

# Programação Paralela

Autores: João Vitor Branquinho, Bruno Locha Gomes Oliveira

## Informações gerais

O trabalho consiste em dado um programa em linguagem C que resolve a adição, subtração e multiplicação dos elementos de um vetor de inteiros, devemos propor uma solução paralela baseada em MPI para o código original seguindo as orientações de cada enunciado.

## Código original

```
#include <stdio.h>

int main () {
    int x, soma=0, subtracao=0, mult=1, TAM = 1000;
    int vet[TAM];
    for (x=0;x<10;x++) {
        vet[x] = x + 1;
    }
    for (x=0;x<10;x++) {
        printf("vet[%d] = %d\n", x, vet[x]);
    }
    for (x=0;x<10;x++) {
        soma = soma + vet[x];
    }
}
```

```

    subtracao = subtracao - vet[x];
    mult = mult * vet[x];
}

printf("Soma = %d\n", soma);
printf("Subtracao = %d\n", subtracao);
printf("Multiplicacao = %d\n", mult);
}

```

## Enunciado A

O primeiro enunciado foi definido por:

**a) uma versão com apenas 4 processos executando. Nesta versão, cada processo faz uma função: 1 soma, 1 subtrai e 1 multiplica. O outro processo é responsável por avisar cada um dos outros 3 a sua função, e ao término imprimir os resultados.**

Com isso, arquitetamos uma solução baseado no seguintes aspectos:

1- Definimos os papéis (Roles) e as etiquetas (Tags) para cada tipo de processo.

```

13  enum Role {
14      ROLE_NONE = 0,
15      ROLE_SUM = 1,
16      ROLE_SUB = 2,
17      ROLE_MUL = 3
18  };
19
20 // tags para send / recv
21 enum Tags {
22     TAG_ROLE = 100, // função atribuida
23     TAG_N = 101, // tamanho do vetor
24     TAG_VET = 102, // dados do vetor
25     TAG_RESULT = 200 // resultado de volta ao mestre
26 };
27

```

2- Inicializamos as variáveis gerais para serem utilizadas bem como o sistema do MPI

```
28 int main(int argc, char *argv[]) {
29     int rank, nprocs;
30     MPI_Status state; // status de retorno para o MPI_Recv
31
32     MPI_Init(&argc, &argv); // inicia o MPI
33     MPI_Comm_rank(MPI_COMM_WORLD, &rank); // id do processo
34     MPI_Comm_size(MPI_COMM_WORLD, &nprocs); // total de processos
35
36     const int TAM = 10;
37     int N = 10;
38     int vet[TAM];
39
40     long long res_sum = 0, res_sub = 0, res_mul = 0;
41     double t0 = 0.0, t1 = 0.0;
42 }
```

3- Definimos o caso do processo mestre, onde o programa vai inicializar o vetor, inicializar a marcação de tempo (necessário ser sempre no processo 0 para evitar medições inconsistentes), atribuição de Roles para cada um dos processos slaves, envio da variável de tamanho N do vetor, envio do vetor inicializado. Com isso garantimos a consistência inicial para o processamento.

```
43 if (rank == 0) { // mestre
44
45     // inicializa o vetor
46     for (int i = 0; i < N; i++)
47         vet[i] = i + 1;
48
49     // printa
50     for (int i = 0; i < N; i++) {
51         printf("vet[%d] = %d\n", i, vet[i]);
52     }
53
54     t0 = MPI_Wtime(); // início da medição
55
56     // atribui roles e envia para cada slave
57     int roles[3] = { ROLE_SUM, ROLE_SUB, ROLE_MUL };
58     int dest[3] = { 1, 2, 3 };
59
60     for (int i = 0; i < 3; i++) {
61
62         // envia papel
63         MPI_Send(&roles[i], 1, MPI_INT, dest[i], TAG_ROLE, MPI_COMM_WORLD);
64         // envia N
65         MPI_Send(&N, 1, MPI_INT, dest[i], TAG_N, MPI_COMM_WORLD);
66         // envia vetor (apenas os N primeiros elementos)
67         MPI_Send(vet, N, MPI_INT, dest[i], TAG_VET, MPI_COMM_WORLD);
68     }
69 }
```

4- Em seguida, o processo mestre deve aguardar o recebimento de mensagens com a resposta e armazenar de acordo com o Role e Tag do processo que enviou.

```
70 // recebe os resultados de cada slave (3)
71 for (int i = 0; i < 3; i++) {
72     long long tmp = 0;
73     MPI_Status st;
74     MPI_Recv(&tmp, 1, MPI_LONG_LONG, MPI_ANY_SOURCE, TAG_RESULT, MPI_COMM_WORLD, &st);
75     if (st.MPI_SOURCE == 1) res_sum = tmp;
76     else if (st.MPI_SOURCE == 2) res_sub = tmp;
77     else if (st.MPI_SOURCE == 3) res_mul = tmp;
78 }
79
80 t1 = MPI_Wtime();
81
82 printf("Soma = %lld\n", res_sum);
83 printf("Subtracao = %lld\n", res_sub);
84 printf("Multiplicacao = %lld\n", res_mul);
85 printf("Tempo (segundos): %.6f\n", t1 - t0);
```

5- Já no caso dos slaves, cada um deve receber seu papel enviado pelo master e alocar um buffer para processar o vetor recebido.

```
86 } else { // slaves
87     int my_role = ROLE_NONE;
88     int n_local = 0;
89
90     MPI_Recv(&my_role, 1, MPI_INT, 0, TAG_ROLE, MPI_COMM_WORLD, &state);
91     MPI_Recv(&n_local, 1, MPI_INT, 0, TAG_N, MPI_COMM_WORLD, &state);
92
93     int *buffer = (int*)malloc((size_t)n_local * sizeof(int));
94     if (!buffer) {
95         fprintf(stderr, "Rank %d: falha de alocação\n", rank);
96         MPI_Abort(MPI_COMM_WORLD, 3);
97     }
98
99     MPI_Recv(buffer, n_local, MPI_INT, 0, TAG_VET, MPI_COMM_WORLD, &state);
```

6- Em seguida, cada processo deverá realizar a operação que lhe foi designada via Role e Tag, executando o vetor e em seguida enviando o resultado para o master.

```

102     long long result = 0;
103     if (my_role == ROLE_SUM) {
104         long long s = 0;
105         for (int i = 0; i < n_local; i++) s += buffer[i];
106         result = s;
107     } else if (my_role == ROLE_SUB) {
108         long long sub = 0;
109         for (int i = 0; i < n_local; i++) sub -= buffer[i];
110         result = sub;
111     } else if (my_role == ROLE_MUL) {
112         long long mult = 1;
113         for (int i = 0; i < n_local; i++) mult *= buffer[i];
114         result = mult;
115     }
116
117     free(buffer);
118     MPI_Send(&result, 1, MPI_LONG_LONG, 0, TAG_RESULT, MPI_COMM_WORLD);
119 }
120
121 MPI_Finalize();
122 return 0;
123 }
124

```

Prompt de Execução:

```

branq@Branq:~/LPP/trab1$ mpirun -np 4 ./exec_a
vet[0] = 1
vet[1] = 2
vet[2] = 3
vet[3] = 4
vet[4] = 5
vet[5] = 6
vet[6] = 7
vet[7] = 8
vet[8] = 9
vet[9] = 10
Soma = 55
Subtracao = -55
Multiplicacao = 3628800
Tempo (segundos): 0.000082

```

**Enunciado B**

Já o segundo enunciado foi definido por:

- b) uma versão com MPI com qualquer número de processos baseado em mestre/escravo, utilizando operações de comunicação coletiva. A divisão das tarefas deve ser a mais balanceada possível, e a forma de implementação é livre.**

Então, decidimos por uma solução utilizando MPI\_Isend e MPI\_Irecv:

1- Criamos uma função de apoio que divide o vetor para o balanceamento de tarefas

```
15 static void divide_vetor(int N, int P, int *counts, int *displs) {
16     // divide N elementos em P blocos
17     // arg *counts = quantos elementos cada processo recebe
18     // arg *displs = índice inicial de cada processo no vetor
19     int base = N / P;
20     int resto = N % P;
21     int desloc = 0;
22     for (int i = 0; i < P; i++) {
23         counts[i] = base + (i < resto ? 1 : 0);
24         displs[i] = desloc;
25         desloc += counts[i];
26     }
27 }
```

2- Prosseguimos para inicializar os valores necessários para a distribuição inicial

```

27 int main(int argc, char *argv[]) {
28     int rank, nprocs;
29     MPI_Init(&argc, &argv);
30     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
31     MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
32
33     int N = 10;
34     int vet[10];
35     double t0 = 0.0, t1 = 0.0;
36
37     // inicializacoes
38     if (rank == 0) {
39         for (int i = 0; i < N; i++)
40             |   vet[i] = i + 1;
41         printf("vetor:\n");
42         for (int i = 0; i < N; i++)
43             |   printf("vet[%d] = %d\n", i, vet[i]);
44         t0 = MPI_Wtime();
45     }
46
47     // distribui tarefas
48     int *counts = (int*)malloc(nprocs * sizeof(int));
49     int *displs = (int*)malloc(nprocs * sizeof(int));
50     divide_vetor(N, nprocs, counts, displs);
51
52     int local_n = counts[rank];
53     int *local_buf = (int*)malloc(local_n * sizeof(int));
54

```

3- O processo mestre envia para cada um dos slaves disponíveis um pedaço do vetor bem como o tipo de operação que deve ser realizado. Em seguida, o processo mestre deve aguardar o retorno da mesma informação.

```

57 // mestre envia tamanho e pedaço do vetor
58 if (rank == 0) {
59     for (int r = 1; r < nprocs; r++) {
60         MPI_Send(&counts[r], 1, MPI_INT, r, 100, MPI_COMM_WORLD);
61         if (counts[r] > 0) {
62             MPI_Request req;
63             MPI_Isend(&vet[displs[r]], counts[r], MPI_INT, r, 101, MPI_COMM_WORLD, &req);
64             MPI_Wait(&req, MPI_STATUS_IGNORE);
65         }
66     }
67     // copia para o buffer local do mestre a sua própria parte do vetor global.
68     memcpy(local_buf, &vet[displs[0]], local_n * sizeof(int));
69 } else {
70     // slaves recebem tamanho e o pedaco do vetor
71     MPI_Recv(&local_n, 1, MPI_INT, 0, 100, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
72     if (local_n > 0) {
73         local_buf = realloc(local_buf, local_n * sizeof(int));
74         MPI_Request req;
75         MPI_Irecv(local_buf, local_n, MPI_INT, 0, 101, MPI_COMM_WORLD, &req);
76         MPI_Wait(&req, MPI_STATUS_IGNORE);
77     }
78 }
79
80 // cálculos locais para cada processo
81 int local_sum = 0;
82 int local_mul = 1;
83 for (int i = 0; i < local_n; i++) {
84     local_sum += local_buf[i];
85     local_mul *= local_buf[i];
86 }
87

```

4- Com isso, o mestre recebe o resultado de todos os processos e consolida eles nas respectivas variáveis referentes à soma, subtração e multiplicação.

```

88 // envio
89 int results[2] = { local_sum, local_mul };
90 if (rank == 0) {
91     // mestre recebe de todos
92     int (*all_results)[2] = malloc(nprocs * sizeof *all_results);
93     MPI_Request *reqs = malloc(nprocs * sizeof(MPI_Request));
94
95     for (int r = 0; r < nprocs; r++) {
96         MPI_Irecv(all_results[r], 2, MPI_INT, r, 200, MPI_COMM_WORLD, &reqs[r]);
97
98     // mestre envia para si mesmo (poderia copiar direto, mas é só para exemplificar Isend/Irecv)
99     MPI_Request self_req;
100    MPI_Isend(results, 2, MPI_INT, 0, 200, MPI_COMM_WORLD, &self_req);
101
102    // MPI_Waitall eh o equivalente a for (...) {MPI_Wait}
103    MPI_Waitall(nprocs, reqs, MPI_STATUSES_IGNORE);
104
105    // Consolida resultados
106    int global_sum = 0;
107    int global_mul = 1;
108    for (int r = 0; r < nprocs; r++) {
109        global_sum += all_results[r][0];
110        global_mul *= all_results[r][1];
111    }
112    int global_sub = -global_sum;
113
114    t1 = MPI_Wtime();
115
116    printf("\nResultados finais:\n");
117    printf("Soma = %d\n", global_sum);
118    printf("Subtracao = %d\n", global_sub);
119    printf("Multiplicacao = %d\n", global_mul);
120    printf("Tempo (segundos): %.6f\n", t1 - t0);

```

5- Já os processos slave devem enviar o resultado correto, e ao final liberamos a memória e finalizamos o MPI.

```

122     free(all_results);
123     free(reqs);
124 } else {
125     // slaves enviam resultados com Isend
126     MPI_Request req;
127     MPI_Isend(results, 2, MPI_INT, 0, 200, MPI_COMM_WORLD, &req);
128     MPI_Wait(&req, MPI_STATUS_IGNORE);
129 }
130
131     free(local_buf);
132     free(counts);
133     free(displs);
134     MPI_Finalize();
135     return 0;
136 }

```

Prompt de Execução:

```
branq@Branq:~/LPP/trab1$ mpicc -o exec_b b.c
branq@Branq:~/LPP/trab1$ mpirun -np 4 ./exec_b
Vetor:
vet[0] = 1
vet[1] = 2
vet[2] = 3
vet[3] = 4
vet[4] = 5
vet[5] = 6
vet[6] = 7
vet[7] = 8
vet[8] = 9
vet[9] = 10

Resultados finais:
Soma = 55
Subtracao = -55
Multiplicacao = 3628800
Tempo (segundos): 0.000607
```