Exame CP 30/06/20 Parte A

1.

a) O MPI_Ssend obriga a uma comunicação síncrona entre o processo que envia a informação e o que recebe. Isto significa que tanto o processo 0 que envia como o 3 que recebe, tem de esperar que a comunicação conclua antes de avançar para a instrução seguinte, pois a instrução send é blocking quando a comunicação trata muitos dados. se os dados a enviar fossem poucos, então a comunicação seria buffered.
No que diz respeito à informação trocada neste caso, são enviados 2 000 000 números reais presentes no buffer "b". Em cada ciclo a tag é incrementada.

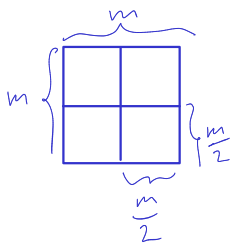
b) MPI_Recv(b, 2000000, MPI_REAL, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE)

c) MPI_Issend(b, 2000000, MPI_REAL, 3, i, MPI_COMM_WORLD, &request)
O parâmetro extra "request" serve para mais tarde verificar o estado da comunicação ou para esperar pela sua conclusão.

d)
```
MPI_Status status;
MPI_Request request;
if ...
  for ...
    MPI_Issend(b, 2000000, MPI_REAL, 3, i, MPI_COMM_WORLD, &request);
    (...)
    MPI_Wait(&request, &status);
    for ...
(
```
Primeiro criam-se os tipos de dados status e request, depois utiliza-se uma comunicação síncrona non-blocking e assim podem-se executar novas instruções durante o envio de dados até chegar ao ponto onde se iriam fazer alterações nesses dados, então aí chama-se MPI_Wait para confirmar o estado da comunicação e verifica se já terminou, se já tiver terminado pode-se avançar, caso contrário é necessário esperar que ela termine.

e) A vantagem do MPI_Issend sobre o MPI_Ssend, é que pode executar novas instruções ao mesmo tempo que envia dados através da comunicação, desde que se tenha o cuidado de evitar que estas instruções alterem os dados enquanto estiverem a ser enviados.

2.



a) MPI_Type_vector($\frac{m}{2}$, $\frac{m}{2}$, m, MPI_DOUBLE, &stype)

MPI_Type_commit(&stype)

MPI_Send(&gr[$\frac{m}{2}$][$\frac{m}{2}$], 1, stype, 0, 0, MPI_COMM_WORLD)

---

b)
```
gsizes[0] = m; gsizes[1] = m;
lsizes[0] = m/2; lsizes[1] = m/2;
start_indices[0] = m/2; start_indices[1] = m/2;
MPI_Type_create_subarray(2, gsizes, lsizes, start_indices, MPI_ORDER_C,
                         MPI_DOUBLE, &stype)

MPI_Type_commit(&stype)
MPI_Send(&gr, 1, stype, 0, 0, MPI_COMM_WORLD)
```

3.

a)
```
MPI_Comm_rank(comm1, &newid)
MPI_Cart_shift(comm1, 1, 1, &left, &right)
```

---

b) Os processos 0 e 3 têm 10001 colunas das quais são responsáveis por actualizar 9999 pontos, pois excluí-se o ponto da fronteira à esquerda em relação ao processo 0 e o da direita em relação ao processo 3, enquanto que a coluna extra que perfaz as 10001 colunas, serve para guardar o ghost point do vizinho à direita do processo 0 e o do vizinho à esquerda do processo 3.

Os processos 1 e 2 têm 10002 colunas das quais são responsáveis por actualizar 10000 pontos, enquanto que as colunas extra que perfazem as 10002 colunas, servem para guardar os ghost points dos vizinhos à esquerda e direita de cada um dos processos.

---

c) Inicialmente define-se a variável mystart para cada processo, que diz respeito ao índice da primeira coluna que cada processo irá tratar, sendo o mystart = 0 para o processo 0, pois a primeira coluna do seu Tlocal diz respeito à condição fronteira à sua esquerda que irá precisar para fazer cálculos.

O MPI_Scatterv pega na linha inicial considerada nos cálculos e divide-a em quatro partes e distribui uma parte para cada processo no comunicador comm1. Apesar de todos os processos executarem esta instrução, apenas o processo 0 irá efectuar a distribuição propriamente dita, enquanto os restantes processos apenas irão receber dados.

O primeiro MPI_Sendrecv pega no valor mais à direita, excluindo ghost points, de cada array de cada processo e envia-o para o vizinho da direita, e em simultâneo cada processo guarda no seu valor mais à esquerda, excluindo ghost points, o valor que recebe do seu vizinho à esquerda.

O segundo MPI_Sendrecv pega no valor mais à esquerda, excluindo ghost points, de cada processo e envia-o para o seu vizinho da esquerda, e em simultâneo cada processo guarda no seu valor mais à direita, excluindo ghost points, o valor que recebe do seu vizinho à direita.

d)

```
for i ...
    for j ...
        (...)
        {
        MPI_Sendrecv(&Tlocal[i+1][cols-2], 1, MPI_DOUBLE, right, 0, &Tlocal[i+1][0], 1,
                     MPI_DOUBLE, left, 0, comm1, MPI_STATUS_IGNORE)

        MPI_Sendrecv(&Tlocal[i+1][1], 1, MPI_DOUBLE, left, 1, &Tlocal[i+1][cols-1], 1,
                     MPI_DOUBLE, right, 1, comm1, MPI_STATUS_IGNORE)
        }
```

e) `MPI_Gather(&Tlocal[Nt-1][mystart], Nx/4, MPI_DOUBLE, &perfilfinal, Nx/4, MPI_DOUBLE, 0, comm1)`

Para que todos os processos recebessem o array perfilfinal, bastaria usar a função MPI_Allgather em vez de MPI_Gather.

4.
```
// 1000000000 inteiros ocupam tanto espaço como 500000000 doubles
N = 1000000000 / 2;
gsizes = 3 * N;
lsizes = N;
start_index = myid * N;
MPI_Type_Create_Subarray(1, gsizes, lsizes, start_index, MPI_ORDER_C, MPI_DOUBLE, &filetype);
MPI_File_open(MPI_COMM_WORLD, "output.sim", MPI_MODE_CREATE | MPI_MODE_WRONLY,
              MPI_INFO_NULL, &fh);
MPI_File_set_view(fh, 0, MPI_DOUBLE, filetype, "native", MPI_INFO_NULL);
if (myid == 0)
  MPI_File_write_all(fh, &a[0], N, MPI_DOUBLE, MPI_STATUS_IGNORE);
if (myid == 1)
  MPI_File_write_all(fh, &a[N], N, MPI_DOUBLE, MPI_STATUS_IGNORE);
if (myid == 2)
  MPI_File_write_all(fh, &b[0], 2*N, MPI_DOUBLE, MPI_STATUS_IGNORE);
MPI_File_close(&fh);
```