Relatório do Capítulo 03 de Introdução à Programação Paralela

Lucas Sousa de Oliveira (10/59491)

29 de agosto de 2013

1 Título do Capítulo

Greetings!

2 Objetivo

Introduzir o aluno ao processamento paralelo e da biblioteca MPI através do estudo de um programa simples chamado *"Greetings!"*, equivalente ao famoso *"Hello World!"*.

3 Resumo

No estudo de processamento paralelo, um dos componentes de maior importância é o sistema de troca de mensagens (*message passing*). Tal componente é responsável por coordenar a troca de informações entre diversos processos. Neste capítulo utilizou-se a biblioteca MPI para tal fim.

Os programas usados neste capítulo usam o paradigma "um-programa, vários-dados" (*single-program multiple-data*, SMPD). Nesse modelo/paradigma, cada processo roda o mesmo executavel. Entretanto, os processos executam instruções diferentes através do uso de estruturas condicionais que selecionam os processos segundo seu identificador (*rank*).

O uso da biblioteca MPI requer que a diretiva

```
#include "mpi.h"
```

seja usada para incluir os protótipos e estruturas que serão usadas. Ainda, antes de se poder usar a biblioteca, faz-se necessário iniciá-la usando a função

```
int MPI_Init(int* argc, char** argv[]);
```

e, simetricamente, terminá-lo usando a função

```
int MPI_Finalize(void);
```

Outra função de grande importância é a

```
int MPI_Comm_size(MPI_Comm comm, int* number_of_processes);
```

usada para se descobrir quantos processos foram instanciados. Note que um comunicador é uma coleção de processos que podem enviar mensagens uns para os outros. O comunicador padrão, predefinido, é o MPI_COMM_WORLD, que consiste de todos os processos em execução quando a execução do programa começa. Um processo pode ainda descobrir seu identificador (*rank*) através da função

```
int MPI_Comm_rank(MPI_Comm comm, int* my_rank);
```

A troca de mensagens na biblioteca MPI acontece através das funções

```
int MPI_Send(void* message, int count, MPI_Datatype datatype, int dest, int
    tag, MPI_Comm comm);
int MPI_Recv(void* message, int count, MPI_Datatype datatype, int source,
    int tag, MPI_Comm comm, MPI_Status* status);
```

onde o parâmetro *message* contém a mensagem enviada/recebida, *count* contém o espaço alocado para a mensagem em *bytes*, *datatype* possui o tipo de dados transmitidos, *tag* é um identificador para diferenciar tipos de mensagens, *comm* é o identificador do canal de comunicação e *status* indica se a mensagem foi de fato recebida. Existem ainda as variáveis curinga para as variáveis *tag* e *source*, MPI_ANY_SOURCE e MPI_ANY_TAG respectivamente, que permitem que permitem que uma chamada do MPI_Recv receba mensagens de qualquer processo com qualquer identificador de mensagem.

O código exemplo utilizado para o desenvolvimento das atividades deste capítulo está apresentado abaixo.

```
1  #include <stdio.h>
2  #include <string.h>
3  #include "mpi.h"
4
5  int main(int argc, char* argv[]) {
6    int          my_rank;
7    int          p;
8    int          source;
9    int          dest;
10   int          tag = 0;
11   char         message[100];
12   MPI_Status   status;
13
14   MPI_Init(&argc, &argv);
15   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
16   MPI_Comm_size(MPI_COMM_WORLD, &p);
17
18   if (my_rank != 0) {
19     sprintf(message, "Greetings from process %d!", my_rank);
20     dest = 0;
21     MPI_Send(message, strlen(message)+1, MPI_CHAR, dest, tag, MPI_COMM_WORLD
           );
22   } else {
23     for (source = 1; source < p; source++) {
24         MPI_Recv(message, 100, MPI_CHAR, source, tag, MPI_COMM_WORLD, &
               status);
25         printf("%s\n", message);
26     }
27   }
28
29   MPI_Finalize();
30 }
```

4 Exercícios

4.1 Crie um arquivo contendo o programa *"Greetings!"*. Descubra como compilar e executar em um número diferente de processadores. Qual é a saída do programa se ele for executado com apenas um processo? Quantos processadores podem ser usados?

O comando usado para compilar um programa que utiliza a biblioteca MPI é o mostrado logo abaixo, onde *code.c* é o código a ser compilado e é *execultable* o executavel resultante. Tal comando usa uma instalação customizada do MPICH para gerar o executavel.

```
/opt/mpich-1.2.7/bin/mpicc code.c -o executable
```

Para rodar o executavel gerado, o comando abaixo deve ser usado. Note que *nproc* deve indicar quantos processos serão iniciados e *executable* indica o programa que será executado.

```
/opt/mpich-1.2.7/bin/mpirun -np nproc executable
```

Uma vez que os códigos para compilação e execução são simples, um *script* pode ser criado para a simplificação destes processos. A única restrição encontrada foi o número de processos que podem ser iniciados em uma mesma máquina. Os testes realizados, apresentados na figura 1, mostrou que apenas 254 processos podem ser executados numa mesma máquina.

```
mpid@L312603:~/Documentos/D/Greetings> mpirun -np 255 ./greetings
[L312603:16653] [[62441,0],0] ORTE_ERROR_LOG: The system limit on number of pipes a pr
--------------------------------------------------------------------------
mpirun was unable to start the specified application as it encountered an error
on node L312603. More information may be available above.
--------------------------------------------------------------------------
255 total processes failed to start
mpid@L312603:~/Documentos/D/Greetings> mpirun -np 254 ./greetings
```

Figura 1: Resultado do teste para descobrir o número máximo de processos.

4.2 Modifique o programa *"Greetings!"* de forma que ele use elementos genéricos para *source* e *tag*. Existe alguma diferença na saída do programa?

Podemos modificar o código (apresentado no final da seção 3) da forma solicitada modificando a linha indicada abaixo.

```
24 MPI_Recv(message, 100, MPI_CHAR, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD
       , &status);
```

A diferença na saída em relação ao código original é mostrado na figura 2 a seguir. Apenas a ordem de processamento de alguns processos foi modificada.

4.3 Tente modificar alguns parametros de MPI_Send e MPI_Recv (e.g., *count*, *datatype*, *source*, *dest*). O que acontece quando se executa o programa? Ele falha? Ele pára?

De forma a melhor analisar as modificações sugeridas, cada uma delas foi realizada individualmente. Elas estão listadas abaixo.

***count* no MPI_Send** Reduzindo *count* em 10 unidades observou-se lixo no resultado, possivelmente devido a remoção do caractere "\0". Aumentar *count* em 10 unidades não gerou mudanças, como esperado.

```
Greetings from process 1!
Greetings from process 2!
Greetings from process 3!
Greetings from process 5!
Greetings from process 9!
Greetings from process 10!
Greetings from process 6!
Greetings from process 15!
Greetings from process 7!
Greetings from process 4!
Greetings from process 8!
Greetings from process 11!
Greetings from process 12!
Greetings from process 13!
Greetings from process 14!
Greetings from process 16!
Greetings from process 17!
Greetings from process 18!
Greetings from process 19!
```

Figura 2: Resultado de se modificar *source* e *tag* para variáveis curingas.

***count* no MPI_Recv** Apesar de se esperar o mesmo problema que no caso passado, aumentar e reduzir *count* em 10 unidades não apresentou mudanças na saída.

***datatype* no MPI_Recv** Modificar o *datatype* para MPI_INT, MPI_FLOAT e MPI_BYTE não causou modificações na saída do programa. Note que tal comportamento condiz com o esperado visto que *datatype* apenas altera a forma como o tamanho do *buffer* de mensagem é visto. Uma vez que o tipo de dados transmitido é um caractere, representado em 8 bits, e todos os tipos disponíveis tem tamanhos maior ou igual a 8 bits, nunca seria possível observar um erro neste caso.

***datatype* no MPI_Send** Modificar o *datatype* para MPI_BYTE e MPI_SHORT não causou modificações na saída do programa, mas verificou-se o erro "0 - MPI_RECV : Message truncated" quando foi usado MPI_INT ou MPI_FLOAT. Note que tal erro era esperado uma vez que MPI_INT e MPI_FLOAT são maiores que MPI_CHAR, fazendo com que o programa espere uma mensagem de tamanho maior. Uma vez que tal mensagem nunca chega, o erro é retornado.

***dest* no MPI_Send** Modificando *dest* para *dest*+1 faz o programa congelar. Isso acontece uma vez que o processo 0 nunca recebe as mensagens que deve imprimir, agora enviadas para o processo 1 que apenas envia mensagens.

***source* no MPI_Recv** Modificando *source* para *source*+1 faz o programa gerar o erro "0 - MPI_RECV : Invalid rank 20". Isso acontece porque o processo 20 não existe, visto que apenas 20 processos (do 0 ao 19) foram criados.

4.4 Modifique o programa *"Greetings!"* de forma que todos os processos enviam uma mensagem para o processo p-1.

A modificação apresentada abaixo faz com que todos os processos enviem uma mensagem para o processo com um *rank* imediatamente inferior.

```
1 #include <stdio.h>
2 #include <string.h>
```

```
3  #include "mpi.h"
4
5  main(int argc, char* argv[]) {
6      int          my_rank;
7      int          p;
8      int          source;
9      int          dest;
10     int          tag = 0;
11     char         message[100];
12     MPI_Status   status;
13
14     MPI_Init(&argc, &argv);
15     MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
16     MPI_Comm_size(MPI_COMM_WORLD, &p);
17
18     dest = ((my_rank != 0)?(my_rank-1):p-1;
19     sprintf(message, "Greetings from %d to %d!", my_rank, dest);
20
21     MPI_Send(message, strlen(message)+1, MPI_CHAR, dest, tag, MPI_COMM_WORLD
           );
22     MPI_Recv(message, 100, MPI_CHAR, MPI_ANY_SOURCE, tag, MPI_COMM_WORLD, &
           status);
23     printf("%s\n",message);
24
25     MPI_Finalize();
26 }
```

5 Trabalho de programação

5.1 Escreva um programa em que cada processo i envia mensagem ao processo (i+1)%p. (Cuidado em como i calcula de quem deve receber a mensagem). O processo i deverá enviar uma mensagem para i+1 e depois receber de i-1, ou o contrário? Faz diferença? O que acontece quando o programa é rodado em um processador?

O código com as devidas modificações está mostrado abaixo. A saída do programa está na figura 3.

```
18 #include <stdio.h>
19 #include <string.h>
20 #include "mpi.h"
21
22 main(int argc, char* argv[]) {
23     int          my_rank;
24     int          p;
25     int          source;
26     int          dest;
27     int          tag = 0;
28     char         message[100];
29     MPI_Status   status;
30
31     MPI_Init(&argc, &argv);
32     MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
33     MPI_Comm_size(MPI_COMM_WORLD, &p);
34
35     dest = (my_rank+1)%p;
36
37     sprintf(message, "Greetings from %d to %d!", my_rank, dest);
```

```
38
39      MPI_Send(message, strlen(message)+1, MPI_CHAR, dest, tag, MPI_COMM_WORLD
            );
40      MPI_Recv(message, 100, MPI_CHAR, MPI_ANY_SOURCE, tag, MPI_COMM_WORLD, &
            status);
41      printf("%s\n",message);
42
43      MPI_Finalize();
44  }
```

Note que cada processo deve primeiro enviar a mensagem para depois esperar, uma vez que se o contrário for feito, todos os processos esperarão indefinidamente uns pelos outros, o que caracteriza um *deadlock*.

```
Greetings from 19 to 0!
Greetings from 12 to 13!
Greetings from 2 to 3!
Greetings from 10 to 11!
Greetings from 4 to 5!
Greetings from 17 to 18!
Greetings from 9 to 10!
Greetings from 6 to 7!
Greetings from 8 to 9!
Greetings from 7 to 8!
Greetings from 18 to 19!
Greetings from 1 to 2!
Greetings from 13 to 14!
Greetings from 5 to 6!
Greetings from 14 to 15!
Greetings from 16 to 17!
Greetings from 0 to 1!
Greetings from 15 to 16!
Greetings from 3 to 4!
Greetings from 11 to 12!
```

Figura 3: Resultado do programa modificado.

Quando o programa é rodado em apenas um processador espera-se que as respostas passem a ficar ordenadas, uma vez que os fatores externos que geravam o desordenamento, tais como atrasos conexão e cargas de utilização dos outros processadores, deixam de existir.

6 Conclusão

Conclui-se que a biblioteca MPI possui as ferramentas necessárias para o desenvolvimento de programas paralelos, inclusive com seus processos distribuidos entre várias máquinas. Através do uso desta biblioteca pode-se construir programas segundo o modelo SPMD que trocavam informações de variadas formas.

7 Referências

Pacheco, P.S., (1997) *Parallel Programming with MPI.* Morgan Kaufmann.