www.lev.ieav.cta.br/erad2011

(data limite de submissões: 15/maio)

MPI:
Tipos Derivados
(3. Point to point communication)

Tipos Derivados

- Todas as comunicações em MPI transferem dados de um mesmo tipo e contíguos em memória a partir do endereço base *bfr*. Não contemplam:
  1. Dados não contíguos
     - coluna de uma matriz em C
     - linha de uma matriz em Fortran
  2. Dados de tipos distintos

- Há soluções simples, como
  1. Copiar dados para/de *bfr* contíguo e
  2. Uma mensagem por tipo

- Solução MPI: crie um novo tipo MPI, derivado dos existentes
  – Use o tipo na comunicação

Tipos Derivados

- MPI possui múltiplas operações para criar tipos derivados
- Cada tipo (intrínseco ou derivado) é representado por um inteiro em Fortran e por um MPI_Datatype em C ("*handle*")
- Qualquer tipo pode ser utilizado em comunicações MPI, não apenas os intrínsecos
- Antes de usar um novo tipo, seu uso futuro deve ser comunicado ao MPI, invocando MPI_Type_commit

Criar e Destruir Tipos Derivados

- Comunicar a criação de um novo tipo:

  MPI_Type_commit (newtype, ierr)

  integer, intent(inout) :: newtype

  – MPI "compila" uma representação interna para o novo tipo

- Comunicar a destruição de um novo tipo:

  MPI_Type_free (newtype, ierr)

  integer, intent(out) :: newtype

  – MPI remove a representação interna para o novo tipo. Ao término desta operação, *newtype* tem o valor MPI_DATATYPE_NULL

- São operações locais
  – Tipos conhecidos apenas nos processos MPI que os criam

Tipos Derivados

- Um tipo MPI é uma seqüência de pares
  <tipo, deslocamento>
  - Tipo é um tipo MPI anteriormente definido
  - Deslocamento é a distância na memória, em bytes, de um endereço base (tipicamente o endereço do *bfr*)
  - Por exemplo, MPI_COMPLEX corresponde a
    - {(MPI_REAL, 0), (MPI_REAL, 4)}
  - Um tipo é um mapa de memória a partir de endereço base
- Nos operações MPI com tipos derivados, deslocamento é expresso ora em bytes ora em unidades do tipo básico
  - Depende da função MPI invocada
- Há muitas funções MPI para definir novos tipos; veremos apenas duas.

Tipo com dados não contíguos

- Comunicar blocos igualmente espaçados de elementos contíguos
- Todos os blocos de mesmo tipo e tamanho

```
MPI_Type_vector (count, blocklength, stride, oldtype, newtype, ierr)
integer, intent(in) :: count ! Quantos blocos
integer, intent(in) :: blocklength! Quantos elementos por bloco
integer, intent(in) :: stride ! Espaçamento entre blocos
integer, intent(in) :: oldtype
integer, intent(out) :: newtype
```

- Espaçamento em unidades do tipo básico (não em bytes!)

Tipo com dados não contíguos

MPI_Type_vector (count, blocklength, stride, oldtype, newtype, ierr)
integer, intent(in) :: count ! Quantos blocos
integer, intent(in) :: blocklength! Quantos elementos por bloco
integer, intent(in) :: stride ! Espaçamento entre blocos

- Exemplos em Fortran sobre matriz real (10,10):
  - Diagonal
    - MPI_TYPE_VECTOR(10, 1, 11, MPI_DIAG, MPI_REAL, ierr)
  - Dois primeiros elementos de cada coluna da matriz:
    - MPI_TYPE_VECTOR(10, 2, 10, MPI_DUAS, MPI_REAL, ierr)

- Exemplo em C
  - Uma coluna de matriz de inteiros com 10 linhas:
    - MPI_Type_vector(10, 1, 10, MPI_INT, mpi_col)

Exemplo em C

- Seja a matriz A[4][4] com "Ghost Zone":

```
(0,0) (0,0) (0,0) (0,0) (0,0) (0,0)
(0,0) (1,1) (1,2) (1,3) (1,4) (0,0)
(0,0) (2,1) (2,2) (2,3) (2,4) (0,0)
(0,0) (3,1) (3,2) (3,3) (3,4) (0,0)
(0,0) (4,1) (4,2) (4,3) (4,4) (0,0)
(0,0) (0,0) (0,0) (0,0) (0,0) (0,0)
```

- Particione a matriz por colunas em 2 processos MPI:

```
#define nRow 4
#define nColLocal 2
int A[nRow+2][nColLocal+2];
```

- Comunique a “Ghost Zone” (coluna em C) entre os processos

Estado Inicial

- Matriz A[4][4] particionada com "Ghost Zone“ desatualizada:

Código

```
MPI_Type_vector(nRow, 1, nColLocal+2, MPI_INT, &mpi_col);
MPI_Type_commit(&mpi_col);

if (rank == 0) {
  MPI_Isend(&A[1][nColLocal  ], 1, mpi_col, 1, tag1,
  MPI_COMM_WORLD, &req[0]);
  MPI_Irecv(&A[1][nColLocal+1], 1, mpi_col, 1, tag2,
  MPI_COMM_WORLD, &req[1]);
}
else {
  MPI_Irecv(&A[1][0], 1, mpi_col, 0, tag1, MPI_COMM_WORLD, &req[0]);
  MPI_Isend(&A[1][1], 1, mpi_col, 0, tag2, MPI_COMM_WORLD, &req[1]);
}
MPI_Waitall(2, req, status);

MPI_Type_free(&mpi_col);
```

Resultado

- Matriz A[4][4] particionada com "Ghost Zone“ atualizada:

proc 0:				proc 1:			
(0,0)	(0,0)	(0,0)	(0,0)	(0,0)	(0,0)	(0,0)	(0,0)
(0,0)	(1,1)	(1,2)	(1,3)	(1,2)	(1,3)	(1,4)	(0,0)
(0,0)	(2,1)	(2,2)	(2,3)	(2,2)	(2,3)	(2,4)	(0,0)
(0,0)	(3,1)	(3,2)	(3,3)	(3,2)	(3,3)	(3,4)	(0,0)
(0,0)	(4,1)	(4,2)	(4,3)	(4,2)	(4,3)	(4,4)	(0,0)
(0,0)	(0,0)	(0,0)	(0,0)	(0,0)	(0,0)	(0,0)	(0,0)

Tipo com dados de tipos distintos

- Blocos de elementos contíguos
- Cada bloco com elementos de único tipo
- Tipo pode variar entre blocos
- Número de elementos pode variar entre blocos
- Deslocamento do primeiro elemento do bloco varia entre blocos

```
MPI_Type_struct (count, blocklengths, displacements, types,
    newtype, ierr)
integer, intent(in) :: count ! Quantos blocos
integer, intent(in) :: blocklengths(count)! Quantos elementos por bloco
integer, intent(in) :: displacements(count)! Espaçamento entre blocos
integer, intent(in) :: types(count)! Tipo em cada bloco
integer, intent(out) :: newtype
```

Empacotar Dados

- Outra forma de enviar dados de tipos distintos sem criar um novo tipo MPI é copiá-los em um array de bytes
- MPI_Pack faz a cópia, um tipo por vez
- MPI_Unpack desfaz a cópia, um tipo por vez
- Transferência no tipo MPI_PACKED

Outras Funções

- Há muitas outras funções para tipos derivados
- Consulte o padrão quando necessário

MPI:
Grupos e Comunicadores
(5. Groups, Contexts and Communicators)

Comunicador

- MPI implementa o conceito de comunicador
- Um comunicador é um conjunto ordenado de *n* processos, enumerados de *0* a *n-1* (com *n > 0*)
- O comunicador cria um contexto (grupo de processos) no qual ocorrem comunicações. O comunicador enumera os processos do grupo, permitindo sua identificação e a gerência das mensagens entre processos
- Um processo pode pertencer a múltiplos comunicadores simultaneamente; comunicações em um comunicador são independentes das comunicações em outro comunicador

Grupos e Comunicadores: Motivação

- Suponha:
  - Um conjunto de *p* processadores com um processo por processador
  - Duas aplicações paralelas independentes, funcionando corretamente para qualquer número de processos
- Como garantir correção de uma nova aplicação composta pela execução simultânea das duas anteriores?
  - Por exemplo, com *m* processos na primeira e *p-m* processos na segunda
- O que ocorre se a primeira aplicação contiver a invocação

  CALL MPI_BARRIER(MPI_COMM_WORLD, ierr)

Grupos e Comunicadores: Motivação

- Para garantir a execução simultânea de duas aplicações, é necessário definir contextos de comunicação (comunicadores), envolvendo grupos de processos.

- Um grupo é um conjunto ordenado de processos
  - Cada processo em um grupo é univocamente denotado por um identificador (o inteiro *rank*)
  - Identificadores são inteiros consecutivos e começam em 0

- MPI implementa o conceito de grupo por meio de objetos opacos, acessíveis por "handles"
  - Um mesmo processo pode estar em múltiplos grupos

- Grupos não são suficientes para resolver o problema?
  - Não é imediato distinguir *comunicador* de *grupo*

Distinguir Comunicador de Grupo

- Suponha:
  - que um comunicador é apenas um grupo de processos
- Admita:
  - que uma aplicação invoque a rotina de uma biblioteca paralela que resolva um sistema de equações lineares
  - todos os processos participando nas duas computações
  - que imediatamente antes de invocar a rotina, a aplicação emita um IRECV para antecipar a recepção de uma mensagem enviada após a solução do sistema
  - que o IRECV utilize MPI_ANY_SOURCE e MPI_ANY_TAG
- Como garantir que o IRECV não receberá a primeira mensagem emitida pela rotina da biblioteca?

Grupos e Comunicadores: Motivação

- Um comunicador é um grupo de processos e um contexto de comunicação
- O mesmo grupo de processos pode ser utilizado em múltiplos comunicadores.
- Comunicações em um comunicador são independentes e não interferem em comunicações de outro comunicador.
  - Resolve o problema da biblioteca de rotinas
- Comunicadores são objetos opacos, acessíveis por "handle"

Criação de Comunicador

```
MPI_Comm_create (comm, group, newcomm, ierr)
integer, intent(in) :: comm ! Comunicador pré-existente
integer, intent(in) :: group ! Grupo (subconjunto do grupo de comm)
integer, intent(out) :: newcomm ! Novo comunicador
```

- Cria novo comunicador com todos os processos no grupo.
- Observe que:
  - O grupo é um subconjunto do grupo de comm
  - Tipicamente, comm é MPI_COMM_WORLD
    - Para garantir que grupo é um subconjunto

Operações com Comunicadores

- Comunicadores pré-existentes:
  - MPI_COMM_WORLD: Todos os processos
  - MPI_COMM_SELF: Apenas este processo
  - MPI_COMM_NULL: Comunicador inválido

- Liberação de comunicador:

  MPI_COMM_FREE (comm, ierr)

  integer, intent(inout) :: comm ! Comunicador a liberar
  - comm recebe MPI_COMM_NULL

- Tamanho do comunicador e rank deste processo:

  MPI_COMM_SIZE(comm, size, ierr)

  MPI_COMM_RANK(comm, rank, ierr)
  - Velhos conhecidos

Obter Grupo de um Comunicador

```
MPI_Comm_group (comm, group, ierr)
integer, intent(in) :: comm ! Comunicador pré-existente
integer, intent(out) :: group ! Grupo deste comunicador
```

- Retorna o grupo do comunicador

- Grupos pré-existentes
  - MPI_GROUP_EMPTY: Grupo sem elementos
  - MPI_GROUP_NULL: Grupo inválido

Criar um Grupo

- Para criar um grupo a partir de ranks do grupo atual:
  MPI_Group_incl(group, n, ranks, newgroup, ierr)
  integer, intent(in) :: group ! Pre-existente
  integer, intent(in) :: n ! Quantos processos no novo grupo
  integer, intent(in) :: ranks(n) ! Quais processos
  integer, intent(out) :: newgroup ! Novo grupo
  - Todos os ranks tem que fazer parte de group
  - Todos os ranks tem que ser distintos

- Ou ainda
  MPI_Group_excl(group, n, ranks, newgroup, ierr)
  - Exclui os processos do grupo

Exemplo

- Dados n processos (n>2), crie dois comunicadores:
  - Comm_io, com os dois primeiros processos, para I/O
  - Comm_comp, com os demais processos, para computação

CALL MPI_COMM_GROUP(MPI_COMM_WORLD, group_all, ierr)

rank_io = (/0, 1/)

CALL MPI_GROUP_INCL (group_all, 2, rank_io, group_io, ierr)

CALL MPI_GROUP_EXCL (group_all, 2, rank_io, group_comp, ierr)

CALL MPI_COMM_CREATE (MPI_COMM_WORLD, group_io, comm_io, ierr)

CALL MPI_COMM_CREATE (MPI_COMM_WORLD, group_comp, comm_comp, ierr)

Outras Características

- Há muitas outras operações em grupos e comunicadores
  - Quando necessário, estude o padrão
- Conceito interessante é o de inter-comunicador:
  - A junção de dois comunicadores por meio de funções de MPI específicas forma um novo comunicador (o inter-comunicador)
  - Sends e Receives no inter-comunicador
- Comunicadores são extensíveis
  - É possível acrescentar atributos a comunicadores

MPI:
Topologia de Processos
(6. Process Topologies)

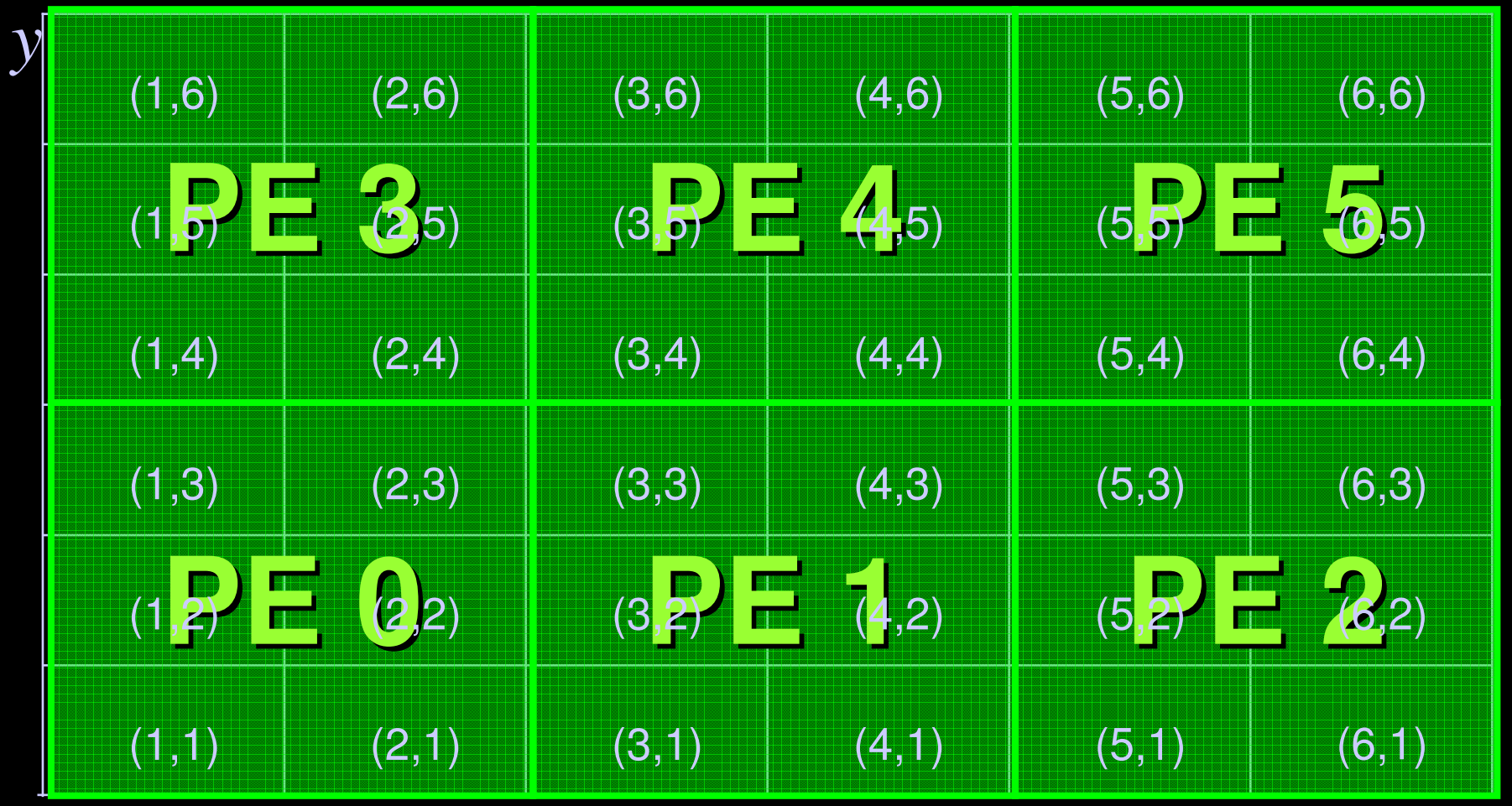
Topologia de Processos

- MPI impõe topologia unidimensional de processos
  - Enumeração unidimensional de processos (“rank”)
  - Todas as comunicações utilizam a enumeração unidimensional para identificar processos
  - Adequado quando o paralelismo é unidimensional, mesmo que o problema seja multidimensional
- Muitas aplicações exploram paralelismo multidimensional
  - Modelos meteorológicos dividem a superfície em retângulos

Superfície de Modelo Meteorológico

Paralelismo 1D com 6 Processadores

Superfície de Modelo Meteorológico



Paralelismo 2D com 6 Processadores

Dividir n processos em m dimensões

MPI_Dims_create (nproc, ndims, dims, ierr)

integer, intent(in) :: nproc	Numero total de processos
integer, intent(in) :: ndims	Quantas dimensões
integer, intent(inout) :: dims(ndims)	Processos por dimensão
integer, intent(out) :: ierr	Código de retorno

- Tenta decompor nProc em nDims fatores
- Retorna fatores em Dims (**Π** Dims = nDims)
- Detalhes no padrão

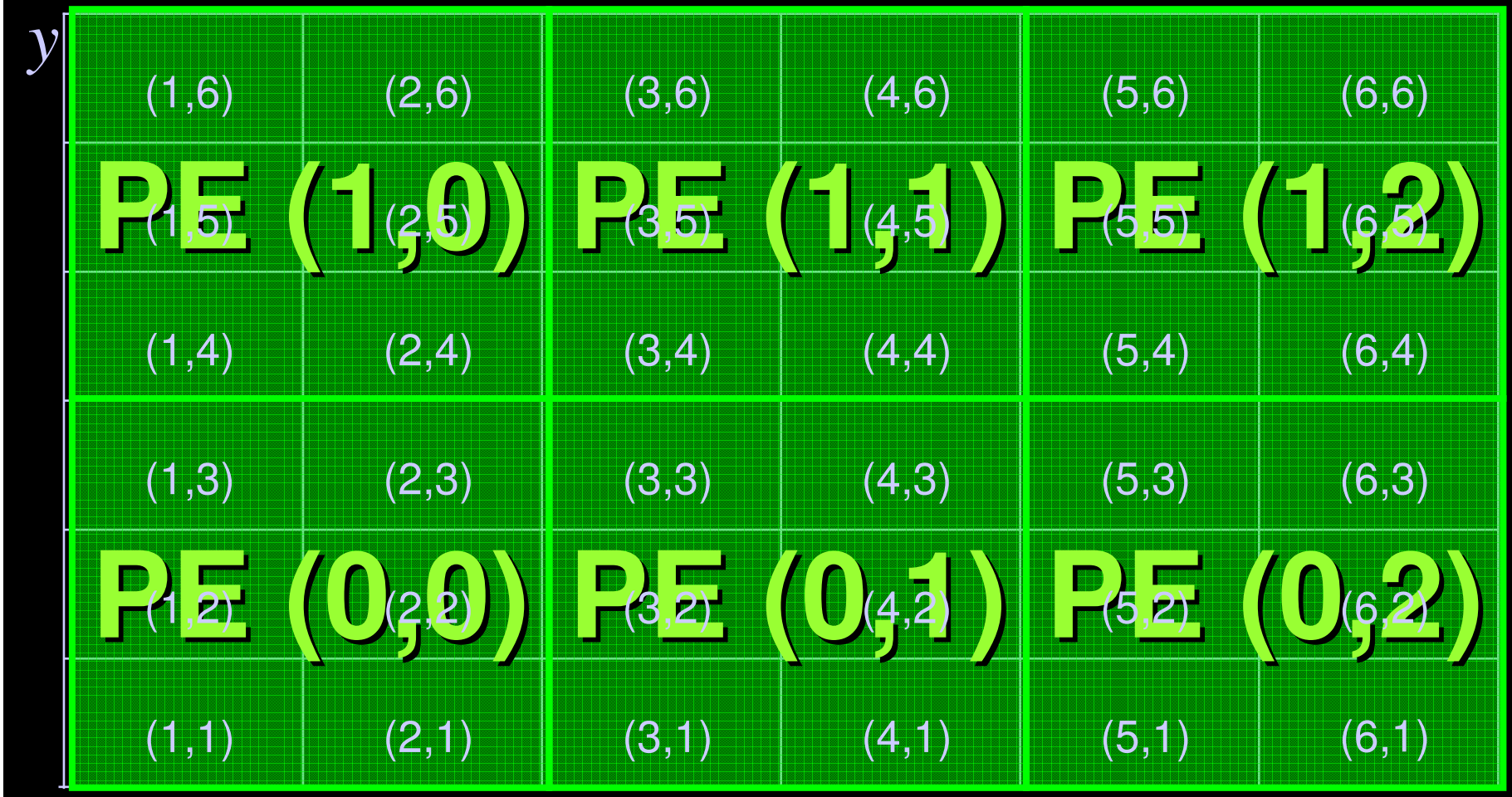
Exemplo (nDims=2)

nProc	Dims(1)	Dims(2)
2	2	1
3	3	1
4	2	2
5	5	1
6	3	2
7	7	1
8	4	2
9	3	3
10	5	2

Topologia de Processos

- Quer paralelismo 1D, quer 2D, a enumeração dos processos MPI é 1D (rank é um inteiro)
- Que tal enumerar 2D?

Superfície de Modelo Meteorológico



Paralelismo 2D com 6 Processadores (numerado 2D)

Topologia de Processos

- Facilita programação?
  - SEND/RECV usa numeração 1D (*rank* é um escalar!)
- É preciso manter as duas numerações:
  - Numeração 2D para facilitar a programação
  - Numeração 1D para comunicação
- MPI apresenta topologia *n*-dimensional
  - Implementada no comunicador
  - Operações de conversão entre rank e topologia
  - Operações para obtenção de vizinhos na topologia
- MPI também apresenta topologia de grafos
  - Funcionalidades similares à topologia n-dimensional
- Quando necessário, estude o padrão

MPI: SUMÁRIO

Sumário

- MPI-1 implementa paralelismo de troca de mensagens
- Padrão MPI permite programas paralelos portáteis
- Conjunto rico de 128 operações
  - Não cobrimos todas as operações, mas
  - Apresentamos todos os conceitos básicos
- Programação de baixo nível
  - Poderosa, mas difícil
  - MPI é conhecido como “assembly” de programação paralela
- Proposta mais utilizada para programas paralelos
  - Embora existam outras propostas interessantes

MPI-2

MPI-2

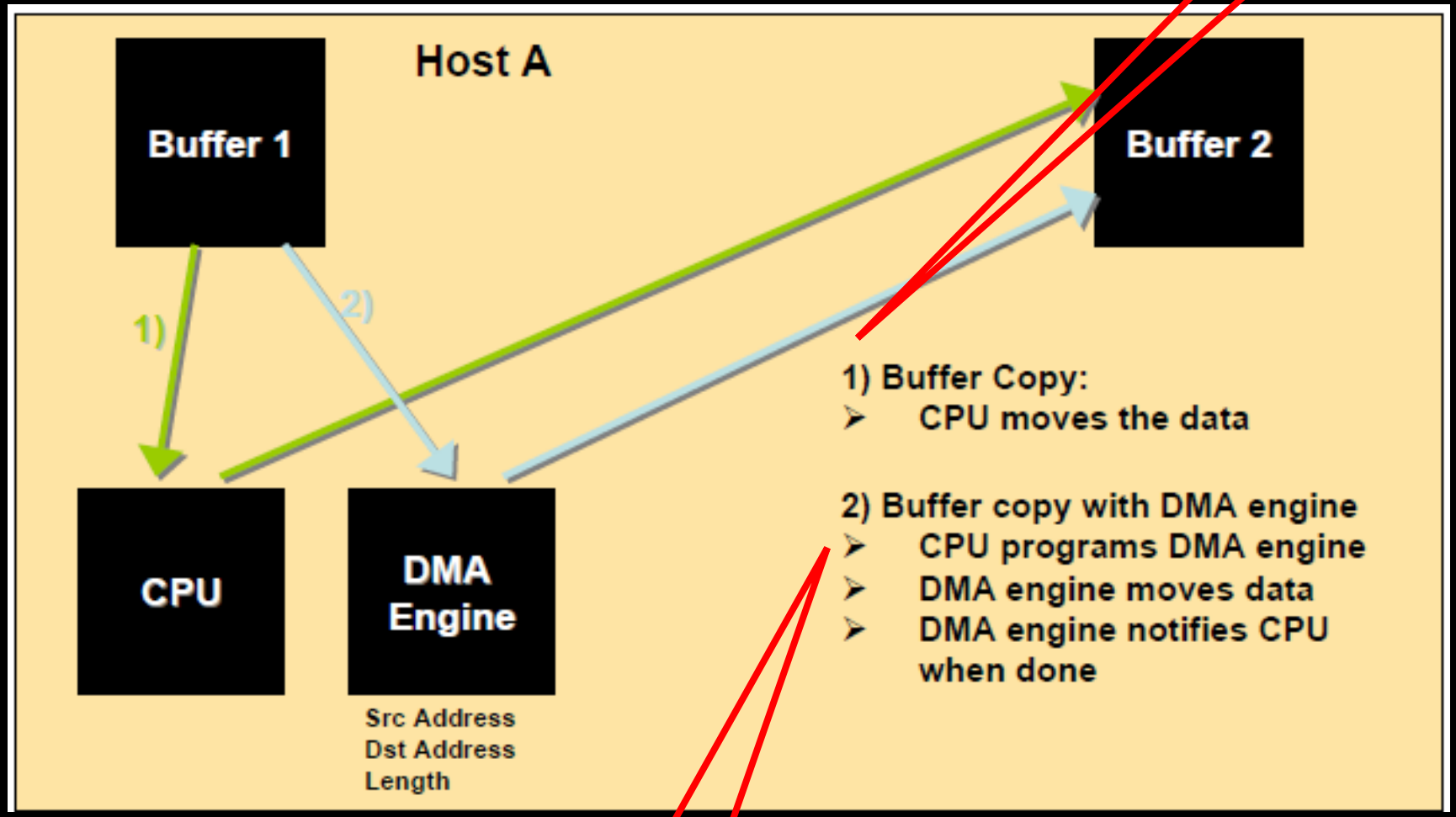
- Extensão de MPI-1 gerada em 1997
  - Indisponibilidade de implementações atrasou disseminação
  - Hoje disponível nas implementações (sw livre) MPICH e OpenMPI
- Extensões:
  - Processos dinâmicos
    - criação e destruição dinâmica de processos
  - I/O paralelo
    - operações de I/O paralelo padronizadas
  - Comunicação unilateral
    - um único processo define a comunicação ponto a ponto
- Veremos apenas comunicação unilateral

Comunicação unilateral

- Na forma de comunicação que vimos, um processo envia explicitamente (send) e outro recebe explicitamente (recv)
  - Denominada two-sided communication
- Nesta nova forma, um processo envia (ou recebe) dados do outro sem a interferência explícita do outro
  - Denominada one-sided communication
- Para tanto, um processo tem que citar a memória do outro
  - Para enviar dados, o processo que emite a comunicação (envio) deve citar onde colocar os dados na memória do outro
  - Para receber dados, o processo que emite a comunicação (recepção) deve citar onde retirar os dados da memória do outro
- Motivação: redes rápidas permitem escrever/ler diretamente na memória de outros nós via RDMA (Remote Direct Memory Access) sem a participação da outra CPU

Cópia com DMA

(Direct Memory Access na mesma placa)

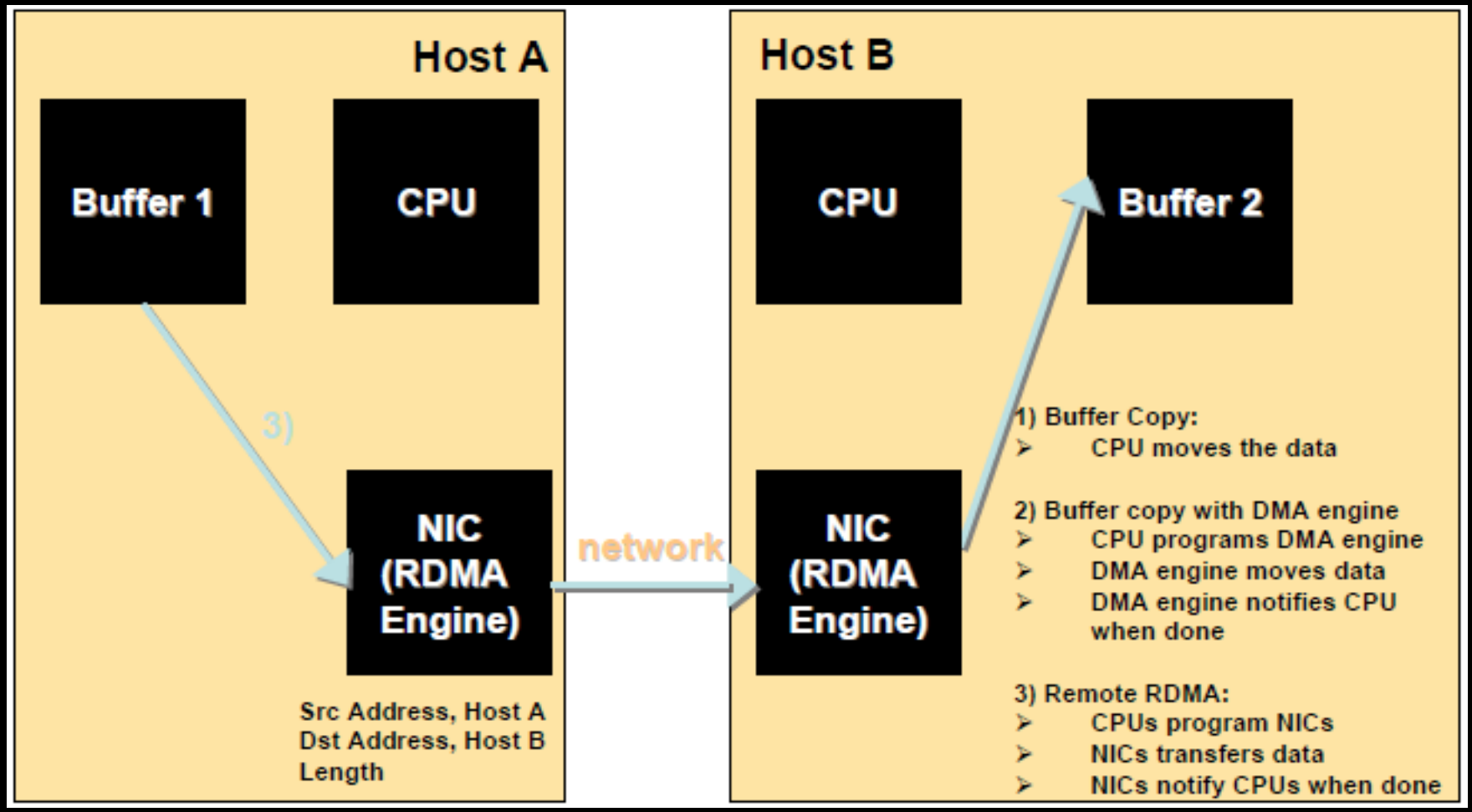


Fonte:

The case for RDMA, RDMA Consortium

Cópia com RDMA

(Remote Direct Memory Access entre placas conectadas pela rede via NIC – Network Interface Card - específicos)



Fonte:

The case for RDMA, RDMA Consortium

Utilidade de comunicação unilateral

- Em tese, simplifica programação
  - Um único lado (processo) define a comunicação
- Potencialmente mais eficiente – pode evitar os buffers de send/receive impostos pela semântica de MPI, existentes mesmo em máquinas com RDMA
- Semântica separa comunicação de sincronismo
  - Já separados em comunicações não bloqueantes:
    - SEND e RECV bloqueantes contém comunicação e sincronismo
    - Mas ISEND e IRECV realizam comunicação enquanto WAIT sincroniza
- Entretanto, cria condições de corrida (*race conditions*)
  - Pois duas entidades atuam sobre a mesma memória

Comunicação Unilateral em MPI-2

- Processos MPI definem um trecho de sua memória visível aos outros processos do comunicador
  - Janela de comunicação
- Processos sincronizam, indicando o início de trecho do programa com comunicação entre os processos
- Processos MPI escrevem (“put”), lêem (“get”) ou acumulam (“accumulate”) dados na janela de comunicação dos outros processos
- Processos sincronizam, indicando o término de trecho do programa com comunicação entre os processos

Janela de Comunicação

- O conceito de janela (window) de memória expõe trecho da memória de um processo a outros
  - necessário para que outros processos possam citar memória alheia
  - também define quais processos podem acessar a janela
  - processos distintos podem expor trechos distintos de memória (cada processo expõe sua janela)

- Em operações de envio/recepção de mensagens, o trecho da memória alheia é especificado por:
  - target_disp: *displacement* da posição inicial da janela alheia
  - target_count: quantos elementos do tipo target_datatype serão enviados
  - target_datatype: tipo MPI no destino

- O trecho de memória alheia obrigatoriamente está contido na janela exposta pelo processo alvo

Janela de Comunicação

MPI_Win_create (

void *base, MPI_Aint size, int disp_unit,

MPI_Info info, MPI_Comm comm, MPI_Win *win)

- Cria uma janela em cada processo do comunicador
- A janela é visível por todos os processos do comunicador
- A janela é a região contígua de memória com *size* bytes iniciada no endereço *base*
- Cada processo cria a sua janela (tamanho e base distintas)
- O conjunto de todas as janelas é armazenado em *win*
- Operação coletiva no comunicador

Janela de Comunicação

MPI_Win_free (MPI_Win *win)

- Destroi o conjunto de janelas indicado por *win*
- Todos os processos no comunicador utilizado na criação da janela devem invocar MPI_Win_free
- Há uma barreira implícita

Escrever em Memória Alheia

```
MPI_Win_Put (
void *origin_addr, int origin_count, MPI_Datatype origin_datatype,
int target_rank,
MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype,
MPI_Win *win)
```

- Escreve na janela exposta pelo processo *target_rank*
- Escreve no trecho da janela definido por (*target_disp*, *target_count*, *target_datatype*)
- Escreve o conteúdo da memória local definido por (*origin_addr*, *origin_count*, *origin_datatype*)
  - memória local não precisa estar na janela
- Operação não bloqueante; o retorno da invocação significa que o envio foi solicitado

Ler de Memória Alheia

MPI_Win_Get (
void *origin_addr, int origin_count, MPI_Datatype origin_datatype,
int target_rank,
MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype,
MPI_Win *win)

- Lê da janela exposta pelo processo *target_rank*
- Lê o trecho da janela definido por (*target_disp*, *target_count*, *target_datatype*)
- Armazena na memória local definida por (*origin_addr*, *origin_count*, *origin_datatype*)
- Operação não bloqueante; o retorno da invocação significa que a recepção foi solicitada

Acumular em Memória Alheia

MPI_Win_Accumulate (
void *origin_addr, int origin_count, MPI_Datatype origin_datatype,
int target_rank,
MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype,
MPI_OP op, MPI_Win *win)

- Executa a operação de redução op (ex. MPI_SUM) na janela alheia
- Posições a enviar e a acumular definidas como em Put e Get
- Operação não bloqueante; o retorno da invocação significa que a recepção foi solicitada

Sincronismo e Coerência

- Put, Get e Accumulate retornam antes do término da comunicação
- É necessária operação de sincronismo que garanta que as comunicações iniciadas já terminaram
- Mas nada impede que o processo “dono” da janela também atue sobre a memória contida na “sua” janela. Logo, dois processos podem atuar sobre a mesma posição de memória, gerando:
  - condições de corrida
  - inconsistências entre memória, cache e registradores, similar à OpenMP (resolvida por *fence* em OpenMP)
- Solução MPI:
  - Operação de sincronismo também garante consistência de memória

Sincronismo e Coerência

- MPI-2 fornece múltiplas operações de sincronismo; veremos apenas MPI_Fence
- Semântica de MPI_Fence:
  - Completa as operações “one-sided” desde a última invocação à MPI_Fence;
  - Coerência de memória:
    - Garante que qualquer escrita local à janela é completada
    - Elimina as cópias locais das variáveis modificadas da janela
- Em suma, sincronismo e coerência em uma mesma operação
  - Não há ordem entre sincronismo e coerência, por isso:
- Entre duas invocações à MPI_Fence, o programa deve conter apenas one-sided ou apenas acessos locais
  - Separa regiões do programa entre acessos locais e remotos

Sincronismo e Coerência

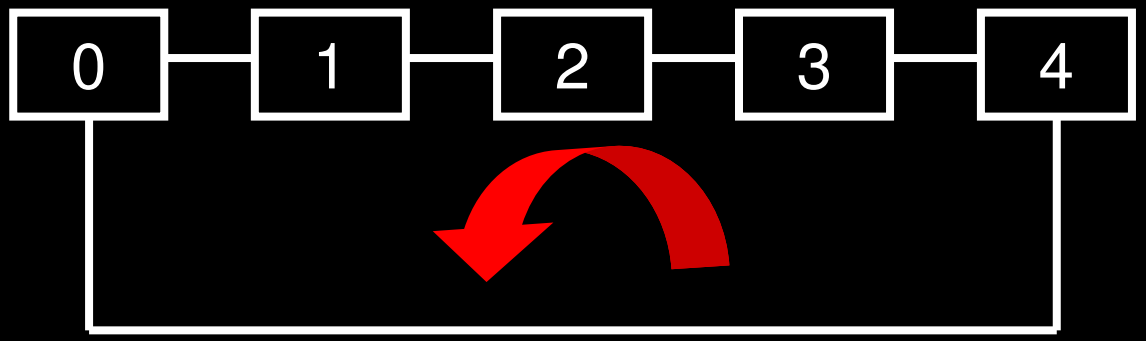
MPI_Win_fence (int assert, MPI_Win *win)

- Operação coletiva sobre todos os processos no comunicador da janela
  - Há barreira implícita
- Completa todas as operações one-sided iniciadas após a última invocação a MPI_Win_fence
- Garante que todas as atualizações à janela (operações locais) sejam completadas antes que qualquer operação one-sided que sucede MPI_Win_fence seja executada e que cópias locais de variáveis alteradas na janela sejam atualizadas
- Uso: invocação a fence delimita regiões:
  - apenas com one-sided
  - apenas com atualizações locais

Condição de Corrida

- Exemplo de condição de corrida sutil:

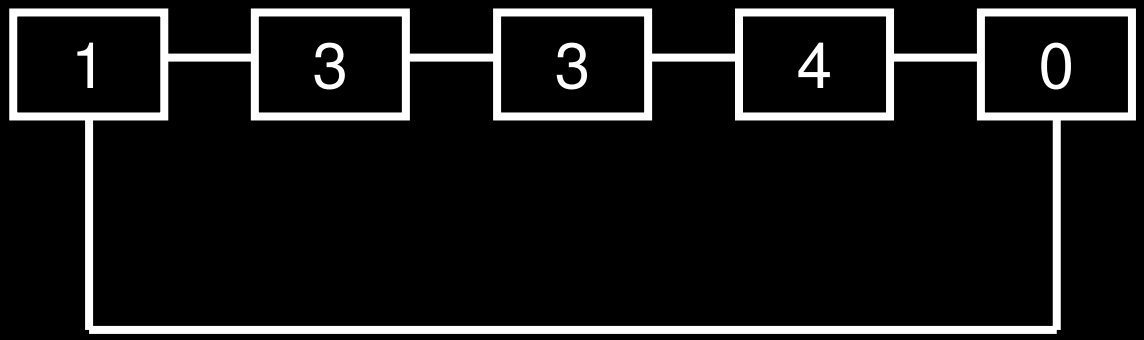
- Codifique um programa em que cada processo MPI envie sua identidade (“rank”) ao seu vizinho de rank imediatamente anterior em um anel



Condição de Corrida

- Suponha a codificação:
  - Cada processo exporta, em sua janela, a variável “otherRank”
  - A variável “otherRank” é inicializada para o rank do próprio processo
  - “MPI_Put” copia otherRank em otherRank na janela do processo vizinho
  - “MPI_Put” é orlado por duas invocações a “MPI_Fence”

- Eis um resultado possível:



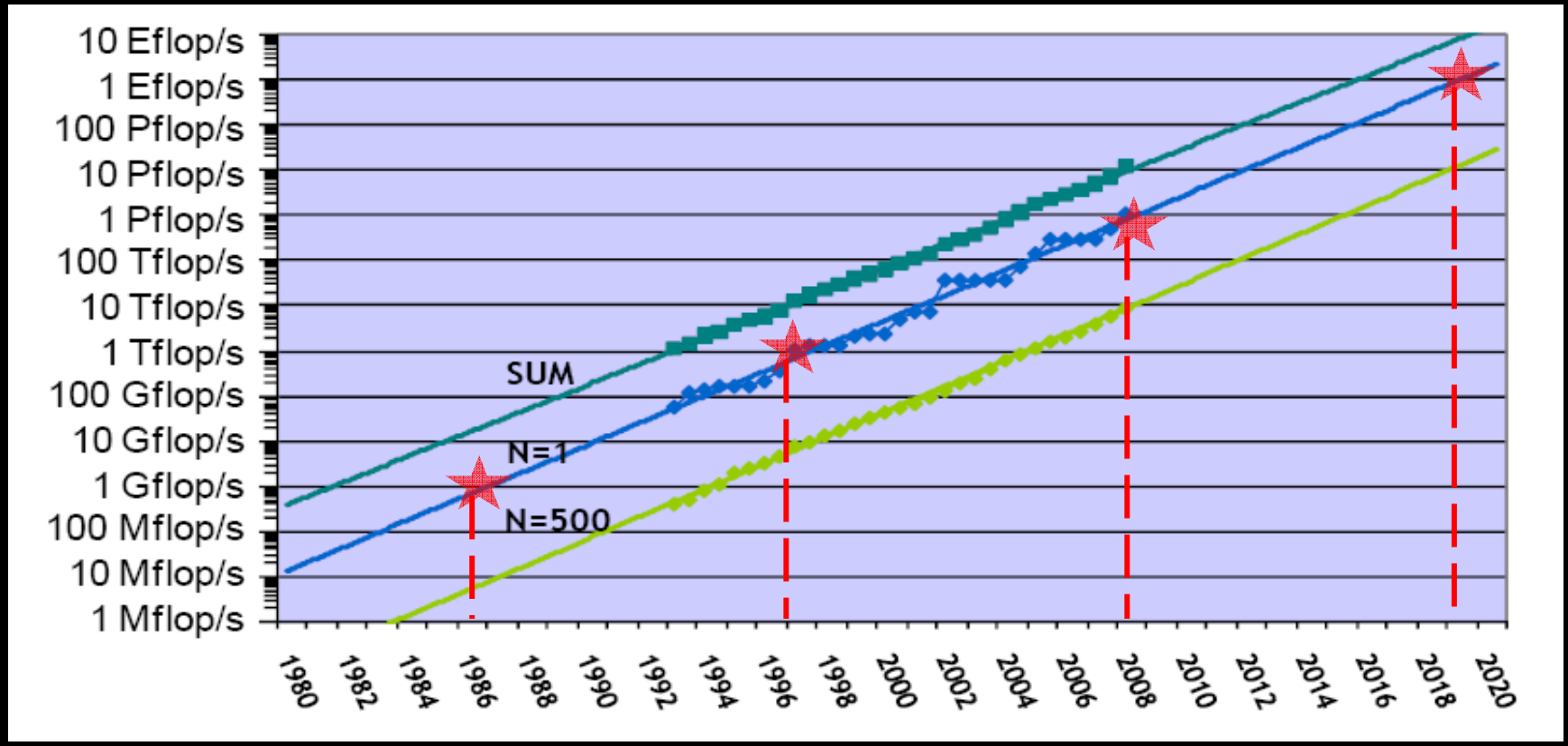
Condição de Corrida

- Pois é possível que o “MPI_Put” do processo 3 no processo 2 atualize otherRank na janela do processo 2 antes do “MPI_Put” do processo 2 enviar otherRank para o processo 1

- Em suma, comunicação unilateral é forma poderosa de comunicação, mas gera problemas de condição de corrida e consistência de memória

Desafios Futuros e seu Impacto em MPI

Extrapolação do Top500 (cópia da Semana 1)



Cray 2	ASCI Red	Roadrunner	??
1 GFlop/s	1 TFlop/s	1 PFlop/s	1 EFlop/s
O(1) Thread	O(10^3) Threads	O(10^6) Threads	O(10^9) Threads

Jack Dongarra, Invited Talk, SIAM 2008 Annual Meeting

Máquinas Exaflop

- Departamentos de Defesa e de Energia Americanos fomentam máquinas de Exaflop em 2018
  - Data advém de extrapolação do Top500
- Limites impostos pelo DARPA à máquina de exaflop:
  - Disponível em 2018
  - Custo da ordem de US$200M
  - Energia limitada a 20MW
- Indústria, Academia e usuários trabalham para atingir esses limites
- O desafio é enorme

Referências à Máquinas Exaflop

- Estudos exploratórios encomendados pelo DARPA:
  - ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems, Peter Kogge, editor, DARPA, 2008
    - Dificuldades de Hardware
  - ExaScale Software Study: Software Challenges in Extreme Scale Systems, Vivek Sarkar, editor, DARPA, 2009
    - Dificuldades de Software
- Reação da academia e da indústria:
  - www.exascale.org
    - Projeto comunitário internacional

Petaflop x Exaflop

Systems	2010	2018	Difference Today & 2018
System peak	2 Pflop/s	1 Eflop/s	O(1000)
Power	6 MW	~20 MW	
System memory	0.3 PB	32 - 64 PB [.03 Bytes/Flop]	O(100)
Node performance	125 GF	1,2 or 15TF	O(10) – O(100)
Node memory BW	25 GB/s [.20 Bytes/Flop]	2 - 4TB/s [.002 Bytes/Flop]	O(100)
Node concurrency	12	O(1k) or 10k	O(100) – O(1000)
Total Node Interconnect BW	3.5 GB/s	200-400GB/s (1:4 or 1:8 from memory BW)	O(100)
System size (nodes)	18,700	O(100,000) or O(1M)	O(10) – O(100)
Total concurrency	225,000	O(billion) [O(10) to O(100) for latency hiding]	O(10,000)
Storage	15 PB	500-1000 PB (>10x system memory is min)	O(10) – O(100)
IO	0.2 TB	60 TB/s (how long to drain the machine)	O(100)
MTTI	days	O(1 day)	- O(10)

Impactos em MPI

Future of MPI

A Presentation at HPC Advisory Council Workshop, Lugano 2011

by

Dhabaleswar K. (DK) Panda
The Ohio State University
E-mail: panda@cse.ohio-state.edu
http://www.cse.ohio-state.edu/~panda

Sayantan Sur
The Ohio State University
E-mail: surs@cse.ohio-state.edu
http://www.cse.ohio-state.edu/~surs

What are the basic design challenges for Exascale Systems?

- DARPA Exascale Report – Peter Kogge, Editor and Lead
- Energy and Power Challenge
  - Hard to solve power requirements for data movement
- Memory and Storage Challenge
  - Hard to achieve high capacity and high data rate
- Concurrency and Locality Challenge
  - Management of very large amount of concurrency (***billion*** threads)
- Resiliency Challenge
  - Low voltage devices (for low power) introduce more faults

How does MPI plan to meet these challenges?

- Power required for data movement operations is one of the main challenges
- Non-blocking collectives
  - Overlap computation and communication
- Much improved One-sided interface
  - Reduce synchronization of sender/receiver
- Manage concurrency
  - Improved interoperability with PGAS (e.g. UPC, Global Arrays)
- Resiliency
  - New interface for detecting failures

Major New Features

(MPI 3 em elaboração)

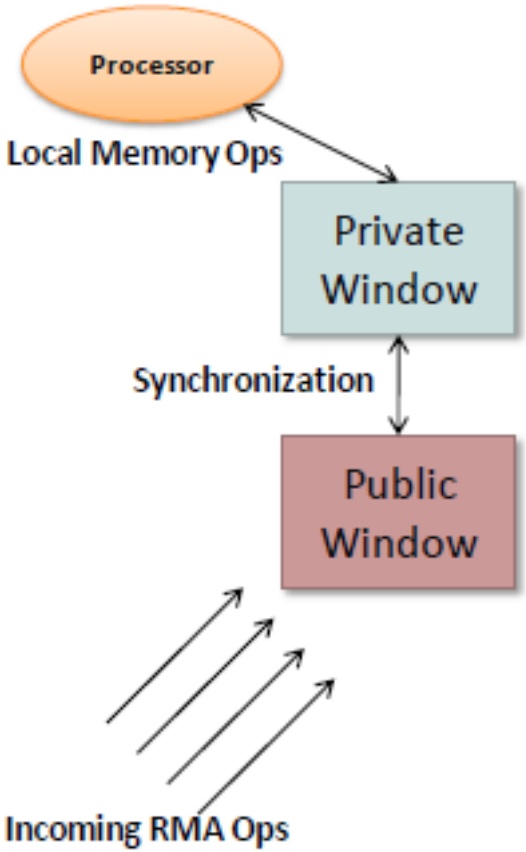
- Non-blocking Collectives
- Improved One-Sided (RMA) Model
- Fault-Tolerance

Non-blocking Collective operations

- Enables overlap of computation with communication
- Removes synchronization effects of collective operations (exception of barrier)
- Completion when local part is complete
- Completion of one non-blocking collective does not imply completion of other non-blocking collectives
- No “tag” for the collective operation
- Issuing many non-blocking collectives may exhaust resources
  - Quality implementations will ensure that this happens for only pathological cases

Improved One-sided Model

- Remote Memory Access (RMA)
- New proposal has major improvements
- MPI-2: public and private windows
  - Synchronization of windows explicit
- MPI-2: works for non-cache coherent systems
- MPI-3: two types of windows
  - Unified and Separate
  - Unified window leverages hardware cache coherence



Highlights of Fault tolerance in MPI-3

- MPI-2: if an error is detected, the state of MPI is undefined
  - Program aborts
- MPI-3: ranks have various states
  - MPI_RANK_STATE_OK
  - MPI_RANK_STATE_FAILED
  - MPI_RANK_STATE_NULL (not yet recognized as failed)
- MPI_Comm_validate
  - Return new unrecognized failures
- MPI_Comm_get_state_info
  - Get status of ranks on comm
- MPI_Comm_set_state_null
  - Recognize failure of a particular rank

Sumário

- Atualmente, exaflop é foco central de muitas pesquisas
  - Consumo de energia
  - Volume de paralelismo
- Padrão MPI-3, em elaboração, tenta contemplar esses desafios
  - Mantendo compatibilidade com o vasto legado de MPI-1
- Principais extensões em discussão:
  - Operações coletivas não bloqueantes
  - Melhoras em comunicação unilateral
  - Tolerância a Falhas