Message Passing Interface

Gonzalo Travieso

2010

Sumário

1 Conceitos Básicos

O MPI (*Message Passing Inteface*) é um padrão de biblioteca de passagem de mensagens para sistemas paralelos. Foi desenvolvido procurando fornecer uma base comum de desenvolvimento de programas paralelos em plataformas distintas. Apresentaremos aqui uma descrição parcial e resumida do padrão. Mais detalhes podem ser encontrados em [1] e [2].

De acordo com a intenção do comite de padronização (o *MPI Forum* [3]) o padrão especifica apenas uma interface de programação e sua concretização em algumas linguagens (correntemente C, C++, Fortran 77 e Fortran 90); detalhes de implementação são deixados totalmente por conta do implementador, de forma a flexibilizar o sistema e possibilitar implementações eficientes. Programas C, Fortran 77 e C++ que desejam utilizar MPI devem incluir os arquivos `mpi.h`, `mpif.h` e `mpi++.h`, respectivamente; programas Fortran 90 devem usar (`USE`) o módulo `mpi`.[1]

As rotinas de MPI são construídas em torno dos conceitos de *processos*, *mensagens*, *comunicadores* e *tipos de dados*.

Inicialização e Finalização Em um programa MPI, antes da chamada a qualquer das rotinas MPI este deve ser inicializado, e antes do término do programa ele deve ser finalizado. Para isto são definidas as rotinas a seguir (será usada a interface C neste texto).

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

[1] O *header* `mpi++.h` define a interface C++ de MPI. Programas C++ podem também utilizar as rotinas da interface C, caso em que devem incluir `mpi.h`. Nesta apostila é apresentada apenas a interface C. Da mesma forma, programas Fortran 90 podem usar a interface Fortran 77.

Os parâmetros `argc` e `argv` passados na interface C devem ser os tradicionais parâmetros recebidos por `main`, e devem ser passados para `MPI_INIT` antes da sua utilização no programa.

O valor de retorno é um código de erro: se não houve erro, o valor é `MPI_SUCCESS`; caso haja um erro os valores serão diferentes, mas não são especificados pelo MPI.

A finalização é feita pela rotina:

```
int MPI_Finalize();
```

Processos O elemento básico de computação em MPI é chamado *processo*. Um processo é uma entidade de execução seqüencial, que pode ser comparada com um programa seqüencial em execução. A diferença consiste em que um processo MPI tem a possibilidade de cooperar com outros processos MPI para a execução de alguma tarefa. Um programa MPI consiste então na especificação do código a ser executado por um conjunto de processos que cooperam na solução de um problema.

Comunicadores Diferentes processos são interligados em MPI através de uma entidade chamada *comunicador* (definida como um objeto opaco, com todos os detalhes de implementação deixados a cargo do implementador). Sempre que precisar existir alguma interação entre processos distintos, eles devem estar associados a um mesmo comunicador. Todos os processos de uma mesma aplicação MPI são automaticamente associados pelo sistema a um comunicador denominado `MPI_COMM_WORLD`; caso necessário, outros comunicadores podem ser definidos pelo usuário.

Cada comunicador tem associado a ele um grupo de processos. Cada um desses processos têm uma numeração própria dentro do comunicador, denominada *rank*. Se um comunicador tem associado a ele P processos, esses processos serão numerados com *ranks* de 0 a $P - 1$.

Em uma mesma aplicação podem ser definidos diversos comunicadores, e desta forma cada processo pode fazer parte de mais do que um comunicador. Correspondentemente, um mesmo processo será identificado por *ranks* possivelmente distintos em comunicadores distintos; só faz sentido se falar do *rank* de um processo em um dado comunicador (ver seção 5).

Para encontrar o *rank* de um processo em um dado comunicador deve-se utilizar a rotina abaixo:

```
int MPI_Comm_rank(MPI_Comm com, int *rank);
```

O parâmetro `com` indica o comunicador, e `rank` devolverá o *rank* do processo nesse comunicador (em C, deve ser passado o endereço da variável onde desejamos que o *rank* seja colocado).

Se desejamos saber o número de processos em um comunicador usamos:

```
int MPI_Comm_size(MPI_Comm com, int *size);
```

No programa do exemplo abaixo, cada processo escreverá uma mensagem na tela, indicando seu *rank* e o número total de processos. O programa pressupõe que todos os processos têm acesso à saída padrão do sistema, o que não é garantido pelo padrão MPI, mas comumente implementado.

Exemplo

```
1  #include <stdio.h>
2  #include <mpi.h>
3
4  int main(int argc, char **argv)
5  {
6    int quantos, rank;
7    MPI_Init(&argc, &argv);
8    MPI_Comm_size(MPI_COMM_WORLD, &quantos);
9    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
10   printf("Processo %d de %d rodando.\n", rank, quantos);
11   MPI_Finalize();
12   return 0;
13 }
```

Mensagens Quando processos precisam coordenar suas operações, se faz necessária a troca de informações. Em MPI a troca de informações é realizada por meio de *mensagens*.

Uma mensagem pode ser enviada entre dois processos por meio de um comunicador em que os dois processos tomem parte. Como diversas mensagens podem estar circulando simultaneamente pelo sistema, inclusive pelo mesmo comunicador, além dos dados a serem transmitidos a mensagem deve conter informações de identificação.

Tabela 1: Lista de Tipos C e seus correspondentes identificadores MPI

Tipo C	Identificador MPI
char	MPI_CHAR
short	MPI_SHORT
int	MPI_INT
long	MPI_LONG
unsigned char	MPI_UNSIGNED_CHAR
unsigned short	MPI_UNSIGNED_SHORT
unsigned int	MPI_UNSIGNED_INT
unsigned long	MPI_UNSIGNED_LONG
float	MPI_FLOAT
double	MPI_DOUBLE
long double	MPI_LONG_DOUBLE

Essas informações, denominadas o *envelope* da mensagem são: processo remetente, processo destinatário, um inteiro denominado *tag*, e o comunicador utilizado.

O *tag* é utilizado para distinguir mensagens distintas provenientes de um mesmo remetente. Isto é importante em alguns algoritmos (veja exemplo 6).

Destinatário e *tag* especificados no remetente devem ser iguais a remetente e *tag* especificados no destinatário (a sintaxe será apresentada adiante, pag. 3). A semântica de MPI garante que mensagens do mesmo destinatário e com o mesmo *tag* serão recebidas *na mesma ordem em que foram enviadas.*

Tipos de Dados Os dados enviados em mensagens têm tipos associados (inteiro, números de ponto flutuante, etc.). Apesar de num sistema paralelo constituído de máquinas iguais os tipos não serem importantes, eles são importantes quando nós distintos do sistema paralelo podem ter arquiteturas distintas.

O mesmo tipo pode ser representado distintamente em arquiteturas diferentes. Cadeias de caracteres podem ser representadas em códigos com 8 bits por caracter (como ASCII) ou 16 bits por caracter (como Unicode); números de ponto flutuante podem utilizar representações diferentes, como IEEE ou do VAX; números inteiros podem ser representados com os bytes menos significativos antes dos mais significativos ou ao contrário. O formato de representação dos vários tipos de dados é dependente da arquitetura do sistema. Assim, se um sistema paralelo tem nós com arquiteturas distintas (o que não é incomum em redes de estações de trabalho), quando os dados são transmitidos de um processo em um nó com arquitetura A para um processo em um nó com arquitetura B, para que os dados sejam corretamente interpretados após a recepção é necessário que seja feita uma conversão de representação. Isto será feito *automaticamente* pelo MPI, utilizando informações de tipo fornecidas pelo usuário.

As conversões realizadas pelo MPI são apenas para lidar com diferenças de representação dos mesmos tipos de dados em arquiteturas distintas e não lidam com conversão entre tipos distintos (por exemplo, de inteiro para ponto flutuante); os tipos especificados pelo usuário em todos os parceiros da comunicação devem ser compatíveis, ou haverá erro na transmissão.

MPI possui vários tipos de dados pré-definidos (ver tabela 1), e possibilita também a definição de novos tipos pelo usuário (ver seção 4).

2 Comunicações Ponto-a-Ponto

O termo *comunicação ponto-a-ponto* é utilizado em MPI para indicar os tipos de comunicação que envolvem um par de processos. Essas comunicações são efetuadas por primitivas *send* (no remetente) e *receive* (no destinatário).

Para especificar uma comunicação devemos especificar os dados a serem comunicados e o envelope. Os dados são especificados indicando o *buffer* de transmissão ou recepção, o número de elementos a transmitir ou receber e o tipo dos elementos. O envelope é especificado indicando o comunicador a utilizar, o *rank* do parceiro da comunicação e o *tag* da comunicação.

A interface C para a transmissão padrão é indicada abaixo.

```
int MPI_Send(void *buffer, int cont, MPI_Datatype tipo, int dest, int tag, MPI_Comm com);
```

Os significados dos parâmetros são: `buffer` indica endereço do primeiro elemento a ser enviado ; `cont` indica o número de elementos a enviar; `tipo` indica o tipo dos dados (tabela 1); `com` indica o comunicador a utilizar para a transmissão; `dest` indica o *rank* do processo destinatário em `com` e `tag` indica o *tag* associado com esta comunicação.

A especificação da recepção é similar, mas envolve um parâmetro adicional para a verificação do *status* da comunicação.

```
int MPI_Recv(void *buffer, int cont, MPI_Datatype tipo, int rem, int tag, MPI_Comm com,
             MPI_Status *status);
```

Na recepção o parâmetro `cont` indica o número *máximo* de elementos que podem ser recebidos no *buffer* especificado. O número de elementos efetivamente recebidos pode ser menor do que esse. O parâmetro `rem` indica o *rank* do processo do qual a mensagem deve ser recebida.

O exemplo abaixo deve ser executado com dois processos. O processo de *rank* 1 prepara uma mensagem (seqüência de caracteres) e a envia ao processo 0, que recebe essa mensagem e a escreve na tela. Note como os dois parceiros especificam a comunicação, e como na recepção o tamanho especificado é o do *buffer* utilizado, e não o tamanho da mensagem efetivamente recebida (que é desconhecido de antemão). Veja o uso da variável `rank` nos dois processo: cada um dos processos tem sua própria cópia dessa variável, que terá valores distintos, por exemplo `rank` receberá o valor 0 em um dos processos e o valor 1 no outro. Note também como todos os processos executam o mesmo código, e testes baseados no valor do *rank* determinam que parte do código cada processo irá executar. Este é o esquema de programação conhecido como SPMD (*single program multiple data*).

Exemplo

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

#define MAX 256    /* Tamanho maximo da mensagem      */
#define TAG 1      /* Tag a utilizar nas comunicacoes */

int main(int argc, char **argv)
{
  int quantos, rank;
  char mensagem[MAX];
  MPI_Status status;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &quantos);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if (quantos != 2) {
    if (rank == 0)
      fprintf(stderr, "Este programa funciona apenas com 2 processos!\n");
  }
  else { /* Rodando com 2 processos */
    if (rank == 0) { /* Rodando no processo 0 */
      MPI_Recv(mensagem, MAX, MPI_CHAR, 1, TAG, MPI_COMM_WORLD, &status);
      printf("Recebi a mensagem: %s\n",mensagem);
    }
    else { /* Rodando no processo 1 */
      snprintf(mensagem, MAX, "Um ola de seu amigo, processo %d", rank);
      MPI_Send(mensagem, strlen(mensagem)+1, MPI_CHAR,
               0, TAG, MPI_COMM_WORLD);
    }
  }
  MPI_Finalize();
  return 0;
}
```

Normalmente especificam-se na recepção os valores do remetente e do *tag* explicitamente, o que resulta em que apenas mensagens com o valor de *tag* especificado e provenientes do remetente indicado serão recebidas. Se outras mensagens que não concordam com essa especificação estiverem disponíveis elas serão ignoradas e armazenadas para uma leitura posterior que possivelmente combine com seus parâmetros. Esta regra pode ser abrandada pelo uso de especificações de *tag* ou remetente genéricos, utilizando as constantes `MPI_ANY_TAG` e `MPI_ANY_SOURCE`, tanto individualmente como em conjunto. Por exemplo, se no lugar do rementente for especificado `MPI_ANY_SOURCE`, então uma mensagem com o *tag* especificado proveniente de *qualquer* processo

do comunicador será aceita para recepção. Se desejarmos posteriormente saber qual o remetente da mensagem (ou qual o *tag* utilizado, caso haja sido especificado `MPI_ANY_TAG`) podemos utilizar o valor retornado na variável passada ao parâmetro `status`, como descrito a seguir.

O `status` retorna informações sobre a comunicação efetuada, incluindo o *rank* do processo de origem, o *tag* utilizado na comunicação e o número de dados efetivamente recebidos. O modo de acesso dessas informações depende da linguagem utilizada: em C o `status` é uma `struct`, que possui campos denominados `MPI_SOURCE` e `MPI_TAG`, esses valores podem ser, portanto, acessados como `status.MPI_SOURCE` e `status.MPI_TAG`; em Fortran, `status` é um vetor, e `MPI_SOURCE` e `MPI_TAG` representam índices nesse vetor, sendo que as informações podem então ser acessadas através de `status(MPI_SOURCE)` e `status(MPI_TAG)`.

Para acessar o número de elementos recebidos, é necessário realizar uma chamada à rotina:

```
int MPI_Get_count(MPI_Status *status, MPI_Datatype tipo, int *cont);
```

Nesta chamada `status` é a variável de *status* retornada pela rotina de recepção, `tipo` é o tipo de dados usado na recepção, e `cont` é a variável onde será colocado o número de elementos efetivamente lidos (passar endereço em C).

No exemplo abaixo, generalizamos o programa do exemplo anterior para funcionar com um número variável de processos. O processo de *rank* 0 fica aguardando uma mensagem de cada um dos outros processos, e a imprime na tela. Como a ordem na qual os diversos processos serão executados não é garantida, o processo 0 aguarda mensagem de qualquer outro processo, utilizando `MPI_ANY_SOURCE` na recepção. Também utiliza `MPI_ANY_TAG`, pois os *tags* utilizados no envio são variantes de acordo com o *rank* do processo que está enviando. Ao imprimir a mensagem, o processo 0 informa de quem ela foi recebida, o *tag* utilizado na sua recepção e o número de caracteres da mensagem.

Exemplo

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

#define MAX 256    /* Tamanho maximo da mensagem      */

int main(int argc, char **argv)
{
  int quantos, rank;
  char mensagem[MAX];
  MPI_Status status;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &quantos);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if (rank == 0) { /* Rodando no processo 0 */
    int i, tamanho;
    for (i = 1; i < quantos; i++) {
      MPI_Recv(mensagem, MAX, MPI_CHAR, MPI_ANY_SOURCE,
               MPI_ANY_TAG, MPI_COMM_WORLD, &status);
      MPI_Get_count(&status, MPI_CHAR, &tamanho);
      printf("Mensagem recebida de %2d, com tag %3d, tamanho %2d:\n",
             status.MPI_SOURCE, status.MPI_TAG, tamanho);
      printf("   >>>>> %-40s <<<<<\n\n",mensagem);
    }
  }
  else { /* Rodando em processo diferente de 0*/
    char *enfase = (rank%2 == 0) ? "grande" : "bom";
    snprintf(mensagem, MAX, "Um ola de seu %s amigo, processo %d",
            enfase, rank);
    MPI_Send(mensagem, strlen(mensagem)+1, MPI_CHAR, 0,
             rank+100, MPI_COMM_WORLD);
  }
  MPI_Finalize();
  return 0;
}
```

Uma operação de comunicação ponto a ponto bastante útil é uma combinação de transmissão e recepção. Ela permite que um processo envie dados para um segundo processo e simultaneamente receba dados de um terceiro processo. A interface é:

```
int MPI_Sendrecv(void *sbuffer, int scont, MPI_Datatype stipo, int dest, int stag,
                 void *rbuffer, int rcont, MPI_Datatype rtipo, int rem, int rtag,
                 MPI_Comm com, MPI_Status *status);
```

onde `sbuffer`, `scont`, `stipo`, `dest` e `stag` são as informações sobre a transmissão, enquanto `rbuffer`, `rcont`, `rtipo`, `rem`, `rtag` e `status` são informações sobre a recepção. A grande vantagem no uso dessa operação é que, como transmissão e recepção ocorrem como se fossem executadas simultaneamente, não é necessário se preocupar com a ordem necessária para evitar *deadlock* quando temos um conjunto de processos trocando dados. O exemplo abaixo usa `MPI_Sendrecv` para processos de *rank* i trocarem dados com processos de *ranks* $i-1$ e $i+1$. Esse exemplo também demostra o uso de `MPI_PROC_NULL` que é um *rank* de processo para um processo inexistente. Quando uma operação de comunicação recebe `MPI_PROC_NULL` como o *rank* do parceiro, a comunicação simplesmente não é realizada. Isso ajuda a facilitar o desenvolvimento do código para processos nas bordas.

Exemplo

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

int main(int argc, char **argv)
{
  int quantos, rank;
  int da_esquerda = -1, da_direita = -1, meu;
  int esquerda, direita;

  MPI_Status status;

  MPI_Init(&argc, &argv);

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

  esquerda = rank - 1;
  if (esquerda < 0)
    esquerda = MPI_PROC_NULL;

  direita = rank + 1;
  if (direita == quantos)
    direita = MPI_PROC_NULL;

  meu = rank * rank;

  MPI_Sendrecv(&meu, 1, MPI_INT, direita, 1,
               &da_esquerda, 1, MPI_INT, esquerda, 1, MPI_COMM_WORLD, &status);
  MPI_Sendrecv(&meu, 1, MPI_INT, esquerda, 2,
               &da_direita, 1, MPI_INT, direita, 2, MPI_COMM_WORLD, &status);

  printf("Rank: %d, da esquerda: %d, meu: %d, da direita %d\n", rank, da_esquerda, meu, da_direita);

  MPI_Finalize();
  return 0;
}
```

Além dessas rotinas existem diversas outras relacionadas a comunicação ponto-a-ponto, que não serão tratadas aqui. Elas definem, entre outros, variantes para lidar com os seguintes tipos de comunicação:

Comunicação bufferizada onde o usuário explicita o *buffer* a ser utilizado para armazenar os dados da comunicação, podendo desta forma evitar o estouro de *buffers* definidos pelo sistema.

Comunicação síncrona onde a comunicação somente ocorre quando tanto transmissor como receptor estiverem prontos.

Recepção pronta onde é garantido *pelo programador* que a recepção já estará pronta quando a transmissão for efetuada. Isto possibilita algumas otimizações, como a escrita direta no *buffer* de recepção dos dados recebidos.

Comunicação não-bloqueante pode ser associada com qualquer dos modos acima, e é utilizada quando desejamos iniciar uma comunicação mas prosseguir com o processamento enquanto a comunicação ocorre.

O modo de funcionamento das rotinas `MPI_Send` e `MPI_Recv` vistas acima, chamado *modo padrão*, não é definido pelo MPI, podendo ser tanto bufferizado como síncrono, por exemplo. Em geral, as implementações realizam um modo que consiste em utilizar um *buffer* reservado pelo sistema; se os dados a transmitir cabem nesse *buffer*, então a transmissão funciona como bufferizada, mas se a quantidade de dados exceder o *buffer*, a transmissão funciona como síncrona. Desta forma, um programa MPI que usa as rotinas `MPI_Send` e `MPI_Recv` deve estar preparado para funcionar tanto no caso em que o envio seja bloqueante, aguardando a recepção, como no caso em que a mensagem a ser enviada seja bufferizada pelo sistema.

Para maiores detalhes sobre os modos de transmissão e sobre a semântica das rotinas veja [1].

3 Comunicações Coletivas

Comunicações coletivas são aquelas que envolvem um grupo de processos, ao invés de apenas um par de processos. Em MPI, o grupo envolvido na comunicação é especificado através de um comunicador: todos os processos associados ao comunicador utilizado participarão da comunicação. As operações coletivas em MPI são as seguintes: barreira; *broadcast*; coleta; distribuição; redistribuição; redução; prefixo. Em todas essas operações coletivas, *todos* os processos do comunicador utilizado para especificar o grupo devem realizar uma chamada para a rotina da operação. Passaremos a apresentar as diversas variantes dessas operações.

Barreira Uma *barreira* é utilizada quando desejamos sincronizar a operação de todos os processos de um grupo. Os processos somente passam pela barreira quando todos os processos do grupo a tiverem atingido. A sintaxe é:

```
int MPI_Barrier(MPI_Comm com);
```

Note que a barreira não implica que os diversos processos executarão as operações seguintes à barreira no mesmo instante. Ela indica apenas que essas operações seguintes somente serão efetuadas após todos os processos terem chegado na barreira. Esta operação é raramente necessária em programas MPI.

Broadcast Um *broadcast* é uma operação em que um dado presente em um dos processos é transmitido para todos os outros processos do grupo. Nesta operação, o processo que inicialmente possui o dado a ser transmitido é denominado *raiz* (*root*) do *broadcast*. A sintaxe em MPI é:

```
int MPI_Bcast(void *buffer, int cont, MPI_Datatype tipo, int raiz, MPI_Comm com);
```

Os parâmetros `buffer`, `cont` e `tipo` especificam o dado a ser transmitido (como nas rotinas já apresentadas); `com` especifica o comunicador associado ao grupo de processos que estarão envolvidos na comunicação; `raiz` especifica o *rank* do processo raiz.

O exemplo a seguir mostra como um *broadcast* pode ser utilizado para distribuir um valor calculado (no caso lido de um arquivo) de um dos processos para todos os outros de uma aplicação.

Exemplo

```
1  /* Criar um arquivo entrada.dat com um numero inteiro
2   * antes de rodar este programa
3   */
4
5  #include <stdio.h>
6  #include <mpi.h>
7
8  int main(int argc, char **argv)
9  {
10   int rank, dado = -1;
11   MPI_Init(&argc, &argv);
12   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
13
14   if (rank == 0) { /* Rodando no processo 0 */
15     FILE *entrada;
16     entrada = fopen("entrada.dat","r");
17     if (entrada != NULL) {
18       fscanf(entrada," %d ", &dado);
19       fclose(entrada);
```

```
20      }
21    }
22
23    printf("Processo %d meu dado por enquanto e %d\n", rank, dado);
24    MPI_Bcast(&dado, 1, MPI_INT, 0, MPI_COMM_WORLD);
25
26    printf("Processo %d calculando com dado %d\n", rank, dado);
27    /* etc... */
28    MPI_Finalize();
29    return 0;
30  }
```

Coleta Uma operação de coleta recolhe dados de diversos processos e os armazena em um vetor em um processo específico, chamado *raiz* da coleta. O processo de *rank* i terá seu dado armazenado na posição i do vetor; o processo raiz também contribuirá com um elemento para o vetor que ele irá coletar.

Existem diversas variantes de coleta. A primeira coleta o mesmo número de elementos de cada processo:

```
int MPI_Gather(void *sbuffer, int scont, MPI_Datatype stipo,
               void *rbuffer, int rcont, MPI_Datatype rtipo,
               int raiz, MPI_Comm com);
```

Os parâmetros `sbuffer`, `scont` e `stipo` especificam os dados que serão enviados por cada processo; `rbuffer`, `rcont` e `rtipo` especificam os dados recebidos pelo processo raiz, e somente têm validade no próprio processo raiz; `com` especifica os processos envolvidos e `raiz` o *rank* do processo raiz em `com`. É importante notar que `rbuffer` é o *buffer* onde todos os dados serão armazenados no processo raiz, mas `rcont` indica o número de elemento que serão recebidos de *cada um* dos processos. Portanto, `rbuffer` deve ter capacidade de armazenar um número de elementos igual a `rcont` vezes o número de processos em `com`.

O exemplo abaixo coleta dois valores de ponto flutuante de cada processo e armazena todos os valores coletados num arquivo.

Exemplo

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <math.h>
4  #include <mpi.h>
5
6  int main(int argc, char **argv)
7  {
8    int quantos, rank;
9    float *dados;
10   float originais[2];
11
12   MPI_Init(&argc, &argv);
13   MPI_Comm_size(MPI_COMM_WORLD, &quantos);
14   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
15
16   if (rank == 0) { /* Rodando no processo 0 */
17     dados = (float *)malloc(2 * quantos * sizeof(float));
18   }
19
20   /* Cálculos em todos os processos */
21   originais[0] = pow(2,2*rank);
22   originais[1] = pow(2,2*rank+1);
23
24   MPI_Gather(originais, 2, MPI_FLOAT,
25              dados, 2, MPI_FLOAT,
26              0, MPI_COMM_WORLD);
27
28   if (rank == 0) { /* Rodando no processo 0 */
29     FILE *saida = fopen("resultados.dat", "w");
30     int i;
31     for (i = 0; i < 2*quantos; i++)
32       fprintf(saida, " %f ", dados[i]);
33     fprintf(saida, "\n");
```

```
34  }
35
36  MPI_Finalize();
37  return 0;
38 }
```

Uma outra variante permite a recepção de uma quantidade distinta de dados de cada um dos processos, e também permite armazenar esses dados em uma ordem arbitrária no *buffer* de recepção.

```
int MPI_Gatherv(void *sbuffer, int scont, MPI_Datatype stipo,
                void *rbuffer, int *rconts, int *rdesls, MPI_Datatype rtipo,
                int raiz, MPI_Comm com);
```

A diferença aqui em relação à rotina anterior é que especificamos no processo raiz dois vetores, ao invés de apenas o número de elementos a coletar de cada processo. Esses vetores devem ter um elemento para cada processo no grupo de `com`. A entrada i do vetor `rconts` indica o número de elementos a receber do processo de *rank* i, enquanto que a entrada i do vetor `rdesls` indica o índice do vetor `rbuffer` onde o primeiro elemento recebido do *rank* i será armazenado (os outros elementos serão armazenados seqüencialmente).

Outras variantes de coleta são aquelas em que ao invés da coleta ser realizada apenas por um processo raiz, ela é realizada por todos os processos. Semanticamente, a operação é correspondente a uma coleta seguida de um *broadcast* dos dados coletados. As interfaces são similares às das rotinas anteriores, mas sem a especificação de um processo raiz, que não existe nessas operações:

```
int MPI_Allgather(void *sbuffer, int scont, MPI_Datatype stipo,
                  void *rbuffer, int rcont, MPI_Datatype rtipo,
                  MPI_Comm com);
```

```
int MPI_Allgatherv(void *sbuffer, int scont, MPI_Datatype stipo,
                   void *rbuffer, int *rconts, int *rdesls, MPI_Datatype rtipo,
                   MPI_Comm com);
```

Distribuição A distribuição é de certa forma a operação inversa da coleta: o processo raiz possui um vetor de elementos, e distribui um dos elementos desse vetor para cada um dos processos no grupo, de acordo com o *rank*.

A variante mais simples de distribuição é:

```
int MPI_Scatter(void *sbuffer, int scont, MPI_Datatype stipo,
                void *rbuffer, int rcont, MPI_Datatype rtipo,
                int raiz, MPI_Comm com);
```

Os parâmetros são similares aos de `MPI_Gather`, mas a operação é oposta. Os valores a enviar são especificados por `sbuffer`, `scont` e `stipo`, e são válidos apenas no processo raiz; `scont` especifica o número de elementos a enviar para cada um dos processos do grupo. Os parâmetros de recepção `rbuffer`, `rcont` e `rtipo` devem ser válidos em todos os processos (incluindo o raiz, que receberá aí uma parte do *buffer* original).

Outra variante permite maior flexibilidade no número de elementos distribuídos para cada processo e na ordem em que eles se encontram no *buffer* de transmissão:

```
int MPI_Scatterv(void *sbuffer, int *sconts, int *sdesls, MPI_Datatype stipo,
                 void *rbuffer, int rcont, MPI_Datatype rtipo,
                 int raiz, MPI_Comm com);
```

A interpretação dos parâmetros é similar à de `MPI_Gatherv`.

Veja exemplo de uso na página 10.

Distribuição e Coleta Uma operação adicional combina as operações de distribuição e de coleta. Cada um dos processos de um grupo possui um vetor que ele irá distribuir de acordo com o *rank* para todos os processo do grupo, e cada processo coletará os dados recebidos de todos os outros processos de acordo com o *rank* em um vetor. Isto é, os dados presentes na posição j do *buffer* de envio do processo i serão transmitidos para o processo j e armazenados na posição i de seu *buffer* de recepção.

As interfaces são, para envio e recepção da mesma quantidade de dados de cada processo:

```
int MPI_Alltoall(void *sbuffer, int scont, MPI_Datatype stipo,
                 void *rbuffer, int rcont, MPI_Datatype rtipo,
                 MPI_Comm com);
```

Tabela 2: Operadores de redução pré-definidos em MPI

Nome	Significado
MPI_MAX	máximo
MPI_MIN	mínimo
MPI_SUM	soma
MPI_PROD	produto
MPI_LAND	e lógico
MPI_BAND	e bit-a-bit
MPI_LOR	ou lógico
MPI_BOR	ou bit-a-bit
MPI_LXOR	ou-exclusivo lógico
MPI_BXOR	ou-exclusivo bit-a-bit
MPI_MAXLOC	máximo e localização
MPI_MINLOC	mínimo e localização

A interpretação dos parâmetros é similar à de `MPI_Gather` e `MPI_Scatter`.

Para envio e recepção mais flexíveis:

```
int MPI_Alltoallv(void *sbuffer, int *sconts, int *sdesls, MPI_Datatype stipo,
                  void *rbuffer, int *rconts, int *rdesls, MPI_Datatype rtipo,
                  MPI_Comm com);
```

A interpretação dos parâmetros é similar à de `MPI_Gatherv` e `MPI_Scatterv`.

Redução Operações de redução são aquelas onde cada um dos processos fornece um valor para o cálculo de alguma forma de "total global", como uma somatória ou um produtório, ou a busca de um mínimo. As duas variantes mais usadas de redução são a redução simples, onde o valor total calculado é colocado em um processo raiz especificado e aquela onde todos os processos envolvidos na redução recebem o total.

```
int MPI_Reduce(void *sbuffer, void *rbuffer, int cont, MPI_Datatype tipo,
               MPI_Op op, int raiz, MPI_Comm com);
```

```
int MPI_Allreduce(void *sbuffer, void *rbuffer, int cont, MPI_Datatype tipo,
                  MPI_Op op, MPI_Comm com);
```

Nestas rotinas `sbuffer` indica os valores que serão fornecidos pelo processo para o total a ser calculado, `rbuffer` é o local onde o total será armazenado (somente no processo raiz, caso se trate de `MPI_Reduce`). Os outros parâmetros têm especificação similar à encontrada em outras rotinas, com exceção de `op`, que deve especificar qual a operação a ser executada nos valores para efetuar a redução. Há aqui duas possibilidades: ou usar operadores de redução pré-definidos pelo MPI (o caso mais comum), de acordo com a tabela 2, ou definir um novo operador. Aqui não trataremos da definição de novos operadores de redução, veja discussão em [1, cap. 4].

Os operadores `MPI_MAXLOC` e `MPI_MINLOC`, na tabela 2 diferem de `MPI_MAX` e `MPI_MIN` respectivamente por apresentarem além do valor (máximo ou mínimo, respectivamente) também uma indicação do *rank* do processo em que esse máximo ou mínimo ocorreu. Detalhes sobre seu uso não serão discutidos aqui (veja [1]).

O programa do exemplo abaixo lê dados de um arquivo (número de dados e nome do arquivo especificados na linha de comando), distribui os dados pelos processos disponíveis usando `MPI_Scatterv` (pois o número de dados pode não ser múltiplo exato do número de processos), soma localmente em cada processo todos os valores recebidos e em seguida realiza uma redução para calcular a soma total, que é mostrada na tela.

Exemplo

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <math.h>
4 #include <mpi.h>
5
6 int main(int argc, char **argv)
7 {
8   int quantos, rank, N, n, i, q, r;
```

```
9  float *originais, *dados;
10 float somalocal, somaglobal;
11 int *conts, *desls, desloc;
12
13 MPI_Init(&argc, &argv);
14
15 MPI_Comm_size(MPI_COMM_WORLD, &quantos);
16 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
17
18 /* Verificacao de parametros para o programa */
19 if (argc != 3) { /* Nao foram especificados os parametros */
20   if (rank == 0) /* 0 envia mensagem */
21     fprintf(stderr, "Passe nome de arquivo e "
22             " numero de elementos como parametro!\n");
23   MPI_Finalize(); /* Todos terminam */
24   exit(1);
25 }
26
27 /* Parametro 2 e o numero de elementos */
28 sscanf(argv[2], "%d", &N);
29
30 /* Leitura dos dados (processo 0) */
31 if (rank == 0) {
32   FILE *arqDados;
33   originais = (float *)calloc(N,sizeof(float));
34   /* Parametro 1 e o nome do arquivo */
35   arqDados = fopen(argv[1],"r");
36   if (arqDados != NULL) {
37     for (i = 0; i < N; i++)
38       fscanf(arqDados," %f ", &originais[i]);
39     fclose(arqDados);
40   }
41 }
42
43 /* Calculo do numero de elementos por processo */
44 /* Calculo realizado por todos os processos    */
45 conts = (int *)malloc(quantos * sizeof(int));
46 desls = (int *)malloc(quantos * sizeof(int));
47
48 n = ceil( (float)N / quantos);
49 dados = (float *)malloc(n * sizeof(float));
50
51 q = N / quantos; r = N % quantos;
52 desloc = 0;
53 for (i = 0; i < quantos; i++) {
54   if (i < r)
55     conts[i] = q+1;
56   else
57     conts[i] = q;
58   desls[i] = desloc;
59   desloc += conts[i];
60 }
61
62 /* Distribuicao dos elementos para os processos */
63 MPI_Scatterv(originais, conts, desls, MPI_FLOAT,
64              dados, n, MPI_FLOAT, 0, MPI_COMM_WORLD);
65
66 /* Somatoria local dos elementos recebidos */
67 somalocal = 0;
68 for (i = 0; i < conts[rank]; i++)
69   somalocal += dados[i];
70
71 /* Reducao final para encontrar somatoria global */
72 MPI_Reduce(&somalocal, &somaglobal, 1, MPI_FLOAT,
73            MPI_SUM, 0, MPI_COMM_WORLD);
```

```
74
75    printf("Soma local no processo %d = %f\n", rank, somalocal);
76    /* Impressao do resultado (presente no processo 0) */
77    if (rank == 0)
78      printf("Soma total: %f\n", somaglobal);
79
80    MPI_Finalize();
81    return 0;
82  }
```

Prefixo Operações de prefixo são semelhantes às de redução, mas cada um dos processos recebe o resultado da redução parcial de todos os processos com valores de *rank* menor ou igual ao dele. A interface é muito parecida com a da redução.

```
int MPI_Scan(void *sbuffer, void *rbuffer, int cont, MPI_Datatype tipo,
             MPI_Op op, MPI_Comm com);
```

Os *buffers* de transmissão e recepção devem ser válidos em todos os processos, pois todos eles irão fornecer algum dado para a redução e recolher um valor de redução (parcial).

4 Tipos Definidos Pelo Usuário

Além dos tipos de dados pré-definidos (tabela 1), é possível definir também tipos especiais, bastante úteis para simplificar o código quando certos conjuntos de dados forem transmitidos sempre simultaneamente. Alguns dos construtores de tipos serão discutidos a seguir (veja os demais em [1]).

Dados Contíguos Quando diversos dados contíguos na memória formam um bloco único (por exemplo toda uma linha de uma matriz em C, ou toda um coluna em Fortran), podemos definir um tipo especial para esse bloco.

```
int MPI_Type_contiguous(int cont, MPI_Datatype tipoant, MPI_Datatype *tiponovo);
```

O número de elementos contíguos é especificado por `cont`; `tipoant` especifica o tipo de cada um desses elementos contíguos e `tiponovo` retornará com o novo tipo de dados.

Vetores Muitas vezes os dados que devem ser transmitidos não estão alocados contíguamente na memória, mas apresentam um espaçamento regular. Por exemplo, se tentarmos pegar todos os elementos de uma linha de uma matriz em Fortran, ou todos os elementos de uma coluna de uma matriz estática em C, cada elemento terá um espaçamento fixo do anterior, determinado pelo número de elementos em uma coluna (para Fortran) ou em uma linha (para C). Esses casos podem ser tratados com a rotina seguinte.

```
int MPI_Type_vector(int cont, int compr, int stride, MPI_Datatype tipoant, MPI_Datatype *tiponovo);
```

Para tornar a rotina mais geral, ela permite que blocos contíguos de elementos sejam agrupados, com os blocos então regularmente separados entre si. O parâmetro `compr` indica o número de elementos contíguos em cada bloco, enquanto que `stride` indica o número de elementos entre o *início* de um bloco e o *início* do bloco seguinte; `cont` indica o número de blocos.

Note que a chamada:

```
MPI_Type_contiguous(c, tipo1, &tipo2);
```

é equivalente a:

```
MPI_Type_vector(c, 1, 1, tipo1, &tipo2);
```

ou então a:

```
MPI_Type_vector(1, c, c, tipo1, &tipo2)
```

e portanto `MPI_Type_contiguous` é um caso especial de `MPI_Type_vector`.

Blocos Gerais Os blocos formados por `MPI_Type_vector` são de um formato bem específico, pois todos os blocos têm o mesmo tamanho e todos são regularmente espaçados entre si. A rotina `MPI_Type_indexed` permite a generalização desses dois fatores.

```
int MPI_Type_indexed(int cont, int *comps, int *desls, MPI_Datatype tipoant,
                     MPI_Datatype *tiponovo);
```

O tipo gerado será constituído por `cont` blocos, sendo que cada elemento do vetor `comps` diz o número de elementos em um bloco, e cada elemento correspondente em `desls` diz o início do bloco em relação ao início geral do tipo (e não em relação ao último bloco, desta forma é possível gerar um novo tipo que tem seus elementos em outra ordem em relação aos elementos originais na memória).

Uma chamada para `MPI_Type_vector` pode ser emulada por uma chamada para `MPI_Type_indexed`, desde que todos os elementos de `comps` tenham o mesmo valor, e que os elementos de `desls` sejam em ordem crescente e com o espaçamento adequado e portanto `MPI_Type_vector` é um caso especial de `MPI_Type_indexed`.

Diferentes Tipos A rotina `MPI_Type_indexed` ainda tem uma limitação que faz com que ela não seja geral: todos os elementos incluídos no novo tipo devem ser de um mesmo tipo original. A rotina `MPI_Type_struct` elimina essa limitação. O novo tipo será formado por blocos de elementos de tipos possivelmente distintos, especificados por três vetores: um dizendo o tamanho do bloco (número de elemento), outro dizendo o deslocamento desse bloco em relação ao começo do tipo, e outro dizendo o tipo de dados original nesse bloco.

```
int MPI_Type_struct(int cont, int *comps, MPI_Aint *desls, MPI_Datatype *tipos,
                    MPI_Datatype *tiponovo);
```

O parâmetro `cont` indica o número de blocos; `comps` o número de elementos em cada um dos blocos; e `tipos` o tipo de dados em cada bloco. O tipo `MPI_Aint` é utilizando em C sempre que desejamos guardar endereços ou deslocamentos em termos de endereços (e não de número de elementos) o que é o caso aqui, pois podemos ter elementos de tipos distinto, e não faz sentido então falar de distância em termos de número de elementos.

Para calcular endereços, podemos utilizar a rotina:

```
int MPI_Address(void *local, MPI_Aint *endereco);
```

que, dado um vetor ou endereço de uma variável (`local`) fornece um `MPI_Aint` (em C) ou `INTEGER` (em Fortran) que corresponde ao endereço dessa variável, que pode então ser utilizado para os cálculos de deslocamento.

Após definir um novo tipo de dados, e antes que ele possa ser utilizado em operações de comunicação, é necessário que ele seja "registrado" no sistema:

```
int MPI_Type_commit(MPI_Datatype *tipo);
```

Como primeiro exemplo vejamos como definir tipos para linhas e colunas de uma matriz estática. Este exemplo demonstra também um outro ponto importante de tipos em MPI: o tipo usado na transmissão não precisa ser exatamente o mesmo utilizado na recepção, basta que a quantidade de elementos do tipo básico transmitidos e recebidos seja a mesma. Este programa se utiliza desse fato para implementar um algoritmo (ineficiente) de transposição de matrizes: uma matriz é enviada linha por linha utilizando um tipo definido para as linhas e em seguida recebida coluna por coluna, utilizando um tipo definido para as colunas. Outro ponto demonstrado nesse programa é que nada impede que um processo envie dados para ele mesmo.

Exemplo

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <mpi.h>
4
5  #define N 10
6  #define TAG 7
7
8  int main(int argc, char **argv)
9  {
10   int quantos, i, j;
11   double matorig[N][N], matnova[N][N];
12   MPI_Datatype tLinha, tColuna;
13   MPI_Status status;
14
15   MPI_Init(&argc, &argv);
16
17   MPI_Comm_size(MPI_COMM_WORLD, &quantos);
```

```
18  if (quantos != 1) {
19    int rank;
20    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
21    if (rank == 0)
22      fprintf(stderr,"Rodar com um processo apenas.\n");
23    MPI_Finalize();
24    exit(1);
25  }
26
27  /* Inicializa matriz original */
28  for (i = 0; i < N; i++)
29    for (j = 0; j < N; j++)
30      matorig[i][j] = i-j;
31
32  /* Cria um tipo para as linhas */
33  MPI_Type_contiguous(N, MPI_DOUBLE, &tLinha);
34  MPI_Type_commit(&tLinha);
35
36  /* Cria um tipo para as colunas */
37  MPI_Type_vector(N, 1, N, MPI_DOUBLE, &tColuna);
38  MPI_Type_commit(&tColuna);
39
40  /* Transmite linha por linha,
41   * recebe coluna por coluna
42   */
43  for (j = 0; j < N; j++) {
44    MPI_Sendrecv(&matorig[j][0], 1, tLinha,  0, TAG,
45                 &matnova[0][j], 1, tColuna, 0, TAG, MPI_COMM_WORLD, &status);
46  }
47
48  printf("Matriz original:\n");
49  for (i = 0; i < N; i++) {
50    for (j = 0; j < N; j++)
51      printf("%5.1f", matorig[i][j]);
52    printf("\n");
53  }
54
55  printf("\nMatriz transposta:\n");
56  for (i = 0; i < N; i++) {
57    for (j = 0; j < N; j++)
58      printf("%5.1f", matnova[i][j]);
59    printf("\n");
60  }
61
62  MPI_Finalize();
63
64  return 0;
65 }
```

O exemplo abaixo usa `MPI_Type_struct` para definir um tipo utilizado para a transmissão de dados de uma estrutura.

Exemplo

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <string.h>
4 #include <mpi.h>
5
6 #define MAX 64
7
8 struct Pessoa {
9   long id;
10  char nome[MAX];
11 };
12
```

```
13 int main(int argc, char **argv)
14 {
15   int quantos, rank;
16
17   MPI_Datatype tipoPessoa[2] = {MPI_LONG, MPI_CHAR};
18   int compPessoa[2]          = {1,        MAX      };
19   MPI_Aint deslPessoa[2];
20   MPI_Datatype tPessoa;
21
22   struct Pessoa *pessoas, umapessoa;
23
24   MPI_Init(&argc, &argv);
25
26   MPI_Comm_size(MPI_COMM_WORLD, &quantos);
27   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
28
29   /* Inicializacao dos dados */
30   if (rank == 0) {
31     int i;
32     pessoas = (struct Pessoa *)malloc(quantos * sizeof(struct Pessoa));
33     printf("Digite dados de %d pessoas:\n",quantos);
34     for (i = 0; i < quantos; i++) {
35       printf("Nome:       ");
36       fgets(pessoas[i].nome,MAX,stdin);
37       pessoas[i].nome[strlen(pessoas[i].nome)-1] = '\0';
38       printf("Identidade: ");
39       scanf("%ld", &pessoas[i].id);
40       fgetc(stdin);
41     }
42   }
43
44   /* Construcao de um tipo para a transmissao */
45   MPI_Address(&umapessoa,     &deslPessoa[0]);
46   MPI_Address(umapessoa.nome, &deslPessoa[1]);
47   deslPessoa[1] -= deslPessoa[0];
48   deslPessoa[0] = 0;
49
50   MPI_Type_struct(2, compPessoa, deslPessoa, tipoPessoa,
51                   &tPessoa);
52
53   MPI_Type_commit(&tPessoa);
54
55   /* Distribuicao dos dados para os processos */
56   MPI_Scatter(pessoas,    1, tPessoa,
57               &umapessoa, 1, tPessoa, 0, MPI_COMM_WORLD);
58
59   printf("Processo %d recebeu: %-30s %10ld\n",
60          rank, umapessoa.nome, umapessoa.id);
61
62   MPI_Finalize();
63   return 0;
64 }
```

5 Grupos e Comunicadores

Para muitos problemas o uso do comunicador pré-definido `MPI_COMM_WORLD` é suficiente. Em alguns casos, no entanto, isso não ocorre, especialmente quando desejamos realizar comunicações coletivas usando apenas uma parte dos processos disponíveis para a aplicação ou quando desejamos implementar rotinas paralelas, que serão úteis em diversos programas. No primeiro caso, devemos ter novos comunicadores pois em todas as comunicações coletivas *todos* os processos do comunicador devem participar. No segundo caso, o uso de comunicadores especiais para as rotinas paralelas facilita o seu desenvolvimento (pois não há perigo de confusão entre mensagens dessas rotinas e mensagens de outros trechos do programa) e facilita a utilização modular das rotinas, pois elas podem ser consideradas como operando em espaços de comunicação distintos.

A criação de novos comunicadores é realizada com a ajuda de outros objetos chamados *grupos*, que têm informações sobre os processos pertencentes a um comunicador.

Para descobrir qual o grupo associado a um comunicador utilizamos:

```
int MPI_Comm_group(MPI_Comm com, MPI_Group *grupo);
```

Para manipular grupos existem diversas rotinas.

```
int MPI_Group_union(MPI_Group grupo1, MPI_Group grupo2, MPI_Group *novogrupo);
```

que gera um grupo que contém todos os processos da união dos dois grupos fornecidos.

```
int MPI_Group_intersection(MPI_Group grupo1, MPI_Group grupo2, MPI_Group *novogrupo);
```

que gera um grupo com a intersecção dos dois grupos fornecidos.

```
int MPI_Group_difference(MPI_Group grupo1, MPI_Group grupo2, MPI_Group *novogrupo);
```

que gera um novo grupo com todos os elementos de `grupo1` que não estão em `grupo2`.

Podemos também indicar explicitamente quais os processos de um grupo que devem ser incluídos no ou excluídos do novo grupo.

```
int MPI_Group_incl(MPI_Group grupoant, int n, int *ranks, MPI_Group *novogrupo);
```

Aqui `n` indica o número de processos do grupo `grupoant` que serão incluídos em `novogrupo`, e `ranks` é um vetor com os *ranks* dos processos a incluir.

```
int MPI_Group_excl(MPI_Group grupoant, int n, int *ranks, MPI_Group *novogrupo);
```

Neste caso, são incluídos todos os processos de `grupoant` em `novogrupo`, menos aqueles especificados por `n` e `ranks`.

Como os grupos são frequentemente criados manipulando grupos anteriormente existentes, a constante `MPI_GROUP_EMPTY` é definida como um grupo vazio (sem nenhum processo).

Uma vez construído um grupo com os processos que farão parte do novo comunicador, podemos construir o comunicador correspondente:

```
int MPI_Comm_create(MPI_Comm comantigo, MPI_Group grupo, MPI_Comm *comnovo);
```

Aqui `comantigo` é o comunicador a partir do qual `grupo` foi criado (todos os processos de `grupo` fazem parte de `comantigo`). O novo comunicador pode então passar a ser utilizado diretamente para operações de comunicação.

Comunicadores também podem ser construídos diretamente de outros comunicadores. Uma forma é criar uma duplicata de um comunicador, isto é, um comunicador com os mesmos processos que um comunicador dado, mas que define um contexto de comunicação distinto. Isto é feito através de:

```
int MPI_Comm_dup(MPI_Comm comantigo, MPI_Comm *comnovo);
```

Também é possível criar novos comunicadores dividindo os processos de um comunicador original em um série de comunicadores que ficarão com processos distintos:

```
int MPI_Comm_split(MPI_Comm comantigo, int cor, int chave, MPI_Comm *comnovo);
```

O parâmetro `cor` é utilizado para determinar a separação dos processos nos novos comunicadores: processos do comunicador original que fornecerem o mesmo valor de `cor` serão colocados no mesmo comunicador, processos com valor de `cor` distinto serão colocados em comunicadores distintos. O parâmetro `chave` é usado para determinar o *rank* dos processos nos novos comunicadores: processos que fornecem o mesmo valor de `cor` são ordenados no novo comunicador de acordo com o valor de `chave` fornecido.

No exemplo abaixo, são criados dois novos comunicadores: um englobando todos os processos de *rank* ímpar e outro englobando todos os processos de *rank* par. Veja como os dois conjuntos de processos (pares e ímpares), apesar de estarem executando o mesmo cógido, estão trabalhando para a construção de dois comunicadores distintos, e depois fazendo reduções nesses dois comunicadores. Note também como o *rank* dos processos é dependente do comunicador utilizado: no comunicador `subcom` dos pares, a raiz da redução é seu *rank* 0, que corresponde ao *rank* 0 de `MPI_COMM_WORLD`, mas no `subcom` dos ímpares o processo de *rank* 0 é aquele que em `MPI_COMM_WORLD` tem *rank* 1.

Exemplo

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

int main(int argc, char **argv)
{
  int quantos, rank;
  int i, j;
  int *incluidos;
  int meu, total;
  MPI_Group todos, subgrupo;
  MPI_Comm  subcom;

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

  MPI_Comm_group(MPI_COMM_WORLD, &todos);

  incluidos = (int *)malloc(ceil((float)quantos/2)*sizeof(int));
  for (i = rank%2, j = 0; i < quantos; i+=2, j++)
    incluidos[j] = i;

  MPI_Group_incl(todos, j, incluidos, &subgrupo);
  MPI_Comm_create(MPI_COMM_WORLD, subgrupo, &subcom);

  meu = pow(2,rank);
  MPI_Reduce(&meu, &total, 1, MPI_INT,
             MPI_SUM, 0, subcom);

  if (rank == 0)
    printf("Soma das potencias pares:   %10d\n", total);
  else if (rank == 1)
    printf("Soma das potencias impares: %10d\n", total);

  MPI_Finalize();
  return 0;
}
```

O programa poderia ser simplificado com o uso de `MPI_Comm_split`. (Você sabe como?)

6 Conclusão

Descrevemos aqui algumas das primitivas de MPI, mais explicitamente da versão 1 do padrão MPI. Estas foram selecionadas por serem bastante úteis, suficientes para muitos problemas, e por apresentarem um bom apanhado do funcionamento da interface. Diversos outros detalhes não foram cobertos, e podem ser encontrados nos textos básicos [1] e [2].

Exercícios

1. Compile e execute os programas de exemplo desta apostila. Execute-os para diversos números de processos, e assegure-se de que você entendeu seu funcionamento e se familiarizou com o uso da implementação MPI empregada.

2. Escreva um programa usando MPI em que os diversos processos enviem mensagens à tela de acordo com o seguinte padrão: o processo de *rank* 0 envia a mensagem:

   `Eu sou o primeirao.`

   o processo de *rank* $P-1$ (para P processos rodando) envia a mensagem:

`Eu sou o lanterninha.`

entre os outros processos, os pares enviam a mensagem:

`Vamos passear juntos?`

e os ímpares enviam a mensagem

`Sou um sujeito ímpar!`

3. Escreva um programa que realize o seguinte: o processo de *rank* i calcula o valor i^2, envia esse valor para o processo de *rank* $i+1$ (se for o último processo, envia para o processo de *rank* 0), recebe o valor enviado pelo processo de *rank* $i-1$ (se for processo 0, recebe do último processo), subtrai do valor calculado localmente o valor recebido e imprime o resultado.

4. Desenvolva um programa de produto escalar paralelo. O processo 0 deve ler os dados de dois arquivos, em seguida os dados são distribuídos pelos diversos processos. Cada processo então calcula o produto escalar de seu trecho de vetores. Finalmente, uma redução é realizada para calcular o produto escalar total, que deve ser enviado para a tela.

   (O produto escalar de dois vetores a_i e b_i é a somatória

   $$\sum_{i=0}^{N-1} a_i b_i$$

   onde N é o número de elementos dos vetores.)

5. Desenvolva um programa para o seguinte problema de diferenças finitas unidimensional:

   Dados os valores de $X_i(0)$ para $1 \leq i \leq N$ (lidos de um arquivo), calcular o valor dos $X_i(T)$ (que deverão ser escritos em um arquivo). T é um valor fornecido ao programa. A equação de recursão é:

   $$X_i(t+1) = \frac{X_{i-1}(t) + 2X_i(t) + X_{i+1}(t)}{4}, \qquad 1 \leq i \leq N, \quad 0 \leq t \leq T-1$$

   e as condições de contorno são:

   $$X_0(t) = X_N(t), \qquad 0 \leq t \leq T$$

   e

   $$X_{N+1}(t) = X_1(t), \quad 0 \leq t \leq T$$

   (condições de contorno periódicas).

   Sendo P o número de processadores disponíveis, implemente um programa para as seguintes três situações:

   (a) $N = P$;

   (b) $N \geq P$;

   (c) Caso geral: N é independente de P.

6. Desenvolva uma nova versão de produto escalar paralelo, utilizando o esquema gerente/trabalhador. O processo 0 servirá de gerente, enquanto os outros são os trabalhadores.

   O código geral será da forma:

Pseudo-Código

```
1  Se rank igual a 0:
2    Execute gerente
3  Senão:
4    Execute trabalhador
```

   Um trabalhador executará um código da forma:

Pseudo-Código

```
1 Loop até receber mensagem de término:
2   Envia pedido de trabalho ao gerente
3   Recebe mensagem do gerente
4   Verifica tipo da mensagem:
5     Se bloco de trabalho:
6       Executa cálculos
7       Envia resultado ao gerente
8     Se mensagem de término:
9       Termina loop
```

Já o gerente deve executar código da forma:

Pseudo-Código

```
1 Lê dados dos vetores
2 Inicializa total geral com 0
3 Loop até não haver mais blocos a enviar:
4   Espera mensagem de algum trabalhador
5   Verifica tipo da mensagem:
6     Se resultado de cálculo:
7       Pega valor retornado e soma no total geral
8     Se pedido de trabalho:
9       Envia próximo bloco de ambos os vetores ao trabalhador
10 Inicializa n em 0
11 Enquanto n menor do que o número de trabalhadores
12   Espera mensagem de algum trabalhador
13   Verifica tipo da mensagem:
14     Se resultado de cálculo:
15       Pega valor retornado e soma no total geral
16     Se pedido de trabalho:
17       Envia aviso de término
18       Incrementa n
19 Imprime total geral
```

Os diversos "tipos de mensagens" indicados acima podem ser distinguidos pelo uso de *tags* distintos nas comunicações. Assim, por exemplo, as transmissões no código do trabalhador, linhas 2 e 7 utilizarão *tags* distintos. As correspondentes recepções no gerente, linhas 4 e 12 devem utilizar `MPI_ANY_TAG`, e após a recepção deve ser realizado um teste para verificar qual o *tag* efetivamente recebido. Algo similar ocorrerá com as transmissões nas linhas 9 e 17 do gerente e a correspondente recepção na linha 3 dos trabalhadores.

Nas linhas 4 e 12 do gerente, deve também ser utilizado `MPI_ANY_SOURCE`, pois o gerente não sabe de início qual a velocidade relativa de execução dos trabalhadores.

Além disso os trabalhadores, ao receberem um pacote de trabalho, devem utilizar `MPI_Get_count` para determinar o número de elementos efetivamente recebidos, pois o número total de elementos no vetor pode não ser um múltiplo exato do tamanho do bloco, caso em que o número de elementos enviados no último bloco será menor do que o dos outros blocos.

O código das linhas 10 a 18 do gerente é necessário para terminar a execução, após haverem sido enviados todos os blocos de dados existentes: deve-se esperar o resultado do último trabalho de cada trabalhador, e então enviar um aviso de término quando ele solicitar novo trabalho.

Obs: O pseudo-código acima apresenta a idéia central do funcionamento do esquema gerente/trabalhador. É possível reduzir o número de mensagens trocadas entre o gerente e os trabalhadores, com correspondente ganho de desempenho e também, neste caso, simplificação do código, através do seguinte método: utilizar a própria mensagem de retorno de valor por um trabalhador como indicação de um novo pedido de trabalho. Considere a possibilidade de implementar o seu programa usando esse método.

7. Considere o problema de Poisson [6] caracterizado pela equação diferencial parcial elíptica

$$\frac{\partial^2 u(x,y)}{\partial x^2} + \frac{\partial^2 u(x,y)}{\partial y^2} = f(x,y),$$

com condições de contorno

$$u(x,y) = g(x,y).$$

O cálculo aproximado da solução pode ser feito através da discretização do espaço em uma malha de pontos (x_i, y_j) uniforme na duas direções: $x_{i+1} = x_i + h$, $y_{j+1} = y_j + h$. Usando uma aproximação em diferenças finitas centrais encontramos

$$\frac{u(x_{i+1}, y_j) - 2u(x_i, y_j) + u(x_{i-1}, y_j)}{h^2} + \frac{u(x_i, y_{j+1}) - 2u(x_i, y_j) + u(x_i, y_{j-1})}{h^2} = f(x_i, y_i).$$

Essa expressão pode ser utilizada num método iterativo, onde os valores atuais de u são usados para calcular uma nova aproximação para u, até que se detecte convergência. A expressão para a iteração fica [usando $u_{i,j}$ para representar $u(x_i, y_j)$ e correspondentemente para as funções f e g]:

$$u_{i,j}^{k+1} = \frac{1}{4}\left(u_{i+1,j}^{k} + u_{i-1,j}^{k} + u_{i,j+1}^{k} + u_{i,j-1}^{k} - h^2 f_{i,j}\right),$$

onde k é o número da iteração atual.

Note que para a execução desse método iterativo necessitamos de duas matrizes, uma para armazenar o valor atual de $u_{i,j}^{k}$ e outra para armazenar o novo valor $u_{i,j}^{k+1}$.

Desenvolva um programa que resolva esse problema para o seguinte caso:

- A extensão do espaço é unitária (isto é, temos um quadrado de lado 1), extendendo-se de $(0, 0)$ a $(1, 1)$.
- A discretização do espaço é feita com N divisões em cada direção, totalizando N^2 elementos discretos e resultando em $h = 1/N$.
- As condições de contorno são dadas por
$$g_{i,j} = 0$$
em todo o contorno.
- Para os pontos que não estão no contorno, temos
$$f(x, y) = \begin{cases} 1 & \text{se } \frac{3}{8} < x < \frac{5}{8}, \frac{3}{8} < y < \frac{5}{8} \\ 0 & \text{caso contrário} \end{cases}$$
Note que determinamos a posição de um elemento usando a posição de seu centro.
- As condições iniciais serão
$$u_{i,j} = 0.$$
- Para terminar o processo iterativo, use um valor fixo M de iterações.
- Os valores de N e M são entradas para o programa.

Calcular os valores $u_{i,j}^{M}$ (que deverão ser escritos em um arquivo).

8. Altere o programa anterior para que o critério de parada não seja mais uma quantidade fixa de iterações, mas que o término seja decidido pelo seguinte critério de convergência:

   O programa termina a execução quando $\left|u_{i,j}^{k+1} - u_{i,j}^{k}\right| < \epsilon$ para $1 \leq i \leq N$, $1 \leq j \leq N$, sendo ϵ um valor de precisão fornecido ao programa.

   Note que neste caso as entradas do programa são N e ϵ.

Referências

[1] MPI Forum, *MPI: A Message-Passing Interface Standard*, 1995.

[2] MPI Forum, *MPI-2: Extensions to the Message-Passing Interface*, 1997.

[3] MPI Forum, `http://www.mpi-forum.org`.

[4] `http://www.mcs.anl.gov/mpi/mpich`

[5] `http://www.open-mpi.org`

[6] J. Dongarra, I. Foster, G. Fox, W. Gropp, K. Kennedy, L. Torczon, A. White, *Sourcebook of Parallel Computing*, Morgan Kaufmann, 2003.