Message Passing Interface

Nicolas Maillard
nicolas@inf.ufrgs.br

Instituto de Informática
Universidade Federal do Rio Grande do Sul

Curso de extensão Programação Paralela
para Arquiteturas Multicores
I.I. / INTEL

Bibliografia

- `http://www.mpi-forum.org`
- Gropp, William *et al.*, **Using MPI**, MIT Press.
- Gropp, William *et al.*, **Using MPI-2**, MIT Press.
- Snir, M. *et al.*, Dongarra, J., **MPI: The Complete Reference**.

Interface de Passagem de Mensagens

Message Passing Interface

- Histórico:
  - PVM (Parallel Virtual Machine)
  - 1995 — MPI 1.1
  - 1997 — MPI 1.2
  - 1998 — MPI-2
- 2 principais distribuições livres: LAM-MPI (www.lam-mpi.org) e MPI-CH (www-unix.mcs.anl.gov/mpi/mpich/).
- Distribuições de vendedores.
- “MPI is as simple as using 6 functions and as complicated as a user wishes to make it.”

Filosofia de MPI

- Single Program Multiple Data;
  - Vários processos executam todos o mesmo fluxo de instruções;
  - Cada um dos processos é identificado por seu rank.
- Troca de Mensagens:
  - existem primitivas especiais para comunicações “via rede”;
  - oposto a um modelo de memória compartilhada.
- Acrescentar linguagens squenciais via biblioteca
  - Novos tipos de dados (MPI_Status, MPI_Communicator,...)
  - C, C++, Fortran.
  - Java (?)

Seis instruções mágicas

- MPI_Init — Inicializa os processos.
- MPI_Finalize — Finaliza os processos.
- MPI_Comm_size — determina o número de processos executando.
- MPI_Comm_rank — determina o rank de um processo.
- MPI_Send — Manda uma mensagem.
- MPI_Recv — Recebe uma mensagem.

MPI_Init

- Declaração (em C): `int MPI_Init(int*, char***);`
- Usado para inicializar os processos participando à execução, as estruturas de dados e para repassar os argumentos do `main` a todos os processos.
- Exemplo de chamada: `MPI_Init(&argc, &argv);`
- Usa-se no início do programa!

MPI_Finalize

- Declaração (em C): `int MPI_Finalize();`
- Libera os recursos, termina com o programa concorrente.
- Exemplo de chamada: `MPI_Finalize();`
- Usa-se no fim do programa!

MPI_Comm_size

- Declaração (em C): `int MPI_Comm_size(MPI_Communicator, int*);`
- Determina o número de processos em execução dentro do *Communicator* MPI.
- Exemplo de chamada: `MPI_Comm_size(MPI_COMM_WORLD, &p)`.
- Usa-se em qualquer lugar do programa — em geral no início.

MPI_Comm_rank

- Declaração (em C): `int MPI_Comm_rank(MPI_Communicator, int*);`
- Determina o número (rank) do processo em execução dentro do *Communicator* MPI. O rank varia de 0 a $p - 1$.
- Exemplo de chamada: `MPI_Comm_rank(MPI_COMM_WORLD, &r)`.
- Usa-se em qualquer lugar do programa — em geral no início, para personalizar o comportamento do programa em função do rank.

Meu primeiro programa MPI

- Veja /Ufrgs/Disciplinas/ProgParalelaPad/hello.c
- mpicc / mpirun no LAM.

MPI_Send

- `int MPI_Send(void*, int, MPI_Datatype, int, int, MPI_Comm)`.
- Manda o conteúdo de um *buffer* do processo corrente para um processo destino.
- O buffer é determinado:
  - pelo tipo de dados no buffer (MPI_Datatype);
  - por seu tamanho (número de itens no buffer);
- A mensagem é identificada por um tag.
- Exemplo de chamada: MPI_Send(&work, 1, MPI_INT, rank_dest, WORKTAG, MPI_COMM_WORLD);

MPI_Recv

- `int MPI_Recv(void*, int, MPI_Datatype, int, int, MPI_Comm, MPI_Status*)`.
- Recebe o conteúdo de uma mensagem em um *buffer* do processo corrente, vindo de um processo fonte.
- O buffer é determinado:
  - pelo tipo de dados no buffer (MPI_Datatype);
  - por seu tamanho (número de itens no buffer);
- A mensagem é identificada por um tag.
- Existe um *Status* que possibilita a recuperação de informações sobre a mensagem após sua recepção.
- Exemplo de chamada: MPI_Recv(&result, 1, MPI_DOUBLE, 1, tag, MPI_COMM_WORLD, &status);

Meu segundo programa MPI

- Veja /Ufrgs/Disciplinas/ProgParalelaPad/ping-pong.c
- Obs: só roda com $p \geq 2$! (PÉSSIMO EXEMPLO!)
- Obviamente, cada processo deve alocar/inicializar apenas as variáveis de que ele precisará, sem fazer o trabalho para as demais.

Observações & Dicas (1)

- MPI != Mestre/escravo.
- Esquema freqüente: testar a paridade do rank para diferenciar o comportamento.
- Os tags devem ser pre-definidos pelo programador e servem para evitar a colisão entre mensagens. Existe a constante MPI_ANY_TAG.
- O buffer é contíguo na memória. Se os dados a serem transmetidos estão numa estrutura espalhada, deve haver primeiro “serialização” dos mesmos. Cuidado com ponteiros. . .
- O buffer é de tamanho fixo. Caso se transmita valores em número desconhecido à compilação, deve-se usar 2 mensagens:
  - 1 primeira, de 1 int, para mandar o tamanho do buffer;
  - 1 segunda, contendo o buffer.

Observações & Dicas (1)

- MPI != Mestre/escravo.
- Esquema freqüente: testar a paridade do rank para diferenciar o comportamento.
- Os tags devem ser pre-definidos pelo programador e servem para evitar a colisão entre mensagens. Existe a constante MPI_ANY_TAG.
- O buffer é contíguo na memória. Se os dados a serem transmetidos estão numa estrutura espalhada, deve haver primeiro “serialização” dos mesmos. Cuidado com ponteiros. . .
- O buffer é de tamanho fixo. Caso se transmita valores em número desconhecido à compilação, deve-se usar 2 mensagens:
  - 1 primeira, de 1 int, para mandar o tamanho do buffer;
  - 1 segunda, contendo o buffer.

Observações & Dicas (1)

- MPI != Mestre/escravo.
- Esquema freqüente: testar a paridade do rank para diferenciar o comportamento.
- Os tags devem ser pre-definidos pelo programador e servem para evitar a colisão entre mensagens. Existe a constante MPI_ANY_TAG.
- O buffer é contíguo na memória. Se os dados a serem transmetidos estão numa estrutura espalhada, deve haver primeiro “serialização” dos mesmos. Cuidado com ponteiros. . .
- O buffer é de tamanho fixo. Caso se transmita valores em número desconhecido à compilação, deve-se usar 2 mensagens:
  - 1 primeira, de 1 int, para mandar o tamanho do buffer;
  - 1 segunda, contendo o buffer.

Observações & Dicas (1)

- MPI != Mestre/escravo.
- Esquema freqüente: testar a paridade do rank para diferenciar o comportamento.
- Os tags devem ser pre-definidos pelo programador e servem para evitar a colisão entre mensagens. Existe a constante MPI_ANY_TAG.
- O buffer é contíguo na memória. Se os dados a serem transmetidos estão numa estrutura espalhada, deve haver primeiro “serialização” dos mesmos. Cuidado com ponteiros. . .
- O buffer é de tamanho fixo. Caso se transmita valores em número desconhecido à compilação, deve-se usar 2 mensagens:
  - 1 primeira, de 1 int, para mandar o tamanho do buffer;
  - 1 segunda, contendo o buffer.

Observações & Dicas (1)

- MPI != Mestre/escravo.
- Esquema freqüente: testar a paridade do rank para diferenciar o comportamento.
- Os tags devem ser pre-definidos pelo programador e servem para evitar a colisão entre mensagens. Existe a constante MPI_ANY_TAG.
- O buffer é contíguo na memória. Se os dados a serem transmetidos estão numa estrutura espalhada, deve haver primeiro “serialização” dos mesmos. Cuidado com ponteiros. . .
- O buffer é de tamanho fixo. Caso se transmita valores em número desconhecido à compilação, deve-se usar 2 mensagens:
  - 1 primeira, de 1 int, para mandar o tamanho do buffer;
  - 1 segunda, contendo o buffer.

Observações & Dicas (2)

- No MPI_Recv, pode-se usar o valor MPI_ANY_SOURCE como rank do processo de envio. Pode-se receber alguma-coisa de qualquer um, com qualquer tag! Neste caso, usa-se o MPI_Status para recuperar as informações:
  - stat.MPI_TAG
  - stat.MPI_SOURCE
- O MPI_Recv é bloqueante: o processo vai ficar esperando até ter recebido alguma coisa. Cuidado com Deadlocks.
- o MPI_Send, logicamente, é bloqueante. Nas implementações, em geral ele não é (vide exemplo).

Comunicações coletivas — definições

- São todas as comunicações que implicam mais de dois processos.
  - Oposto às comunicações “ponto-a-ponto”.
- Sua prototipação no MPI permite a otimização, na biblioteca, da comunicação.
  - Nem sempre a implementação é tão eficiente assim...
  - Depende (a) da instalação; (b) da arquitetura.
- Exemplos: broadcast (one-to-all), all-to-all, all-to-one (reduce), split...
- Lembra muito do HPF!

Sincronização global: barreira

- Pode ser preciso sincronizar todos (ou parte de) os processos.
  - Garantir que x processos passaram em um dado ponto do programa;
  - Medir um tempo.
- MPI_Barrier(MPI_Comm) — MPI_Barrier(MPI_COMM_WORLD);
- Obviamente, isso leva a perda de tempo!

Broadcast (difusão)

- Um processo ”mestre” difunde uma mensagem para x outros.
  - Vide aulas passadas para vários algoritmos (árvore binária, árvore de Fibonacci. . .);
  - obs: nada obriga a mandar para todos os $p - 1$ outros processos!
- MPI_Bcast(void*, int, MPI_Datatype, int, MPI_Comm)
- Manda os dados contidos no *buffer*, de tipo ’datatype’, a partir do processo ’mestre’, para todos os processos.

Exemplo: Bcast de 1 int e de n *doubles*

```
MPI_Bcast(&size, 1, MPI_INT, 0, MPI_COMM_WORLD);
#define RAIZ 0
MPI_Bcast(dados, n, MPI_DOUBLE, RAIZ, MPI_COMM_WORLD);
```

- Observações:
  - Todos os processos chamam MPI_Bcast!
  - É uma sincronização global.
  - O buffer muda de semântica conforme o rank

Juntar dados

- Cada processo, inclusive o “mestre”, manda dados para o mestre, que os armazena na ordem dos ranks.
- Exemplo típico de uma facilidade: pode-se fazer a mesma coisa com *x* MPI_Send/MPI_Recv !
- MPI_Gather(void*, int, MPI_Datatype, void*, int, MPI_Datatype, int, MPI_Comm)
- Argumentos: *buffer* de emissão, buffer de recepção, *rank* do mestre, comunicador.
- O argumento recvcount é o número de elementos recebidos / mandados **por processo**.

Exemplo: Gather de 10 doubles

```
double* sbuf, rbuf;
int mestre = 2;
if (rank == mestre) rbuf = malloc(p*10*sizeof(double));
else sbuf = malloc(10*sizeof(double));
MPI_Gather(sbuf, 10, MPI_DOUBLE, rbuf, 10, MPI_DOUBLE,
mestre, MPI_COMM_WORLD);
```

Variações sobre as comunicações globais

- MPI_Gatherv(. . .);
- MPI_Scatter(. . .);
  - Faz o contrário do Gather!
  - Argumentos: *buffer* de emissão, buffer de recepção, *rank* do mestre, comunicador.
  - Existe também o Scatterv.
- MPI_Allgather
  - Gather + cópia em todos os processos (Gather-to-all).
  - Argumentos: *buffer* de emissão, buffer de recepção, SEM mestre, comunicador.

All-to-All

- Versão mais “avançada” do Allgather, com dados distintos.
- O bloco *j* mandado pelo processo *i* vai ser recebido pelo processo *j* e armazenado no bloco *i* de seu *buffer*.
- MPI_Alltoall(void*, int, MPI_Datatype, void*, int, MPI_Datatype, MPI_Comm)
- Argumentos: *buffer* de emissão, de recepção, comunicador.

Exemplo: MPI_Alltoall de 10 doubles

```
int i,j;
double* sbuf, *rbuf;
rbuf = malloc(p*10*sizeof(double));
sbuf = malloc(p*10*sizeof(double));
MPI_Alltoall(sbuf, 10, MPI_DOUBLE, rbuf, 10, MPI_DOUBLE,
MPI_COMM_WORLD); for (i=0 ; i<p ; i++)
    printf(“Recebi de %d : ”, i);
    for (j=0 ; j<10 ; j++) printf(“%lf”, rbuf[i*10+j]);
```

Reduções

- Para efetuar uma operação comutativa e associativa em paralelo.
- MPI_Reduce(void*, void*, int, MPI_Datatype, MPI_Op, int, MPI_Comm)
- Argumentos: *buffer* de emissão, buffer de recepção, comunicador, tipo de operador.
- O operador pode ser MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD, MPI_LAND, MPI_LOR, etc. . .

Exemplo

```
double* sum_buf, resultado;
rbuf = malloc(p*10*sizeof(double));
sum_buf = malloc(10*sizeof(double));
MPI_Reduce(sum_buf, resultado, 10, MPI_DOUBLE, MPI_SUM, 0,
MPI_COMM_WORLD);
```

Comunicações coletivas: conclusões

- Procedimentos extremamente poderosos;
- Observa-se que são parecidos aos mecanimos de paralelismo de dados (HPF) ou de laços (OpenMP): cf. o Reduce, por exemplo;
- Grande progresso rumo à especificação portável e eficiente do paralelismo;
- Obs: é um dos exemplos onde o MPI pode ser visto como modelo de programação mais avançado (!= soquetes, RMI, threads, . . .).

Comunicações ponto-a-ponto avançadas

- As operações básicas MPI_Send e MPI_Recv são bloqueantes e não locais:
  - não retornam até o usuário poder re-usar o *buffer*;
  - pode-se usar buffers intermediários (ou não);
  - o resultado do send depende do comportamento do processo que efetua o receive.
- Para deixar o usuário controlar exatamente o comportamento do programa, MPI oferece 3 modos de comunicação:
  - buffered: obriga as comunicações a usarem *buffers*, com controle de seu tamanho e uso, bem como do fim da operação;
  - síncrono: usa um Rendez-vous e retorna somente quando o Receive tem sido efetuado;
  - ready: o Send começa somente se o Recv tem começado; se não, retorna com erro.
- Existe só um MPI_Recv, além de MPI_Bsend, MPI_Ssend e MPI_Rsend.

Vários MPI_Send

- MPI_Bsend, MPI_Ssend e MPI_Rsend têm o mesmo perfil como o MPI_Send clássico;
- No caso do Bsend, MPI provê mais primitivas para controlar o uso dos *buffers* intermediários:
  - MPI_Buffer_attach(void* buff, int size): define o espaço apontado por buff como sendo um buffer intermediário;
  - MPI_Buffer_detach(void* buff, int size): libera o espaço apontado por buff de ser um buffer intermediário; essa chamada e bloqueante.

Comunicações não-bloqueantes

- São mecanismos cruciais para obter alto-desempenho, a fim de poder mascarar as comunicações por cálculos;
- MPI usa uma estrutura de dados chamada MPI_Request para identificar as comunicações não bloqueantes e poder verificar seu andamento;
- O perfil das chamadas é igual ao do MPI_Send (ou MPI_Recv), mas com um parâmetro final extra de tipo MPI_Request.
- Pode-se mesclar o caráter não-bloqueante com os modos *buffered*, síncronos ou prontos.
- Exemplo:
  MPI_request* req; MPI_Isend(&work, 1, MPI_INT, rank_dest, WORKTAG, MPI_COMM_WORLD, req) ; MPI_Irecv(&result, 1, MPI_DOUBLE, 1, tag, MPI_COMM_WORLD, req);
- Obs: as chamadas Irecv e ISend allocam e inicializam a Request.

Controle do andamento das comunicações

- É preciso poder testar a conclusão das comunicações não bloqueantes para poder re-aproveitar seus *buffers*.
- Duas possibilidades:
  - MPI_Wait(MPI_Request* req, MPI_Status* stat): retorna apenas quando a primitiva não bloqueante identificada por 'req' terminou. Ao retornar, 'stat' contém as informações relevantes. 'req' é liberada pelo Wait.
  - MPI_Test(MPI_Request* req, int* flag, MPI_Status* stat): testa se a comunicação terminou, sem bloquear. Ao retornar, 'flag' é setado e pode ser testado depois. Se o resultado é positivo, a 'req' é liberada e o 'stat' setado.
- Outras primitivas: MPI_Request_free; MPI_Wait_any; MPI_Test_any (com tabelas de requests).

Controle do andamento das comunicações

- É preciso poder testar a conclusão das comunicações não bloqueantes para poder re-aproveitar seus *buffers*.
- Duas possibilidades:
  - MPI_Wait(MPI_Request* req, MPI_Status* stat): retorna apenas quando a primitiva não bloqueante identificada por 'req' terminou. Ao retornar, 'stat' contém as informações relevantes. 'req' é liberada pelo Wait.
  - MPI_Test(MPI_Request* req, int* flag, MPI_Status* stat): testa se a comunicação terminou, sem bloquear. Ao retornar, 'flag' é setado e pode ser testado depois. Se o resultado é positivo, a 'req' é liberada e o 'stat' setado.
- Outras primitivas: MPI_Request_free; MPI_Wait_any; MPI_Test_any (com tabelas de requests).

Controle do andamento das comunicações

- É preciso poder testar a conclusão das comunicações não bloqueantes para poder re-aproveitar seus *buffers*.
- Duas possibilidades:
  - MPI_Wait(MPI_Request* req, MPI_Status* stat): retorna apenas quando a primitiva não bloqueante identificada por 'req' terminou. Ao retornar, 'stat' contém as informações relevantes. 'req' é liberada pelo Wait.
  - MPI_Test(MPI_Request* req, int* flag, MPI_Status* stat): testa se a comunicação terminou, sem bloquear. Ao retornar, 'flag' é setado e pode ser testado depois. Se o resultado é positivo, a 'req' é liberada e o 'stat' setado.
- Outras primitivas: MPI_Request_free; MPI_Wait_any; MPI_Test_any (com tabelas de requests).

Estruturação dos processos

- MPI provê uma definição abstrata de conjuntos de processos:
  - para encapsular uma biblioteca MPI (espaço de nomes);
  - para possibilitar sincronizações de parte dos processos.
- MPI define grupos, contextos, topologia e comunicadores:
  - **Grupo**: coleção ordenada de processos, cada um tendo seu *rank* local ao grupo.
    - Define as comunicações ponto-a-ponto!
    - Define os particpantes em comunicações coletivas.
    - Default: MPI_GROUP_EMPTY.
  - **Contextos**: espaço de nomes para mensagens;
  - **Topologia**: organização virtual dos processos.
  - **Comunicador**: estrutura de dados que encapsula todo o resto.
    - Intra-comunicador: comunicação interna a um grupo;
    - Inter-comunicador: comunicação entre grupos;
    - Defaults: MPI_COMM_WORLD, MPI_COMM_SELF.
  - Obs: MPI não define o relacionamento rank/hostname.

Estruturação dos processos

- MPI provê uma definição abstrata de conjuntos de processos:
  - para encapsular uma biblioteca MPI (espaço de nomes);
  - para possibilitar sincronizações de parte dos processos.
- MPI define grupos, contextos, topologia e comunicadores:
  - **Grupo**: coleção ordenada de processos, cada um tendo seu *rank* local ao grupo.
    - Define as comunicações ponto-a-ponto!
    - Define os particpantes em comunicações coletivas.
    - Default: MPI_GROUP_EMPTY.
  - **Contextos**: espaço de nomes para mensagens;
  - **Topologia**: organização virtual dos processos.
  - **Comunicador**: estrutura de dados que encapsula todo o resto.
    - Intra-comunicador: comunicação interna a um grupo;
    - Inter-comunicador: comunicação entre grupos;
    - Defaults: MPI_COMM_WORLD, MPI_COMM_SELF.
  - Obs: MPI não define o relacionamento rank/hostname.

Operações com grupos

- MPI_Group_size(MPI_Group g, int* size) retorna o tamanho de um grupo.
- MPI_Group_rank(MPI_Group g, int* rank) retorna o rank dentro de um grupo.
- MPI_Group_translate_ranks(MPI_Group g1, int n, int* ranks1, MPI_Group g2, int* ranks2) traduz os ranks de um grupo para o outro.
- MPI_Comm_group(MPI_Comm comm, MPI_Group* group) retorna o grupo associado ao Comunicador 'comm'. Usado com o MPI_COMM_World!
- União, interseção, inclusão, . . . de grupos são possíveis.
- MPI_Group_free(MPI_Group g) libera o recurso.

Topologia virtual com MPI

- simplifica a programação em alguns casos;
- exemplos: cartesiana; grafo;
- a topologia vem embutida no comunicador!
- MPI_Cart_create(MPI_Comm old_comm, int nbdims, int* dims, int* periodico, int reorder, MPI_Comm* comm_cart);
- reorder: para re-ordenar os processos dentro da nova topologia.
- periodico[i] indica se a dimensão i é periodicamente mapeada.

Topologia virtual com MPI

- simplifica a programação em alguns casos;
- exemplos: cartesiana; grafo;
- a topologia vem embutida no comunicador!
- MPI_Cart_create(MPI_Comm old_comm, int nbdims, int* dims, int* periodico, int reorder, MPI_Comm* comm_cart);
- reorder: para re-ordenar os processos dentro da nova topologia.
- periodico[i] indica se a dimensão i é periodicamente mapeada.

Operações com comunicadores

- MPI_Comm_size(MPI_Comm c, int* size) retorna o tamanho de um comunicador.
- MPI_Comm_rank(MPI_Comm c, int* rank) retorna o rank dentro de um comunicador.
- MPI_Comm_dup(MPI_Comm c, MPI_Comm new_comm) efetua uma copia profunda da estrutura c.
- MPI_Comm_create(MPI_Comm c, MPI_Group g, MPI_Comm* new_comm) cria um novo comunicador a partir do grupo g.
- MPI_Comm_split(MPI_Comm c, int cor, int chave, MPI_Comm* new_comm) cria um novo comunicador, através da partição de c, conforme for o valor de 'cor'. 'chave' permite definir o rank do processo em seu novo comunicador.
  - Útil para computação "Divisão & Conquista".
- MPI_Comm_free(MPI_Comm c) libera o comunicador.

MPI_Datatypes

- MPI define tipos básicos (MPI_INT, MPI_DOUBLE, . . .)
- MPI_Byte (8 bits)
- MPI_PACKED. Usado junto com MPI_Pack(. . .) e MPI_Unpack(. . .):
  - MPI_Pack(void* inbuf, int count, MPI_Datatype dtype, void* outbuf, int outcount, int* position, MPI_Comm comm)
    - Empacota em ’inbuf’ os dados, atualiza “position” e retorna “outbuf”.
  - MPI_Send(outbuf, MPI_Pack_size(outbuf), MPI_PACK, . . .)
  - MPI_Unpack(. . .).

Datatypes complexos

- O usuário pode definir um tipo complexo para MPI:
  - Define um conjunto de tipos básicos;
  - Define o deslocamento de cada tipo básico.
  - $\{(T_0, D_0), (T_1, D_1), \ldots, (T_n, D_n)\}$.
- A assinatura define a seqüência de tipos usados;
- MPI_Type_struct(count, array_of_block_length, array_of_deslocamento, array_of_types, new_type);

MPI-2

- “Nova´´ norma de MPI que inclui dinamicidade.
  - data de 1998!
  - (muito) recentemente implementada.
- Inclui:
  - criação dinâmica de processos;
  - comunicação entre processos dinamicamente criados;
  - RMA;
  - E/S paralelas.

MPI_Comm_spawn

- Possibilidade de criar processos durante a execução do programa;
- Possibilidade de estabalecer comunicações entre 2 programas MPI rodando em paralelo.
- Usa extensivamente os comunicadores e cria um inter-comunicador para comunicação entre o grupo criador e o grupo criado.
- MPI_Comm_spawn(char* comando, char* argumentos[], int maxprocs, MPI_Info info, int root, MPI_Comm com, MPI_Comm* inter_comm, int erros[]);
- dispara 'maxprocs' processos para executar o 'comando' (= outro programa MPI). Os argumentos devem ter sentido no processo 'root'. Na saída, 'inter_comm' contém um comunicator para possibilitar a comunicação entre os novos processos e os antigos.

Comunicação entre processos em MPI-2

- O mecanismo básico fica igual: troca de mensagens.
- Mensagens ponto-a-ponto: usa-se o inter-comunicator retornado pelo MPI_Comm_Spawn.
  - A maior diferença é que o processo fonte pode acessar *ranks* de processos dependendo do número no segundo comunicador!
- Mensagens coletivas: é uma novidade do MPI-2. Funciona da mesma forma.
- Pode-se também usar MPI_Intercomm_merge para fusionar os comunicadores em um só, e se comunicar normalmente nele depois.

Conectar aplicações MPI-2

- Pode-se usar mecanismos tipo cliente/servidor entre (grupos de) processos MPI-2.
- Lado servidor: Criação de porta e aceitação de conexão a ela;
  - MPI_Open_port(MPI_Info, char* nome-porta) ;
  - MPI_Comm_accept(char* nome-porta, MPI_Info, int root, MPI_Commm, MPI_Comm* inter_com) — chamada coletiva e bloqueante que retorna um inter-comunicador.
- Lado cliente: Conexão a uma porta.
  - MPI_comm_connect(char* nome-porta, MPI_Info info, int root, MPI_Comm comm, MPI_Comm* inter_comm);
- uma vez que foi estabelecida a conexão, pode-se usar os Send/Recv clássicos através do inter-comunicator.
- Obs: existe um MPI_Publish_name e um MPI_Lookup_name.

Acesso distante a memória (RMA)

- Alternativa à troca de mensagens!!!
- Outro modelo de programação paralela
  - Escritas/leituras diretas, remotas, na memória de um outro processo (sem sua cooperação).
- MPI_Win_create para definir uma *janela* de memória
- MPI_Put, MPI_Get para escrever e ler. accessível.

E/S paralelas

- Solução clássica com MPI-1.2: um processo faz as E/S para/de um arquivo. . .
- Melhora: cada processo escreve para um arquivo diferente.
- Solução com MPI-2:
  - MPI_File em lugar de FILE;
  - MPI_File_open — MPI_File_write MPI_File_read — MPI_File_close
  - usa um comunicator para definir quais processos acessam ao arquivo;
  - escreve o conteúdo de um *buffer* no arquivo;
  - Usa MPI_File_view para especificar o deslocamento no arquivo onde cada processo vai escrever.
    - MPI_File_set_view(arquivo, rank*BUFSIZE*sizeof(char), MPI_CHAR,. . .)