Introdução à programação paralela

Gabriel Martins de Miranda – 13/0111350

1. Título do capítulo

Dealing with I/O

2. Objetivo do capítulo

Foi visto que existem uma série de dúvidas em como tratar entrada e saída em sistemas paralelos, grande parte porque a linguagem C não foi pensada para esta situação. Com isto, o MPI padrão já provê suporte para I/O, mesmo que de forma simples e confiável acima de performance. São apresentados caching de atributos, tipos derivados e distribuição de dados.

3. Resumo do capítulo

O objetivo primordial do capítulo foi desenvolver aplicações simples de entrada e saída para programas paralelos. Pelo menos um processo deve ter a capacidade de realizar funções de entrada e saída – printf, scanf, … - e torná-las em funções coletivas, sendo os dados lidos pelo processo I/O e distribuídos aos outros e dados a serem escritos reunidos
dos outros processos pelo processo I/O e por ele escritos.

Todas as funções I/O devem recebem um argumento do comunicador. Como o **rank** dos processos I/O dependem de comunicador, é possível unir estas informações em **caching de atributos**, que são ponteiros para int, que apontam para o conteúdo dos atributos, rank do processo I/O. Como podem ter vários atributos em cache num comunicador, uma **chave** é necessária para identificá-los.

Atributos e chaves são locais, cada processo tem chaves e atributos diferentes. As funções de acesso são locais, atributos podem estar definidos ou não para processos diferentes.

Criamos chaves com **MPI_Keyval_create**. Retorna ponteiro pra chave, útil para achar rank do processo de I/O.

Para duplicar um comunicador, usamos **MPI_Comm_dup**. Para deletar um comunicador, **MPI_Comm_free**. Ambas são passadas como argumento callback de **MPI_Keyval_create**. Ainda há como deletar um atributo com **MPI_Attr_delete**. Existem funções predefinidas para **MPI_Comm_dup** e **MPI_Comm_free**, que são **MPI_DUP_FN** e **MPI_NULL_DELETE_FN**, sendo que a primeira simplesmente copia o ponteiro do atributo e a segunda não faz nada.

Para determinar ranks de processos que podem usar I/O, usamos o atributo **MPI_IO**. Este atributo deve ser cacheado junto com **MPI_COMM_WORLD**. Seu conteúdo pode tanto ser **MPI_PROC_NULL** (não pode usar I/O), **MPI_ANY_SOURCE**(todos podem usar I/O) ou o **rank do processo**. Podemos determinar seu valor ou de qualquer outro atributo com **MPI_Attr_get**. Após encontrar o rank do processo com I/O, podemos cacheá-lo com o comunicador chamando **MPI_Attr_put**.

Foi criado um comunicador separado para I/O, mas podemos querer o rank I/O por meio de outro comunicador. Com **MPI_Comm_compare**, podemos comparar grupos e contextos de dois comunicadores. Se iguais, retorna **MPI_IDENT**, se grupos iguais com contextos diferentes, **MPI_CONGRUENT**, se grupos com mesmos processos em outra ordem, **MPI_SIMILAR**, senão **MPI_UNEQUAL**. Existe também **MPI_Group_translate_ranks**.

A função de entrada **Cscanf** obtém rank do processo I/O, lê dados desse processo e broadcast pra todos os outros processos. A função de saída **Cprintf** obtém o rank de I/O, coleta dados de todos os processo no processo 0 e mostras os dados dele. Em **Cerror**, obtemos códigos de erros de todos os processos e enviamos a todos eles, se algum com problema, o processo I/O mostra ranks deles e todos os processos chamam **MPI_Abort**, que tenta desligar todos os processos do comunicador.

Alguns problemas básicos são abordados. Pelo menos um processo pode acessar stdout e stderr, mas existem implementações que não permite acesso à stdin. São três as soluções: usar linha de argumento, que em muitas implementações compartilha com todos os outros processos, usar um arquivo C separado com entradas ou usar um arquivo que não stdin. Na terceira opção, surgem os problemas dos arquivos em sistemas paralelos, sendo um deles ter certeza de que cada processo acessa um arquivo diferente.

Para ganhar performance usando dados de I/O, podemos usar arrays de I/O, sendo as distribuições em bloco, cíclica ou em bloco e cíclica. Para arrays cíclicos, um único processo pode se encarregar de I/O enquanto que as funções de I/O são operações coletivas. A de entrada, **Read_entries**, obtém rank do processos I/O, lê dados de um array simples e usa **MPI_Scatter**. A de saída, **Print_entries**, obtém rank do I/O e **MPI_Gather** de todos os processos no de I/O, printando na tela. Porém há problemas da não linearidade dos dados na memória e do que é passado pra cada processo. Para passar por estes problemas, usa-se o tipo de dados **MPI_UB**, que muda a extensão de um tipo artificialmente.

4. Solução dos exercícios

1. Ok. As funções são Cache_io_rank, Copy_attr, Get_corresp_rank, Get_io_rank, Cscanf, Cprintf e Cerror_test.

3. Ok. As funções são Initialize_params, Build_cyclic_type, Print_params, Print_entries e Read_entries.

5. Ok.

7. Aplicações bancárias, que têm de esperar por entrada de usuário e mostrar muitas coisas na tela. Jogos, que têm de mostrar diversos frames na tela e receber comandos de controle do usuário.

6. Conclusão

Foram abordados os problemas encontrados em situações em que é necessário manipular entrada e saída em programas distribuídos, sendo que surgem dúvidas em quais processos devem receber uma entrada de usuário, por exemplo, e a ordem em que devem mostrar informações na tela. MPI se preocupa em gerenciar estes tipos de situações usando uma forma inteligente de associar ranks, chaves e comunicadores de processos com o poder de acessar entrada e saída e passar estas informações para outros processos.

7. Referências consultadas

Parallel Programming with MPI by Peter Pacheco ; Dealing with I/O – chapter eight.