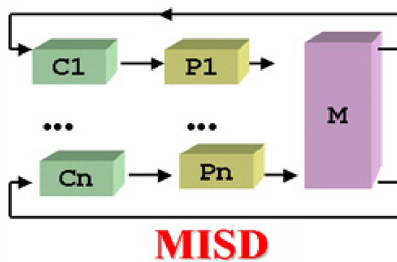
Características:

- Controle único de instruções;
- Cada instrução pode manipular mais de um dado por vez
- Tempo de acesso uniforme (arquitetura UMA - Uniform Memory Access)
- A performance é medida em MFLOPS (milhão de operações em ponto flutuante por segundo)
- Opera no modo síncrono, onde cada instrução é executada simultaneamente em cada processador

Subdivide-se em duas categorias:

SIMD Vetorial	SIMD Paralela
A[7] B[7] A[6] B[6] A[5] B[5] A[4] + B[4] A[3] + B[3] A[2] + B[2] A[1] + B[1]	B[1] A[1] → A[1] + B[1] processador 1 B[2] A[2] → A[2] + B[2] processador 2 B[3] A[3] → A[3] + B[3] processador 3

MISD - Multiple Instruction Single Data

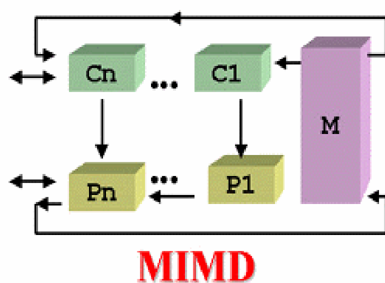


Características:

As máquinas com a arquitetura MISD são classificadas diferentemente segundo vários autores:

- Para alguns, correspondem a processadores vetoriais;
- Outros descrevem-nas como variantes de máquinas SIMD;
- Para a maioria porém, tais máquinas não têm existência prática

MIMD - Multiple Instruction Multiple Data



Características:

- Essencialmente um grupo de computadores interligados e independentes
- Os Sistemas Distribuídos utilizam essa arquitetura
- Cada processador executa suas instruções independente dos outros processadores

Vantagem:

Os processadores podem executar várias instruções simultaneamente.

Desvantagem:

Custo de Balanceamento de Carga.

1.9 - Acesso a Memória

A comunicação entre os processadores depende da arquitetura de memória utilizada no ambiente. Para entendermos como os processadores se comunicam é fundamental compreender os seguintes tipos de arquitetura de memória :

- ***Memória Compartilhada (Shared Memory)***

Características:

- Os processadores operam independentemente mas compartilham o recurso de uma única memória.
- Somente um processador acessa um endereço na memória por vez.
- Implementado em hardware

Vantagem:

- compartilhamento de dados entre os processos é mais rápido

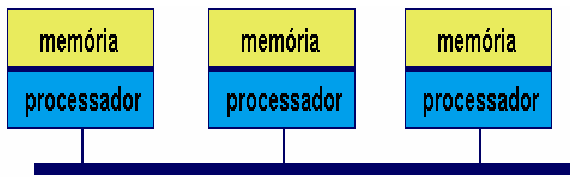
Desvantagens:

- São computadores extremamente caros
- Há um limite no número de processadores
- São necessárias técnicas de sincronização para a leitura e gravação

Exemplo:

- Cray Y-MP (Centro Nacional de Supercomputação UFRGS)

♦ ***Memória Distribuída (Distributed Memory)***



Características:

- Os processadores operam independentemente onde cada um possui sua própria memória.
- Os dados são compartilhados através da rede usando um mecanismo de troca de mensagens (Message Passing).

Vantagens:
- Acesso à memória local sem interferência;
- Não existe limite teórico para o número de processadores.

Desvantagens:
- Dificuldade para mapear as informações;
- usuário é responsável pelo sincronismo e recebimento de dados;
- Elevado c devido a comunicação.

Exemplo:
IBM Risc/6000 SP2 (Cenapad-NE)

1.10 - Considerações de Performance

Lei de Amdahl

A Lei de Amdahl determina o potencial de aumento de velocidade a partir da porcentagem paralelizável do programa.

$$Speedup = \frac{1}{1 - \%paralelizável}$$

Consideremos uma aplicação que leva T unidades de tempo quando executado em modo serial em um único processador.

Quando a aplicação é paralelizável, assumimos que o parâmetro S constitui o percentual serial e P o percentual que pode ser paralelizado.

Assim temos:

Tempo Sequencial:

$$T = Ts$$

Tempo Paralelo:

Assumindo N processadores em implementação paralela, o tempo de execução é dado pela seguinte expressão (assumindo speedup perfeito na parte da aplicação que pode ser paralelizada).

$$Tp = \left(S + \frac{P}{N}\right) \bullet T$$

O speedup que é obtido por N processadores é:

$$Sp = \frac{Ts}{Tp} = \frac{1}{S + \frac{P}{N}}$$

Ou seja:

$$\text{Speedup} = \frac{1}{\frac{P}{N} + S}$$

Onde:

P = % paralelizável
N = Nº de processadores
S = % serial

Se assumimos que a parte serial é fixa, então o speedup para infinitos processadores é limitado em:

$$\alpha = 1/S$$

Por exemplo, se $\alpha = 10\%$, então o speedup máximo é 10, até se usarmos um número infinito de processadores.

Resumindo temos:

- ***Programa Sequencial***

Em um programa que seja paralelizavel, o valor do % paralelizável do programa é igual a zero, logo o valor do speedup é 1, ou seja, não existe aumento na velocidade de processamento.

- ***Programa Totalmente Paralelo***

Num programa no qual ocorra paralelização total, o valor de % paralelizável do programa é igual a 1, logo o valor do speedup teoricamente tende a valores infinitos.

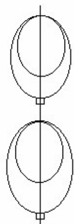
Tabela Comparativa: Para compreender melhor o significado da Lei de Ahmdahl, veja como se comportariam os casos hipotéticos da tabela abaixo:

N	P=0,50	P=0,90	P=0,99
10	1,82	5,26	9,17
100	1,98	9,17	50,25
1.000	1,99	9,91	90,99
10.000	1,99	9,91	99,02

1.11 - GRANULOSIDADE

É uma medida usada para indicar a relação entre o tamanho de cada tarefa e o tamanho total do programa, ou seja, é a razão entre computação e comunicação.

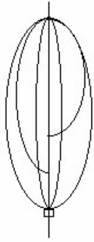
Granulosidade Fina



Características:

- Maior freqüência nas comunicações
- Facilita o balanceamento de carga
- processamento é menor do que a comunicação
- Tende a diminuir a performance do programa
- Alto custo de sincronização

Granulosidade Grossa



Características:

- Menor custo no processamento
- Pode permitir melhorar a performance do programa.
- Dificulta o balanceamento de carga
- O processamento é maior do que a comunicação
- Menor custo de sincronização

Qual a menor granulosidade?

- Tudo depende do grau de dependência entre os processos
- É importante definir o grau de controle (sincronização) necessário
- Finalmente, a própria natureza do problema pode limitar a granulosidade possível

1.12 - BALANCEAMENTO DE CARGA

É um processo específico responsável pela distribuição otimizada dos processos entre os processadores fazendo com que o tempo de execução dos processos em paralelo seja eficiente.

O usuário submete um processo ao software de balanceamento de carga, que irá fazer a distribuição do processo entre o processador que estiver mais disponível no momento. O balanceamento de carga pode fazer com que um processo fique em fila de espera aguardando um processador para ser executado.

Existem alguns softwares que executam o balanceamento de carga:

- LoadLeveler da IBM (instalado no CENAPAD-NE)
- DQS (freeware)

1.13 - ESCALABILIDADE

É a propriedade de um sistema paralelo de aumentar o speedup a medida que aumentamos o número de processadores.

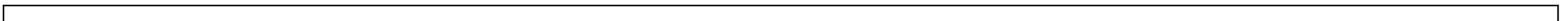
Obstáculos:

- *Hardware*

  Um dos maiores gargalos é o tráfego de informação na rede que interliga os processadores. O aumento do número de processadores pode degradar a capacidade de comunicação da rede, diminuindo a performance do programa.

- *Software*

  Alguns algoritmos trabalham melhor em certas escalas de paralelismo. O aumento do número de processadores pode acarretar uma queda na eficiência durante a execução do programa.



Capítulo 2

MESSAGE PASSING

2.1 - O QUE É O MODELO MESSAGE PASSING?

O modelo de Message Passing é um conjunto de processos que possuem acesso à memória local. As informações são enviadas da memória local do processo para a memória local do processo remoto.

A comunicação dos processos é baseada no envio e recebimento de mensagens.

A transferência dos dados entre os processos requer operações de cooperação entre cada processo de forma que operação de envio deve casar com uma operação de recebimento.

O modelo computacional Message Passing não inclui sintaxe de linguagem nem biblioteca e é completamente independente do hardware.

Message Passing tem a "última palavra" em paralelismo?

Não. O paradigma de passagem de mensagem apresenta-se apenas como uma das alternativas mais viáveis, devido a muitos pontos fortes como:

- Generalidade: Pode-se construir um mecanismo de passagem de mensagens para qualquer linguagem, como ferramentas de extensão das mesmas (bibliotecas).
- Adequação à ambientes distribuídos.

Porém, apresenta limitações:

- Primeiro de tudo, o programador é diretamente responsável pela paralelização
- Custos de comunicação podem tornar extremamente proibitiva a transmissão de mensagens em um dado ambiente.

2.2 - MPI FORUM

O padrão para Message Passing, denominado MPI (Message Passing Interface), foi projetado em um forum aberto constituído de pesquisadores, acadêmicos, programadores, usuários e vendedores, representando 40 organizações ao todo.

Itens discutidos e definidos no MPI Forum:

- Sintaxe
- Semântica
- Conjunto de rotinas padronizadas para Message Passing.

A aceitação e utilização do padrão MPI deve-se a colaboração dos seus membros:

- Representantes da Convex, Cray, IBM, Intel, Meiko, nCUBE, NEC e Thinking Machines;
- Membros de grupos de desenvolvedores de softwares, tais como, PVM, P4, Zipcode,
- Chamaleon, PARMACS, TCGMSG e Express
- Especialistas da área de Processamento Paralelo.

A documentação do MPI foi apresentada em Maio de 1994 (versão 1.0) e atualizada em Junho de 1995 (versão 1.1).

O documento que define o padrão "MPI : A Message-Passing Standard" foi publicado pela Universidade de Tennesee e encontra-se disponível via WWW na home-page do Laboratório Nacional de Argonne.
http://www.mcs.anl.gov/mpi/

O MPI Forum discute, agora, uma nova extensão para o MPI. Trata-se do MPI 2. A documentação do MPI 2 encontra-se em :
http://www.mcs.anl.gov/Projects/mpi/mpi2/mpi2.html

2.3 - PADRÃO MPI

Apresenta as seguintes características:

- ***Eficiência***

  Foi cuidadosamente projetado para executar eficientemente em máquinas diferentes.

  Especifica somente o funcionamento lógico das operações. Deixa em aberto a implementação. Os desenvolvedores otimizam o código usando características específicas de cada máquina.

- ***Facilidade***

  Define uma interface não muito diferente dos padrões PVM, NX, Express, P4, etc. e acrescenta algumas extensões que permitem maior flexibilidade.

- ***Portabilidade***

  É compatível para sistemas de memória distribuída, NOWs (network of workstations) e uma combinação deles.

- ***Transparência***

  Permite que um programa seja executado em sistemas heterogêneos sem mudanças significativas.

- ***Segurança***

  Provê uma interface de comunicação confiável. O usuário não precisa preocupar-se com falhas na comunicação.

- ***Escalabilidade***

  O MPI suporta escalabilidade sob diversas formas, por exemplo: uma aplicação pode criar subgrupos de processos que permitem operações de comunicação coletiva para melhorar o alcance dos processos.

2.4 - ROTINAS DE MESSAGE PASSING

As bibliotecas de Message Passing possuem rotinas com finalidades bem específicas, como:

Rotinas de gerência de processos :

- Inicializar e finalizar processos
- Determinar número de processos
- Identificar processos

Rotinas de comunicação Ponto a Ponto onde a comunicação é feita entre dois processos:

- Comunicação síncrona ou assíncrona
- Blocante ou não-blocante
- Empacotamento de dados

Rotinas de comunicação de grupos:

- Broadcast
- Sincronização de processos (barreiras)
- E outras

2.5 - IMPLEMENTAÇÕES

Message Passing

As duas principais bibliotecas de Message Passing da atualidade são:

- PVM: (Parallel Virtual Machine) - Possui como característica, o conceito de "máquina virtual paralela", dentro da qual processadores recebem e enviam mensagens, com finalidades de obter um processamento global.
- MPI - Nosso conhecido " message passing interface", que está se tornando um padrão na indústria.

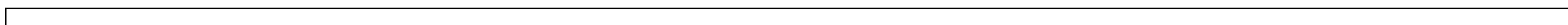
As principais implementações são:

MPI

IBM MPI: Implementação IBM para SP e clusters
MPICH: Argonne National Laboratory/Mississipi State University
UNIFY: Mississipi State University
CHIMP: Edinburgh Parallel Computing Center
LAM: Ohio Supercomputer Center
PMPIO: NASA
MPIX: Mississippi State University NSF Engineering Research Center

Disponível no CENAPADNE:

MPICH: Utilizado neste curso;
LAM MPI: Com utilitários.



Capítulo 3

INTRODUÇÃO AO MPI

3.1 - O QUE É O MPI?

MPI é uma biblioteca de Message Passing desenvolvida para ambientes de memória distribuída, máquinas paralelas massivas, NOWs (network of workstations) e redes heterogêneas.

MPI define um conjunto de rotinas para facilitar a comunicação (troca de dados e sincronização) entre processos paralelos.

A biblioteca MPI é portável para qualquer arquitetura, tem aproximadamente 125 funções para programação e ferramentas para se analisar a performance.

A biblioteca MPI possui rotinas para programas em Fortran 77 e ANSI C, portanto pode ser usada também para Fortran 90 e C++. Os programas são compilados e linkados a biblioteca MPI.

Todo paralelismo é explícito, ou seja, o programador é responsável por identificar o paralelismo e implementar o algoritmo utilizando chamadas aos comandos da biblioteca MPI.

Existem duas divisões para MPI :

- ***MPI Básico***

  O MPI básico contém 6 funções básicas indispensáveis para o uso do programador, permitindo escrever um vasto número de programas em MPI.

- ***MPI Avançado***

  O MPI avançado contém cerca de 125 funções adicionais que acrescentam às funções básicas a flexibilidade (permitindo tipos diferentes de dados), robustez (modo de comunicação non-blocking), eficiência (modo ready), modularidade através de grupos e comunicadores (group e communicator) e conveniência (comunicações coletivas, topologias).

  As funções avançadas fornecem uma funcionalidade que pode ser ignorada até serem necessárias.

Observação:
O MPI pode variar a quantidade de funções, a forma de inicialização de processos, a quantidade de cada sistema de "buffering", os códigos de retorno e erro, de acordo com os desenvolvedores de cada implementação. O MPI padrão não especifica todos os aspectos de implementação.

3.2 - CONCEITOS BÁSICOS DE MPI

Vejamos a seguir alguns conceitos básicos de MPI:

Processo

Por definição, cada programa em execução constitui um processo. Considerando um ambiente multiprocessado, podemos ter processos em inúmeros processadores.

Observação:
O MPI Padrão estipulou que a quantidade de processos deve corresponder à quantidade de processadores disponíveis.

Mensagem (message)

É o conteúdo de uma comunicação, formado de duas partes:

- **Envelope**

  Endereço (origem ou destino) e rota dos dados. O envelope é composto de três parâmetros:

  - Identificação dos processos (transmissor e receptor);
  - Rótulo da mensagem;
  - Comunicator.

- **Dado**

  Informação que se deseja enviar ou receber. É representado por três argumentos:

  - Endereço onde o dado se localiza;
  - Número de elementos do dado na mensagem;
  - Tipo do dado.

Tipos de Dados Básicos no C		Tipos de Dados Básicos no Fortran	
Definição no MPI	Definição no C	Definição no MPI	Definição no Fortran
MPI_CHAR	signed char	MPI_CHARACTER	CHARACTER(1)
MPI_INT	signed int	MPI_INTEGER	INTEGER
MPI_FLOAT	float	MPI_REAL	REAL
MPI_DOUBLE	Double	MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_UNSIGNED_SHORT	Unsigned short int	MPI_LOGICAL	LOGICAL
MPI_UNSIGNED	Unsigned int	-	-
MPI_UNSIGNED_LONG	Unsigned long int	-	-
MPI_LONG_DOUBLE	long double	-	-
MPI_LONG	Signed long int	-	-
MPI_UNSIGNED_CHAR	Unsigned char	-	-
MPI_SHORT	Signed short int	-	-
-	-	MPI_COMPLEX	COMPLEX
MPI_BYTE	-	MPI_BYTE	-
MPI_PACKED	-	MPI_PACKED	-

Rank

Todo o processo tem uma identificação única atribuída pelo sistema quando o processo é inicializado. Essa identificação é contínua representada por um número inteiro, começando de zero até N-1, onde N é o número de processos.

Rank é o identificador único do processo, utilizado para identificar o processo no envio (send) ou recebimento (receive) de uma mensagem.

Group

Group é um conjunto ordenado de N processos. Todo e qualquer group é associado a um communicator muitas vezes já predefinido como "MPI_COMM_WORLD". Inicialmente, todos os processos são membros de um group com um communicator.

Communicator

O communicator é um objeto local que representa o domínio (contexto) de uma comunicação (conjunto de processos que podem ser contactados).

O MPI utiliza esta combinação de grupo e contexto para garantir segurança na comunicação e evitar problemas no envio de mensagens entre os processos.

A maioria das rotinas de PVM exigem que seja especificado um communicator como argumento. O MPI_COMM_WORLD é o comunicador predefinido que inclui todos os processos definidos pelo usuário numa aplicação MPI.

Application Buffer

É o endereço de memória, gerenciado pela aplicação, que armazena um dado que o processo necessita enviar ou receber.

System Buffer

É um endereço de memória reservado pelo sistema para armazenar mensagens.

Dependendo do tipo de operação de envio/recebimento (send/receive), o dado no buffer da aplicação (application buffer) pode necessitar ser copiado de/para o buffer do sistema (system buffer) através das rotinas send buffer/receive buffer. Neste caso a comunicação é assíncrona.

Blocking Comunnication

Em uma rotina de comunicação blocking, a finalização da chamada depende de certos eventos.

Em uma rotina de envio de mensagem, o dado tem que ter sido enviado com sucesso, ou ter sido salvo no buffer do sistema (system buffer), indicando que o endereço do buffer da aplicação (application buffer) pode ser reutilizado.

Em uma rotina de recebimento de mensagem, o dado tem que ser armazenado no buffer do sistema (system buffer), indicando que o dado pode ser utilizado.

Non-Blocking Comunnication

Em uma rotina de comunicação non-blocking, a finalização da chamada não espera qualquer evento que indique o fim ou sucesso da rotina.

As rotinas non-blocking comunnication não esperam pela cópia de mensagens do buffer da aplicação (application buffer) para o buffer do sistema (system buffer), ou a indicação do recebimento de uma mensagem.

Overhead

Overhead é um desperdício de tempo que ocorre durante a execução dos processos. Os fatores que levam ao overhead são: comunicação, tempo em que a máquina fica ociosa (tempo idle) e computação extra.

Existem os seguintes tipos de overhead:

- *System Overhead*

  É o tempo gasto pelo sistema na transferência dos dados para o processo destino.

- *Synchronization Overhead*

  É o tempo gasto para a sincronização dos processos.

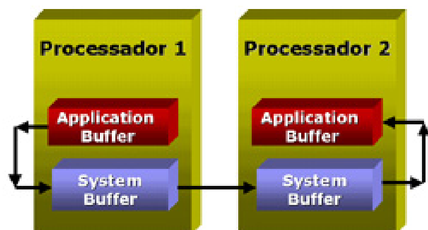
  Observação:
  O MPI sempre apresenta um overhead de sicronização inicial, quando espera até que todas as máquinas estejam disponíveis para iniciar o processamento e inicializar os processos. O mesmo ocorre quando da conclusão: existe um delay até que todos os processos possam encerrar adequadamente e terminarem a execução.

3.3 - COMUNICAÇÃO ENTRE OS PROCESSOS EM MPI

Técnicas de Comunicação entre os Processos

As técnicas de comunicação entre os processos são:

Buffering



É a cópia temporária de mensagens entre endereços de memória efetuada pelo sistema como parte de seu protocolo de comunicação.
A cópia ocorre entre o buffer do usuário (application buffer) definido pelo processo e o buffer do sistema (system buffer) definido pela biblioteca.

Utilização:

- Muitas vezes, pode ser necessário utilizar explicitamente um novo local para armazenar os dados.
- Um exemplo disso ocorre quando se pretende enviar uma grande massa de dados.

Blocking

A rotina de comunicação é blocking quando a finalização da execução da rotina é dependente de determinados eventos, ou seja, espera por determinada ação antes de liberar a continuação do processamento.

Utilização:

- Quando se deseja obter confirmação de recebimento de determinado dados, para continuar o processamento;
- Sempre que se criam "barreiras" de controle entre os processos.

Non-Blocking

A rotina de comunicação é non-blocking quando a finalização da execução da rotina não depende de determinados eventos, ou seja, o processo continua sendo executado normalmente sem haver espera.

Utilização:

- Quando não existem dependências ou necessidade de controle intensivo.

Assíncrono

É a comunicação na qual o processo que envia a mensagem, não espera que haja um sinal de recebimento da mensagem pelo destinatário.

Utilização:

- Em comunicação confiável, principalmente.

Síncrono

É a comunicação na qual o processo que envia a mensagem, não retorna a execução normal enquanto não haja um sinal do recebimento da mensagem pelo destinatário.

Utilização:

- Quando, por questões de segurança (como por exemplo, devido às instabilidades da rede), se torna necessária uma confirmação da comunicação.

Tipos de Comunicação no MPI

Existem os seguintes tipos de comunicação entre os processos:

Ponto a Ponto

Dentre as rotinas básicas do MPI estão as rotinas de comunicação ponto a ponto que executam a transferência de dados entre dois processos.

Existem quatro (4) modos de comunicação ponto a ponto:

♦ ***Synchronous (Síncrono)***

É a comunicação na qual o processo que envia a mensagem, não retorna a execução normal enquanto não haja um sinal do recebimento da mensagem pelo destinatário.

♦ ***Ready (Imediata)***

Neste tipo de comunicação, sabe-se a priori que o recebimento (receive) de uma mensagem já foi efetuado pelo processo destino, podendo-se enviá-la (send) sem necessidade de confirmação ou sincronização, melhorando o desempenho na transmissão.

♦ ***Buffered (Buferizada)***

No modo buffered a operação de envio (send) utiliza uma quantidade de espaço específica para o buffer definida pelo usuário (application buffer). Isto se tornará necessário quando a quantidade de mensagem a ser enviada ultrapassar o tamanho padrão do buffer de 4Kbytes para um processo.

♦ ***Standard (Padrão)***

É um modo padrão para o envio (send) de mensagens, buscando um meio termo entre eficiência e segurança, usando comunicação blocking.

Além dos tipos de comunicação citados acima, temos duas formas de chamadas a rotinas:

♦ ***Blocking (Blocante)***

A rotina de comunicação é blocking quando a finalização da execução da rotina é dependente de determinados eventos, ou seja, espera por determinada ação antes de liberar a continuação do processamento.

♦ ***Non-Blocking (Não-Blocante)***

A rotina de comunicação é non-blocking quando a finalização da execução da rotina não depende de determinados eventos, ou seja, o processo continua sendo executado normalmente sem haver espera.

Coletiva

É a comunicação padrão que invoca todos os processos em um grupo (group) - coleção de processos que podem se comunicar entre si. Normalmente a comunicação coletiva envolve mais de dois processos. As rotinas de comunicação coletiva são voltadas para a comunicação / coordenação de grupos de processos.

Existem os seguintes tipos de comunicação coletiva:

♦ ***Broadcast (Difusão)***

É a comunicação coletiva em que um único processo envia (send) os mesmos dados para todos os processos com o mesmo communicator.

♦ ***Reduction (Redução)***

É a comunicação coletiva onde cada processo no communicator contém um operador, e todos eles são combinados usando um operador binário que será aplicado sucessivamente.

- ***Gather (Coleta)***

A estrutura dos dados distribuídos é coletada por um único processo.

- ***Scatter (Espalhamento)***

A estrutura dos dados que está armazenado em um único processo é distribuído a todos os processos.

Tabela Comparativa:

Operação	Custo Médio
MPI_SEND / MPI_RECV	ts + tw*N
MPI_BARRIER	ts * log P
MPI_BCAST	log P*(ts + tw*N)
MPI_GATHER / MPI_SCATTER	ts*log P + tw*N
MPI_REDUCE	log P*(ts + (tw+top)*N)

3.4 - PROGRAMANDO EM PARALELO UTILIZANDO MPI

Para programar utilizando programação paralela, considere os seguintes pontos:

Análise - Analise o programa nos seguintes aspectos:

Se existem procedimentos independentes que possam ser executados concorrentemente e se existem relações de controle e dependência entre procedimentos.

Qual e em que nível deve se dar a comunicação entre procedimentos. Quais os tipos de dependências existentes entre os processos para só então definir como realizar a sincronização entre os procedimentos.

Dependência

- ***Dados***
  Quando uma mesma variável é referenciada ou modificada dentro do escopo de processos distintos, podendo possuir um resultado inconsistente (diferenciado) ao final de tais processos.
  Podemos subclassificar esse tipo de dependência em :

Dependência real (dependência de fluxo)	Anti-dependência	Dependência de saída
X=3;	Y=X*4;	X=3;
Y=X*4;	X=3;	X=Y*4;

- ***Loops***
  Quando iterações dependem de iterações anteriores.
  Y[I] = Y[I-1] * 2;

- ***Recursos***
  Quando um mesmo recurso ou dispositivo é acessado simultaneamente por processos distintos, como impressoras ou arquivos.

Como saber se dois programas são paralelizáveis?

- ***Condição de Bernstein***

  Se qualquer das saídas de um processo for utilizada por outro processo, estes não poderão ser executados concorrentemente. Esquematicamente temos:

$$I1 \cap O2 = \emptyset \text{ e } I2 \cap O1 = \emptyset \text{ e } O1 \cap O2 = \emptyset$$

3.5 - Primeiros Passos para Utilização do MPI

Depois da paralelização do algoritmo, seja ela de dados ou funcional, é necessário utilizar uma biblioteca que forneça as rotinas e procedimentos de paralelização.

A biblioteca MPI é uma biblioteca de Message Passing que define um conjunto de rotinas/chamadas para facilitar a comunicação (troca de dados e sincronização) entre processos paralelos. Uma vez definida qual a biblioteca a ser utilizada para a implementação do algoritmo paralelo, é necessário fazer a compilação e linkedição do programa em paralelo. A implementação MPICH para MPI padrão definiu um único comando para as tarefas de compilação e linkedição.

Preparação do Ambiente

Criação do arquivo de maquinas que deve conter o nome e domínio das máquinas pertencentes a rede do CENAPAD-NE. As máquinas disponíveis são:

Oros.cenapadne.br	Caico.cenapadne.br	f1n1.cenapadne.br	f1n2.cenapadne.br	f1n3.cenapadne.br
f1n4.cenapadne.br	pacoti.cenapadne.br	Aracati.cenapadne.br	icapui.cenapadne.br	tiangua.cenapadne.br
Juazeiro.cenapadne.br	Ubajara.cenapadne.br	ipu.cenapadne.br	jijoca.cenapadne.br	acopiara.cenapadne.br
Caucaia.cenapadne.br				

Observação: Para se utilizar o switch (concentrador do SP/2) de alta velocidade, devem-se utilizar os IP's dos nós, conforme especificados abaixo:

f1s1.cenapadne.br	f1s2.cenapadne.br	f1s3.cenapadne.br	f1s4.cenapadne.br

3.6 - Como Utilizar o MPI?

Você utiliza diretamente o MPI nos seguintes pontos:

Bibliotecas

Adicione em seus programas um dos seguintes arquivos include:
mpi.h para programas em C
mpif.h para programas em FORTRAN

Rotinas MPI

Escrever seu programa fazendo chamadas às rotinas MPI.

Compilação e linkedição

Utilizaremos Message Passing Interface, através da versão do MPICH, desenvolvido pelo ARNL - Argonne National Laboratory e pelo MSU – Mississipi State University.

Observação: O ambiente do CENAPAD/NE ainda possui o MPL - Message Passing Library - uma biblioteca de passagem de mensagens compatível com o MPI que vêm incorporada às ferramentas do PE - Parallel Environment – da IBM.

Usar o compilador com a sintaxe adequada, incluindo os parâmetros desejados.

Para a compilação e linkedição utilizar os comandos para:

Linguagem C (C++)

```
mpicc [fonte.c] -o [executável] [parâmetros]
```

Obs.: O comando mpicc aceita todos os argumentos de compilação do compilador C

FORTRAN

```
mpif77 [fonte.f] -o [executável] [parâmetros]
```

Obs.: O comando mpif77 aceita todos os argumentos de compilação do compilador Fortran

Atenção: A compilação, tanto em C, quanto em Fortran, no ambiente do CENAPAD/NE deve ser executado ou na máquina oros.cenapadne.br ou na máquina caico. cenapadne.br.

Execução

Na hora da execução utilizar o comando para linguagens C e FORTRAN:

```
mpirun -[argumentos] [executável]
```

Argumentos:

- h - Mostra todas as opções disponíveis
- arch - Especifica a arquitetura da(s) máquina(s)
- machine - Especifica a(s) máquina(s)
- machinefile - Especifica o arquivo que contém o nome das máquinas
- np - Especifica o número de processadores
- leave_pg - Registra onde os processos estão sendo executados
- nolocal - Não executa na máquina local
- t - Testa sem executar o programa, apenas imprime o que será executado
- dbx - Inicializa o primeiro processo sobre o dbx

Atenção: A execução dos programas em MPI pode ser feita a partir de qualquer máquina do CENAPAD/NE.

Exemplos:

```
mpirun -np 5 teste
```

Executa o programa teste em 5 processadores

```
mpirun -arch risc6000 -np 5 teste 10
```

Executa o programa teste em 5 processadores de arquitetura RISC 6000 (IBM), passando como parâmetro o valor 10

Observação: Em locais de arquitetura mista, é possível mesclar os parâmetros, de forma a executar sob o MPI, um mesmo programa (compilado diferentemente nas várias arquiteturas).

Exemplo:

`mpirun -arch risc6000 -np 3 -arch sun4 -np 2 teste.%a` onde %a substitui a arquitetura.

3.7 - NOMENCLATURA DAS ROTINAS MPI

Nomenclatura das Rotinas no MPI:

Linguagem C:

rc = MPI_Xxxxx(parâmetros, ...);
Onde rc é uma variável inteira que recebe um código de erro.

Exemplo:

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

Linguagem Fortran:

MPI_XXXXX(parâmetros, ..., ierror);
Onde ierror é uma variável inteira que recebe um código de erro.

Exemplo:

call MPI_INIT (mpierr)

Retorno dos Códigos de Erro:

Observação Inicial:
Todas as rotinas MPI com exceção de MPI_Wtime e MPI_Wtrack, retornam um código de erro. Em programas em linguagem C o código de erro corresponde ao código de retorno da chamada da função, enquanto na linguagem Fortran corresponde ao último argumento da rotina.

Exemplo:

mpierr = MPI_Init(&argc, &argv[ ])
onde mpierr é a variável inteira que contém o status da rotina.

Tabela de Erros:

Abaixo, listamos os principais códigos de erros, comumente encontrados nos programas em MPI.

Código	Significado
MPI_ERR_ARG	Argumento inválido
MPI_ERR_BUFFER	Ponteiro para buffer inválido
MPI_ERR_COMM	Comunicator inválido
MPI_ERR_COUNT	Argumento numérico inválido
MPI_ERR_DIMS	Argumento de dimensão inválida
MPI_ERR_GROUP	Grupo inválida
MPI_ERR_IN_STATUS	Código de erro na variável status
MPI_ERR_INTERN	Erro interno no MPI
MPI_ERR_LASTCODE	Código do último erro
MPI_ERR_OP	Operações inválidas
MPI_ERR_OTHER	Outros erros
MPI_ERR_PENDING	Requisição pendente
MPI_ERR_ROOT	Root (origem) inválido
MPI_ERR_TAG	Declaração inválida
MPI_ERR_TYPE	Tipo de dado inválido
MPI_ERR_TOPOLOGY	Topologia inválida
MPI_ERR_TRUNCATE	Mensagem truncada no recebimento
MPI_ERR_UNKNOWN	Erro de natureza desconhecida
MPI_ERR_SUCCESS	Sem erro. Operação efetuada com sucesso.

Para o MPICH, temos:

```
#define MPI_SUCCESS          0     /* Successful return code */
#define MPI_ERR_BUFFER       1     /* Invalid buffer pointer */
#define MPI_ERR_COUNT        2     /* Invalid count argument */
#define MPI_ERR_TYPE         3     /* Invalid datatype argument */
#define MPI_ERR_TAG          4     /* Invalid tag argument */
#define MPI_ERR_COMM         5     /* Invalid communicator */
#define MPI_ERR_RANK         6     /* Invalid rank */
#define MPI_ERR_ROOT         7     /* Invalid root */
#define MPI_ERR_GROUP        8     /* Null group passed to function */
#define MPI_ERR_OP           9     /* Invalid operation */
#define MPI_ERR_TOPOLOGY    10     /* Invalid topology */
#define MPI_ERR_DIMS        11     /* Illegal dimension argument */
#define MPI_ERR_ARG         12     /* Invalid argument */
#define MPI_ERR_UNKNOWN     13     /* Unknown error */
#define MPI_ERR_TRUNCATE    14     /* message truncated on receive */
#define MPI_ERR_OTHER       15     /* Other error; use Error_string */
#define MPI_ERR_INTERN      16     /* internal error code    */
#define MPI_ERR_IN_STATUS   17     /* Look in status for error value */
#define MPI_ERR_PENDING     18     /* Pending request */
#define MPI_ERR_REQUEST     19     /* illegal mpi_request handle */
#define MPI_ERR_LASTCODE    (256*16+18)      /* Last error code*/
```

Capítulo 4

MPI BÁSICO

4.1 - MPI_INIT

Iniciando um processo MPI:

Definição:

Inicializa um processo MPI. Portanto, deve ser a primeira rotina a ser chamada por cada processo, pois estabelece o ambiente necessário para executar o MPI. Ela também sincroniza todos os processos na inicialização de uma aplicação MPI.

Sintaxe:

C

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

Fortran

call MPI_INIT (mpierr)

Parâmetros:

argc - Apontador para a qtde. de parametros da linha de comando;
argv - Apontador para um vetor de strings

Erro:

MPI_ERR_OTHER - Ocorreu mais de uma chamada dessa rotina no mesmo código.

Exemplo:

```
#include <stdio.h>
#include "mpi.h"
 main(int argc, char *argv[])
 {
  int ret;
  ret = MPI_Init(&argc, &argv);
  if (ret < 0)
   {
    printf ("Nao foi possivel inicializar o processo MPI!\n");
    return;
   }
  else
   ...
 }
```

4.2 - MPI_COMM_RANK

Identificando um processo no MPI:

Definição:

Identifica um processo MPI dentro de um determinado grupo. Retorna sempre um valor inteiro entre 0 e n-1, onde n é o número de processos.

Sintaxe:

C

int MPI_Comm_rank (MPI_Comm comm, int *rank)

Fortran

call MPI_COMM_RANK (comm, rank, mpierr)

Parâmetros:

comm - Comunicador do MPI;
rank - Variável inteira com o numero de identificação do processo.

Erro:

MPI_ERR_COMM - Communicator inválido.

Exemplo:

```
#include <stdio.h>
#include "mpi.h"
 main(int argc, char *argv[])
 {
  int mpierr, rank;
  mpierr = MPI_Init(&argc, &argv);
  if (mpierr < 0)
   {
    printf ("Nao foi possivel inicializar o processo MPI!\n");
    return;
   }
  else
   {
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    printf ("Minha identificação no MPI e':%d\n",rank);
    ...
   }
 }
```

4.3 - MPI_COMM_SIZE

Contando processos no MPI:

Definição:

Retorna o número de processos dentro de um grupo.

Sintaxe:

C

int MPI_Comm_size (MPI_Comm comm, int *size)

Fortran

call MPI_COMM_SIZE (comm, size, mpierr)

Parâmetros:

comm

- Comunicador do MPI;

size

- Variável interna que retorna o numero de processos iniciados pelo MPI.

Erro:

MPI_ERR_COMM	- Communicator inválido.
MPI_ERR_ARG	- Argumento inválido.
MPI_ERR_RANK	- Identificação inválida do processo.

Exemplo:

```
#include <stdio.h>
#include "mpi.h"
 main(int argc, char *argv[])
 {
  int mpierr, rank, size;
  mpierr = MPI_Init(&argc, &argv);
  if (mpierr < 0)
   {
    printf ("Nao foi possivel inicializar o processo MPI!\n");
    return;
   }
  else
   {
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    printf ("Minha identificação no MPI e':%d\n",rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    printf("O numero de processos e': %d\n",size);
    ...
   }
 }
```

4.4 - MPI_Send

Enviando mensagens no MPI:

Definição:

Rotina básica para envio de mensagens no MPI, utiliza o modo de comunicação "blocking send" (envio bloqueante), o que traz maior segurança na transmissão da mensagem. Após o retorno, libera o "system buffer" e permite o acesso ao "application buffer".

Sintaxe:

C

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

Fortran

call MPI_SEND (sndbuf, count, dtype, dest, tag, comm, mpierr)

Parâmetros:

Sndbuf - Identificação do buffer (endereço inicial do "application buffer" - de onde os dados serão enviados);
count - Número de elementos a serem enviados;
dtype - Tipo de dado
dest - Identificação do processo destino;
tag - Rótulo (label) da mensagem;
comm - MPI Communicator

Erros:

MPI_ERR_COMM	- Communicator inválido.
MPI_ERR_COUNT	- Argumento numérico inválido.
MPI_ERR_RANK	- Identificação inválida do processo.
MPI_ERR_TYPE	- Tipo de dado inválido.
MPI_ERR_TAG	- Declaração inválida do label.
MPI_ERR_RANK	- Identificação inválida do processo.

Exemplo:

```
#include <stdio.h>
#include "mpi.h"
main(int argc, char *argv[])
{
 int mpierr, rank, size, i;
 char message[20];
 mpierr = MPI_Init(&argc, &argv);
 if (mpierr < 0)
  {
   printf ("Nao foi possivel inicializar o processo MPI!\n");
   return;
  }
 else
  {
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   printf ("Minha identificação no MPI e':%d\n",rank);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   printf("O numero de processos e': %d\n",size);
   if (rank == 0)
    {
     strcpy(message,"Ola', Mundo!\n");
     for (i=1; i<size; i++)
      MPI_Send(message, 13, MPI_CHAR, i, tag, MPI_COMM_WORLD);
    }
   else
    ...
  }
}
```

4.5 - MPI_RECV

Recebendo mensagens no MPI:

Definição:

É uma rotina básica para recepção de mensagens no MPI, que utiliza o modo de comunicação "blocking receive" (recepção bloqueante), de forma que o processo espera até que a mensagem tenha sido recebida. Após o retorno, libera o "system buffer", que pode ser então, novamente utilizado.

Sintaxe:

C

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

Fortran

call MPI_RECV (recvbuf, count, dtype, source, tag, comm, status, mpierr)

Parâmetros:

sndbuf - Identificação do buffer (endereço inicial do "application buffer" - de onde os dados estão sendo enviados);
count - Número de elementos a serem recebidos;
dtype - Tipo de dado
source - Identificação do processo emissor;
tag - Rótulo (label) da mensagem;
comm - MPI Communicator
status - Vetor de informações envolvendo os parâmetros source e tag.

Erros:

MPI_ERR_COMM	- Communicator inválido.
MPI_ERR_COUNT	- Argumento numérico inválido.
MPI_ERR_RANK	- Identificação inválida do processo.
MPI_ERR_TYPE	- Tipo de dado inválido.
MPI_ERR_TAG	- Declaração inválida do label.
MPI_ERR_RANK	- Identificação inválida do processo.

Exemplo:

```
#include <stdio.h>
 #include "mpi.h"
  main(int argc, char *argv[])
  {
   int mpierr, rank, size, tag, i;
   MPI_Status status;
   char message[20];
   mpierr = MPI_Init(&argc, &argv);
   if (mpierr < 0)
    {
     printf ("Nao foi possivel inicializar o processo MPI!\n");
     return;
    }
   else
    {
     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
     printf ("Minha identificação no MPI e':%d\n",rank);
     MPI_Comm_size(MPI_COMM_WORLD, &size);
     printf("O numero de processos e': %d\n",size);
     if (rank == 0)
      {
       strcpy(message,"Ola', Mundo!\n");
       for (i=1; i<size; i++)
        MPI_Send(message, 13, MPI_CHAR, i, tag, MPI_COMM_WORLD);
      }
     else
      MPI_Recv(message, 20, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);
      printf("Mensagem do no' %d : %s\n", rank, message);
      ...
    }
  }
```

4.6 - MPI_FINALIZE

Encerrando um processo no MPI:

Definição:

Finaliza um processo MPI. Portanto deve ser a última rotina a ser chamada por cada processo. Sincroniza todos os processos na finalização de uma aplicação MPI.

Sintaxe:

C

int MPI_Finalize (void)

Fortran

call MPI_FINALIZE (mpierr)

Parâmetros:

(Nenhum)

Erros:

(Nenhum)

Exemplo:

```
#include <stdio.h>
 #include "mpi.h"
  main(int argc, char *argv[])
  {
   int mpierr, rank, size, tag, i;
   MPI_Status status;
   char message[20];
   mpierr = MPI_Init(&argc, &argv);
   if (mpierr < 0)
    {
     printf ("Nao foi possivel inicializar o processo MPI!\n");
     return;
    }
   else
    {
     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
     printf ("Minha identificação no MPI e':%d\n",rank);
     MPI_Comm_size(MPI_COMM_WORLD, &size);
     printf("O numero de processos e': %d\n",size);
     if (rank == 0)
      {
       strcpy(message,"Ola', Mundo!\n");
       for (i=1; i<size; i++)
        MPI_Send(message, 13, MPI_CHAR, i, tag, MPI_COMM_WORLD);
      }
     else
      MPI_Recv(message, 20, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);
      printf("Mensagem do no' %d : %s\n", rank, message);
      MPI_Finalize();
    }
  }
```

4.7 - Programas Exemplos

hello.c

```
#include <stdio.h>
#include <string.h>
#include "mpi.h"
main(int argc, char *argv[])
{
   int ret, rank, size, i, tag;
   MPI_Status status;
   char message[12];
   ret = MPI_Init(&argc, &argv);
   ret = MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   ret = MPI_Comm_size(MPI_COMM_WORLD, &size);
   tag=100;
   if (rank == 0)
    {
     strcpy(message,"Ola, Mundo!");
     for (i=1; i<size; i++)
       ret = MPI_Send(message, 12, MPI_CHAR, i, tag, MPI_COMM_WORLD);
    }
   else
    ret = MPI_Recv(message, 12, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);
    printf("Mensagem do no %d : %s\n", rank, message);
    ret = MPI_Finalize();
}
```

Características:
- Inclusão das bibliotecas padrões (stdio.h, string.h);
- Verificar os parâmetros de linha de comando (argc, argv);
- Mensagem com o caractere final NULL ('/0');
- Códigos de retorno podem ser ignorados.

hello.f

```
program hello
include "mpif.h"
integer ret, rank, size, i, tag
integer status(MPI_STATUS_SIZE)
character(11) message
call MPI_INIT(ret)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ret)
call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ret)
tag=100
if (rank .eq. 0) then
   message = 'Ola, Mundo!'
   do i=1, size-1
    MPI_SEND(message, 11, MPI_CHARACTER, i, tag, MPI_COMM_WORLD, ret)
   enddo
else
   MPI_RECV(message, 11, MPI_CHARACTER, 0, tag, MPI_COMM_WORLD, status, ret)
endif
print*,'Mensagem do no',rank,':',message
call MPI_FINALIZE(ret)
end
```

Características:
- Include básico: mpif.h (diferente do header .c);
- Parâmetros diferentes para as rotinas .c e .f (ex.: MPI_INIT);
- Distinções das linguagens (ex.: a string message);
- Códigos de retorno obrigatórios (parâmetros)

Capítulo 5

MPI AVANÇADO

Conceitos Básicos:

- ***Comunicação Ponto a Ponto***

Dentre as rotinas básicas do MPI estão as rotinas de comunicação ponto a ponto que executam a transferência de dados entre dois processos.

- ***Comunicação Ponto a Ponto Blocking Send***

Liberam o system buffer após o envio da mensagem, permitindo então a continuação do processamento.

- ***Comunicação Ponto a Ponto Non-blocking Send***

Nessas rotinas, a finalizaão da execução da rotina não depende de determinados eventos, ou seja, a mensagem continua sendo enviada normalmente sem haver espera.

- ***Comunicação Ponto a Ponto Non-blocking Receive***

Nessa rotina, o processo não fica bloqueado esperando receber dados: se eles já estiverem disponíveis são imediatamente carregados; caso contrário, continua sua execução ainda que nada tenha sido recebido.

- ***Comunicação Ponto a Ponto Buffer***

São rotinas responsáveis pela criação e desalocação de um espaço na memória para a recepção ou envio da mensagem.

- ***Comunicação Ponto a Ponto de Empacotamento***

Relacionam-se com o agrupamento de dados de diferentes tipos que são armazenados em um espaço contíguo para permitir um único envio/recebimento dos dados.

- ***Comunicação Coletiva***

É a comunicação padrão que invoca todos os processos em um grupo (group) - coleção de processos que podem se comunicar entre si. Normalmente a comunicação coletiva envolve mais de dois processos. As rotinas de comunicação coletiva são voltadas para a comunicação / coordenação de grupos de processos.

- ***Comunicação Coletiva de Sincronização***

Os processos executados em paralelo, num determinado instante, aguardam a ocorrência mútua de um determinado evento para continuarem sua execução.

- ***Comunicação Coletiva de Transporte de Dados***

Diferentemente da comunicação ponto-a-ponto, na comunicação coletiva é possível enviar/receber dados simultaneamente de/para vários processos.

- ***Comunicação Coletiva de Computação Global***

Além de receber dados, são realizadas operações sobre os mesmos, como maximização, soma, e outras.

- ***Rotinas Auxiliares***

São rotinas de finalidade geral, fornecendo muitos dos dados necessários a uma boa aplicação. Grande parte das 125 rotinas do MPI dedicam-se a esta funcionalidade.

♦ *Rotinas de Temporização*

São rotinas relacionadas com a medição do tempo de execução dos processos no MPI.

♦ *Rotina de Identificação*

Fornece dados úteis para identificação do ambiente onde os processos estão sendo executandos.

A seguir, destacaremos as principais rotinas avançadas do MPI, que dizem respeito tanto à comunicação *ponto-a-ponto*, quanto à comunicação coletiva e às rotinas de auxílio à programação.

5.1 - MPI_SSEND

Envio blocante e síncrono:

Definição:

Utiliza o modo de comunicação **Blocking Synchronous Send**, onde o processo que envia a mensagem avisa ao processo receptor que está pronto e esperando um sinal de OK do receptor e só então enviará os dados.

Sintaxe:

C
int MPI_Ssend (void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
Fortran
call MPI_SSEND (buf, count, datatype, dest, tag, comm, mpierr)

Parâmetros:

buf	- Identificação do buffer (endereço inicial do "application buffer" - de onde os dados estão sendo enviados);
count	- Número de elementos a serem enviados;
datatype	- Tipo de dado
dest	- Identificação do processo destino;
tag	- Rótulo (label) da mensagem;
comm	- MPI Communicator

Erros:

MPI_ERR_COUNT	- Argumento numérico inválido.
MPI_ERR_RANK	- Identificação inválida do processo.
MPI_ERR_TYPE	- Tipo de dado inválido.
MPI_ERR_TAG	- Declaração inválida do label.

Observações:

É necessário que se tenha cuidado com os **overheads** (custos) na comunicação do sistema, principalmente com:

- **System overhead**: Devido à cópia da mensagem do buffer do processo emissor para a rede e desta para o buffer do processo receptor;
- **Syncronization overhead**: Devido ao tempo de espera de um dos processos pelo sinal de OK do outro.

Exemplo:

```
#include <stdio.h>
#include "mpi.h"
 main(int argc, char *argv[])
 {
  int mpierr, rank, size, i, tag;
  MPI_Status status;
  char message[40];
  mpierr = MPI_Init(&argc, &argv);
  if (mpierr >= 0)
   {
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    if (rank == 0)
     {
      strcpy(message,"Enviando mensagens sincronas no MPI!\n");
      for (i=1; i<size; i++)
       MPI_Ssend(message, 40, MPI_CHAR, i, tag, MPI_COMM_WORLD);
     }
    else
     MPI_Recv(message, 40, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);
    printf("Mensagem do processo' %d : %s\n", rank, message);
    MPI_Finalize();
   }
 }
```

5.2 - MPI_RSEND

Envio blocante e imediato:

Definição:

Utiliza o modo de comunicação Blocking Ready Send. Quando uma rotina de MPI_Rsend é executada a mensagem é enviada imediatamente para a rede. Neste caso, é obrigatório que o processo receptor tenha emitido um sinal de OK.

Sintaxe:

C

int MPI_Rsend (void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)

Fortran

call MPI_RSEND (buf, count, datatype, dest, tag, comm, mpierr)

Parâmetros:

buf	- Identificação do buffer (endereço inicial do "application buffer" - de onde os dados estão sendo enviados);
count	- Número de elementos a serem enviados;
datatype	- Tipo de dado
dest	- Identificação do processo destino;
tag	- Rótulo (label) da mensagem;
comm	- MPI Communicator

Erros:

MPI_ERR_COUNT	- Argumento numérico inválido.
MPI_ERR_RANK	- Identificação inválida do processo.
MPI_ERR_TYPE	- Tipo de dado inválido.
MPI_ERR_TAG	- Declaração inválida do label.

Observações:

Quanto ás características de sincronização, percebemos que:

- **System overhead mínimo**: Como o processo receptor estaria pronto para o recebimento dos dados, o único overhead neste caso seria do processo transmissor dos dados para a rede;
- **Syncronization overhead**: Ocorre quando o Receive é dado antes do Send, fazendo com que o processo receptor fique esperando os dados.

Exemplo:

```
#include <stdio.h>
#include "mpi.h"
 main(int argc, char *argv[])
 {
  int mpierr, rank, size, i, tag;
  MPI_Status status;
  char message[40];
  mpierr = MPI_Init(&argc, &argv);
  if (mpierr >= 0)
   {
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    if (rank == 0)
     {
      strcpy(message,"Enviando mensagens imediatas no MPI!\n");
      for (i=1; i<size; i++)
       MPI_Rsend(message, 40, MPI_CHAR, i, tag, MPI_COMM_WORLD);
     }
    else
     MPI_Recv(message, 40, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);
    printf("Mensagem do processo' %d : %s\n", rank, message);
    MPI_Finalize();
   }
 }
```

5.3 - MPI_BSEND

Envio blocante e bufferizado:

Definição:

Utiliza o modo de comunição **Blocking Buffering Send**. Quando uma rotina de MPI_Bsend é executada a mensagem é copiada do "application buffer" para um buffer definido pelo usuário. A rotina retorna a execução normal e só envia a mensagem quando receber um sinal de OK do processo receptor.

Sintaxe:

C

int MPI_Bsend (void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)

Fortran

call MPI_BSEND (buf, count, datatype, dest, tag, comm, mpierr)

Parâmetros:

buf - Identificação do buffer (endereço inicial do "application buffer" - de onde os dados estão sendo enviados);
count - Número de elementos a serem enviados;
datatype - Tipo de dado
dest - Identificação do processo destino;
tag - Rótulo (label) da mensagem;
comm - MPI Communicator

Erros:

MPI_ERR_COUNT	- Argumento numérico inválido.
MPI_ERR_RANK	- Identificação inválida do processo.
MPI_ERR_TYPE	- Tipo de dado inválido.
MPI_ERR_TAG	- Declaração inválida do label.

Observações:

Quanto às características de sincronização, temos que o MPI_Bsend tem:

- **System overhead considerável**: Devido à cópia da mensagem do Application buffer para o buffer do usuário;
- **Syncronization overhead considerável**: Isto porque o processo receptor pode se antecipar demais ao send, ficando, portanto, à espera de dados.

Com a rotina MPI_Bsend, o MPI encarrega-se de armazenar os dados a serem transmitidos (que podem estar em outras variáveis), no novo buffer previamente alocado com MPI_Buffer_attach.

Exemplo:

```
#include <stdio.h>
#include "mpi.h"
#define MAX_BUF 100
#define MSGLEN 40
 main(int argc, char *argv[])
 {
  int mpierr, rank, size, i, bufsize=MAX_BUF, tag=10;
  MPI_Status status;
  char buffer[MAX_BUF], message[MSGLEN];
  mpierr = MPI_Init(&argc, &argv);
  if (mpierr >= 0)
   {
    MPI_Buffer_attach(buffer, bufsize);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    if (rank == 0)
     {
      strcpy(message,"Envindo mensagens bufferizadas no MPI!\n");
      for (i=1; i<size;i++)
       MPI_Bsend(message, MSGLEN, MPI_CHAR, i, tag, MPI_COMM_WORLD);
     }
    else
     MPI_Recv(message, MSGLEN, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);
    printf("Mensagem do processo' %d : %s\n", rank, message);
    MPI_Buffer_detach(buffer, &bufsize);
    MPI_Finalize();
   }
 }
```

5.4 - MPI_ISEND

Envio não blocante padrão:

Definição:

Utiliza o modo de comunicação **Non-Blocking Standard Send**. É uma rotina padrão para envio de mensagens non-blocking no MPI.

Sintaxe:

C

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

Fortran

call MPI_ISEND (buf, count, datatype, dest, tag, comm, request, ierror)

Parâmetros:

buf - Identificação do buffer (endereço inicial do "application buffer" - de onde os dados estão sendo enviados);
count - Número de elementos a serem enviados;
datatype - Tipo de dado
dest - Identificação do processo destino;
tag - Rótulo (label) da mensagem;
comm - MPI Communicator
request - Identificação, perante o MPI do evento em questão

Retorno:

request - Identificação do evento.
mpierr - Código de erro.

Erros:

MPI_ERR_COMM	- Comunicador inválido.
MPI_ERR_COUNT	- Argumento numérico inválido.
MPI_ERR_RANK	- Identificação inválida do processo.
MPI_ERR_TYPE	- Tipo de dado inválido.
MPI_ERR_TAG	- Declaração inválida do label.
MPI_ERR_INTERN	- Erro interno no MPI (memória insuficiente).

Exemplo:

```
#include <stdio.h>
#include "mpi.h"
 main(int argc, char *argv[])
 {
  int mpierr, rank, size, i, tag;
  char message[40];
  MPI_Status status;
  MPI_Request request;
  mpierr = MPI_Init(&argc, &argv);
  if (mpierr >= 0)
   {
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    if (rank == 0)
      {
      strcpy(message,"Enviando mensagens em modo nao blocante!\n");
      for (i=1; i<size; i++)
        MPI_Isend(message, 40, MPI_CHAR, i, tag, MPI_COMM_WORLD, &request);
     }
    else
      MPI_Recv(message, 40, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);
    printf("Mensagem do processo' %d : %s\n", rank, message);
    MPI_Finalize();
   }
 }
```

5.5 - MPI_ISSEND

Envio não blocante síncrono:

Definição:

Utiliza o modo de comunicação **Non-Blocking** em modo síncrono: avisa quando uma dada mensagem foi enviada com sucesso.

Sintaxe:

C

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

Fortran

call MPI_ISSEND (buf, count, datatype, dest, tag, comm, request, ierror)

Parâmetros:

buf	- Identificação do buffer (endereço inicial do "application buffer" - de onde os dados estão sendo enviados);
count	- Número de elementos a serem enviados;
datatype	- Tipo de dado
dest	- Identificação do processo destino;
tag	- Rótulo (label) da mensagem;
comm	- MPI Communicator
request	- Identificação, perante o MPI do evento em questão

Retorno:

request	- Identificação do evento.
mpierr	- Código de erro.

Erros:

MPI_ERR_COMM	- Comunicador inválido.
MPI_ERR_COUNT	- Argumento numérico inválido.
MPI_ERR_RANK	- Identificação inválida do processo.
MPI_ERR_TYPE	- Tipo de dado inválido.
MPI_ERR_TAG	- Declaração inválida do label.
MPI_ERR_INTERN	- Erro interno no MPI (memória insuficiente).

Exemplo:

```
#include <stdio.h>
#include "mpi.h"
 main(int argc, char *argv[])
 {
  int mpierr, rank, size, i, tag;
  char message[40];
  MPI_Status status;
  MPI_Request request;
  mpierr = MPI_Init(&argc, &argv);
  if (mpierr >= 0)
   {
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    if (rank == 0)
      {
      strcpy(message,"Enviando mensagens nao-blocante sincronas!\n");
      for (i=1; i<size; i++)
        MPI_Issend(message, 40, MPI_CHAR, i, tag, MPI_COMM_WORLD, &request);
      }
    else
      MPI_Recv(message, 40, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);
    printf("Mensagem do processo' %d : %s\n", rank, message);
    MPI_Finalize();
   }
 }
```

5.6 - MPI_IRSEND

Envio não blocante imediato:

Definição:

Utiliza o modo de comunicação **Non-Blocking** em modo imediato: pressumpõe que o receive correspondente já foi dado.

Sintaxe:

C

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

Fortran

call MPI_IRSEND (buf, count, datatype, dest, tag, comm, request, ierror)

Parâmetros:

buf - Identificação do buffer (endereço inicial do "application buffer" - de onde os dados estão sendo enviados);
count - Número de elementos a serem enviados;
datatype - Tipo de dado
dest - Identificação do processo destino;
tag - Rótulo (label) da mensagem;
comm - MPI Communicator
request - Identificação, perante o MPI do evento em questão

Retorno:

request - Identificação do evento.
mpierr - Código de erro.

Erros:

MPI_ERR_COMM	- Comunicador inválido.
MPI_ERR_COUNT	- Argumento numérico inválido.
MPI_ERR_RANK	- Identificação inválida do processo.
MPI_ERR_TYPE	- Tipo de dado inválido.
MPI_ERR_TAG	- Declaração inválida do label.
MPI_ERR_INTERN	- Erro interno no MPI (memória insuficiente).

Exemplo:

```
#include <stdio.h>
#include "mpi.h"
 main(int argc, char *argv[])
 {
  int mpierr, rank, size, i, tag;
  char message[45];
  MPI_Status status;
  MPI_Request request;
  mpierr = MPI_Init(&argc, &argv);
  if (mpierr >= 0)
    {
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    if (rank == 0)
      {
      strcpy(message,"Enviando mensagens nao-blocantes imediatas!\n");
      for (i=1; i<size; i++)
        MPI_Irsend(message, 45, MPI_CHAR, i,tag, MPI_COMM_WORLD, &request);
      }
    else
      MPI_Recv(message, 45, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);
    printf("Mensagem do processo' %d : %s\n", rank, message);
    MPI_Finalize();
   }
 }
```

Observações:

Note algumas observações que o usuário deve ter em mente ao usar a rotina MPI_Irsend:

- Deve-se estar atento ao fato de que o receive já deve ter sido dado, da parte do processo que receberá o s dados;
- MPI_Irsend é a rotina mais rápida que existe, pois diminui drasticamente o syncronization e o system overhead.

5.7 - MPI_IBSEND

Envio não blocante bufferizado:

Definição:

Utiliza o modo de comunicação **Non-Blocking** em modo imediato bufferizado, isto é, permite que o espaço de buffer possa ser estendido, para além dos limites padrões (4KB), para acomodar uma maior quantidade de dados.

Sintaxe:

C

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

Fortran

```
call MPI_IBSEND (buf, count, datatype, dest, tag, comm, request, ierror)
```

Parâmetros:

buf - Identificação do buffer (endereço inicial do "application buffer" - de onde os dados estão sendo enviados);
count - Número de elementos a serem enviados;
datatype - Tipo de dado
dest - Identificação do processo destino;
tag - Rótulo (label) da mensagem;
comm - MPI Communicator
request - Identificação, perante o MPI do evento em questão

Retorno:

request - Identificação do evento.
mpierr - Código de erro.

Erros:

MPI_ERR_COMM	- Comunicador inválido.
MPI_ERR_COUNT	- Argumento numérico inválido.
MPI_ERR_RANK	- Identificação inválida do processo.
MPI_ERR_TYPE	- Tipo de dado inválido.
MPI_ERR_TAG	- Declaração inválida do label.
MPI_ERR_BUFFER	- Buffer inválido ou vazio.

Exemplo:

```
#include <stdio.h>
#include "mpi.h"
#define MAX_BUF 100
#define MSGLEN 50
 main(int argc, char *argv[])
 {
  int mpierr, rank, size, i,bufsize=MAX_BUF, tag=10;
  char buffer[MAX_BUF], message[MSGLEN];
  MPI_Status status;
  MPI_Request request;
  mpierr = MPI_Init(&argc, &argv);
  if (mpierr >= 0)
    {
    MPI_Buffer_attach(buffer, bufsize);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    if (rank == 0)
      {
      strcpy(message,"Enviando msg bufferizadas nao blocantes no MPI!\n");
      for (i=1; i<size; i++)
        MPI_Ibsend(message, MSGLEN, MPI_CHAR, i, tag, MPI_COMM_WORLD, &request);
      }
    else
      MPI_Recv(message, MSGLEN, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);
    printf("Mensagem do processo' %d : %s\n", rank, message);
    MPI_Buffer_detach(buffer, &bufsize);
    MPI_Finalize();
   }
 }
```

Observação: Note que, ao usar a rotina MPI_Ibsend: O usuário deve continuar atento à criação e remoção do buffer (usando rotinas como MPI_Buffer_Attach e MPI_Buffer_detach

5.8 - MPI_IRECV

Recepção não blocante:

Definição:

Rotina básica para recepção de mensagens de forma não-blocante no MPI, de modo que a rotina não fica bloqueada, à espera da mensagem.

Sintaxe:

C

int MPI_Irecv (void *recvbuf, int count, MPI_Datatype dtype, int source, int tag, MPI_Comm comm, MPI_Request *request)

Fortran

call MPI_IRECV (recvbuf, count, dtype, source, tag, comm, request, mpierr)

Parâmetros:

recvbuf	- Identificação do buffer (endereço inicial do "application buffer" - de onde os dados estão sendo recebidos);
count	- Número de elementos a serem recebidos;
dtype	- Tipo de dado
source	- Identificação do processo emissor;
tag	- Rótulo (label) da mensagem;
comm	- MPI Communicator
request	- Identificação, perante o MPI do evento em questão

Retorno:

request	- Identificação do evento.
mpierr	- Código de erro.

Exemplo:

```
#include <stdio.h>
#include "mpi.h"
 main(int argc, char *argv[])
 {
  int mpierr, rank, size;
  char message[20];
  MPI_Request request;
  MPI_Status status;
  mpierr = MPI_Init(&argc, &argv);
  if (mpierr >= 0)
    {
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    if (rank == 0)
      {
      strcpy(message,"Mensagem recebida em modo nao-blocante!\n");
      for (i=1; i<size; i++)
        MPI_Send(message, 50, MPI_CHAR, i, tag, MPI_COMM_WORLD);
      }
    else
      {
       MPI_Irecv(message, 20, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &request);
       MPI_Wait(&request, &status);
      }
    printf("Mensagem do processo' %d : %s\n", rank, message);
    MPI_Finalize();
   }
 }
```

5.9 - MPI_BUFFER_ATTACH

Criação de um buffer de usuário:

Definição:

Inicializa para o MPI uma região de memória, de tamanho especificado, que poderá ser usado pelo usuário como um buffer para o envio / recepção de mensagens maiores do que o tamanho definido pelo MPI.

Sintaxe:

C

int MPI_Buffer_attach (void *buf, int size)

Fortran

call MPI_BUFFER_ATTACH (buf, size, mpierr)

Parâmetros:

buf - Definição do buffer de usuário;
size - Tamanho do buffer

Erros:

MPI_ERR_INTERN	- Erro interno no MPI (memória insuficiente).

Exemplo:

```
#include <stdio.h>
#include "mpi.h"
#define MAX_BUF 100
#define MSGLEN 40
main(int argc, char *argv[])
{
 int mpierr, rank, size, i, bufsize=MAX_BUF, tag=10;
 MPI_Status status;
 char buffer[MAX_BUF], message[MSGLEN];
 mpierr = MPI_Init(&argc, &argv);
 if (mpierr >= 0)
   {
    MPI_Buffer_attach(buffer, bufsize);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    if (rank == 0)
      {
       strcpy(message,"Alocando buffer!\n");
       for (i=1; i<size;i++)
         MPI_Bsend(message, MSGLEN, MPI_CHAR, i, tag, MPI_COMM_WORLD);
      }
    else
     MPI_Recv(message, MSGLEN, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);
    printf("Mensagem do processo' %d : %s\n", rank, message);
    ...
    MPI_Finalize();
   }
}
```

Observação:

Abaixo, listamos os tamanhos padrões para o espaço de buffer (dependendo da quantidade de processos criados):

Número de Processos	Tamanho do Buffer (bytes)
1 a 16	4096
17 a 32	2048
33 a 64	1024
65 a 128	512

5.10 - MPI_BUFFER_DETACH

Remoção de um buffer de usuário:

Definição:

Elimina para o MPI uma região de memória, usada para a troca de mensagens, previamente criada pela rotina Buffer_attach.

Sintaxe:

C

int MPI_Buffer_detach (void *buf, int *size)

Fortran

call MPI_BUFFER_DETACH (buf, size, mpierr)

Parâmetros:

buf - Definição do buffer de usuário;

size - Tamanho do buffer

Exemplo:

```
#include <stdio.h>
#include "mpi.h"
#define MAX_BUF 100
#define MSGLEN 40
 main(int argc, char *argv[])
 {
  int mpierr, rank, size, i, bufsize=MAX_BUF, tag=10;
  MPI_Status status;
  char buffer[MAX_BUF], message[MSGLEN];
  mpierr = MPI_Init(&argc, &argv);
  if (mpierr >= 0)
   {
    MPI_Buffer_attach(buffer, bufsize);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    if (rank == 0)
     {
      strcpy(message,"Desalocando o buffer!\n");
      for (i=1; i<size;i++)
       MPI_Bsend(message, MSGLEN, MPI_CHAR, i, tag, MPI_COMM_WORLD);
     }
    else
     MPI_Recv(message, MSGLEN, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);
    printf("Mensagem do processo' %d : %s\n", rank, message);
    MPI_Buffer_detach(buffer, &bufsize);
    MPI_Finalize();
   }
}
```

5.11 - MPI_PACK

Empacotamento de dados para Transmissão:

Definição:

Empacota um conjunto de dados contíguos na memória, esses dados podem pertencer a vários tipos de dados.

Sintaxe:

C

```
int MPI_Pack (void* inbuf, int insize, MPI_Datatype datatype, void* outbuf, int outsize, int* position, MPI_Comm comm)
```

Fortran

```
call MPI_PACK (inbuf, insize, datatype, outbuf, outsize, position, comm, mpierr)
```

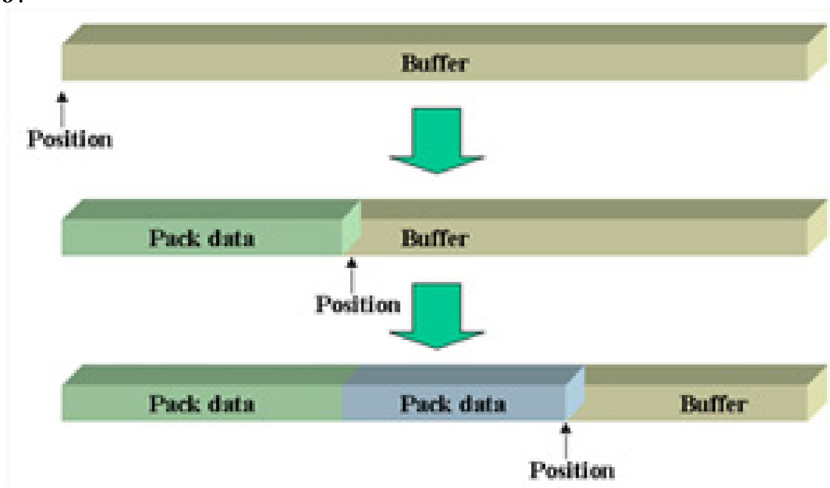
Parâmetros:

inbuf	- Buffer de entrada;
request_vector	- Vetor dos processos a serem verificados;
insize	- Tamanho do buffer de entrada (bytes);
datatype	- Tipo de dado de cada elemento do buffer;
outbuf	- Buffer de saída;
outsize	- Tamanho do buffer de saída;
position	- Posição incial do buffer de saída;
comm	- Identificação do communicator;
mpierr	- Código de erro com o status da rotina.

Erros:

MPI_ERR_ARG	- Argumento inválido;
MPI_ERR_COMM	- Comunicator inválido;
MPI_ERR_COUNT	- Argumento numérico inválido;
MPI_ERR_TYPE	- Tipo inválido.

Funcionamento:



Exemplo:

```
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
typedef struct pessoa
 {
  char nome[30];
  int idade;
  float renda;
 }
 pessoa;
main(int argc, char *argv[])
 {
  int mpierr, rank, size, tag, i, position;
  MPI_Status status;
  char message[100];
  pessoa p1;
  mpierr = MPI_Init(&argc, &argv);
  if (mpierr >= 0)
   {
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    if (rank == 0)
     {
      strcpy(p1.nome,"Joao da Silva");
      p1.idade=35;
      p1.renda=1235.50;
      MPI_Pack(p1.nome, 30, MPI_CHAR, message, 100, &position, MPI_COMM_WORLD);
      MPI_Pack(&p1.idade, 1, MPI_INT, message, 100, &position, MPI_COMM_WORLD);
      MPI_Pack(&p1.renda, 1, MPI_FLOAT, message, 100, &position, MPI_COMM_WORLD);
      for (i=1; i<size; i++)
       MPI_Send(message, 100, MPI_PACKED, i, tag, MPI_COMM_WORLD);
     }
    else
     {
      MPI_Recv(message, 100, MPI_PACKED, 0, tag, MPI_COMM_WORLD, &status);
      ...
     }
    printf("Mensagem do processo' %d : %s\n", rank, message);
    MPI_Finalize();
   }
 }
```

5.12 - MPI_PACK_SIZE

Obtendo o tamanho de um pacote de dados:

Definição:

Retorna o limite superior da quantidade de espaço necessária para empacotar uma mensagem.

Sintaxe:

C

```
int MPI_Pack_size (int incount, MPI_Datatype datatype, MPI_Comm comm, int* size)
```

Fortran

```
call MPI_PACK_SIZE (incount, datatype, comm, size, mpierr)
```

Parâmetros:

- incount - Quantidade de elementos a serem empacotados;
- datatype - Tipo de dado do argumento a ser empacotado;
- comm - Identificação do communicator;
- size - Limite superior do tamanho da mensagem a ser empacotada;
- mpierr - Código de erro com o status da rotina.

Erros:

MPI_ERR_ARG	- Argumento inválido;
MPI_ERR_COMM	- Comunicator inválido;
MPI_ERR_TYPE	- Tipo inválido.

Exemplo:

```
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
typedef struct pessoa
 {
  char nome[30];
  int idade;
  float renda;
 }
 pessoa;
main(int argc, char *argv[])
 {
  int mpierr, rank, size, position, tambuf, tambuftmp, i, tag;
  MPI_Status status;
  char *message;
  pessoa p1;
  mpierr = MPI_Init(&argc, &argv);
  if (mpierr >= 0)
   {
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    tambuf=0;
    MPI_Pack_size(30, MPI_CHAR, MPI_COMM_WORLD, &tambuftmp);
    tambuf=tambuf+tambuftmp;
    MPI_Pack_size(1, MPI_INT, MPI_COMM_WORLD, &tambuftmp);
    tambuf=tambuf+tambuftmp;
    MPI_Pack_size(1, MPI_FLOAT, MPI_COMM_WORLD, &tambuftmp);
    tambuf=tambuf+tambuftmp;
    message=(char*)malloc(tambuf*sizeof(char));
    if (rank == 0)
     {
      strcpy(p1.nome,"Joao da Silva");
      p1.idade=35;
      p1.renda=1235.50;
      MPI_Pack(p1.nome, 30, MPI_CHAR, message, 100, &position, MPI_COMM_WORLD);
      MPI_Pack(&p1.idade, 1, MPI_INT, message, 100, &position, MPI_COMM_WORLD);
      MPI_Pack(&p1.renda, 1, MPI_FLOAT, message, 100, &position, MPI_COMM_WORLD);
      for (i=1; i<size; i++)
       MPI_Send(message, tambuf, MPI_PACKED, i, tag, MPI_COMM_WORLD);
     }
    else
     {
      ...
      MPI_Recv(message, tambuf, MPI_PACKED, 0, tag, MPI_COMM_WORLD, &status);
     }
    printf("Mensagem do processo' %d : %s\n", rank, message);
    MPI_Finalize();
   }
 }
```

5.13 - MPI_UNPACK

Desempacotamento de dados na recepção:

Definição:

Desempacota um conjunto de dados contíguos na memória. Estes dados podem ser de vários tipos.

Sintaxe:

C

int MPI_Unpack (void* inbuf, int insize, int* position, void* outbuf, int outsize, MPI_Datatype datatype, MPI_Comm comm)

Fortran

call MPI_UNPACK (inbuf, insize, position, outbuf, outsize, datatype, comm, mpierr)

Parâmetros:

inbuf	- Buffer de entrada;
request_vector	- Vetor dos processos a serem verificados;
insize	- Tamanho do buffer de entrada (bytes);
datatype	- Tipo de dado de cada elemento do buffer;
outbuf	- Buffer de saída;
outsize	- Tamanho do buffer de saída;
position	- Posição incial do buffer de saída;
comm	- Identificação do communicator;
mpierr	- Código de erro com o status da rotina.

Erros:

MPI_ERR_ARG	- Argumento inválido;
MPI_ERR_COMM	- Comunicator inválido;
MPI_ERR_COUNT	- Argumento numérico inválido;
MPI_ERR_TYPE	- Tipo inválido.

Observação:

Existe uma diferença no significado do argumento relacionado com o número de elementos a serem recebidos pelas rotinas MPI_Recv e MPI_Unpack: na primeira rotina, o que interessa é o máximo de elementos e na segunda, a quantidade exata dos elementos a serem desempacotados.

Exemplo:

```
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
typedef struct pessoa
 {
  char nome[30];
  int idade;
  float renda;
 }
 pessoa;
main(int argc, char *argv[])
 {
  int mpierr, rank, size, position, tambuf, tambuftmp, i, tag;
  MPI_Status status;
  char *message;
  pessoa p1;
  mpierr = MPI_Init(&argc, &argv);
  if (mpierr >= 0)
   {
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    tambuf=0;
    MPI_Pack_size(30, MPI_CHAR, MPI_COMM_WORLD, &tambuftmp);
    tambuf=tambuf+tambuftmp;
    MPI_Pack_size(1, MPI_INT, MPI_COMM_WORLD, &tambuftmp);
    tambuf=tambuf+tambuftmp;
    MPI_Pack_size(1, MPI_FLOAT, MPI_COMM_WORLD, &tambuftmp);
    tambuf=tambuf+tambuftmp;
    message=(char*)malloc(tambuf*sizeof(char));
    if (rank == 0)
     {
      strcpy(p1.nome,"Joao da Silva");
      p1.idade=35;
      p1.renda=1235.50;
      MPI_Pack(p1.nome, 30, MPI_CHAR, message, 100, &position, MPI_COMM_WORLD);
      MPI_Pack(&p1.idade, 1, MPI_INT, message, 100, &position, MPI_COMM_WORLD);
      MPI_Pack(&p1.renda, 1, MPI_FLOAT, message, 100, &position, MPI_COMM_WORLD);
      for (i=1; i<size; i++)
       MPI_Send(message, tambuf, MPI_PACKED, i, tag, MPI_COMM_WORLD);
     }
    else
     {
      position=0;
      MPI_Recv(message, tambuf, MPI_PACKED, 0, tag, MPI_COMM_WORLD, &status);
      MPI_Unpack(message, tambuf, &position, p1.nome, 30, MPI_CHAR, MPI_COMM_WORLD);
      MPI_Unpack(message, tambuf, &position, &p1.idade, 1, MPI_INT, MPI_COMM_WORLD);
      MPI_Unpack(message, tambuf, &position, &p1.renda, 1, MPI_FLOAT, MPI_COMM_WORLD);
     }
    printf("Mensagem do processo' %d : %s\n", rank, message);
    MPI_Finalize();
   }
 }
```

5.14 - MPI_WAIT

Bloqueio de execução de um processo:

Definição:
Bloqueia a execução de um programa até que seja completada a ação identificada pela variável request.

Sintaxe:
C
int MPI_Wait (MPI_Request *request, MPI_Status *status)
Fortran
call MPI_WAIT (request, status, mpierr)

Parâmetros:
request - Variável que indica se o evento em questão com aquele request foi executado;
status - Retorno do status do evento definido por request;
mpierr - Código de erro com o status da rotina.

Erros:

MPI_ERR_REQUEST	- Requisição não persistente;
MPI_ERR_ARG	- Argumento inválido.

Observação:
Os valores de request podem ser null (nulo), inactive (inativo) ou active (ativo).

Exemplo:

```
#include
 #include "mpi.h"
   main(int argc, char *argv[])
   {
    int mpierr, rank, size;
    int i, send_reques, sucessor, predecessor, tag=10;
    MPI_Request send_request, recv_request;
    MPI_Status status;
    char message[20]="Mensagem do MPI";
    mpierr = MPI_Init(&argc, &argv);
    if (mpierr < 0)
     {
      printf ("Nao foi possivel inicializar o processo MPI!\n");
      return;
     }
    else
     {
      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
      MPI_Comm_size(MPI_COMM_WORLD, &size);
      if (rank==0)
        predecessor=size-1;
      else
        predecessor=rank-1;
      if (rank==size-1)
        sucessor=0;
      else
        sucessor=rank+1;
      MPI_Isend(message, 20, MPI_CHAR, sucessor, tag, MPI_COMM_WORLD, &send_request);
      MPI_Irecv(message, 20, MPI_CHAR, predecessor, tag, MPI_COMM_WORLD, &recv_request);
      MPI_Wait(&send_request, &status);
      MPI_Wait(&recv_request, &status);
      printf("Processo = %d - Enviou para %d e recebeu de %d\n", rank, sucessor, predecessor);
      MPI_Finalize();
     }
   }
```

5.15 - MPI_WAITANY

Bloqueio de Execução de um Processo por qualquer outro:

Definição:

Bloqueia a execução de um processo até que alguma determinada operação de comunicação - envio (send) ou recebimento (receive) – seja completada. Se mais de uma operação for completada, a rotina arbitrariamente escolhe um valor.

Sintaxe:

C

int MPI_Waitany (int incount, MPI_Request *request_vector, int* index, int* flag, MPI_Status *status_vector)

Fortran

call MPI_WAITANY (incount, request_vector, index, flag, status_vector, mpierr)

Parâmetros:

incount	- Quantidade de processos a serem verificados;
request_vector	- Vetor dos processos a serem verificados;
index	- Índice do processo que teve a operação completada.
flag	- Valor booleano que indica se a operação foi concluída.
status_vector	- Vetor com o status do evento em cada processo.
mpierr	- Código de erro com o status da rotina.

Erros:

MPI_ERR_REQUEST	- Requisição não persistente;
MPI_ERR_ARG	- Argumento inválido.

Exemplo:

```
#include <stdio.h>
#include "mpi.h"
  main(int argc, char *argv[])
  {
   int mpierr, rank, size, *index, tag, completed;
   MPI_Request *request, rec_request, send_request;
   MPI_Status *status;
   char message[20]="Mensagem do MPI";
   mpierr = MPI_Init(&argc, &argv);
   if (mpierr < 0)
    {
     printf ("Nao foi possivel inicializar o processo MPI!\n");
     return;
    }
   else
    {
     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
     MPI_Comm_size(MPI_COMM_WORLD, &size);
     if (rank == 0)
      {
       request=(MPI_Request*)malloc(size*sizeof(MPI_Request));
       index=(int*)malloc(size*sizeof(int));
       status=(MPI_Status*)malloc(size*sizeof(MPI_Status));
       printf("Iniciando o Waitany...\n");
       MPI_Waitany(size, request, &index, &status);
       printf("...Encerrando o Waitany!\n");
      }
     else
      {
       tag=0;
       MPI_Isend(message, 20, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &send_request);
      }
     MPI_Finalize();
    }
  }
```

5.16 - MPI_WAITSOME

Bloqueio de execução de um processo por algum outro:

Definição:

Bloqueia a execução de um processo até que uma determinada operação de comunicação - envio (send) ou recebimento (receive) – seja completada por um processo específico.

Sintaxe:

C

int MPI_Waitsome (int incount, MPI_Request *request_vector, int* outcount, int* index_vector, MPI_Status *status_vector)

Fortran

call MPI_WAITSOME (incount, request_vector, outcount, index_vector, status_vector, mpierr)

Parâmetros:

incount	- Quantidade de processos a serem verificados;
request_vector	- Vetor dos processos a serem verificados;
outcount	- Quantidade de processos com a operação completada;
index_vector	- Índice dos processos que tiveram a operação completada;
status_vector	- Vetor com o status do evento em cada processo;
mpierr	- Código de erro com o status da rotina.

Erros:

MPI_ERR_REQUEST	- Requisição não persistente;
MPI_ERR_ARG	- Argumento inválido.

Exemplo:

```
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
  main(int argc, char *argv[])
  {
   int mpierr, rank, size, *index, tag, completed;
   MPI_Request *request, send_request;
   MPI_Status *status;
   char message[20]="Mensagem do MPI";
   mpierr = MPI_Init(&argc, &argv);
   if (mpierr < 0)
    {
     printf ("Nao foi possivel inicializar o processo MPI!\n");
     return;
    }
   else
    {
     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
     MPI_Comm_size(MPI_COMM_WORLD, &size);
     if (rank == 0)
      {
       request=(MPI_Request*)malloc(size*sizeof(MPI_Request));
       index=(int*)malloc(size*sizeof(int));
       status=(MPI_Status*)malloc(size*sizeof(MPI_Status));
       printf("Iniciando o Waitsome...\n");
       MPI_Waitsome(size, request, &completed, index, status);
       printf("...Encerrando o Waitsome!\n");
      }
     else
      {
       tag=0;
       MPI_Isend(message, 20, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &send_request);
      }
     MPI_Finalize();
    }
  }
```

5.17 - MPI_WAITALL

Bloqueio de execução de um processo por todos os outros:

Definição:

Bloqueia a execução de um processo até que todos os processos na lista tenham executado uma operação receive.

Sintaxe:

C

int MPI_Waitall (int count, MPI_Request *request_vector, MPI_Status *status_vector)

Fortran

call MPI_WAITALL (count, request_vector, status_vector, mpierr)

Parâmetros:

incount	- Quantidade de processos a serem verificados;
request_vector	- Vetor dos processos a serem verificados;
status_vector	- Vetor com o status do evento em cada processo.
mpierr	- Código de erro com o status da rotina.

Erros:

MPI_ERR_REQUEST	- Requisição não persistente;
MPI_ERR_ARG	- Argumento inválido.

Exemplo:

```
#include <stdio.h>
#include "mpi.h"
 main(int argc, char *argv[])
 {
  int mpierr, rank, size, *index;
  MPI_Request request[], send_request;
  MPI_Status *status;
  char message[20]="Mensagem do MPI";
  mpierr = MPI_Init(&argc, &argv);
  if (mpierr < 0)
   {
    printf ("Nao foi possivel inicializar o processo MPI!\n");
    return;
   }
  else
   {
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    if (rank == 0)
     {
      request=(MPI_Request*)malloc(size*sizeof(MPI_Request));
      status=(MPI_Status*)malloc(size*sizeof(MPI_Status));
      printf("Iniciando o Waitall...\n");
      MPI_Waitall(size-1, request, status);
      printf("...Encerrando o Waitall.\n");
     }
    else
     {
      tag = 0;
      MPI_Isend(message, 20, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &send_request);
     }
    MPI_Finalize();
   }
 }
```

5.18 - MPI_BARRIER

Sincronizando Processos num Grupo:

Definição:

Essa rotina sincroniza todos os processos de um grupo. Um processo de um grupo que utiliza MPI_Barrier interrompe a sua execução até que que todos os processos do mesmo grupo executem também um MPI_Barrier.

Sintaxe:

C

int MPI_Barrier (MPI_Comm comm)

Fortran

call MPI_BARRIER (comm, mpierr)

Parâmetros:

comm - Identificação do communicator;
mpierr - Código de erro com o status da rotina.

Erros:

MPI_ERR_COMM	- Comunicator inválido.

Exemplo:

```
#include <stdio.h>
#include "mpi.h"
#include <math.h>
  main(int argc, char *argv[])
    {
     int mpierr, rank, size;
     float i, j;
     mpierr = MPI_Init(&argc, &argv);
     if (mpierr < 0)
      {
       printf ("Nao foi possivel inicializar o processo MPI!\n");
       return;
      }
     else
      {
       MPI_Comm_rank(MPI_COMM_WORLD, &rank);
       MPI_Comm_size(MPI_COMM_WORLD, &size);
       if (rank==0)
        {
         printf("Iniciando calculo no processo 0...\n");
         printf("(Outros processos estarao em espera)\n");
         for (i=0;i<10000000;i++)
          j=sqrt(i);
         printf("...Calculo encerrado.\n");
         printf("(Sincronizando os processos)\n");
        }
       MPI_Barrier(MPI_COMM_WORLD);
       MPI_Finalize();
      }
  }
```

Observação:

Existe uma diferença no significado do argumento relacionado com o número de elementos a serem recebidos pelas rotinas MPI_Recv e MPI_Unpack: na primeira rotina, o que interessa é o máximo de elementos e na segunda, a quantidade exata dos elementos a serem desempacotados.

5.19 - MPI_BCAST

Enviando dados para todos os processos:

Definição:

Diferentemente da comunicação ponto-a-ponto, na comunicação coletiva é possível enviar/receber dados simultaneamente de/para vários processos.

Sintaxe:

C

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

Fortran

call MPI_BCAST (buffer, count, datatype, root, comm, mpierr)

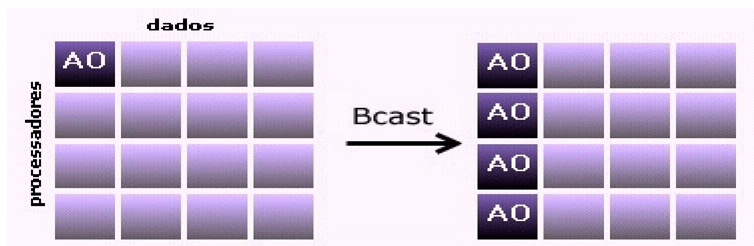
Parâmetros:

buffer	- Endereço do dado a ser enviado;
count	- Número de elementos a serem enviados;
datatype	- Tipo do dado;
root	- Identifica o processo que irá efetuar o broadcast (origem);
tag	- Variável inteira com o rótulo da mensagem
comm	- Identifica o Communicator;

Erros:

MPI_ERR_BUFFER	- Buffer inválido ou vazio.
MPI_ERR_COMM	- Comunicator inválido.
MPI_ERR_COUNT	- Argumento numérico inválido.
MPI_ERR_TYPE	- Tipo de dado inválido;
MPI_ERR_ROOT	- Processo root (origem) fora da faixa do rank (número total de processos).

Ilustração:



<u>*Exemplo:*</u>

```
#include <stdio.h>
#include "mpi.h"
main(int argc, char *argv[])
{
 int mpierr, rank, size, *index;
 char message[20];
 mpierr = MPI_Init(&argc, &argv);
 if (mpierr < 0)
   {
    printf ("Nao foi possivel inicializar o processo MPI!\n");
    return;
   }
 else
   {
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    if (rank==0)
      strcpy(message,"Mensagem do no 0\n");
    MPI_Bcast(message, 20, MPI_CHAR, 0, MPI_COMM_WORLD);
    printf("A mensagem recebida foi: %s", message);
    MPI_Finalize();
   }
}
```

5.20 - MPI_SCATTER

Enviando mensagens coletivamente:

Definição:

É uma rotina para o envio de mensagens para um subgrupo de processos. Com a rotina MPI_Scatter a mensagem pode ser segmentada e enviada para processos diferentes.

Sintaxe:

C

int MPI_Scatter (void *sbuf,int scount,MPI_Datatype stype,void *rbuf, int rcount,MPI_Datatype rtype,int root, MPI_Comm comm)

Fortran

call MPI_SCATTER (sbuf, scount,stype, rbuf, rcount, rtype, root, comm, mpierr)

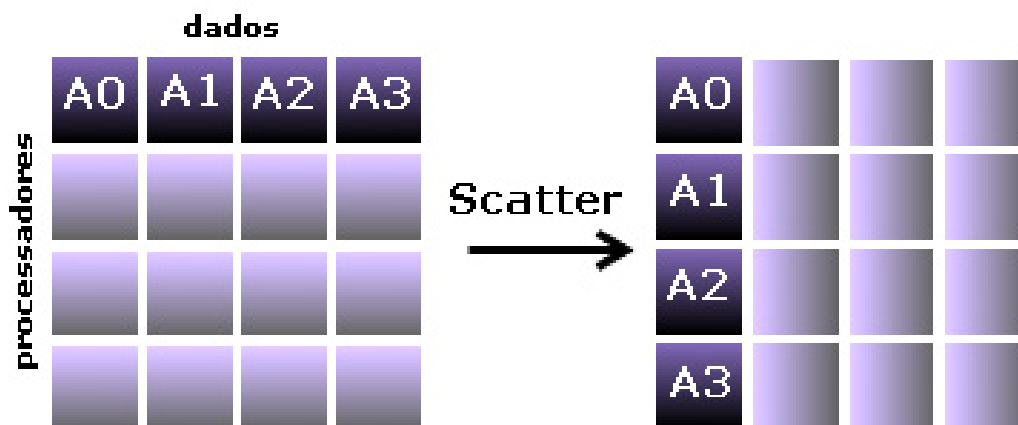
Parâmetros:

sbuf - Endereço dos dados a serem distribuídos;
scount - Número de elementos enviados para cada processo;
stype - Tipo do dado a ser enviado;
rbuf - Endereço onde os dados serão armazenados;
rcount - Quantidade de dados recebidos;
rtype - Tipo do dado recebido;
root - Identifica o processo que irá distribuir os dados;
comm - Identifica o Communicator;

Erros:

MPI_ERR_BUFFER	- Buffer inválido ou vazio.
MPI_ERR_COMM	- Comunicator inválido.
MPI_ERR_COUNT	- Argumento numérico inválido.
MPI_ERR_TYPE	- Tipo de dado inválido;

Ilustração:



Exemplo:

```
#include "mpi.h"
#include <stdio.h>
#define SIZE 4
int main(argc,argv)
 int argc;
 char *argv[];
  {
   int numtasks, rank, sendcount, recvcount, source;
   float sendbuf[SIZE][SIZE] = {
     {1.0, 2.0, 3.0, 4.0},
     {5.0, 6.0, 7.0, 8.0},
     {9.0, 10.0, 11.0, 12.0},
     {13.0, 14.0, 15.0, 16.0}  };
   float recvbuf[SIZE];
   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
   if (numtasks == SIZE)
    {
     source = 0;
     sendcount = SIZE;
     recvcount = SIZE;
     MPI_Scatter(sendbuf, sendcount, MPI_FLOAT, recvbuf, recvcount, MPI_FLOAT,
                 source, MPI_COMM_WORLD);
     printf("rank= %d  Results: %f %f %f %f\n",rank,recvbuf[0],
            recvbuf[1],recvbuf[2],recvbuf[3]);
    }
   else
     printf("Must specify %d processors. Terminating.\n",SIZE);
  MPI_Finalize();
 }
```

5.21 - MPI_GATHER

Coletando mensagens de processos:

Definição:
Rotina para coleta de mensagens de um subgrupo de processos.

Sintaxe:
C
int MPI_Gather (void *sbuf,int scount, MPI_Datatype stype,void *rbuf,int rcount,MPI_Datatype rtype,int root,MPI_Comm comm)
Fortran
call MPI_GATHER (sbuf, scount,stype, rbuf, rcount, rtype, root, comm, mpierr)

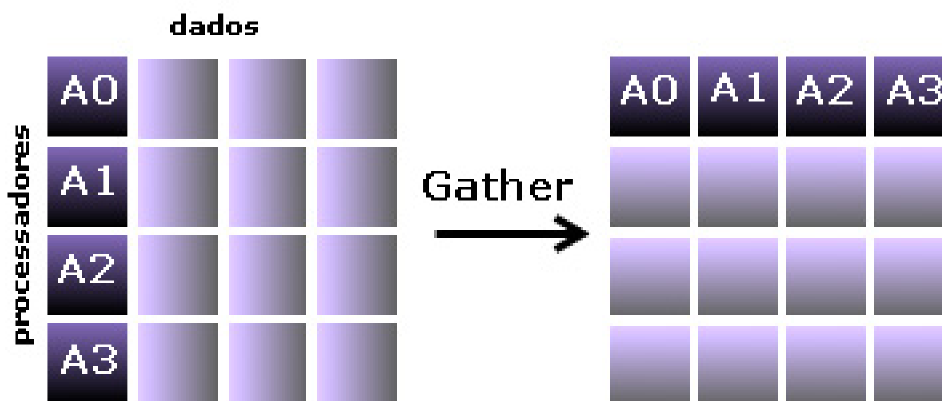
Parâmetros:

sbuf	- Endereço inicial do dado a ser coletado;
scount	- Número de dados a serem coletados;
stype	- Tipo do dado a ser coletado;
rbuf	- Endereço onde os dados serão armazenados;
rcount	- Número de elementos recebidos por processo;
rtype	- Tipo do dado recebido;
root	- Identifica o processo que irá efetuar a coleta;
comm	- Identifica o Communicator;

Erros:

MPI_ERR_BUFFER	- Buffer inválido ou vazio.
MPI_ERR_COMM	- Comunicator inválido.
MPI_ERR_COUNT	- Argumento numérico inválido.
MPI_ERR_TYPE	- Tipo de dado inválido;

Ilustração:



Exemplo:

```
#include "mpi.h"
    #include <stdio.h>
    #define SIZE 4
      int main(argc,argv)
      int argc;
      char *argv[];
       {
        int numtasks, rank, sendcount, recvcount, source, i;
        float sendbuf[SIZE][SIZE];
        float recvbuf[SIZE];
        MPI_Init(&argc,&argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
        if (numtasks == SIZE)
         {
          for (i=0;i<4;i++)
            recvbuf[i]=(float)rank*SIZE + i;
          source = 0;
          sendcount = SIZE;
          recvcount = SIZE;
          MPI_Gather(recvbuf, recvcount, MPI_FLOAT, sendbuf, sendcount, MPI_FLOAT,
                     source, MPI_COMM_WORLD);
          if (rank==source)
            for (i=0;i<SIZE;i++)
              printf("Results: %f %f %f %f\n",sendbuf[i][0],
                     sendbuf[i][1],sendbuf[i][2],sendbuf[i][3]);
         }
        else
          printf("Especifique %d processadores. Abortado.\n",SIZE);
       MPI_Finalize();
     }
```

5.22 - MPI_ALLGATHER

Coleta global de dados:

Definição:
Faz com que todos os processos coletem dados de cada processo da aplicação.

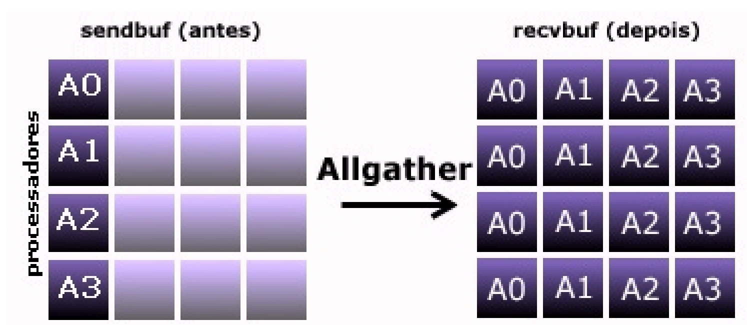
Sintaxe:
C
```
int MPI_Allgather (void *sbuf,int scount,MPI_Datatype stype,void *rbuf, int rcount,MPI_Datatype rtype, MPI_Comm comm)
```
Fortran
```
call MPI_ALLGATHER (sbuf, scount,stype, rbuf, rcount, rtype, comm, mpierr)
```

Parâmetros:
sbuf - Endereço do dado a ser enviado;
scount - Número de elementos a serem enviados;
stype - Tipo do dado a ser enviado;
rbuf - Endereço onde os dados serão coletados;
rcount - Número de elementos recebidos por processo;
rtype - Tipo do dado recebido;
comm - Identifica o Communicator;

Erros:

MPI_ERR_BUFFER	- Buffer inválido ou vazio.
MPI_ERR_COMM	- Comunicator inválido.
MPI_ERR_COUNT	- Argumento numérico inválido.
MPI_ERR_TYPE	- Tipo de dado inválido;

Ilustração:



Exemplo:

```
 #include "mpi.h"
#include <stdio.h>
#define SIZE 4
int main(argc,argv)
 int argc;
 char *argv[];
  {
   int numtasks, rank, sendcount, recvcount, i;
   char sendbuf[SIZE], recvbuf[SIZE];
   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
   for (i=0;i<SIZE;i++)
     sendbuf[i]=65+i;
   if (numtasks == SIZE)
    {
     sendcount = 1;
     recvcount = 1;
     MPI_Allgather(sendbuf, sendcount, MPI_CHAR, recvbuf, recvcount,
                   MPI_CHAR, MPI_COMM_WORLD);
     printf("rank= %d  Results all: %c %c %c %c\n",rank,recvbuf[0],
             recvbuf[1],recvbuf[2],recvbuf[3]);
    }
   else
     printf("Must specify %d processors. Terminating.\n",SIZE);
   MPI_Finalize();
  }
```

Observação:

Na rotina MPI_Allgatter, todos os processos são envolvidos na coleta recíproca de dados.

5.23 - MPI_ALLTOALL

Comunicação global.

Definição:
Faz com que cada processo envie seus dados para todos os processos da aplicação.

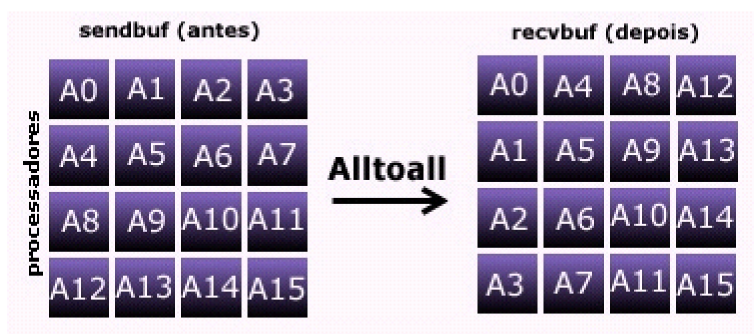
Sintaxe:
C
```
int MPI_Alltoall (void *sbuf, int scount, MPI_Datatype stype, void *rbuf, int rcount,MPI_Datatype rtype, MPI_Comm comm)
```
Fortran
```
call MPI_ALLTOALL (sbuf, scount,stype, rbuf, rcount, rtype, comm, mpierr)
```

Parâmetros:
sbuf - Endereço do dado a ser enviado;
scount - Número de elementos a serem enviados;
stype - Tipo do dado a ser enviado;
rbuf - Endereço onde os dados serão armazenados;
rcount - Número de elementos recebidos de cada processo;
rtype - Tipo do dado recebido;
comm - Identifica o Communicator;

Erros:

MPI_ERR_BUFFER	- Buffer inválido ou vazio.
MPI_ERR_COMM	- Comunicator inválido.
MPI_ERR_COUNT	- Argumento numérico inválido.
MPI_ERR_TYPE	- Tipo de dado inválido;

Ilustração:



Exemplo:

```
#include "mpi.h"
      #include <stdio.h>
      #define SIZE 4
      int main(argc,argv)
       int argc;
       char *argv[];
        {
         int numtasks, rank, sendcount, recvcount, i;
         char sendbuf[SIZE], recvbuf[SIZE];
         MPI_Init(&argc,&argv);
         MPI_Comm_rank(MPI_COMM_WORLD, &rank);
         MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
         for (i=0;i<SIZE;i++)
           sendbuf[i]=65+i;
         if (numtasks == SIZE)
          {
           sendcount = 1;
           recvcount = 1;
           MPI_Alltoall(sendbuf, sendcount, MPI_CHAR, recvbuf, recvcount,
                         MPI_CHAR, MPI_COMM_WORLD);
           printf("rank= %d  Results all: %c %c %c %c\n",rank,recvbuf[0],
                   recvbuf[1],recvbuf[2],recvbuf[3]);
          }
         else
           printf("Must specify %d processors. Terminating.\n",SIZE);
         MPI_Finalize();
        }
```

Observação:

A rotina MPI_Alltoall é a mais genérica para a distribuição dos dados.

5.24 - MPI_Reduce

Realizando uma computação global:

Definição:

Rotina que faz com que todos os processos executem uma operação, onde o resultado parcial de cada processo é combinado e retorna para um processo específico.

Sintaxe:

C

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

Fortran

call MPI_REDUCE (sbuf, rbuf, count, datatype, op, root, comm, mpierr)

Parâmetros:

sbuf	- Endereço do dado a ser enviado;
rbuf	- Endereço do dado a ser recebido;
count	- Número de elementos a serem distribuídos;
datatype	- Tipo do dado a ser computado;
op	- Operaçâo a ser executada;
root	- Processo que irá receber o resultado da operação;
comm	- Identifica o Communicator;

Erros:

MPI_ERR_BUFFER	- Buffer inválido ou vazio.
MPI_ERR_COMM	- Comunicator inválido.
MPI_ERR_COUNT	- Argumento numérico inválido.
MPI_ERR_TYPE	- Tipo de dado inválido;

Observação: As operações possíveis são listadas na tabela abaixo:

Função	Significado	C	Fortran
MPI_MAX	Valor máximo	Int, float	integer, real, complex
MPI_MIN	Valor míximo	Int, float	integer, real, complex
MPI_SUM	Somatório dos valores	Int, float	integer, real, complex
MPI_PROD	Produtório dos valores	Int, float	integer, real, complex
MPI_LAND	E (and) lógico	Int	Boolean
MPI_BAND	E (and) lógico a nível de BIT	Int	Boolean
MPI_LOR	OU (or) lógico	Int	Boolean
MPI_BOR	OU (or) lógico a nível de BIT	Int	Boolean
MPI_LXOR	OU-EXCLUSIVO (xor) lógico	Int	Boolean
MPI_BXOR	OU-EXCLUSIVO (xor) lógico a nível de BIT	Int	Boolean
MPI_MAXLOC	Valor máximo de maior índice	Int, float	integer, real, complex
MPI_MINLOC	Valor mínimo de menor índice	Int, float	integer, real, complex

Exemplo:

```
#include "mpi.h"
 #include <stdio.h>
 #define SIZE 4
 int main(argc,argv)
  int argc;
  char *argv[];
   {
    int numtasks, rank, sendcount, recvcount, i, maior;
    int sendbuf, recvbuf;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
    sendbuf=rank;
    if (numtasks == SIZE)
     {
      MPI_Reduce(&sendbuf, &recvbuf, 1, MPI_INT, MPI_MAX, 0, MPI_COMM_WORLD);
      if (rank==0) printf("Maior = %d\n", recvbuf);
     }
    else  printf("Must specify %d processors. Terminating.\n",SIZE);
    MPI_Finalize();
   }
```

5.25 - MPI_WTIME

Computando o tempo decorrido:

Definição:

Retorna (em precisão numérica dupla) o número de segundos decorridos desde algum tempo no passado.

Sintaxe:

C

double MPI_Wtime(void)

Fortran

MPI_WTIME(tempo)

Parâmetros:

(Nenhum)

Erros:

(Nenhum)

Observação:

A função MPI_Wtime retornará o tempo decorrido desde o início do programa. Sua precisão, pode ser obtida através da rotina MPI_Wtick.

Exemplo:

```
#include <stdio.h>
#include "mpi.h"
main(int argc, char *argv[])
{
 int mpierr, rank, size;
 double tempo;
 mpierr = MPI_Init(&argc, &argv);
 if (mpierr < 0)
  {
   printf ("Nao foi possivel inicializar o processo MPI!\n");
   return;
  }
 else
  {
   tempo=MPI_Wtime();
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Barrier(MPI_COMM_WORLD);
   tempo=MPI_Wtime()-tempo;
   printf("Tempo decorrido no processo %d = %f\n", rank, tempo);
   MPI_Finalize();
  }
}
```

5.26 - MPI_WTICK

Dando a precisão do tempo:

Definição:

Retorna (em valor real) a precisão de tempo computada pelo comando MPI_Wtime.

Sintaxe:

C

double MPI_Wtick(void)

Fortran

MPI_WTICK(tempo)

Parâmetros:

(Nenhum)

Erros:

(Nenhum)

Observação:

A função MPI_Wtime retornará a precisão em múltiplos ou submúltiplos de segundo. Por exemplo: se MPI_Wtime for incrementado a cada milisegundo, esta rotina retornará 0.001.

Exemplo:

```
#include <stdio.h>
 #include "mpi.h"
  main(int argc, char *argv[])
  {
   int mpierr, rank, size;
   double utempo;
   mpierr = MPI_Init(&argc, &argv);
   if (mpierr < 0)
     {
      printf ("Nao foi possivel inicializar o processo MPI!\n");
      return;
     }
   else
     {
      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
      MPI_Comm_size(MPI_COMM_WORLD, &size);
      utempo=MPI_Wtick();
      printf("A unidade de tempo no processo %d é dada em %f s.\n", rank, utempo);
      MPI_Finalize();
     }
  }
```

5.27 - MPI_GET_PROCESSOR_NAME

Identificando onde um processo executa:

Definição:

Retorna o nome da máquina onde um dado processo está executando.

Sintaxe:

C

double MPI_Get_processor_name (char *procname, int *namelen)

Fortran

MPI_GET_PROCESSOR_NAME (procname, namelen, mpierr)

Parâmetros:

procname - Nome do processador (var. de saída);

namelen - Tamanho da string a ser retornada

Erros:

(Nenhum)

Exemplo:

```
#include <stdio.h>
 #include "mpi.h"
  main(int argc, char *argv[])
  {
   int mpierr, rank, size;
   char procname[MPI_MAX_PROCESSOR_NAME];
   int namelen;
   mpierr = MPI_Init(&argc, &argv);
   if (mpierr < 0)
    {
     printf ("Nao foi possivel inicializar o processo MPI!\n");
     return;
    }
   else
    {
     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
     MPI_Comm_size(MPI_COMM_WORLD, &size);
     MPI_Get_processor_name(procname, &namelen);
     printf("O processo %d está executando em %s\n", rank, procname);
     MPI_Finalize();
    }
  }
```

CONCLUSÃO

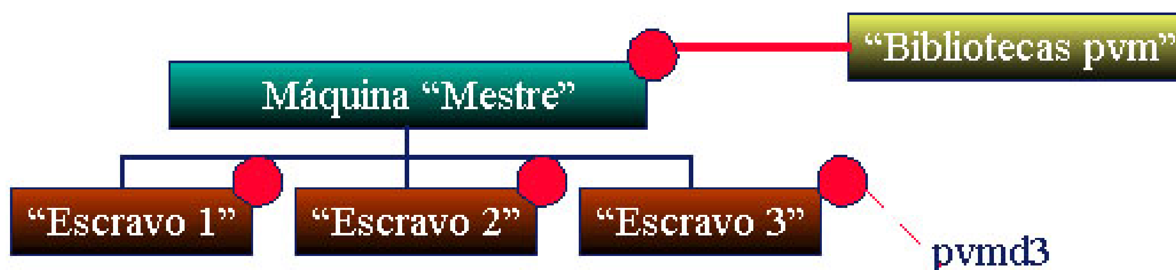
6.1 - UM POUCO DE PVM

6.1.1 - História do PVM

Surgimento: 1989 - Vaidy Sunderam - Emory University & Oak Ridge National Laboratory;
Motivação: Ferramenta usada para criar e executar aplicações paralelas, usando o hardware já existente;
Versão Atual: 3.3.11 - Instalada no CENAPAD/NE - Ambiente nativo: UNIX.

6.1.2 - Características do PVM:

- Capacidade de utilização efetiva de computação paralela, ainda que em arquitetura não-paralela;
- Capacidade de paralelização escalável e dinâmica;
- Grande difusão e aceitação;
- Flexibilidade:
  - Variedade de arquiteturas e redes de trabalho;
  - Recurso computacional expansível;
  - Independência das aplicações.
- Facilidade de utilização e de programação (Linguagens C e Fortran);
- Elementos constituintes: daemon pvmd3 e bibliotecas do sistema:



6.1.3 - O conceito de máquina virtual paralela:

- A abstração em um elevado nível de arquitetura;
- Significado da máquina virtual paralela:

Conjunto de computadores conectados por uma rede de comunicação que trabalham cooperativamente para resolver um grande problema computacional.

- Portabilidade foi considerada mais importante do que performance;
- PVM não somente suporta o nível de portabilidade, mas o expande para o nível de interoperabilidade: um mesmo programa pode estar interagindo com outro gerado para uma arquitetura completamente diferente.

6.2 - PVM VERSUS MPI

6.2.1 - Portabilidade versus Interoperabilidade:

PVM	MPI
Nivel de Interoperabilidade: Além da portabilidade (como o MPI), os programas permitem nível de interoperabilidade, tornando possível execuções em arquiteturas distintas quaisquer modificações.	Nível de Portabilidade: Programas escritos para uma arquitetura podem ser compilados para uma outra arquitetura e executados sem quaisquer modificações.

6.2.3 - Máquina Virtual:

PVM	MPI
Completa Abstração: O PVM permite que se veja a rede como uma coleção dinâmica de recursos computacionais (potencialmente heterogêneos), administrados como um computador paralelo único. Controle de processos: capacidade de iniciar, interromper e controlar processos, em tempo de execução; Controle de recursos: Bastante dinâmico - permite abstração, graças ao conceito da máquina virtual paralela; Topologia: O PVM exige que o programador arranje manualmente tarefas em grupos, segundo a organização desejada.	(Não suportado): O MPI não possui o conceito da máquina virtual paralela, centralizando-se no conceito de message-passing. Controle de processos: Bastante restrito - permite somente o controle de grupos de tarefas; Controle de recursos: Essencialmente estático - não suporta qualquer nível de abstração; Topologia: Ainda que não possua o conceito de PVM, o MPI provê um alto nível de abstração, em termos de topologia.

6.2.3 - Tolerância a Falhas:

PVM	MPI
Básico: Existem esquemas básicos de notificação de falha, para alguns casos. Porém, permite flexibilidade, de forma que, ainda em certas situações onde não existe resposta de uma máquina, uma aplicação receba resultados das outras.	Rudimentar: As versões iniciais, não incluiam qualquer esquema de tolerância, mas a partir das ultimas especificações, esquemas similares ao de PVM serão providos. Infelizmente, o modelo seguido é estático.

6.2.4 - Segurança na Comunicação:

PVM	MPI
Existente: Visto que o PVM possui um daemon mantendo a máquina virtual, ele pode usá-lo para criar rótulos únicos de contexto. Processos podem se comunicar com grupos, permitindo recuperação de falhas. As novas versões pretendem fazer uso de Comunicadores, como os do MPI.	Excelente: O conceito de communicators no MPI, permite prover um elevado nível de segurança na comunicação, permitindo diferenciar mensagens de bibliotecas de mensagens de usuários.

Abaixo, segue-se uma comparação entre as semelhanças de comunicadores do MPI e do PVM3.4:

Função	PVM	MPI
Criação de comunicador	pvm_static_group	MPI_COMM_CREATE
Destrução de comunicador	pvm_lvgroup	MPI_COMM_FREE
Novo contexto	pvm_newcontext	MPI_COMM_DUP
Inter-comunicação	Sem restrições	Algumas restrições
Suporte de Tolerância a Falhas	Sim	Não

6.3 – PVMPI

6.3.1 - A tendência do futuro:

Proposta: Oak Ridge National Laboratory e University of Tennessee- Viabilizar uma mistura entre as características do PVM e do MPI.

Funções:
- Utilizar implementações específicas de hardware, quando disponíveis em máquinas multiprocessadoras;
- Permitir acesso à idéia de uma máquina virtual, com controle de recursos e tolerância a falhas;
- Usar a rede de comunicação PVM transparentemente para transferir dados entre diferentes implementações de MPI, permitindo a interoperabilidade.

Característica Final: Seria um misto entre as funções do PVM, MPI, mantendo, tanto quanto possível, as identidades desses dois message-passing.

6.3.2 - Enquanto o PVMPI não vem... o que usar?

Como decidir o que usar?
Ponderar as características / necessidade da aplicação;
Verificar as configurações do ambiente, arquiteturas, etc;
Analisar cautelosamente prós e contras do MPI e do PVM.

Regra Básica:

Message Passing	PVM	MPI
Características	Em aplicações a serem executadas em ambientes heterogêneos; Quando a idéia de uma máquina virtual se torna presente; Em situações onde a interoperabilidade seja necessária; Quando se torna necessário um administrador de recursos e funções de controle para aplicações portáveis; Para situações onde a tolerância a falhas é essencial.	Quando a boa performance é o objetivo; Em situações onde rotinas específicas (muitas vezes do próprio fabricante) devam ser usadas, para melhorar o desempenho. Quando se a aplicação deve executar em um ambiente homogêneo.

6.4 - JAVA E MPP

Explode a Java:

O que percebemos atualmente é que a Java tem se constituído num dos principais focos de atenção quando se fala em processamento via rede. Sua própria definição já surge como um atrativo:

"Uma linguagem simples, orientada a objetos, distribuída, interpretada, robusta, segura, independente de arquitetura, portável, de alto desempenho, multi-tarefa e dinâmica".

Desenvolvida a partir da idéia de código neutro (bytecodes), as aplicações e os applets (que rodam sob um web browser) possuem internamente java virtual machines (JVM´s) responsáveis pela tradução deste formato para instruções finais de máquina.

Como os JVM´s podem ser implementados em praticamente qualquer arquitetura, temos uma independência de plataformas extremamente eficiente.

MPP:

Com a Java, surgiu também o conceito de Massively Parallel Processing (MPP). A computação massivamente paralela se caracteriza por ser extremamente distribuída e aplica-se a problemas que possuem granulosidade fina e a máxima independência possível (o que implica em baixíssima comunicação).

A esta altura, pode-se perguntar:

Mas com tudo isso, seria a Java a resposta definitiva?

A resposta é negativa. Muitas aplicações tornariam-se proibitivas (ou impossíveis, devido às fortes restrições de segurança da linguagem. E essa lacuna, pelo menos tão cedo, não pode ser superada, a não ser com o uso de bibliotecas de message passing (como o MPI).

Java na Internet

Maiores informações sobre Java podem ser obtidas em:
http://www.javasoft.com/
http://www.java.sun.com/

Algumas informações sobre MPP podem ainda ser obtidas em:
http://www.javaworld.com/

6.5 - MPI NA INTERNET

Implementações MPI

Temos várias implementações de MPI free disponíveis para download

- Implementação desenvolvido pelo Laboratório Nacional de Argonne e Universidade Estadual de Mississipe
  ftp://info.mcs.anl.gov
- Implementação LAM desenvolvido no Centro de Supercomputação de Ohio
  ftp://ftp.osc.edu/pub/lam
- Implementação CHIMP desenvolvido no Centro de Computação Paralela de Edinburgh
  ftp://ftp.epcc.edu.ac.uk/pub/chimp/release
- Existem duas implementações de MPI que rodam em Windows.
  - Para Windows 3.1
    ftp://csftp.unomaha.edu/pub/rewini/WinMPI
  - Para Win32
    ftp://pandora.uc.pt/w32mpi

FAQ MPI

A lista de perguntas mais frequentes (FAQ) está disponínivel em:
http://www.erc.msstate.edu/mpi/mpi-faq.html

Páginas Web MPI

- Laboratório Nacional de Argonne
  http://www.mcs.anl.gov/mpi
- Universidade Estadual de Mississipi
  http://www.erc.msstate.edu/mpi
- Laboratório Nacional de Oak Ridge
  http://www.epm.ornl.gov/~walker/mpi

MPI Newsgroup

Usenet Newsgroup voltado para MPI
comp.parallel.mpi