Computação Paralela – MPI — 2019/2020

Exercício 1
Comunicação ponto-a-ponto bloqueante

Na primeira aula de do módulo de MPI de Computação Paralela analisámos sem profundidade um primeiro programa MPI para ficar com uma ideia geral de como se usam as bibliotecas MPI e de como se compilam e executam os programas.

Curiosamente, não usámos as funções MPI de comunicação ponto-a-ponto, ou seja, aquelas que servem para trocar mensagens entre 2 processos de ordens (*ranks*) especificadas. A subrotina/função padrão MPI para enviar uma mensagem ponto-a-ponto é

```
MPI_Send(address, count, datatype, destination, tag, comm)
```

Dos seis argumentos, apenas o quarto e o quinto não foram discutidos na aula anterior:

- `destination` é a ordem, dentro do comunicador `comm`, do processo a que se destina a mensagem.
- `tag` (etiqueta) é um inteiro que identifica a mensagem e permite ao destinatário distinguir diferentes mensagens provenientes de um mesmo remetente.

Para que o destinatário receba a mensagem, tem que chamar a função

```
MPI_Recv(address, maxcount, datatype, source, tag, comm, status)
```

Na lista de argumentos,

- `maxcount` é o número máximo de elementos do tipo `datatype` que o destinatário aceita receber. É possível que o destinatário não tenha informação prévia sobre o tamanho da mensagem que vem do remetente.
- `source` é a ordem, dentro do comunicador `comm`, do processo que emitiu a mensagem. É possível, em alternativa, usar `MPI_ANY_SOURCE` como argumento e, assim, aceitar uma mensagem de qualquer proveniência.
- `tag` é o inteiro que o destinatário usou para identificar a mensagem. É possível, em alternativa, usar `MPI_ANY_TAG` como argumento e, assim, aceitar uma mensagem com qualquer etiqueta.
- `status` contém informação sobre o tamanho, origem e etiqueta da mensagem efetivamente recebida. Vimos que é possível que qualquer um destes valores possa não ser conhecido à partida. Quando não estamos interessados nesta informação, podemos usar `MPI_STATUS_IGNORE`.

Os vínculos (*bindings*) destas funções em C são:

```
int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int
    ↪ dest, int tag, MPI_Comm comm)
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source,
    ↪ int tag, MPI_Comm comm, MPI_Status *status)
```

Algoritmo 1: Comunicação ponto-a-ponto

Input: Número de elementos n de um vetor b de inteiros (0 para sair).
Output: Mensagens com informações sobre as comunicações.

```
begin
    inicialização do MPI
    while n ≠ 0 do
        if rank = 0 then                    /* Tarefa exclusiva para o processo 0) */
            leitura de n
        o master faz o broadcast de n, os outros recebem
        se n = 0, todos os processos terminam
        todos os processos definem o vetor b
        esperam uns pelos outros, para começar a contar o tempo no
          mesmo instante
        if rank = 0 then                    /* Tarefa exclusiva para o processo 0) */
            atribui valores ao vetor b
            envia o vetor b ao processo 1
            avisa quando está pronto para continuar
        else if rank = 1 then               /* Tarefas exclusivas para o processo 1) */
            faz-se difícil por 5 segundos
            recebe o vetor b do processo 0
            avisa quando está pronto para continuar
        else                                /* Tarefa exclusiva para os outros processos) */
            não fazem nada
```

O objetivo principal deste exercício é perceber como funciona a comunicação ponto-a-ponto bloqueante. Como vamos observar no fim deste exercício, o problema das funções padrão de comunicação ponto-a-ponto bloqueante é que o seu funcionamento não está completamente definido e depende da implementação. Vamos começar por usar, em vez da função padrão `MPI_Send`, a função

```
MPI_Ssend(address, count, datatype, destination, tag, comm)
```

Esta função define uma comunicação síncrona. Isto quer dizer que a mensagem é enviada diretamente ao destinatário e que o programa do remetente só poderá avançar para a seguinte instrução quando o destinatário tiver confirmado que acabou de receber a mensagem. Note que o destinatário continua a usar a função `MPI_Recv`.

O algoritmo que vamos usar até ao fim do exercício é o Algoritmo 1. Vamos estudar a diferença de resultados quando usamos diferentes modos de comunicação ponto-a-ponto bloqueante. O essencial do algoritmo é o seguinte: o processo de ordem 0 envia ao processo de ordem 1 uma mensagem de tamanho escolhido pelo utilizador, mas o segundo processo decide esperar 5 segundos antes de executar a função que recebe a mensagem.

Compile e execute o Programa 1, que usa comunicação bloqueante síncrona. Irá observar que o processo 0 só irá estar pronto para continuar ao fim de mais de 5 segundos, porque para

além do tempo usado no processo de comunicação, tem que ficar à espera que o destinatário inicie a receção. O programa está pensado para correr apenas em 2 processos: aqueles que usar a mais vão apenas ocupar memória sem necessidade (seria fácil corrigir isto). Note que o Programa 1 introduz pela primeira vez mais uma função MPI. Para podermos analisar os tempos medidos, temos que sicronizar o instante em que o processo 0 começa a enviar a mensagem[1] e o instante em que o processo 1 inicia os seus 5 segundos de ócio. Para isso, usa-se a função

---

`MPI_Barrier(comm)`

Cada processo do comunicador informa os outros quando chega a esta instrução e depois fica à espera. Quando é recebida a notícia de que todos os processos do comunicador estão neste ponto do programa, eles avançam em simultâneo para a seguinte instrução (que poderá ser diferente para cada um).

Ainda não discutimos a questão mais importante: porquê bloquear o avanço do programa do remetente até completar a comunicação? A explicação é simples. Se a execução avançasse, a informação que está na memória correspondente ao bloco a ser enviado poderia ser alterada pelo programa. Quando a comunicação finalmente acontecesse, o destinatário iria receber informação diferente daquela que se pretendia enviar.

Programa 1: Comunicação ponto-a-ponto sincronizada

```c
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <unistd.h>
int main(int argc, char *argv[])
{
  int n, myid, numprocs;
  int *bloco;
  double MPI_Wtime(), MPI_Wtick();
  double starttime, endtime;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);

  /* Queremos repetir com vários tamanhos de mensagem. */
  while (1) {

    /* O master lê o nº de inteiros na mensagem, n. */
     usleep(3E5);
    if (myid == 0) {
      printf("Número de inteiros a enviar: (0 p/ sair) ");
      (void)! scanf("%d",&n);
    }
```

[1] Na verdade, ainda tem que escrever os valores do vetor antes, mas não vamos ser preciosistas.

```
26
27      /* Valor de n é distribuido por todos. */
28      MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
29      if (n == 0) break; /* Todos terminam.*/
30
31      /* Todos definem um vetor de inteiros.
32       * Não havia necessidade nenhuma de os processos com myid > 1
33       * fazerem isto. Só estamos a usar memória sem necessidade.*/
34      bloco = (int*)malloc(n * sizeof(int));
35      /* Todos começam a contar o tempo no mesmo instante. */
36      MPI_Barrier(MPI_COMM_WORLD);
37      starttime = MPI_Wtime();
38
39      if (myid == 0) {
40        for (int i=0; i < n; ++i) bloco[i]=2*i;
41        MPI_Ssend(bloco, n, MPI_INT, 1, 0, MPI_COMM_WORLD);
42        endtime=MPI_Wtime();
43        printf("Sou o %d. Passaram aproximadamente %f segundos "
44               "desde que comecei a enviar o bloco "
45               "e agora estou pronto para continuar.\n", myid, endtime-starttime);
46      }
47      else if (myid == 1) {
48        usleep(5E6);
49        MPI_Recv(bloco, n, MPI_INT, 0, 0, MPI_COMM_WORLD,MPI_STATUS_IGNORE);
50        printf("Sou o %d. Estive 5 segundos a dormir e acabei agora de "
51               "receber o bloco.\n", myid);
52      }
53      else {
54        /* Não se faz nada. */
55      }
56
57       free(bloco);
58       /* Não quero que o 0 comece a pedir o n antes de o 1 fazer o output. */
59       MPI_Barrier(MPI_COMM_WORLD);
60    }
61    MPI_Finalize();
62    return 0;
63  }
```

Para evitar que possa ser desperdiçado tempo de CPU enquanto se espera que um envio sincrono seja concluído, o processo remetente pode em alternativa criar uma cópia do conteúdo da mensagem e colocá-la num buffer. O programa pode agora continuar, alterando à vontade a versão local do conteúdo em questão, porque é a cópia que está no buffer que vai ser enviada ao destinatário quando este estiver preparado para a receber. Formalmente, continua a tratar-se de uma comunicação ponto-a-ponto bloqueante. A função é a seguinte:

```
MPI_Bsend(address, count, datatype, destination, tag, comm)
```

Será que basta substituir `MPI_Ssend` por `MPI_Bsend` no Programa 1 para passar a usar comunicação com buffer em vez de comunicação síncrona? Vamos experimentar. No linux

não é necessário abrir o ficheiro para fazer a substituição, basta executar o seguinte comando:

```
$ sed -i 's/Ssend/Bsend/g' Ex1Prog1.c
```

Compile e execute o programa. Deverá encontrar um erro, porque o buffer de envio tem tamanho zero. Isto quer dizer que quando um processo quer usar um buffer para enviar mensagens tem que o preparar previamente com a função

---

`MPI_Buffer_attach(buffer, size)`

Cada processo só pode ter associado a si um único buffer. Tem que se ter cuidado na escolha do tamanho do buffer. Tem que ser suficientemente grande para ir guardando as mensagens que ainda não foram enviadas com sucesso, mas não deverá ser demasiado grande, de forma a não ter que reservar muita memória. Lembre-se que pode haver outros processos no mesmo computador a usarem os seus próprios buffers.

Quando o buffer deixar de ser necessário, pode ser desativado com

---

`MPI_Buffer_detach(buffer, size)`

Pode fazer modificações no Programa 2 para confirmar que invocar `MPI_Buffer_detach` não é suficiente para libertar a memória. Compare o Programa 2 com o Programa 1 para ver as modificações que tiveram de ser feitas para usar comunicação ponto-a-ponto com buffer. Compile e execute o Programa 2. Vai verificar que o processo 0 já não vai ter que esperar que o processo 1 receba a mensagem.

Programa 2: Comunicação ponto-a-ponto com buffer

```c
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <unistd.h>
int main(int argc, char *argv[])
{
  int n, myid, numprocs;
  int *bloco;
  double MPI_Wtime(), MPI_Wtick();
  double starttime, endtime;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);
  /* Queremos repetir com vários tamanhos de mensagem. */
  while (1) {
    /* O master lê o nº de inteiros na mensagem, n. */
    usleep(3E5);
    if (myid == 0) {
      printf("Número de inteiros a enviar: (0 p/ sair) ");
```

```
22        (void)! scanf("%d",&n);
23      }
24
25      /* Valor de n é distribuido por todos. */
26      MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
27      if (n == 0) break; /* Todos terminam.*/
28
29      /* Todos definem um vetor de inteiros.
30       * Não havia necessidade nenhuma de os processos com myid > 1
31       * fazerem isto. Só estamos a usar memória sem necessidade.*/
32      bloco = (int*)malloc(n * sizeof(int));
33      /* Todos começam a contar o tempo no mesmo instante. */
34      MPI_Barrier(MPI_COMM_WORLD);
35      starttime = MPI_Wtime();
36
37      if (myid == 0) {
38        for (int i=0; i < n; ++i) bloco[i]=2*i;
39        /* É preciso criar o buffer. */
40        int buffer_size = (MPI_BSEND_OVERHEAD + n*sizeof(int));
41        char* buffer = malloc(buffer_size);
42        MPI_Buffer_attach(buffer, buffer_size);
43        MPI_Bsend(bloco, n, MPI_INT, 1, 0, MPI_COMM_WORLD);
44        endtime=MPI_Wtime();
45        printf("Sou o %d. Passaram aproximadamente %f segundos "
46               "desde que comecei a enviar o bloco "
47               "e agora estou pronto para continuar.\n", myid, endtime-starttime);
48        MPI_Buffer_detach(&buffer, &buffer_size);
49        free(buffer);
50      }
51      else if (myid == 1) {
52        usleep(5E6);
53        MPI_Recv(bloco, n, MPI_INT, 0, 0, MPI_COMM_WORLD,MPI_STATUS_IGNORE);
54        printf("Sou o %d. Estive 5 segundos a dormir e acabei agora de "
55               "receber o bloco.\n", myid);
56      }
57      else {
58        /* Não se faz nada. */
59      }
60
61       free(bloco);
62       /* Não quero que o 0 comece a pedir o n antes de o 1 fazer o output. */
63       MPI_Barrier(MPI_COMM_WORLD);
64    }
65    MPI_Finalize();
66    return 0;
67  }
```

Estamos finalmente preparados para analisar o funcionamento da função padrão de comunicação ponto-a-ponto bloqueante. Assumindo que ainda não reverteu o comando que estragou o Programa 1, faça

```
$ sed -i 's/Bsend/Send/g' Ex1Prog1.c
```

Compile e corra o Programa 1. Deverá verificar que para mensagens grandes, o comportamento vai ser o de uma comunicação sincronizada. No entanto, para mensagens pequenas o resultado deverá ser semelhante ao de uma comunicação com buffer. Como o Programa 1 não define nenhum buffer de utilizador, isto significa que a implementação de MPI gere um buffer de sistema que é usado para algumas comunicações ponto-a-ponto padrão. Há um quarto modelo de comunicações ponto-a-ponto bloqueante: pode fazer uma pesquisa se estiver curioso. No próximo exercício vamos estudar a comunicação ponto-a-ponto não bloqueante.