4

Message-Passing Interface Avançado

"*Lessons from MPI: Make hard things possible, making easy things easy is not relevant*"
Willian Gropp (EuroPVM/MPI 2003)

Professores:

Nicolas Maillard[1]
(nicolas@inf.ufrgs.br)
Márcia Cristina Cera[2]
(marcia.cera@inf.ufrgs.br)

Resumo:

Este capítulo trata de questões relacionadas ao uso avançado das características do MPI (Message-Passing Interface). Este texto considera que o leitor já aprendeu os fundamentos sobre a programação com troca de mensagens no curso básico de Programação Paralela, tradicionalmente ministrado em todas edições da ERAD. O objetivo desse capítulo é demostrar a evolução das características do MPI e principalmente seu potencial para a solução de problemas complexos.

Nas próximas seções, o leitor encontrará desde a descrição de primitivas a exemplos de código demostrando o emprego de técnicas avançadas do MPI e os ganhos que se pode obter em aplicações de Alto Desempenho. Como seria inviável mostrar todas as possibilidades em um único curso, nós selecionamos alguns pontos que consideramos relevantes dentro da nossa experiência com o MPI: particionamento dos processos e de suas trocas de mensagens com Comunicadores; comunicações não bloqueantes; comunicação de tipos de dados abstratos; e criação dinâmica de processos. Buscou-se, assim, fugir do modelo básico de programa

[1] Doutor em Sciences et Technologies de l'Information - Univ. Joseph Fourier (2001). Atualmente é professor adjunto no Instituto de Informática da UFRGS. Sua pesquisa envolve: programação e algoritmos paralelos, computação em Clusters e Grids, MPI, escalonamento de processos. Em seu currículo há um número significativo de publicações envolvendo MPI e aplicações em PAD.

[2] Doutoranda em Computação na UFRGS onde é bolsista CAPES, mestre na área de Tecnologia de Informação pela UFSM (2005) e graduada em Ciência da Computação na UFSM (2002). Fez seu doutorado sanduíche no Laboratoire d'Informatique de Grenoble, França (2008/2009). Atua em: PAD, escalonamento e balanceamento de carga, MPI e aplicações maleáveis. Seu doutorado envolve estudos sobre MPI e às aplicações maleáveis, sob co-orientação do Prof. Nicolas.

Mestre/Escravo que é comumente associado à programas MPI, embora esta norma possibilite muito mais. Esta é a mensagem que gostaríamos de passar através desse curso, despertando o interesse dos leitores a irem além do conhecimento básico.

4.1. Os Comunicadores

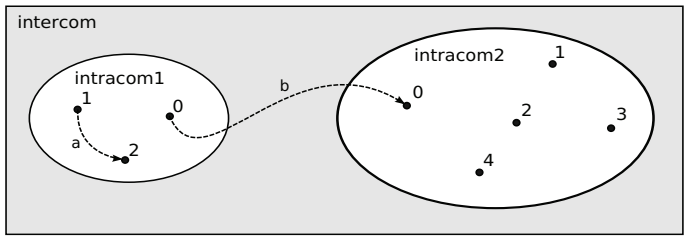
Todo processo MPI possui um par `(rank, grupo de processos)`. O *rank* é um valor único (identificador) do processo, relativo ao seu grupo. Cada processo pode pertencer a mais de um grupo, e portanto possuir mais de um *rank*. Um comunicador encapsula os *ranks* e os grupos. Em outras palavras, um comunicador é uma estrutura que define um grupo de processos e um contexto para comunicações entre eles. O contexto é uma propriedade dos comunicadores que permite um particionamento seguro do espaço de comunicações, garantindo que uma mensagem enviada num contexto não poderá ser recebida noutro.

As operações de comunicação do MPI referenciam-se aos comunicadores para determinar o escopo e o "universo de comunicação" (contexto) no qual acontecerão comunicações ponto a ponto ou coletivas. No caso ponto a ponto, a origem e o destino de uma mensagem são determinados pelos identificadores únicos dos processos dentro de seu grupo. Para as comunicações coletivas, o comunicador especifica o conjunto de processos que participam da operação e a ordem deles, quando esta for significante. Existem 2 tipos de comunicadores: *(i)* os intracomunicadores para troca de mensagens entre processos de um mesmo grupo local e *(ii)* os intercomunicadores para troca de mensagens entre grupos diferentes, um local e e outro remoto (segundo a ótica de um processo).

A Figura 4.1 exemplifica os dois tipos de comunicadores. Nela, têm-se dois intracomunicadores (`intracom1` e `intracom2`), e cada um deles tem seu grupo local de processos. O intercomunicador (`intercom`) estabelece um canal de comunicação entre os dois intracomunicadores. Na figura há duas trocas de mensagens, conforme foi detalhado na sua parte inferior pelos dois conjuntos de `MPI_Send`/`MPI_Recv`. Na comunicação que acontece em *a)*, o processo 1 (ou seja, o processo com o identificador ou *rank* igual a 1) envia uma mensagem para o processo 2 (`MPI_Send(&i, 1, MPI_INT, 2, tag, intracom1)`), o qual receberá a mensagem vinda do processo 1 (`MPI_Recv(&i, 1, MPI_INT, 1, tag, intracom1, &st)`). Ambos os processos pertencem ao mesmo grupo e usam o mesmo `intracom1` determinando o contexto em que a mensagem está sendo enviada. A comunicação em *b)* acontece entre processos que estão em grupos distintos. O processo 0 no grupo representado pelo `intracom1` envia uma mensagem para o processo 0 no grupo do `intracom2`. Tal comunicação é possível graças ao intercomunicador `intercom` sobre o qual as operações de envio e recebimento da mensagem são realizadas.

O `MPI_COMM_WORLD` é um comunicador pré-definido pelo MPI que permite a comunicação entre todos os processos de uma mesma aplicação. Logo, ele é um intracomunicador que engloba automaticamente todos os processos de uma aplicação num mesmo grupo, após a execução do `MPI_Init`. O `MPI_COMM_SELF` é outro intracomunicador pré-definido, o qual refere-se apenas ao processo local, ou seja, ele é um intracomunicador que representa um grupo composto apenas por um processo, o qual possui seu identificador único igual a 0. Pode-se usar o `MPI_COMM_WORLD` ou `MPI_COMM_SELF` para criar comunicadores mais complexos por união ou interseção de comunicadores.

Para classes de aplicações simples, possuir um único comunicador global



a) MPI_Send(&i, 1, MPI_INT, **2**, tag, **intracom1**);
MPI_Recv(&i, 1, MPI_INT, **1**, tag, **intracom1**, &st);

b) MPI_Send(&j, 1, MPI_INT, **0**, tag, **intercom**); MPI_Recv(&j, 1, MPI_INT, **0**, tag, **intercom**, &st);

Figura 4.1: Mensagens entre: (a) processos de um mesmo intracomunicador e (b) em intracomunicadores diferentes usando um intercomunicador.

(`MPI_COMM_WORLD`) supre todas as necessidades. Entretanto, existem classes de aplicações que terão mais de um grupo de processos gerando a necessidade da manipulação de comunicadores para particionar os processos em grupos - tipicamente, isso vai acontecer quando o programa paralelo executa em máquinas com alto número de *cores*, chegando a incluir milhares de processos. A seguir há uma breve descrição de primitivas de manipulação de comunicadores de acordo com o padrão MPI-2.2 [MPI 2009].

- Acesso às informações dos comunicadores (operações locais que não requerem comunicação entre os processos pertencentes aos comunicadores):
  - `MPI_Comm_size(MPI_Comm comm, int *size)` retorna o número total de processos do grupo representado pelo comunicador `comm`;
  - `MPI_Comm_rank(MPI_Comm comm, int *rank)` retorna o identificador do processo que chamou a função no grupo representado pelo comunicador `comm`;
  - `MPI_Comm_compare(MPI_Comm comm1, MPI_Comm comm2, int *result)` compara os dois comunicadores passados como argumento retornando um inteiro que pode ser: `MPI_IDENT` se e somente se `comm1` e `comm2` possuem grupos idênticos e o mesmo contexto; `MPI_CONGRUENT` se os comunicadores diferem apenas pelo contexto; `MPI_SIMILAR` se os membros dos grupos de ambos os comunicadores são os mesmos mas a ordem dos identificadores não; e `MPI_UNEQUAL` caso não hajam similaridades.
- Criação de comunicadores (estas operações são coletivas e requerem comunicações entre os processos):
  - `MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)` duplica um comunicador `comm` com todos os valores chaves associados. O novo co-

municador `newcomm` possui o mesmo grupo ou grupos de `comm` e qualquer informação copiada em cache, mas terá um novo contexto;

- `MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm)` se `comm` é um intracomunicador, a função retorna um novo comunicador `newcomm` com um grupo comunicante definido pelo argumento `group` sem copiar nenhuma informação em cache a partir de `comm`. Se `comm` for um intercomunicador, `newcomm` também será um intercomunicador onde o grupo local terá apenas os processos definidos em `group`;
- `MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)` o grupo associado a `comm` será particionado em grupos disjuntos, um para cada valor de `color`. Cada subgrupo conterá processos de uma mesma cor e seus identificadores estarão ordenados conforme `key`. Um novo comunicador é criado para cada grupo e retornado em `newcomm`.

- Destruição de comunicadores (operação coletiva):
  - `MPI_Comm_free(MPI_Comm comm)` marca o comunicador `comm` como uma estrutura a ser desalocada e lhe atribui `MPI_COMM_NULL`.

Em função do tipo de particionamento dos processos que se quer, pode-se escolher qual dessas primitivas é a mais apropriada (uma outra opção é manipular os grupos). Por exemplo, pode-se partir de `MPI_COMM_WORLD` e "recortá-lo"em subcomunicadores através de `MPI_Comm_split`. A Listagem 4.1 ilustra seu uso.

Listing 4.1: Uso do `MPI_Comm_split`.

```
1 MPI_Comm novo_comm;
2 int rank, cor, chave = 0;
3
4 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
5 cor = rank % 2;
6 MPI_Comm_split(MPI_COMM_WORLD, cor, chave, &novo_comm);
7 MPI_Bcast(&chave, 1, MPI_INT, 0, novo_comm);
```

Se 5 processos executarem este código MPI, todos vão recuperar em `rank` um valor distinto ($0, 1, 2, 3$ ou 4). Nos processos com `rank` par, o cálculo de `cor` retornará 0, e nos ímpares, `cor` terá o valor 1. Dessa forma, a chamada (coletiva) a `MPI_Comm_split` vai fazer com que, após seu retorno, os processos com `rank` par tenham um comunicador `novo_com` que os inclue, e os ímpares tenham um outro comunicador, porém com mesmo nome – lembre-se que se trata de um programa SPMD. O potencial do emprego do `MPI_Comm_split`, nesse exemplo, fica claro quando analisamos a execução do `MPI_Bcast` (linha 7). Ele é executado por todos os processos e irá fisicamente disparar duas difusões: o processo de `rank 0` no comunicador dos

processos pares irá mandar `chave` para todos os processos de seu grupo (os pares); o processo de `rank 0` no comunicador dos ímpares (provavelmente o processo de `rank 1` no `MPI_COMM_WORLD`) irá mandar seu valor de `chave` a todos os processos de seu grupo (os ímpares).

Após a definição do padrão MPI-2 que prevê a criação dinâmica de processos, os conceitos de grupos de processos, intra e intercomunicadores passaram a ser mais evidentes. Isto porque, quando um processo é criado em tempo de execução, ele fará parte de um grupo diferente (remoto) do grupo do processo que o criou. Logo, existirão 2 intracomunicadores, um para cada grupo de processos, e intercomunicadores permitirão trocas de mensagens entre eles. Além de aplicações com criação dinâmica de processos, aplicações modulares e multidisciplinares são implementadas com grupos de processos distintos. Nelas, diferentes grupos de processos executam módulos distintos e processos dentro de diferentes módulos comunicam-se na forma de um *pipeline* ou de um grafo mais generalizado [MPI 2009]. Nesse tipo de aplicações, os intercomunicadores permitirão as intercomunicações ou as comunicações entre processos de grupos distintos. Cabe salientar que nas intercomunicações o processo alvo é determinado pelo par `(comunicador, identificador)` sendo o identificador referente ao grupo remoto – onde se encontra o processo alvo. A seguir, as primitivas que possibilitam a manipulação de intercomunicadores são brevemente descritas.

- Acesso as informações de intercomunicadores (operações locais):
  - `MPI_Comm_test_inter(MPI_Comm comm, int *flag)` através dessa primitiva o processo pode identificar se `comm` é um intra ou intercomunicador. O valor de `flag` será `true` se é um intercomunicador ou `false` no caso contrário;
  - `MPI_Comm_remote_size(MPI_Comm comm, int *size)` retorna o número de processos pertencentes ao grupo remoto do intercomunicador `comm`. O número de processos no grupo local é determinado por `MPI_Comm_size(MPI_Comm comm, int *size);`
  - `MPI_Comm_remote_group(MPI_Comm comm, MPI_Group *group)` retorna o grupo remoto do intercomunicador `comm`. O grupo local pode ser acessado por `MPI_Comm_group(MPI_Comm comm, MPI_Group *group)`.
- Operações sobre intercomunicadores (operações coletivas bloqueantes):
  - `MPI_Intercomm_create (MPI_Comm local_comm, int local_leader, MPI_Comm peer_comm, int remote_leader, int tag, MPI_Comm *newintercomm)` cria o intercomunicador (`newintercomm`) ligando os dois intracomunicadores (`local_comm` e `peer_comm`) passados em argumento;
  - `MPI_Intercomm_merge (MPI_Comm intercomm, int high, MPI_Comm *newintracomm,)` une os grupos local e remoto associados a um intercomunicador, retornando um novo intracomunicador entre eles;

– As operações coletivas `MPI_Comm_dup` e `MPI_Comm_free`, mantém o mesmo funcionamento conforme descrito anteriormente, duplicando e liberando intercomunicadores respectivamente.

A Listagem 4.2 é uma continuação do exemplo apresentado na Listagem 4.1. Além de dividir o `MPI_COMM_WORLD` em dois intracomunicadores (processos pares e ímpares), um intercomunicador foi criado entre eles (linha 20). Após, a chamada `MPI_Intercomm_merge` irá unir os processos em um: único intracomunicador `merge_comm`.

Listing 4.2: Exemplo de manipulação de comunicadores.

```
1  MPI_Init(&argc, &argv);
2  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
3  MPI_Comm_size(MPI_COMM_WORLD, &size);
4  cor = rank % 2;
5  printf("[MPI_COMM_WORLD: %d] Processo: %d, cor: %d\n",
6          size, rank, cor);
7  MPI_Comm_split(MPI_COMM_WORLD, cor, chave, &novo_comm);
8  MPI_Bcast(&chave, 1, MPI_INT, 0, novo_comm);
9
10 MPI_Comm_rank(novo_comm, &rank);
11 MPI_Comm_size(novo_comm, &size);
12 printf("[novo_comm %d] Processo %d \n", size, rank);
13 MPI_Intercomm_create(novo_comm, 0, MPI_COMM_WORLD,
14                       1 - cor, 123, &intercomm);
15 MPI_Comm_free(&novo_comm);
16 MPI_Comm_rank(intercomm, &rank);
17 MPI_Comm_size(intercomm, &size);
18 MPI_Comm_remote_size(intercomm, &rsize);
19 printf("[intercomm: local=%d remoto=%d] Processo %d\n",
20         size, rsize, rank);
21 MPI_Intercomm_merge(intercomm, cor, &merge_comm);
22 MPI_Comm_free(&intercomm);
23 MPI_Comm_rank(merge_comm, &rank);
24 MPI_Comm_size(merge_comm, &size);
25 printf("[merge_comm: %d] Processo %d\n", size, rank);
26 MPI_Comm_free(&merge_comm);
27 MPI_Finalize();
28 }
```

Logo após o código, tem-se a saída textual de uma execução com 5 processos. Nela, é possível identificar tanto a manipulação dos comunicadores quanto a distinção entre grupos remotos e locais.

```
----------------------------------------
[MPI_COMM_WORLD: 5] Processo: 4, cor: 0
[MPI_COMM_WORLD: 5] Processo: 3, cor: 1
[MPI_COMM_WORLD: 5] Processo: 1, cor: 1
[MPI_COMM_WORLD: 5] Processo: 2, cor: 0
[MPI_COMM_WORLD: 5] Processo: 0, cor: 0
[novo_comm 2] Processo 0
[novo_comm 3] Processo 0
[novo_comm 2] Processo 1
[novo_comm 3] Processo 2
[novo_comm 3] Processo 1
[intercomm: local=3 remoto=2] Processo 0
[intercomm: local=3 remoto=2] Processo 1
[intercomm: local=3 remoto=2] Processo 2
[intercomm: local=2 remoto=3] Processo 0
[intercomm: local=2 remoto=3] Processo 1
[merge_comm: 5] Processo 3
[merge_comm: 5] Processo 0
[merge_comm: 5] Processo 4
[merge_comm: 5] Processo 2
[merge_comm: 5] Processo 1
----------------------------------------
```

4.2. Comunicações não Bloqueantes

Comunicações não bloqueantes são essenciais numa aplicação MPI realista, para possibilitar a paralelização entre os cálculos e as comunicações. A não ser nos casos onde o paralelismo é trivial, esta será em geral a única forma de não ter um baixo desempenho na execução de programas paralelos.

O MPI provê a primitiva `MPI_Irecv(void*, int, MPI_Datatype, int, int, MPI_Comm, MPI_Comm, MPI_Request*)` para efetuar recepções não bloqueantes. Seus parâmetros são exatamente os mesmos como no caso do `MPI_Recv` clássico, com a diferença importante do último: no recebimento bloqueante este é do tipo `MPI_Status`, no caso não bloqueante é um `MPI_Request` passado por referência. Trata-se de um *handle* sobre a comunicação não bloqueantes disparado pelo `MPI_Irecv`, o qual servirá em outras primitivas como um identificador da comunicação.

Um exemplo típico de chamada será: `MPI_Irecv(&result, 1, MPI_DOUBLE, 1, tag, MPI_COMM_WORLD, &request);` onde `result` e `tag` serão de tipo *int*, e se terá uma declaração anterior `MPI_Request request`.

O `MPI_Irecv` recebe qualquer mensagem mandada por um outro processo, que esteja de acordo com a assinatura esperada. O uso de recepção não bloqueante não obriga o uso de envios não bloqueantes.

Uma vez que o `MPI_Irecv` é não bloqueante, a execução do programa continuará tão logo a recepção tiver sido disparada. Isso significa que o programador está livre para usar as variáveis envolvidas na recepção, inclusive alterando seu conteúdo. Nada impede, por exemplo, que se use `result` na linha imediatamente após o `MPI_Irecv` acima descrito, quer tenha chegado a mensagem, ou não. Claro, isso pode levar a se usar dados inconsistentes. Por isso, o MPI fornece ao usuário outras primitivas, que serão usadas para testar o andamento e eventual término de uma comunicação não bloqueante:

- `MPI_Test(MPI_Request * req, int * flag, MPI_Status* st)` possibilita testar a finalização da comunicação não bloqueante apontada por `req` (ou seja, por um `MPI_Request` retornado por exemplo pela chamada `MPI_Irecv`). Caso tenha terminado, o `flag` passa a valer `true` e `st` conterá as informações relativas à comunicação (ou seja, o *rank* do processo remetente e o `tag` da mensagem). Caso contrário, `flag` terá seu valor igual a `false`.
- `MPI_Wait(MPI_Request * req, MPI_Status * st)` bloquea a execução, na espera do término da comunicação não bloqueante apontada por `req`. Assim, usar um `MPI_Wait` logo a após um `MPI_Irecv`, com o `MPI_Request` por ele retornado, é totalmente equivalente a se usar um `MPI_Recv` bloqueante tradicional (e, por isso, inútil).

O uso típico de recepções não bloqueantes irá então:

1 Disparar um `MPI_Irecv`, para receber alguma mensagem com dados úteis para o processo, porém não necessários imediatamente no cálculo. Isso supõe que se possa ter uma "reserva" de trabalho a ser executada localmente pelos processos enquanto aguardam as mensagens. Essa reserva de trabalho local depende da velocidade de processamento e de comunicação da aplicação;

2 Executar esses procedimentos locais, enquanto a recepção está sendo feita. A eventual chegada da mensagem será testada periodicamente com o `MPI_Test`, tipicamente a cada iteração quando o trabalho local pode ser efetuado em um laço;

3 Quando não houver mais nada a ser processado localmente, em geral o processo deverá bloquear-se a espera do término da recepção, através de um `MPI_Wait`. Se o programador conseguir mesclar bem cálculos e comunicações (quando a aplicação permite isso), a fase anterior já terá possibilitado que boa parte da mensagem tenha chegado, ou talvez toda ela, durante os cálculos locais.

A listagem 4.3 e 4.4 mostram uma versão de um programa genérico de Mestre/Escravo, usando recepções não bloqueantes. (Por questões de formatação, elas foram posicionadas no fim do capítulo.) Nota-se que do lado do escravo, não tem como usar mensagens não bloqueantes, uma vez que o ele apenas espera por uma tarefa, a processa e deve retornar o resultado antes de receber uma nova tarefa.

A primeira operação realizada pelo mestre é o disparo de uma recepção não bloqueante (linha 30) de um resultado parcial, procedente de qualquer processo escravo. O

Listing 4.3: Programa Mestre/Escravo com MPI, versão não-bloqueante.

```
1  void mestre() {
2    double res_parcial, res_total = 0;
3    int p, p_id, recebido_um_resultado, dest = 1;
4    int task = 1;   /* indice da ultima tarefa enviada*/
5    int feito = 0;  /* número de tarefas processadas*/
6    MPI_Status st;
7    MPI_Request req;
8    MPI_Comm_size(MPI_COMM_WORLD, &p);
9    MPI_Comm_rank(MPI_COMM_WORLD, &p_id);
10   for (dest = 1; dest < p; dest++) {
11     MPI_Send(&task, 1, MPI_INT, dest, TAG_INDICE,
12                                       MPI_COMM_WORLD);
13     task++;
14   }
15   /* Dispara uma recepcao nao-bloqueante ... */
16   MPI_Irecv(&res_parcial, 1, MPI_DOUBLE,MPI_ANY_SOURCE,
17                       TAG_RES, MPI_COMM_WORLD, &req);
18   /* ... e testa as chegadas ateh ter mandado tudo */
19   while (task < NUMERO_TAREFAS) {
20     MPI_Test(&req, &recebido_um_resultado, &st);
21     if (recebido_um_resultado) {
22       /* Manda nova tarefa a quem enviou resultado */
23       printf("Mestre␣recebeu␣um␣resultado\n");
24       MPI_Send(&task, 1, MPI_INT, st.MPI_SOURCE,
25                             TAG_INDICE, MPI_COMM_WORLD);
26       task++; feito++;
27       res_total += res_parcial;
28       /* e dispara uma nova recepcao nao bloqueante */
29       MPI_Irecv(&res_parcial, 1, MPI_DOUBLE,
30         MPI_ANY_SOURCE, TAG_RES, MPI_COMM_WORLD, &req);
31     } else {
32       /* Enquanto espera, o mestre tambem processa */
33       res_total += calculo(task);
34       printf("Mestre␣processa␣tarefa␣%d.\n", task);
35       task++; feito++;
36     }
37   }                 /* Continua ... */
```

identificador da mensagem não bloqueante é `req` Enquanto ela vai sendo efetuada, o mestre entra no laço principal, onde logo testa se chegou alguma resultado (linha 34). Nota-se o uso de `MPI_Test`, com um teste booleano imediato a seguir (linha 35) para determinar se chegou ou não um resultado e tomar a providência adequada:

Listing 4.4: Programa Mestre/Escravo com MPI, versão não-bloqueante.

```
  /* Acabou. Envia última mensagens aos escravos */
  task = ACABOU;
  for (dest = 1; dest < p; dest++) {
    printf("Mestre␣mandou␣mesg␣de␣fim␣a␣%d.\n",dest);
    MPI_Send(&task, 1, MPI_INT, dest, TAG_INDICE,
                                      MPI_COMM_WORLD);
  }
  /* E espera pelos ultimos resultados. Primeiro, o
   * ultimo Irecv disparado */
  MPI_Wait(&recv_req, &st);
  res_total += res_parcial; feito++;
  /* Depois, espera pelos que ainda não retornaram */
  while (feito < NUMERO_TAREFAS - 1) {
    MPI_Recv(&res_parcial, 1,MPI_DOUBLE,MPI_ANY_SOURCE,
                        TAG_RES, MPI_COMM_WORLD, &st);
    res_total += res_parcial; feito++;
  }
  printf("Resultado:␣%lf.", res_total);
  if (res_total != (NUMERO_TAREFAS *
                            (NUMERO_TAREFAS - 1) / 2))
    printf("...␣ERRADO.\n");
  else printf("...␣CERTO.\n");
}                                /* Fim mestre */

void escravo(){
  int i = 0, r;
  MPI_Status st;
  double res_parcial;
  MPI_Comm_rank(MPI_COMM_WORLD, &r);
  while (i != ACABOU) {
    MPI_Recv(&i, 1, MPI_INT, 0, TAG_INDICE,
                                MPI_COMM_WORLD, &st);
    if (i != ACABOU) {
      res_parcial = calculo(i);
      MPI_Send(&res_parcial, 1, MPI_DOUBLE, 0,TAG_RES,
                MPI_COMM_WORLD);
      printf("Escravo␣%d:␣processou␣tar.␣%d\n", r, i);
    }
  }
}
```

- Se chegou uma mensagem, manda-se uma nova tarefa para o escravo que enviou o resultado, da mesma forma como na versão anterior. Note que o mestre deve

disparar um novo `MPI_Irecv` (linha 43), para receber de forma não bloqueante os demais resultados;

- se não chegou nada ainda, o mestre passa a calcular parte das tarefas (linhas 45–50). Dessa forma, o mestre não perde tempo com a espera ativa, e sim participa do cálculo (ele dá o bom exemplo!), além de dar um tempo para os resultados dos escravos chegarem.

O mestre repete esse comportamento até enquanto existem tarefas a serem processadas. Ao final, ele deve ainda mandar um sinal de encerramento a cada processo escravo (linhas 53–58). Depois disso, basta ele esperar pelos últimos resultados. No entanto, o mestre deve primeiro esperar pelo término da recepção não bloqueante que (talvez) esteja recebendo dados, ou vá fazê-lo até o fim do programa. É importante "consumir" esta comunicação para que não sobre uma mensagem pendente. Por isso, usa-se o `MPI_Wait` sobre o `MPI_request` usado pelo programa (linha 61), e só depois se usa uma série de recepções bloqueantes para esperar o que ainda falta (linhas 64–68). Como o mestre não tem mais nada a fazer a não ser esperar pelos resultados, não há justificativas para o uso de recepções não bloqueantes.

Observa-se que, numa execução típica, o mestre acabará calculando muito mais tarefas que os escravos, pelo menos se o tempo de cálculo delas é pequeno. Isso acontecerá porque o tempo de comunicação será muito maior do que o tempo de processamento: se a latência de uma comunicação típica é em torno de 10 ms., enquanto a freqüência de relógio do processador onde executa o mestre é de 1 Ghz, o mesmo pode executar 1 bilhão de instruções por segundo, ou seja, 10 milhões de instruções apenas no período de tempo necessário à inicialização da comunicação. Logo, se uma tarefa envolve apenas algumas dezenas ou centenas de instruções *assembly*, o mestre irá poder calcular muita coisa só durante o envio de algumas tarefas aos seus escravos.

Nota-se que isso é um princípio geral da programação paralela: é sempre bom extrair o máximo de paralelismo que se pode numa aplicação, para poder usar o maior número possível p de processadores, na esperança de poder dividir o tempo de execução por p. No entanto, esse desejo de obter muito paralelismo sempre deve ser ponderado pelos atrasos gerados pela execução em paralelo, por exemplo devidos à alta latência de uma rede. Existem casos onde, simplesmente não se ganha nada ao usar mais de um processador. Nota-se que, mesmo nestes casos, um programa tal como nosso último não irá perder muito tempo, pois o mestre irá calcular quase todas as tarefas se os escravos levarem muito tempo para lhe responder. O único ônus para o mestre, quando comparado com uma execução seqüencial do programa, consiste no envio de uma mensagem inicial por escravo, e em testes periódicos por recepções de mensagens. Assim, este programa paralelo inclui a opção de auto limitar-se no grau de paralelismo caso a melhor opção seja executar sequencialmente.

Uma possível melhoria seria mandar mais de uma tarefa por vez ao escravo. Assim, logo que ele terminar o processamento de uma delas, ele poderá mandar um retorno (de forma não bloqueante) ao mestre enquanto computa a tarefa reserva.

4.3. Comunicação de Dados Complexos

Supondo-se que um programa paralelo envolva o seguinte tipo abstrato:

```
struct Dados {
   int idf;
   double valores[10];
};
```

Enviar uma variável `dado` do tipo descrito acima, através de uma mensagem implica numa série de dificuldades:

- ela possui campos de tipos diferentes;
- cada campo envolve um número diferente de elementos (1 `int`, e 10 `doubles`);
- não existe nenhuma garantia de que o compilador, ao reservar espaço na memória para os campos `idf` e `valores`, o faça de forma contígua. Tipicamente, pode haver alinhamento dos campos da *struct* que faça com que hajam lacunas entre os campos.

Sendo assim, não há como usar um `MPI_Send` clássico, o qual apontaria para o endereço de uma área contígua na memória que conteria n ocorrências de um dado tipo.

4.3.1. MPI_Pack/Unpack

Uma primeira opção seria *serializar* `dado`, ou seja copiá-lo num *buffer* à mão. Seria alguma coisa do tipo:

```
char * buffer = (char *)malloc(sizeof(int) + 10 *
                                        sizeof(double));
memcpy(buffer, &(dado.idf), sizeof(int));
memcpy(buffer+sizeof(int), &(dado.valores), 10 *
                                        sizeof(double));
```

A partir daí, pode-se efetuar um `MPI_Send(buffer, sizeof(int) + 10 * sizeof(double), MPI_CHAR, ...)` e o `MPI_Recv` correspondente. Ao receber `buffer`, o processo receptor deverá efetuar o desempacotamento dos dados. Entretanto, observa-se rapidamente as limitações de uma tal operação: ela depende diretamente da representação local a cada máquina dos tipos envolvidos, além de levar facilmente a erros.

Para resolver o primeiro problema, o MPI fornece uma interface universal de empacotamento e desempacotamento de dados: `MPI_Pack`/`MPI_Unpack`. O usuário define um *buffer* e usa chamadas sucessivas a `MPI_Pack` para copiar seus dados nele. Cada

chamada atualiza um índice inteiro que aponta para a posição atual no *buffer* onde se pode continuar a empacotar outros dados.

`MPI_Pack` tem o seguinte perfil: `MPI_Pack(void *buf, int count, MPI_Datatype dtype, void *packbuf, int packsize, int *packpos, MPI_Comm comm)`. Os três primeiros parâmetros especificam os elementos a serem empacotados. Eles são semelhantes aos argumentos do `MPI_Send`, por exemplo. Depois vêm os parâmetros que descrevem o *buffer* para empacotamento: `void* packbuf` é o ponteiro para o *início* do buffer; `packsize` é seu tamanho. O argumento seguinte é importantíssimo: `packpos` deve valer 0 na primeira chamada a `MPI_Pack`; depois, cada chamada o atualiza em função do que foi empacotado previamente, para apontar para a próxima entrada no *buffer* onde se pode empacotar novos dados. Por fim, o último argumento é o comunicador que se espera nas primitivas MPI.

Um *buffer* que foi preenchido por chamadas ao `MPI_Pack` com os dados a serem enviados, será transmitido através de mensagens associadas ao *datatype* especial `MPI_PACKED`. O tamanho da mensagem é igual ao último valor retornado em `packpos`. O recebimento da mensagem deve estar de acordo com o perfil do dado empacotado, por exemplo usando um `MPI_Recv` esperando por dados do tipo `MPI_PACKED`.

Após o recebimento de *buffer*, usa-se a primitiva `MPI_Unpack` para a extração dos dados. Os argumentos de `MPI_Unpack` são semelhantes aos do `MPI_Pack`. Por exemplo, `MPI_Unpack(packbuf, packsize, &packpos, buf, 1, MPI_INT, MPI_COMM_WORLD)`, desempacota 1 `MPI_INT` do dado `packbuf`, o qual será retornado em `buf` (no caso, um `int*`). Nota-se que os dados devem ser desempacotados na mesma ordem em que foram empacotados. Assim, na primeira chamada `packpos` deverá ser igual a 0, o qual será atualizado como saída do `MPI_Unpack`, em função do tamanho do dado que foi desempacotado. O valor atualizado deverá ser passado como entrada da próxima chamada, e assim sucessivamente.

A Listagem 4.5 ilustra o uso conjunto de `MPI_Pack` e `MPI_Unpack`, com uma troca de mensagem entre dois processos, para transmitir uma *struct* que agrega dados heterogêneos. (Ver final do capítulo.)

4.3.2. Tipo de Dados Definidos pelo Usuário

Apesar de `MPI_Pack` ser uma opção interessante, ele requer o uso de uma área reservada na memória para empacotar os dados. Uma boa alternativa é fornecer ao MPI apenas um descritor, uma função de percurso do tipo composto, que possa ser usada na implementação de uma comunicação para vasculhar uma variável daquele tipo e recuperar os componentes de cada campo. Seria possível copiar diretamente para os *buffers* internos usados pela MPI, ou para a camada de rede caso não haja "buferização" interna; melhor ainda, o mesmo descritor poderia ser usado do lado do receptor. Tal mecanismo acaba com o risco do programador desempacotar de forma inconsistente os dados. É também mais eficiente do que o empacotamento.

Isso é possível no MPI graças à definição de `MPI_Datatype` s pelo usuário, complementando os `MPI_Datatype` s básicos padrões do MPI. Isso pode se fazer,

Listing 4.5: Uso de `MPI_Pack` e `MPI_Unpack`.

```
struct task_desc {
   int idf;
   double values[SIZE];
};

int main(int argc, char* argv[]) {
   int rank, i, packpos, test;
   int* packbuf;
   int packsize = (SIZE*sizeof(double)+sizeof(int))*2;
   struct task_desc tarefa;
   MPI_Status st;

   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   if (rank == 0) {
      tarefa.idf = 1;
      for (i=0 ; i<SIZE; i++) tarefa.values[i] = i;
      packpos = 0;
      packbuf = (int*)malloc(packsize*sizeof(int));
      MPI_Pack(&tarefa.idf, 1, MPI_INT, packbuf,
                  packsize, &packpos, MPI_COMM_WORLD);
      MPI_Pack(&tarefa.values, SIZE, MPI_DOUBLE,
          packbuf, packsize, &packpos, MPI_COMM_WORLD);
      MPI_Send(&packpos, 1, MPI_INT, 1, SIZE,
                                       MPI_COMM_WORLD);
      MPI_Send(packbuf, packpos, MPI_PACKED, 1, TASK,
                                       MPI_COMM_WORLD);
   }
   else {
      MPI_Recv(&packpos, 1, MPI_INT, 0, SIZE,
                                  MPI_COMM_WORLD, &st);
      MPI_Recv(packbuf, packpos, MPI_PACKED, 0, TASK,
                              MPI_COMM_WORLD, &status);
      packpos = 0 ;
      MPI_Unpack(packbuf, packsize, &packpos,
              &tarefa.idf, 1, MPI_INT, MPI_COMM_WORLD);
      MPI_Unpack(packbuf, packsize, &packpos,
                      tarefa.values, SIZE, MPI_DOUBLE,
                                       MPI_COMM_WORLD);
   }
   return(0);
}
```

no caso do exemplo anterior, pelo `MPI_Type_create_struct(int, int, MPI_Aint, MPI_Datatype, MPI_Datatype *)`, cujos parâmetros representam:

1 o número de campos da *struct* que está sendo descrita;

2 3 vetores, cada qual com tantas entradas quantos campos especificados no primeiro parâmetro. Para cada um desses três vetores, a entrada i indica sucessivamente:

- **2.1** o tamanho (ou seja o número de elementos) do campo i;
- **2.2** o deslocamento, a partir do endereço inicial da variável, para se chegar ao campo i;
- **2.3** o `MPI_Datatype` básico dos elementos no campo i (caso não seja um tipo básico, deve-se definir primeiro um novo `MPI_Datatype` associado a este campo).

3 o último parâmetro é um ponteiro para um `MPI_Datatype`, o qual conterá, na saída, o nome do novo `MPI_Datatype` definido.

Para calcular o deslocamento (terceiro parâmetro), deve-se usar a primitiva `MPI_Get_Address(void*, MPI_Aint*)`, que retorna numa variável de tipo `MPI_Aint`, o endereço da variável apontada pelo primeiro argumento. Assim, pode-se determinar o endereço do início de uma *struct* (ou seja o endereço da variável inteira), o endereço de cada campo da variável, e por substração o deslocamento esperado.

Na Listagem 4.6, (ver final do capítulo) o trecho entre as linhas 20–28 é um exemplo do uso da primitiva `MPI_Type_create_struct` para definir um novo `MPI_Datatype` chamado `mpi_task_desc` (declarado na linha 13), que descreve uma *struct* tal qual a mencionada anteriormente.

Na linha 26, o `mpi_task_desc` é definido como sendo o descritor de uma *struct* de dois campos (primeiro parâmetro), sendo que seus tamanhos, seus deslocamentos a partir do início da *struct* e tipos básicos estão nos três vetores `size`, `desloc` e `tipo`. Na linha 20, informa-se os tamanhos de cada um dos dois campos e na linha 21 seu tipo. As linhas 22–25 efetuam o cálculo dos deslocamentos respectivos de cada campo.

Uma vez criado, o descritor de tipo deve ser efetivado por uma chamada a `MPI_Type_commit`, de acordo com a linha 28. Depois disso, e conforme ilustrado a partir da linha 31, pode-se usar este novo `MPI_Datatype` para trocar mensagens que envolvam uma variável de tipo `struct task_desc`, exatamente da mesma forma como se usaria um `MPI_Datatype` básico. Nota-se, por exemplo, o `MPI_Send` da linha 34.

Listing 4.6: Uso de `MPI_Datatype` definido pelo usuário.

```
struct task_desc {
  int idf;
  double values[SIZE];
};
int main(int argc, char* argv[]) {
  struct task_desc tarefa;
  MPI_Aint start, field1, field2;
  MPI_Datatype mpi_task_desc;
  MPI_Aint desloc[2]; /* uma entrada por campo */
  int size[2];        /* em 'task_desc' */
  MPI_Datatype tipo[2];
  int rank, i, test = 1;
  MPI_Status st;
  MPI_Init(&argc, &argv);
  size[0] = 1; size[1] = SIZE;
  tipo[0] = MPI_INT ; tipo[1] = MPI_DOUBLE;
  MPI_Get_address(&tarefa, &start);
  MPI_Get_address(&(tarefa.idf), &field1);
  MPI_Get_address(&(tarefa.values), &field2);
  desloc[0] = field1-start; desloc[1] = field2-start;
  MPI_Type_create_struct(2, size, desloc, tipo,
                                    &mpi_task_desc);
  MPI_Type_commit( &mpi_task_desc );
  /* usando o novo MPI_Datatype 'mpi_task_desc' */
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if (rank == 0) {
    tarefa.idf = SIZE;
    for (i =0 ; i<SIZE ; i++) tarefa.values[i] = i;
    MPI_Send(&tarefa, 1, mpi_task_desc, 1, TASK,
                                   MPI_COMM_WORLD);
    printf("Processo␣0:␣Envio␣da␣tarefa␣ok.\n");
  } else {
    MPI_Recv(&tarefa, 1, mpi_task_desc, 0, TASK,
                                MPI_COMM_WORLD, &st);
    printf("Processo␣1:␣recebeu␣a␣tarefa␣%d\n",
                                         tarefa.idf);
    for (i=0 ; i<SIZE ; i++)
      test &= (tarefa.values[i] == i) ;
    if (test) printf("␣Conteudo␣ok.\n");
    else printf("␣Erro␣no␣conteudo!\n");
  }
}
```

4.4. Criação Dinâmica de Processos

O principal foco do MPI é a comunicação, mas a eficiência de tais operações passa por cuidados com a criação e o gerenciamento dos processos. Nesse sentido, MPI possibilita dois modelos de criação de processos: *(i)* estático e *(ii)* dinâmico. O primeiro deles, originalmente suportado no padrão MPI-1 [GRO 94] com exclusividade, cria todos os processos de uma aplicação no início de sua execução incluindo-os no comunicador global `MPI_COMM_WORLD`. O segundo, definido a partir do padrão MPI-2 [GRO 99], permite a criação e a finalização cooperativa de processos depois que uma aplicação MPI tenha iniciado sua execução. Esta seção destaca como é prevista a criação dinâmica e a comunicação entre processos criados dinâmica e estaticamente.

4.4.1. Criando Processos Dinamicamente

Um processo pode ser criado em tempo de execução através da primitiva:

```
int MPI_Comm_spawn(
 char *command,     --> nome do binário do novo processo
 char *argv[],       --> argumentos para o novo processo
 int maxprocs,   --> número de processos a serem criados
 MPI_Info info,      --> informações para runtime system
 int root,             --> identificador do processo pai
 MPI_Comm comm,     --> intracomunicador do processo pai
 MPI_Comm *intercomm, --> intercomunicador: pai e filho
 int array_of_errcodes[]             --> vetor de erros
)
```

Numa visão geral, o `MPI_Comm_spawn` usará o binário cujo nome foi definido em `command`, passando-lhe os argumentos de `argv` e criando tantos processos quanto definido em `maxprocs`. Informações sobre a localização dos processos podem ser passadas em `info` através de pares `(chave, valor)`. O identificador e o intracomunicador do processo que chama a primitiva são também passados como argumento. Convencionalmente, o processo que chama o `MPI_Comm_spawn` é denominado *pai* e os processos que são criados em tempo de execução são chamados de *filhos*. Como retorno da primitiva há, além de um vetor de erros refletindo o estado dos processos filhos, um intercomunicador entre o processo pai e seu(s) filho(s). O `MPI_Comm_spawn` é uma operação coletiva sobre o intracomunicador `comm` e somente retornará após os processos filhos executarem o `MPI_Init`. Uma vez que os filhos tenham sido criados e que existe um intercomunicador ligando-os com o processo pai, eles podem obter esse intercomunicador através da primitiva `MPI_Comm_get_parent(MPI_Comm *parent)`. Assim, pai e filhos podem trocar mensagens mutuamente.

Para exemplificar a criação dinâmica de processos, tomaremos por base o calculo da sequência de Fibonacci definido por:

$$\text{fib}(n): \begin{cases} \text{if } n < 2 \rightarrow \text{fib}(n) = n \\ \text{else fib}(n) = \text{fib}(n-1) + \text{fib}(n-2) \end{cases}$$

o qual pode ser resolvido recursivamente da seguinte forma (cabe ressaltar que embora essa não seja a forma mais eficiente de calcular Fibonacci, já que repete muitas computações, ela é um exemplo didático, largamente utilizado, com um algoritmo simples e que atende as necessidades desse curso):

```
int fib(int n) {
   if (n < 2) return n;
   else {
      int x, y;
      x = fib(n-1);
      y = fib(n-2);
      return(x + y);
   }
}
```

A Listagem 4.7 (ver final do capítulo) mostra uma implementação de Fibonacci usando processos dinâmicos. Nele, as chamadas recursivas são substituídas pela criação de 2 novos processos para executar $fib(n-1)$ e $fib(n-2)$. Cada novo processo repete este procedimento até que se tenha um valor de n menor que 2. Por questões de espaço, foi omitido o código do processo que inicia a execução da aplicação, criando dinamicamente o primeiro processo para calcular $fib(n)$.

Ao iniciar sua execução, o processo irá obter o intercomunicador para comunicações com seu pai e irá testar o valor de n recebido como argumento (parâmetro `argv` do `MPI_Comm_spawn`). Se $n < 2$, o processo filho envia o valor de n a seu pai, já que, neste caso $fib(n) = n$ (`MPI_Send` na linha 18) e finaliza sua execução (`MPI_Finalize` linha 40). Caso contrário o processo filho se tornará pai de 2 novos processos (linhas 19 a 38) para executar $fib(n-1)$ e $fib(n-2)$ (os valores de n passados aos filhos é definido nas linhas 21 e 25). Em seguida, o processo pai bloqueia sua execução a espera do resultado dos filhos (`MPI_Recv` nas linhas 29 e 32). Quando o resultado dos filhos estiver disponível, o pai computa o seu valor de $fib(n)$ (linha 35) e o retorna ao seu pai (`MPI_Send` na linha 37) finalizando assim sua execução (`MPI_Finalize` linha 40).

A Figura 4.2 ilustra a execução do código da Listagem 4.7 para $fib(4)$. Nela, as elipses representam os processos e o primeiro deles tem $n = 4$, o qual criará 2 novos processos para calcular $fib(3)$ e $fib(2)$ (as execuções do `MPI_Comm_spawn` estão representadas pelas setas descendentes). As criações de processos se repetem até que $n < 2$. A partir daí, os processos retornam seus respectivos valores de $fib(n)$ a seus pais (as trocas de mensagens estão representados pelas setas ascendentes).

A primitiva `MPI_Comm_spawn_multiple (int count, char *array_of_commands[], char **array_of_argv[], int array_of_maxprocs[], MPI_Info array_of_info[], int root, MPI_Comm comm, MPI_Comm *intercomm, int array_of_errcodes[])` atende aos casos em que é necessário criar dinamicamente múltiplos processos a partir de

Listing 4.7: Fibonacci implementado com processos dinâmicos.

```
1  int main(int argc, char **argv){
2    long n, fibn, x, y;
3    int rank;
4    MPI_Comm ch_comm[2];
5    MPI_Comm parent;
6    int err[1];
7
8    MPI_Init(&argc, &argv);
9    MPI_Comm_get_parent(&parent);
10   MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
11   argv += 1;
12   n = atol(argv[0]);
13   if (n < 2) {
14     /* devolve o valor de n: fib(n) = n para n < 2 */
15     MPI_Send(&n, 1, MPI_LONG, 0, 1, parent);
16   } else {
17     /* cria um novo processo para calcular fib(n-1) */
18     sprintf(argv[0], "%ld", (n - 1));
19     MPI_Comm_spawn("fib", argv, 1, MPI_INFO_NULL, rank,
20                    MPI_COMM_SELF, &ch_comm[0], err);
21     /* cria um novo processo para calcular fib(n-2) */
22     sprintf(argv[0], "%ld", (n - 2));
23     MPI_Comm_spawn("fib", argv, 1, MPI_INFO_NULL, rank,
24                    MPI_COMM_SELF, &ch_comm[1], err);
25     /* bloqueia aguardando fib(n-1) */
26     MPI_Recv(&x, 1, MPI_LONG, MPI_ANY_SOURCE, 1,
27                      ch_comm[0], MPI_STATUS_IGNORE);
28     /* bloqueia aguardando fib(n-2) */
29     MPI_Recv(&y, 1, MPI_LONG, MPI_ANY_SOURCE, 1,
30                      ch_comm[1], MPI_STATUS_IGNORE);
31     /* calcula fib(n) = fib(n-1) + fib(n-2) */
32     fibn = x + y;
33     /* retorna fib(n) a seu pai */
34     MPI_Send(&fibn, 1, MPI_LONG, 0, 1, parent);
35   }
36   /* finaliza sua execucao */
37   MPI_Finalize();
38 }
```

binários diferentes ou múltiplos processos com o mesmo binário mas passando múltiplos conjuntos de argumentos. Ele é idêntico ao `MPI_Comm_spawn` exceto por apresentar múltiplas especificações, cuja quantidade é determinada em `count`.

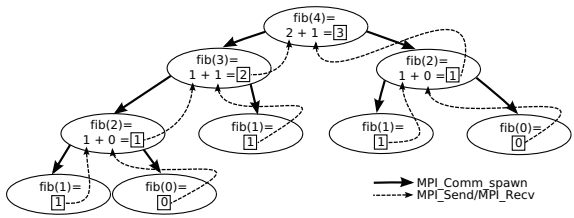


Figura 4.2: $fib(4)$ com processos dinâmicos: setas descendentes representam os `MPI_Comm_spawn` e as ascendentes comunicações.

4.4.2. Estabelecendo Comunicação

Como vimos na seção anterior, o retorno da primitiva `MPI_Comm_spawn` é um intercomunicador entre pai e filhos. Assim, o padrão de comunicação entre processos dinâmicos recai em um modelo hierárquico: há um canal de comunicação de pai para filho e vice-versa. Para os casos em que existe a necessidade de troca de mensagens entre processos que não possuem uma relação pai/filho, o MPI possibilita a criação de uma relação cliente/servidor. O estabelecimento de um canal de comunicação entre dois grupos de processos que não compartilham um comunicador é uma operação coletiva porém assimétrica. Um grupo de processos indica seu interesse em aceitar conexões (servidor), enquanto que outro grupo conecta-se a ele (cliente) [MPI 2009].

O servidor abre uma porta para receber conexões pelo `MPI_Open_port (MPI_Info info, char *port_name)`. Em seguida, ele publica um par `(nome_do_serviço, nome_da_porta)` para que os clientes possam conectar-se a ele – `MPI_Publish_name(char *service_name, MPI_Info info, char *port_name)`. Então, o servidor bloqueia a espera de conexões executando `MPI_Comm_accept(char *port_name, MPI_Info info, int root, MPI_Comm comm, MPI_Comm *newcomm)`. Já o cliente irá pesquisar pela porta previamente aberta pelo servidor com `MPI_Lookup_name(char *service_name, MPI_Info info, char *port_name)`. Então, o cliente estabelece conexão com o servidor em `MPI_Comm_connect(char *port_name, MPI_Info info, int root, MPI_Comm comm, MPI_Comm *newcomm)`. Ao final deste procedimento existe um intercomunicador ligando cliente e servidor (`newcomm` retornado tanto em `MPI_Comm_accept` quanto em `MPI_Comm_connect`). A Figura 4.3 ilustra as chamadas as primitivas detalhadas nesse parágrafo para estabelecer um intercomunicador cliente/servidor.

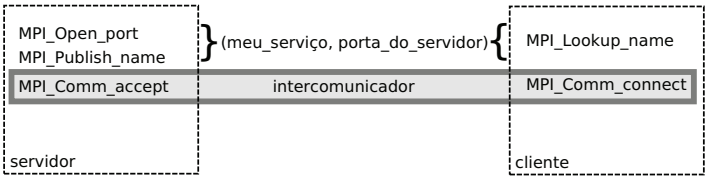


Figura 4.3: Estabelecendo uma conexão cliente/servidor.

4.5. Conclusão

Neste capítulo foram apresentados quatro recursos avançados do MPI.

O agrupamento em comunicadores diferentes de `MPI_COMM_WORLD` é fundamental quando o programa MPI escala até executar centenas de processos. Cabe lembrar que os clusters mais poderosos do top500 atual integram centenas de milhares de cores. Um exemplo típico de uso é a implementação avançada de algoritmos matriciais, onde há necessidade de distribuir as matrizes por blocos. Isso envolve em geral comunicações globais entre os processos que armazenam linhas de blocos, ou colunas de blocos. É clássico definir um comunicador por linha ou coluna.

As comunicações não bloqueantes são imprescindíveis, uma vez que o envio de dados entre processos, por definição, implica num sobrecusto devido ao paralelismo quando comparado com o caso sequencial. Para minimizar este sobrecusto, a única solução é tramitar as comunicações enquanto cálculos são executados na CPU.

Os tipos de dados de usuário são necessários, uma vez que é rara uma aplicação que apenas se limite a usar dados de tipos básicos. Ao comparar a situação com o caso da programação paralela, limitar-se a comunicar inteiros ou números com ponto flutuante significa voltar a programar com linguagens e estruturas de dados da década de 70. Qualquer programa que use, por exemplo, `structs` para agrupar tipos heterogêneos, árvores, grafos, pontos geométricos, deverá usar tais tipos de dados abstratos.

Por fim, a criação dinâmica de processos tem se tornado fundamental para dar conta da necessidade de maleabilidade das aplicações paralelas, seja em máquinas heterogêneas e/ou dinâmicas (Grids, clusters sob demanda), seja em máquinas paralelas compartilhadas, onde a carga de cada CPU não é predizível. Outro aspecto que impulsiona o uso da criação dinâmica de processos é visando o desenvolvimento de aplicações segundo o modelo de paralelismo de tarefas. Este modelo de paralelismo é melhor descrito no capítulo seguinte.

Apesar da dificuldade extra de programação que esses recursos do MPI impõem, é importante dominá-los para poder usá-los quando cabível.

4.6. Bibliografia

[GRO 94] GROPP, W.; LUSK, E.; SKJELLUM, A. **Using MPI**: Portable Parallel Programming with the Message Passing Interface. Cambridge, Massachusetts, USA: MIT Press, 1994.

[GRO 99] GROPP, W.; LUSK, E.; THAKUR, R. **Using mpi-2 advanced features of the message-passing interface**. Cambridge, Massachusetts, USA: The MIT Press, 1999.

[MPI 2009] MPI Forum. **MPI: a message-passing interface standard**. disponível em http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf – último acesso em dezembro 2009.