

## Referências

1. Pacheco, P., An Introduction to Parallel Programming, Morgan Kaufmann Publishers, 2011.
2. Diverio, T., Toscani, L., Veloso P., Análise e Complexidade de Algoritmos Paralelos, anais da Escola Regional de Alto Desempenho, 2002. São Leopoldo - RS.

Ping pong

## Ping pong

```
vizinho = (id + 1) % 2;
cont = 0;
while (cont < LIMITE) {
    if (id == cont % 2) {
        cont++;
        MPI_Send(&cont, 1, MPI_INT, vizinho,
                 tag, MPI_COMM_WORLD);
        printf("%d enviou valor %d para %d\n", id, cont, vizinho);

    } else {
        MPI_Recv(&cont, 1, MPI_INT, vizinho,
                 tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("%d recebeu valor %d de %d\n", id, cont, vizinho);
    }
}
```

## Ping pong

Para compilar:

```
$ mpicc -o ping_pong ping_pong.c
```

Para executar:

```
$ mpirun -np 2 ./ping_pong
```

## Ping pong

```
0 enviou valor 1 para 1
0 recebeu valor 2 de 1
0 enviou valor 3 para 1
0 recebeu valor 4 de 1
0 enviou valor 5 para 1
0 recebeu valor 6 de 1
0 enviou valor 7 para 1
0 recebeu valor 8 de 1
1 recebeu valor 1 de 0
1 enviou valor 2 para 0
1 recebeu valor 3 de 0
1 enviou valor 4 para 0
1 recebeu valor 5 de 0
1 enviou valor 6 para 0
....
```

Anel

## Anel

```
int token, tag = 0;
if (id != MESTRE) {
    MPI_Recv(&token, 1, MPI_INT, id - 1, tag, MPI_COMM_WORLD,
            MPI_STATUS_IGNORE);
    printf("%d recebeu token %d de %d\n", id, token, id - 1);
} else {
    token = 123;
}
MPI_Send(&token, 1, MPI_INT, (id + 1) % p, tag, MPI_COMM_WORLD);
if (id == 0){
    MPI_Recv(&token, 1, MPI_INT, p - 1, tag, MPI_COMM_WORLD,
            MPI_STATUS_IGNORE);
    printf("%d recebeu token %d de %d\n", id, token, p - 1);
}
```

### Anel

```
$mpicc -o anel anel.c
```

```
$mpirun -np 4 ./anel
```

```
1 recebeu token 123 de 0
```

```
2 recebeu token 123 de 1
```

```
3 recebeu token 123 de 2
```

```
0 recebeu token 123 de 3
```



## Situação de Deadlock

## Situação de Deadlock

```
if (id == MESTRE) { a = 0;  b = 1; }
                    else { a = 1;  b = 0; }

if (id == MESTRE) {
    MPI_Send(&a, 1, MPI_INT, 1, tag, MPI_COMM_WORLD);
    MPI_Recv(&b, 1, MPI_INT, 1, tag, MPI_COMM_WORLD,
            MPI_STATUS_IGNORE);
} else {
    MPI_Recv(&a, 1, MPI_INT, 0, tag, MPI_COMM_WORLD,
            MPI_STATUS_IGNORE);
    MPI_Send(&b, 1, MPI_INT, 0, tag, MPI_COMM_WORLD);
}
printf("id = %d, a = %d, b = %d\n", id, a, b);
```

## Situação de Deadlock

Quando há troca de mensagens entre dois processos, pode ocorrer a situação de deadlock, pois as operações de send e receive são bloqueantes.

## Solução mais adequada

```
if (id == MESTRE) { a = 0;  b = 1; }
                    else { a = 1;  b = 0; }

if (id == MESTRE) {
    MPI_Sendrecv(&a, 1, MPI_INT, 1, tag,
                 &b, 1, MPI_INT, 1, tag, MPI_COMM_WORLD,
                                     MPI_STATUS_IGNORE);
} else {
    MPI_Sendrecv(&b, 1, MPI_INT, 0, tag,
                 &a, 1, MPI_INT, 0, tag, MPI_COMM_WORLD,
                                     MPI_STATUS_IGNORE);
}
printf("id = %d, a = %d, b = %d\n", id, a, b);
```

Ordenação usando primitivas send/receive

## Ordenação usando primitivas send/receive

p0

8	23	19	67	45	35	1	24	13	30	3	5
---	----	----	----	----	----	---	----	----	----	---	---

p0

8	19	23	35	45	67
---	----	----	----	----	----

$O(n \lg n)$

p1

1	3	5	13	24	30
---	---	---	----	----	----

p0

8	19	23	35	45	67	1	3	5	13	24	30
---	----	----	----	----	----	---	---	---	----	----	----

p0

1	3	5	8	13	19	23	24	30	35	45	67
---	---	---	---	----	----	----	----	----	----	----	----

intercala

## Ordenação usando primitivas send/receive

```
// inicialização
if (id == MESTRE){
    gerar_vetor(v, N);
    MPI_Send(&v[N/2], N/2, MPI_INT, 1, tag, MPI_COMM_WORLD);
    qsort(v, N/2, sizeof(int), &compare);
    MPI_Recv(&v[N/2], N/2, MPI_INT, 1, tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    merge(v, N);
    mostrar_vetor(v, N);
} else {
    MPI_Recv(v, N/2, MPI_INT, 0, tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    qsort(v, N/2, sizeof(int), &compare);
    MPI_Send(v, N/2, MPI_INT, 0, tag, MPI_COMM_WORLD);
}
```

## Ordenação usando primitivas send/receive

```
void merge(int v[], int n){
    int i = 0, j = N/2, m = N/2;
    int w[N], k = 0;

    while (i < m && j < n){
        if (v[i] < v[j]) w[k++] = v[i++];
        else if (v[i] >= v[j]) w[k++] = v[j++];
    }

    while(i < m) w[k++] = v[i++];
    while(j < n) w[k++] = v[j++];
    for (int i = 0; i < n; i++) v[i] = w[i];
}
```



## Ordenação bolha

## Ordenação bolha

```
void ordenacao_bolha(int a[],int n) {  
    int tamanho, i, temp;  
    for (tamanho = n; tamanho >= 2; tamanho--) {  
        for (i = 0; i < tamanho - 1; i++) {  
            if (a[i] > a[i+1]) {  
                troca (a, i, i+1);  
            }  
        }  
    }  
}
```

Complexidade:  $O(n^2)$

## Ordenação bolha

Difícil de paralelizar, pois há um ordem de comparação inerentemente sequencial.

## Ordenação por transposição par-ímpar

Variação do algoritmo bolha que possui oportunidades consideráveis de paralelismos.

**ideia:** desacoplar as comparações e trocas.

## Ordenação por transposição par-ímpar

O algoritmo consiste de uma sequência comparações e trocas alternando entre as fases par e ímpar.

Durante a **fase par**, executar comparações e trocas entre os pares  $(a[0], a[1]), (a[2], a[3]), (a[4], a[5]), \dots$

Durante a **fase ímpar**, executar comparações e trocas entre os pares  $(a[1], a[2]), (a[3], a[4]), (a[5], a[6]), \dots$

## Ordenação por transposição par-ímpar

Exemplo: [5, 9, 4, 3, 2, 7, 8, 6]

fases	vetor
par	compare e troque os pares (5, 9), (4, 3), (2, 7), (8, 6) obtendo [5, 9, 3, 4, 2, 7, 6, 8]

## Ordenação por transposição par-ímpar

Exemplo: [5, 9, 4, 3, 2, 7, 8, 6]

fases	vetor
par	compare e troque os pares (5, 9), (4, 3), (2, 7), (8, 6) obtendo [5, 9, 3, 4, 2, 7, 6, 8]
ímpar	compare e troque os pares (9, 3), (4, 2), (7, 6) obtendo [5, 3, 9, 2, 4, 6, 7, 8]

## Ordenação por transposição par-ímpar

Exemplo: [5, 9, 4, 3, 2, 7, 8, 6]

fases	vetor
par	compare e troque os pares (5, 9), (4, 3), (2, 7), (8, 6) obtendo [5, 9, 3, 4, 2, 7, 6, 8]
ímpar	compare e troque os pares (9, 3), (4, 2), (7, 6) obtendo [5, 3, 9, 2, 4, 6, 7, 8]
par	compare e troque os pares (5, 3), (9, 2), (4, 6), (7, 8) obtendo [3, 5, 2, 9, 4, 6, 7, 8]



## Ordenação por transposição par-ímpar

Exemplo: [5, 9, 4, 3, 2, 7, 8, 6]

fases	vetor
par	compare e troque os pares (5, 9), (4, 3), (2, 7), (8, 6) obtendo [5, 9, 3, 4, 2, 7, 6, 8]
ímpar	compare e troque os pares (9, 3), (4, 2), (7, 6) obtendo [5, 3, 9, 2, 4, 6, 7, 8]
par	compare e troque os pares (5, 3), (9, 2), (4, 6), (7, 8) obtendo [3, 5, 2, 9, 4, 6, 7, 8]
ímpar	compare e troque os pares (5, 2), (9, 4), (6, 7) obtendo [3, 2, 5, 4, 9, 6, 7, 8]

## Ordenação por transposição par-ímpar

Exemplo: [5, 9, 4, 3, 2, 7, 8, 6]

fases	vetor
par	compare e troque os pares (5, 9), (4, 3), (2, 7), (8, 6) obtendo [5, 9, 3, 4, 2, 7, 6, 8]
ímpar	compare e troque os pares (9, 3), (4, 2), (7, 6) obtendo [5, 3, 9, 2, 4, 6, 7, 8]
par	compare e troque os pares (5, 3), (9, 2), (4, 6), (7, 8) obtendo [3, 5, 2, 9, 4, 6, 7, 8]
ímpar	compare e troque os pares (5, 2), (9, 4), (6, 7) obtendo [3, 2, 5, 4, 9, 6, 7, 8]
par	compare e troque os pares (3, 2), (5, 4), (9, 6), (7, 8) obtendo [2, 3, 4, 5, 6, 9, 7, 8]

## Ordenação por transposição par-ímpar

Exemplo: [5, 9, 4, 3, 2, 7, 8, 6]

fases	vetor
par	compare e troque os pares (5, 9), (4, 3), (2, 7), (8, 6) obtendo [5, 9, 3, 4, 2, 7, 6, 8]
ímpar	compare e troque os pares (9, 3), (4, 2), (7, 6) obtendo [5, 3, 9, 2, 4, 6, 7, 8]
par	compare e troque os pares (5, 3), (9, 2), (4, 6), (7, 8) obtendo [3, 5, 2, 9, 4, 6, 7, 8]
ímpar	compare e troque os pares (5, 2), (9, 4), (6, 7) obtendo [3, 2, 5, 4, 9, 6, 7, 8]
par	compare e troque os pares (3, 2), (5, 4), (9, 6), (7, 8) obtendo [2, 3, 4, 5, 6, 9, 7, 8]
ímpar	compare e troque os pares (3, 4), (5, 6), (9, 7) obtendo [2, 3, 4, 5, 6, 7, 9, 8]

## Ordenação por transposição par-ímpar

Exemplo: [5, 9, 4, 3, 2, 7, 8, 6]

fases	vetor
par	compare e troque os pares (5, 9), (4, 3), (2, 7), (8, 6) obtendo [5, 9, 3, 4, 2, 7, 6, 8]
ímpar	compare e troque os pares (9, 3), (4, 2), (7, 6) obtendo [5, 3, 9, 2, 4, 6, 7, 8]
par	compare e troque os pares (5, 3), (9, 2), (4, 6), (7, 8) obtendo [3, 5, 2, 9, 4, 6, 7, 8]
ímpar	compare e troque os pares (5, 2), (9, 4), (6, 7) obtendo [3, 2, 5, 4, 9, 6, 7, 8]
par	compare e troque os pares (3, 2), (5, 4), (9, 6), (7, 8) obtendo [2, 3, 4, 5, 6, 9, 7, 8]
ímpar	compare e troque os pares (3, 4), (5, 6), (9, 7) obtendo [2, 3, 4, 5, 6, 7, 9, 8]
par	compare e troque os pares (2, 3), (4, 5), (6, 7), (9, 8) obtendo [2, 3, 4, 5, 6, 7, 8, 9]

## Ordenação por transposição par-ímpar

```
void ordenacao_par_impair(int a[], int n) {
    for (int fase = 0; fase < n; fase++)
        if (fase % 2 == 0) { /* fase par */
            for (int i = 1; i < n; i += 2)
                if (a[i - 1] > a[i]) {
                    troca(a, i - 1, i);
                }
        } else { /* fase ímpar */
            for (int i = 1; i < n - 1; i += 2)
                if (a[i] > a[i + 1]) {
                    troca(a, i, i + 1);
                }
        }
    }
}
```

## Ordenação por transposição par-ímpar

Ordenacao par-ímpar paralelo:

1. Cada processo contém um sub-vetor com  $r = n/p$  elementos.
2. Cada processo ordena o seu sub-vetor local.
3. Repita  $p$  fases
  - Fase par, processos com  $id$  ímpares trocam dados com processos de  $id$  pares:  $(p_0, p_1), (p_2, p_3), (p_4, p_5), \dots$
  - Fase ímpar, processos com  $id$  pares trocam dados com processos de  $id$  ímpares:  $(p_1, p_2), (p_3, p_4), (p_5, p_6), \dots$

## Ordenação por transposição par-ímpar

tempo	processo			
	0	1	2	3
início	15, 11, 9, 16	3, 14, 8, 7	4, 6, 12, 10	5, 2, 13, 1
ordenação	9, 11, 15, 16	3, 7, 8, 14	4, 6, 10, 12	1, 2, 5, 13
fase 0	3, 7, 8, 9	11, 14, 15, 16	1, 2, 4, 5	6, 10, 12, 13
fase 1	3, 7, 8, 9	1, 2, 4, 5	11, 14, 15, 16	6, 10, 12, 13
fase 2	1, 2, 3, 4	5, 7, 8, 9	6, 10, 11, 12	13, 14, 15, 16
fase 3	1, 2, 3, 4	5, 6, 7, 8	9, 10, 11, 12	13, 14, 15, 16

## Ordenação por transposição par-ímpar

Custo total:

- Ordenação local :  $O(n \lg n)$
- Pelo menos  $p$  fases
  - ▷ processos vizinhos troca  $n/p$  elementos.
  - ▷ executam o merge e split em dois sub-vetores de tamanho  $n/p$

$$T_{par} = (\text{ordenação local}) \\ + (p \text{ rodadas de comunicação}) \\ + (p \text{ merges/splits}).$$



## Ordenação por transposição par-ímpar

Custo total:

- Ordenação local :  $O(n \lg n)$
- Pelo menos  $p$  fases
  - ▷ processos vizinhos troca  $n/p$  elementos.
  - ▷ executam o merge e split em dois sub-vetores de tamanho  $n/p$

$$\begin{aligned} T_{par} = & (O((n/p) \lg(n/p))) \\ & + (p * O(n/p)) \\ & + (p * O(n/p)). \end{aligned}$$

## Ordenação por transposição par-ímpar

Custo total:

- Ordenação local :  $O(n \lg n)$
- Pelo menos  $p$  fases
  - ▷ processos vizinhos troca  $n/p$  elementos.
  - ▷ executam o merge e split em dois sub-vetores de tamanho  $n/p$

$$\begin{aligned} T_{par} = & (O((n/p) \lg(n/p))) \\ & + O(n) \\ & + O(n). \end{aligned}$$

## Ordenação por transposição par-ímpar

Custo total:

- Ordenação local :  $O(n \lg n)$
- Pelo menos  $p$  fases
  - ▷ processos vizinhos troca  $n/p$  elementos.
  - ▷ executam o merge e split em dois sub-vetores de tamanho  $n/p$

$$T_{par} = (O((n/p) \lg(n/p))) + O(2n)$$

## Ordenação por transposição par-ímpar

`ordenacao_par_impar(a_local, n_local)`

(1) `ordene(a_local, n_local)`

(2) **para** *fase* = 0 **até** *p* **faça**

(3)       `vizinho = compute_o_vizinho(fase, id)`

(4)       **se** *vizinho* é válido **então**

(5)               `envia(a_local, n_local, vizinho)`

(6)               `recebe(b_local, n_local, vizinho)`

(7)               **se** (*id* < *vizinho*) **então**

(8)                       intercala e mantém os menores

(9)               **senão**

(10)                   intercala e mantém os maiores

## computar o vizinho

```
if (fase % 2 == 0) /* fase par */
    if (id % 2 != 0) /* id impar */
        vizinho = id - 1;
    else /* id par */
        vizinho = id + 1;
else /* fase impar */
    if (id % 2 != 0) /* id impar */
        vizinho = id + 1;
    else /* id par */
        vizinho = id - 1;

if (vizinho == -1 || vizinho == p)
    vizinho = MPI_PROC_NULL;
```

## Enviar e receber do vizinho

```
MPI_Sendrecv(a, n, MPI_INT, vizinho, tag,  
             b, n, MPI_INT, vizinho, tag,  
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

Enviando o vetor  $a$  de tamanho  $n$  para o vizinho e recebendo o vetor  $b$  de tamanho  $n$  do vizinho.

Útil para evitar deadlock.

## Ordenação por transposição par-ímpar

*ordenacao\_par\_impar(a\_local, n\_local)*

(1) *ordene(a\_local, n\_local)*

(2) **para** *fase = 0* **até** *p* **faça**

(3)       *vizinho = compute\_o\_vizinho(fase, id)*

(4)       **se** *vizinho* é válido **então**

(5)             *envia(a\_local, n\_local, vizinho)*

(6)             *recebe(b\_local, n\_local, vizinho)*

(7)       **se** (*id < vizinho*) **então**

(8)             intercala e mantém os menores\*

(9)       **senão**

(10)            intercala e mantém os maiores\*

\* como implementar isto?

## Ordenação por transposição par-ímpar

```
void intercala_e_mantem_menores_valores(int a[],
                                         int b[],
                                         int c[], int n){

    int i, j, k;

    i = j = k = 0;
    while (k < n) {
        if (a[i] <= b[j])
            c[k++] = a[i++];
        else c[k++] = b[j++];
    }

    for (i = 0; i < n; i++)
        a[i] = c[i];
}
```



## Ordenação por transposição par-ímpar

```
void intercala_e_mantem_maiores_valores(int a[],
                                         int b[],
                                         int c[], int n){

    int i, j, k;

    i = j = k = n - 1;
    while (k >= 0) {
        if (a[i] >= b[j])
            c[k--] = a[i--];
        else c[k--] = b[j--];
    }

    for (i = 0; i < n; i++)
        a[i] = c[i];
}
```

```
void ordenacao_par_impar(int a[], int n, int id, int p) {
    int vizinho, b[N], c[N], tag = 0;
    qsort(a, n, sizeof(int), compare);
    for (int fase = 0; fase < p+1; fase++){
        vizinho = computar_vizinho(id, p, fase);
        if (vizinho >= 0) { // não está ocioso
            MPI_Sendrecv(a, n, MPI_INT, vizinho, tag,
                        b, n, MPI_INT, vizinho, tag,
                        MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            if (id < vizinho)
                intercala_e_mantem_menores_valores(a, b, c, n);
            else
                intercala_e_mantem_maiores_valores(a, b, c, n);
        }
    }
}
```

## Ordenação por transposição par-ímpar

```
int main(int argc, char *argv[]){
    int id, p, n = N, v[N], v_local[N], n_local;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &id);
    MPI_Comm_size(MPI_COMM_WORLD, &p);

    n_local = n / p;

    if (id == MESTRE){
        gerar_vetor(v, n);
    }
```

```
MPI_Scatter(v,          n_local, MPI_INT,  
            v_local, n_local, MPI_INT,  
            MESTRE, MPI_COMM_WORLD);
```

```
MPI_Barrier(MPI_COMM_WORLD);  
ordenacao_par_impar(v_local, n_local, id, p);  
MPI_Gather(v_local, n_local, MPI_INT,  
           v,        n_local, MPI_INT,  
           MESTRE, MPI_COMM_WORLD);
```

```
if (id == MESTRE) {  
    mostrar_vetor(v, n);  
}
```

```
MPI_Finalize();
```

```
}
```

## Medindo o tempo de execução

Use `MPI_Wtime()` que devolve o tempo decorrido desde algum ponto no passado.

```
double inicio, fim;

inicio = MPI_Wtime();
// codigo a ser medido
...
fim = MPI_Wtime();

printf("tempo decorrido = %e segundos \n", fim - inicio);
```

Rank sort

## Rank sort

**Ideia:** Comparar cada elemento da lista a ser ordenada com todos os outros, fazendo a contagem de todos os elementos que são menores que o elemento que está sendo calculado.

## Rank sort sequencial

**algoritmo** Rank sort

Entrada: O tamanho  $n$  do vetor  $v$  e seus elementos

Saída: O vetor em ordem crescente.



## Rank sort sequencial

**algoritmo** Rank sort

```
1  para  $i = 0$  até  $n$  faça  
2       $contador = 0$   
3      para  $j = 1$  até  $n$  faça  
4          se  $v[i] > v[j]$  então  
5               $contador = contador + 1$   
6       $vetor\_ordenado[contador] = v[i]$ 
```

**Complexidade algoritmo sequencial:**  $O(n^2)$

## Rank sort sequencial

O algoritmo envolve três tarefas principais:

- ▶ **contagem**: para cada elemento o número de elementos menores que ele é computado. (por simplicidade, todos os elementos são distintos)
- ▶ **rank**: a posição final do elemento no vetor ordenado é determinado pela soma das comparações verdadeiras obtidas na fase de contagem.
- ▶ **reorganização dos dados**: cada elemento é colocado na sua posição final determinado pelo seu rank.

## Rank sort sequencial

Exemplo:  $n = 4$ ,  $v = \{4, 2, 3, 1\}$

$i = 1$			
	$j = 1$	$v[1] = 4 > v[1] = 4$	$contador = 0$
	$j = 2$	$v[1] = 4 > v[2] = 2$	$contador = 1$
	$j = 3$	$v[1] = 4 > v[3] = 3$	$contador = 2$
	$j = 4$	$v[1] = 4 > v[4] = 1$	$contador = 3$
$vetor\_ordenado[contador] = v[1]$			

$vetor\_ordenado = [0, 0, 0, 4]$

## Rank sort sequencial

Exemplo:  $n = 4$ ,  $v = \{4, 2, 3, 1\}$

$i = 2$			
	$j = 1$	$v[2] = 2 > v[1] = 4$	$contador = 0$
	$j = 2$	$v[2] = 2 > v[2] = 2$	$contador = 0$
	$j = 3$	$v[2] = 2 > v[3] = 3$	$contador = 0$
	$j = 4$	$v[2] = 2 > v[4] = 1$	$contador = 1$
$vetor\_ordenado[contador] = v[2]$			

$vetor\_ordenado = [0, 2, 0, 4]$

## Rank sort sequencial

Exemplo:  $n = 4$ ,  $v = \{4, 2, 3, 1\}$

$i = 3$			
	$j = 1$	$v[3] = 3 > v[1] = 4$	$contador = 0$
	$j = 2$	$v[3] = 3 > v[2] = 2$	$contador = 1$
	$j = 3$	$v[3] = 3 > v[3] = 3$	$contador = 1$
	$j = 4$	$v[3] = 3 > v[4] = 1$	$contador = 2$
$vetor\_ordenado[contador] = v[3]$			

$vetor\_ordenado = [0, 2, 3, 4]$

## Rank sort sequencial

Exemplo:  $n = 4$ ,  $v = \{4, 2, 3, 1\}$

$i = 4$			
	$j = 1$	$v[4] = 1 > v[1] = 4$	$contador = 0$
	$j = 2$	$v[4] = 1 > v[2] = 2$	$contador = 0$
	$j = 3$	$v[4] = 1 > v[3] = 3$	$contador = 0$
	$j = 4$	$v[4] = 1 > v[4] = 1$	$contador = 0$
$vetor\_ordenado[contador] = v[4]$			

$vetor\_ordenado = [1, 2, 3, 4]$

## Algoritmo baseado em troca de mensagens

A abordagem do algoritmo rank sort, para uma arquitetura de memória distribuída, utilizando-se de troca de mensagens, é a mestre escravo.

**ideia:** o processador mestre faz um *broadcast* do vetor a ser ordenado, cada processador escravo faz a computação do seu vetor local, e devolve o resultado para o mestre.

## Algoritmo baseado em troca de mensagens

**algoritmo** Rank sort por troca de mensagens

Entrada: O tamanho  $n$  do vetor  $v$  e seus elementos, número  $p$  de processos, identificador do processo  $0 \leq id \leq p$

Saída: O vetor em ordem crescente.



**algoritmo** Rank sort paralelo

```
1  broadcast( $v, v\_local, MESTRE$ )
2  se  $id = MESTRE$  então
3      para  $i = 1$  até  $n$  faça
4          recebe( $contador, valor, ANY\_SOURCE$ )
5           $vetor\_ordenado[contador] = valor$ 
6  senão
7       $inicio = (id - 1) * n\_local$ 
8       $fim = inicio + n\_local$ 
9      para  $k = inicio$  até  $fim$  faça
10          $valor = v[k], contador = 0$ 
11         para  $j = 1$  até  $n - 1$  faça
12             se  $v\_local[i] < v\_local[j]$  então
13                  $contador = contador + 1$ 
14         envia( $contador, valor, MESTRE$ )
```

## Rank sort por troca de mensagens (1)

```
/* Iniciando o programa */  
MPI_Init(&argc, &argv);  
MPI_Comm_rank(MPI_COMM_WORLD, &id);  
MPI_Comm_size(MPI_COMM_WORLD, &p);
```

## Rank sort por troca de mensagens (2)

```
if (id == MESTRE){  
    gerar_vetor_sem_repeticao(v, n);  
}  
  
MPI_Bcast(&v, n, MPI_INT, 0, MPI_COMM_WORLD);  
n_local = n/(p-1);          // o mestre não faz o processamento
```

## Rank sort por troca de mensagens (3)

```
if (id == MESTRE){  
    gerar_vetor_sem_repeticao(v, n);  
}  
  
MPI_Bcast(&v, n, MPI_INT, 0, MPI_COMM_WORLD);  
n_local = n/(p-1);          // o mestre não faz o processamento
```

## Rank sort por troca de mensagens (4)

```
if (id != MESTRE){
    int inicio = (id - 1)* n_local;
    int fim    = inicio + n_local;
    for (int k = inicio; k < fim; k++) {
        valor = v[k];
        contador = 0;
        for (int i = 0; i < n; i++){
            if (v[i] < valor) contador++;
        }
        envia(contador, valor, MESTRE);
    }
}
```

## Rank sort por troca de mensagens (5)

```
void envia(int contador, int valor, int destino) {  
    int tag = 0;  
    int posicao = 0;  
  
    MPI_Pack(&contador, 1, MPI_INT, buffer, 100,  
            &posicao, MPI_COMM_WORLD);  
    MPI_Pack(&valor, 1, MPI_INT, buffer, 100,  
            &posicao, MPI_COMM_WORLD);  
    MPI_Ssend(buffer, 100, MPI_PACKED, destino,  
            tag, MPI_COMM_WORLD);  
}
```

## Rank sort por troca de mensagens (6)

```
if (id == MESTRE){
    for (int k = 0; k < n; k++) {
        MPI_Recv(buffer, 100, MPI_PACKED, MPI_ANY_SOURCE, tag,
                MPI_COMM_WORLD, MPI_STATUS_IGNORE);

        posicao = 0;
        MPI_Unpack(buffer, 100, &posicao,
                &contador, 1, MPI_INT, MPI_COMM_WORLD);
        MPI_Unpack(buffer, 100, &posicao,
                &valor, 1, MPI_INT, MPI_COMM_WORLD);
        vetor_ordenado[contador] = valor;
    }
    mostrar_vetor(vetor_ordenado, n);
}
```

## Rank sort por troca de mensagens (7)

```
/* Finalizando o programa */  
MPI_Finalize();
```



Fim