2a. Lista de Exercícios*

1 Comunicação e Roteamento

Roteamento é um tópico que aparece quando estudamos os aspectos referentes a comunicação em um sistema de memória distribuída e em sistemas que utilizam grandes quantidades de memória compartilhada. Se dois nós não estão diretamente conectados ou se um processador não está conectado diretamente ao módulo de memória, como os dados são transmitidos entre os dois? Vamos analisar este problema em sistemas de memória distribuída usando uma rede de interconexão estática. É bom salientar que este problema não é necessariamente solucionado utilizando apenas hardware: muitos sistemas implementam partes de seu roteamento usando software.

O problema de roteamento se subdivide em dois outros subproblemas:

- Se existem múltiplas rotas conectando os dois nós ou processador e memória, como uma rota é escolhida? A mota do caminho mais curto é sempre a escolhida? Muitos sistemas usam um algoritmo de roteamento determinístico para o caminho mínimo. Ou seja, se um nó A se comunica com um nó B, então a rota de comicação utilizada será sempre a mesma, e não existe uma outra rota que utilize menos conexões. Este problema surge quando a rota da comunicação entre A e B passa por outros nós ou *switches*.

- Como os nós intermediários passam as informações? Existem duas abordagens básicas. Vamos supor que o nó A está enviando uma mensagem para o nó C e o nó B fica entre A e C. O nó B tem essencialmente duas escolhas: ele pode ler toda a mensagem, e então envia-la para o nó C, ou ele pode imediatamente passar para o nó C cada

*Este material é para o uso exclusivo do da disciplina MAC 5705 - Topicos de Algoritmos Paralelos usando MPI e BSP/CGM - 2003 do IME-USP e refere-se ao livro Parallel Programming with MPI de Peter Pacheco

	Dados		
Tempo	Nó A	Nó B	Nó C
0	$z\ y\ x\ w$		
1	$z\ y\ x$	w	
2	$z\ y$	$x\ w$	
3	z	$y\ x\ w$	
4		$z\ y\ x\ w$	
5		$z\ y\ x$	w
6		$z\ y$	$x\ w$
7		z	$y\ x\ w$
8			$z\ y\ x\ w$

Figura 1: Roteamento Store-and-forward

	Dados		
Tempo	Nó A	Nó B	Nó C
0	$z\ y\ x\ w$		
1	$z\ y\ x$	w	
2	$z\ y$	x	w
3	z	y	$x\ w$
4		z	$y\ x\ w$
5			$z\ y\ x\ w$

Figura 2: Roteamento Cut-through

parte identificável, ou **pacote**, da mensagem. A primeira abordagem é denominada *store-and-forward routing*. A segunda é denominada *cut-through routing*.

A Figura 1 ilustra os roteamentos *store-and-forward* e a Figura 1 *cut-through*. Nas figuras, a mensagem é composta de quatro pacotes, w, x, y e z. É fácil de ver que a utilização do roteamento *store-and-forward* a mensagem gasta o dobro do tempo relativo ao envio de uma mensagem entre nós adjacentes, enquanto o roteamento *cut-through* adiciona somente o tempo necessário para o envio de um único pacote. Além disso, o roteamento *store-and-forward* utiliza muito mais memória nos nós intermediários, visto que toda a mensagem dever ser armazenada. Desta forma, a maior parte dos sistemas utilizam alguma variante do roteamento *cut-through*.

2 Troca de Mensagens

A *troca de mensagens* é o método mais utilizado na programação de sistemas MIND de memória distribuída. Na troca de mensagens, os processos coordenam suas atividades enviando e recebendo explicitamente mensagens. Por exemplo, o **Message Passing Interface** (MPI) provê uma função para enviar uma mensagem:

```
int MPI_Send(void* buffer          /* in */,
             int count             /* in */,
             MPI_Datatype          /* in */,
             int destination       /* in */,
             int tag               /* in */,
             MPI_Comm communicator /* in */)
```

and a function for receiving a message:

```
int MPI_Recv( void* buffer          /* out */,
              int count             /* in */,
              MPI_Datatype datatype /* in */,
              int source            /* in */,
              int tag               /* in */,
              MPI_Comm communicator /* in */,
              MPI_Status* status    /* out */)
```

A versão arual do MPI assume que os processos estão estaticamente alocados; i.e., o número de processos é fixado no início da execução do programa, e nenhum processo adicional é criado durante a execução. Cada processo é identificado um um único inteiro variando entre $1, 2, \ldots, p-1$, onde p é o número de processos.

Para ilustrar o uso das funçõe, suponhamos que o processo 0 deseje enviar um float `x` para o processo 1. Então o programa pode usar a função `MPI_Send` como segue:

```
MPI_Send(&x, 1, MPI_FLOAT, 1, 0, MPI_COMM_WORLD);
```

Para receber o float `x`, o processo 1 necessita chamar a função `MPI_Recv`. Para receber corretamente os dados da mensagem, é necessário que os argumentos `tag` e `communicator` sejam iguais ao da função `MPI_Send`, e a memória disponível para receber a mensagem, que é especificada pelos parâmetros `buffer`, `count` e `datatype`, devem ser pelo menos do tamanho da mensagem enviada. O parâmetro `status` retorna informação sobre a menagem

recebida, tal como o tamanho. Desta forma, para receber a mensagem, o processo 1 usa a função `MPI_Recv` como segue:

```
MPI_Recv(&x, 1, MPI_FLOAT, 0, 0, MPI_COMM_WORLD, &status);
```

Existem diversos aspectos que necessitam ser abordados nesse momento. Primeiro, note que os comandos executados pelo processo 0 (`MPI_Send`) serão diferentes dos executados pelo processo 1 (`MPI_Recv`). Contudo, isto não significa que os programas necessitem ser diferentes. Podemos simplesmente incluir o seguinte ramo condicional em nosso programa:

```
if (my_process_rank == 0)
   MPI_Send(&x, 1, MPI_FLOAT, 1, 0, MPI_COMM_WORLD);
else if (my_process_rank == 1)
   MPI_Recv(&x, 1, MPI_FLOAT, 0, 0, MPI_COMM_WORLD, &status);
```

Esta abordagem utilizada para programação de sistemas MIMD é denominada **single program, multiple data** (SPMD). Em programas SPMD, o efeito de executar programas diferentes é obtido pelo uso de ramos condicionais dentro do código fonte. Esta é a abordagem mais comum para a programação de sisteams MIMD.

Outro ponto que devemos abordar é com relação a semâtica do para send/receive. Suponhamos que o processo 0 faça uma chamada de `MPI_Send`, mas o processo 1 não chama `MPI_Recv` no mesmo tempo em que o processo 0 fez a chamada para `MPI_Send`, demorando algum tempo para faze-la. Será que o processo 0 simplesmente para e fica esperando até que o processo 1 faça a chamada do `MPI_Recv`? E se o processo 1 tiver algum problema quando estiver fazendo a chamada do `MPI_Recv`? O programa trava? Em geral, as respostas dependem do sistema. O ponto chave é se o software do sistema provê *buffering* das mensagens.

3 Buffering

Vamos assumir que os processos 0 e 1 estão sendo executados em nós distintos, 0 está sendo executado no nó A e 1 está sendo executado no nó B. Neste caso existem diversas abordagens para tratar o caso em que o processo 0 deseja enviar uma mensagem para o processo 1. Uma abordagem, o processo 0 envia uma "requisição para envio" para o processo 1 e espera até receber um "pronto para receber" de 1, nesse ponto, 0 inicia a transmissão da mensagem a ser enviada. Aternativamente, o software do sistema pode armazenar a mensagem em um *buffer*. Isto é, os conteúdos da mensagem podem ser copiados em um bloco de memória controlado pelo sistema (em

A ou B, ou ambos), e 0 pode continuar sendo executado. Quando 1 chega no ponto onde ele está pronto para receber a mensagem, o software do sistema simplesmente copia a mensagem *buffered* em uma posição apropriada de memória controlada por 1. A primeira abordagem, i.e., processo 0 espera até o processo 1 estar pronto, é algumas vezes denominada comunicação **síncrona**. A segunda abordagem é denominada comunicação **assíncrona**.

Uma vantagem clara na comunicação *buffred* é que o processo enviando uma mensagem pode continuar a realizar trabalho útil se o processo que irá receber a mensagem ainda não estiver pronto para faze-lo. As disvantagens são de que essa abordagem utiliza recursos que não seriam seriam necessários (p.ex. espaço de memória para *buffering*, e se o processo que vai receber a mensagem estiver pronto (no momento do envio), a comunicação levará mais tempo, visto que ela envolve a cópia entre o buffre e as posições de memória do programa do usuário.

A maior parte dos sistemas provê algum *buffering*, mas os detalhes de como isso é implementado variam muito. Alguns sistemas tentam *buffer* todas as mensagens. Outros somente *buffer* mensagens relativamente pequenas e usam o protocolo síncrono para mensagens grandes. Outros deixam a cargo do usuário decidir a respeito do *buffer* de mensagens, e quanto de espaço de memória será destinado para *buffering*. Alguns sistemas *buffer* as mensagens nos nós de envio, enquanto outros *buffer* as mensagens nos nós de recebimento.

Note que no caso em que o sistema provê *buffering* de mensagens, então nosso segundo problema - processo 0 executa um send, mas o processo 1 não executa um receive - não deve acarretar o travamento do program; os conteúdos das mensagens simplesmente ficarão armazenadas em um *buffer* do sistema aguardando o final do programa. Se, por outro lado o sistema não provê *buffering*, o processo 0 provavelmente travará; pois ficará aguardando indefinidamente por um "pronto para receber" do processo 1.

4 Mapeamento de Dados

Quando estamos trabalhando com programação paralela, **mapeamento de dados** surge em várias situações. Esse é um ponto crítico em programação tanto de sistemas com memória distribuída como sistemas de memória compartilhada onde o acesso a memória é não uniforme (NUMA). Em geral, a comunicação é muito mais custosa que a computação. Mesmo em sistemas convencionais, instruções que acessam a memória são muito mais lentas que operações que envolvem apenas a CPU. Essa diferença em custo

é muito mais visível se o acesso é efetuado a uma memória não local, i.e., na memória local de um outro nó de um sistema de memória distribuída ou mesmo num módulo de memória "distante" em um sistema de memória compartilhada NUMA. Desta forma, bastante esforço tem sido efetuado com o intuito de otimizar o mapeamento de dados, ou seja, o problema de como distribuir os dados entre os processadores de tal forma que a comunicação seja minimizada. Além disso, necessitamos que essa distribuição possibilite um **balanceamento de carga** entre os processos. Ou seja, qualquer mapeamento deve levar em consideração tanto o balanceamento de carga quanto a localização dos dados.

Nesta seção, analisaremos um dos casos mais simples do problema de mapeamento: como mapear um vetor em uma coleção de nós de um sistema de memória distribuída. Seja o nosso vetor representado por $A = (a_0, a_1, \ldots, a_{n-1})$ e o nosso comjunto de processadores representado por um array linear $P = (q_0, q_1, \ldots, q_{p-1})$. Vamos assumir que a quantidade de computação com cada elemento de A seja a mesma.

Se o número de processadores p, é igual ao número n de elementos do vetor, então existe um único mapeamento que equaliza a carga de trabalho entre os processadores:

$$a_i \rightarrow q_i$$

para cada i, e nosso problema é bastante simples de solucionar. No caso em que p divide n, a primeira vista parece existir apenas dois tipos de mapeamento que equalizam o balanceamento de carga. Um **mapeamento de blocos** particiona os elementos do vetor em blocos de elementos consecutivos e distribui os blocos entre os processadores. Com exemplo, consideremos o caso em que $p = 4$ e $n = 16$. O mapeamento dos blocos é o seguinte:

$$a_0, a_1, a_2, a_3 \rightarrow q_0$$

$$a_4, a_5, a_6, a_7 \rightarrow q_1$$

$$a_8, a_9, a_{10}, a_{11} \rightarrow q_2$$

$$a_{12}, a_{13}, a_{14}, a_{15} \rightarrow q_3$$

Um outro mapemanto "natural"é o **mapeamento cíclico**. Ele distribui o primeiro elemento para o primeiro processador, o segundo elemento para o segundo, e assim sucessivamente. Quando cada processador tem um elemento do vetor, voltamos para o primeiro processador e repetimos o processo de distribuição com os próximos p elementos. Esse procedimento é repetido

até que todos elementos do vetor sejam distribuídos. No caso em que $p = 4$ e $n = 16$, o mapeamento cíclico é o seguinte:

$$a_0, a_4, a_8, a_{12} \rightarrow q_0$$

$$a_1, a_5, a_9, a_{13} \rightarrow q_1$$

$$a_2, a_6, a_{10}, a_{14} \rightarrow q_2$$

$$a_3, a_7, a_{11}, a_{15} \rightarrow q_3$$

Considere agora o seguinte mapeamento:

$$a_0, a_1, a_8, a_9 \rightarrow q_0$$

$$a_2, a_3, a_{10}, a_{11} \rightarrow q_1$$

$$a_4, a_5, a_{12}, a_{13} \rightarrow q_2$$

$$a_6, a_7, a_{14}, a_{15} \rightarrow q_3$$

Este é um **mapeamento bloco-cíclico**. Ele particiona o vetor em blocos de elementos consecutivos como no mapeamento de blocos. Contudo, os blocos não são de tamanho n/p. Os blocos são então mapeados com um mapeamento cíclico. No nosso exemplo, os blocos possuem tamanho 2. Se considerarmos que o tamanho dos blocos e/ou p não dividem necessáriamente n, verificamos que existe um grande número de possibilidades de efetuar diferentes formas de mapeamento dos dados para a topologia de array linear. Se analisarmos arrays com dimensões maiores ou árvores ou grafos em geral, o problema se torna extremamente complexo.

Qual é o mapeamento mais apropriado? É fácil verificar que esta decisão é altamente dependente do problema a ser resolvido. Esse problema será tratado com mais detalhes posteriormente.

5 Exercícios

1. Suponha que um nó A está enviando uma mensagem de n-pacotes para o nó B em um sistema de memória distribuída com uma rede estática. Suponha também que a mensagem deve ser transmitida através de k nós intermediários.

    a. Calcule o custo de enviar a mensagem usandoo roteamento *store-and-forward*.

b. Calcule o custo usando o roteamento *cut-through*.

Escreva a sua estimativa do custo de enviar uma mensagem entre nós diretamente conectados.

2. Os mapeamento abordados para o array linear podem ser generalizados para arrays b-dimensionais.

   a. Um mapeamento *bloco de linhas* corresponde a um mapeamento de blocos de um array linear, exceto que os elementos do array são as linhas de uma matriz. Ilustre uma distribuição de bloco de linhas de uma matriz 6×6 entre três processos.

   b. Definições semelhantes aplicam-se para mapeamento de bloco de colunas. Ilustre uma distribuição de bloco de colunas de uma matriz 6×6 entre três processos.

   c. Uma definição similar aplica-se para mapeamento linha-cíclico. Ilustre uma distribuição linha-cíclico de um matriz 6×6 entre três processos.

   d. Nas definições anteriores, tratamos nossos processos como um array linear "virtual". Se tratarmos nossos processos como um grid "virtual", um mapeamento mais natural é o mapeamento *tabuleiro-bloco*, ou mapeamento *bloco-bloco*. Em um mapeamento bloco-bloco a matriz é particionada em submatrizes retangulares, e as submatrizes são mapeadas aos processos (armazenadas por linha). Ilustre uma distribuição tabuleiro-bloco de uma matriz 6×6 entre quatro processos. Os blocos das submatrizes devem ter ordem 3×3, e o grid virtual deve ter ordem 2×2.

   e. Como poderíamos definir um mapeamento *cíclico-cíclico*? Ilustre sua resposta com uma matriz 6×6 distribuída entre quatro processos.

   f. Como podemos formar mapeamentos híbridos na qual uma dimensão usa uma distribuição de bloco, e a outra usa uma distribuição cíclica? Ilustre sua resposta com uma matriz 6×6 entre quatro processos.