# TSP com MPI e Threads

## Como executar

O projeto tem um `Makefile`. Para compilar o programa basta executar `make mpiapp`, lembrando que é necessário o `mpicc` para compilar programas com MPI.
Para executar o código utilizar `mpirun -np <x> --hostfile host_file ./main`, sendo `x` a quantidade de processos.
Para alterar o número de *threads* basta modificar a constante `NUM_THREADS` no arquivo `main.c`.
Para alterar a instância utilizada no programa, basta alterar o nome do arquivo na constante `FILENAME` no arquivo `main.c`. As opções de instâncias estão na pasta `/instances`.
Ao executar o código deve-se informar o número de cidades da instância escolhida.

## Desenvolvimento

Separei o desenvolvimento nas seguintes etapas:

1. Solução do problema sem o uso de threads e MPI
2. Solução apenas com threads
3. Solução com MPI e threads

### Solução sem threads e MPI

Nessa etapa foram desenvolvidas as estruturas de dados necessárias para executar o programa, assim como a lógica do TSP de acordo com o que foi sugerido no capítulo 6 do livro *An Introduction to Parallel Programming*. As estruturas criadas foram:

- `stack`: utilizada para controlar a descoberta de novos caminhos (como descrito no livro).
- `tour`: responsável por armazenar as cidades de de determinado caminho, assim como o tamanho máximo do caminho e o custo dele.
- `graph`: responsável por armazenar a matriz de adjacências da instância que o problema irá executar.
- `queue`: utilizada para fazer a busca em largura para distribuição dos caminhos pelas *threads* e processos posteriormente.

Nessa primeira etapa tinha apenas um processo e um código sem *threads*, portanto, consistia apenas na inicialização da *stack* com o *tour* inicial (contendo apenas *hometown*) e a busca em profundidade atualizando o melhor caminho.

```
while(!Empty(stack_t)) {
  current_tour = Pop(stack_t);

  if(GetTourCost(current_tour) > *best_tour && *best_tour != -1) {
    continue;
  }

  if(GetTourNumberCities(current_tour) == n_cities) {
     cost = GetEdgeWeight(graph_t, LastCity(current_tour), hometown);
     AddCity(current_tour, hometown, cost);

     if(BestTour(current_tour, best_tour)) {
       printf("Update best tour!\n");
       PrintTourInfo(current_tour);
       best_tour = current_tour;
    }
  } else {
    for (int nbr=n_cities-1; nbr >= 1; nbr--) {
      if(!TourContainCity(current_tour, nbr)) {
        cost = GetEdgeWeight(graph_t, LastCity(current_tour), nbr);
        AddCity(current_tour, nbr, cost);
        PushCopy(stack_t, current_tour);
        RemoveLastCity(current_tour, cost);
      }
    }
  }
}
```

## Solução com threads

Com a primeira etapa consolidada a solução foi extendida para utilizar *pthreads*. O balanceamento das *threads* foi feito inicialmente com uma busca em largura, e depois conforme as *threads* terminavam seus trabalhos elas buscavam novos trabalhos com outras *threads* que ainda não haviam terminado (como foi sugerido no livro).
O código abaixo mostra a busca em largura para o balanceamento inicial, e depois o código para divisão da fila da busca em largura entre as *threads*.

```
void FillBFSQueue(int num_instances, graph* graph_t, queue* bfs_queue,
 tour* initial_tour) {
  int num_cities = NumNodes(graph_t);
  int visited_nodes[num_cities];
  tour* current_tour;

  EnqueueCopy(bfs_queue, initial_tour);

  for(int i=0; i < num_cities; i++) {
    if(TourContainCity(initial_tour, i)) {
      visited_nodes[i] = 1;
    } else {
      visited_nodes[i] = 0;
    }
  }

  while(SizeQueue(bfs_queue) < num_instances) {
    current_tour = Dequeue(bfs_queue);
    int last_city = GetTourLastCity(current_tour);

    for(int nbr=num_cities-1; nbr >= 0; nbr--) {
      int nbr_cost = GetEdgeWeight(graph_t, last_city, nbr);

      if (nbr_cost == 0.0 || visited_nodes[nbr] == 1) { continue; }

      AddCity(current_tour, graph_t, nbr);
      EnqueueCopy(bfs_queue, current_tour);
      RemoveLastCity(current_tour, graph_t);

      visited_nodes[nbr] = 1;
    }
  }
}

void ShareQueue(int num_instances, stack** stacks, queue* queue_t) {
  int i = 0;

  while(!EmptyQueue(queue_t)) {
    stack* current_stack = stacks[i % num_instances];
    PushCopy(current_stack, Dequeue(queue_t));
    i++;
  }
}
```

Com as *stacks* balanceadas era só executar o código da primeira etapa para cada uma das *threads*.

### Solução com MPI e threads

Primeiramente, era preciso iniciar o MPI da maneira correta. Como o programa é *multi thread* era preciso iniciar o MPI com `MPI_Init_thread(NULL, NULL, MPI_THREAD_MULTIPLE, &provided);`. Dessa maneira, múltiplas *threads* podem fazer chamadas para o MPI.

A leitura da instância é feita pelo processo zero e enviada para os demais processos. Como o envio dessas mensagens é necessário para o início da execução. então não tem problema usar o `MPI_Send` e `MPI_Recv` que são bloqueantes.

```
if(process_rank == 0) {
  ReadNCities(&n_cities);
  AllocateInputs(n_cities);
  InitializeInstance();

  for (int dest = 0; dest < num_processes; dest++) {
    if (dest != process_rank) {
      MPI_Send(&n_cities, 1, MPI_INT, dest, 0, MPI_COMM_WORLD);
      MPI_Send(nodes, n_cities, MPI_INT, dest, 0, MPI_COMM_WORLD);
      for(int i=0; i < n_cities; i++)
        MPI_Send(adj_m[i], n_cities, MPI_FLOAT, dest, 0, MPI_COMM_WORLD)
    }
  }
} else {
  MPI_Recv(&n_cities, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORI

  AllocateInputs(n_cities);

  MPI_Recv(nodes, n_cities, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGI
  for(int i=0; i < n_cities; i++)
    MPI_Recv(adj_m[i], n_cities, MPI_FLOAT, 0, 0, MPI_COMM_WORLD, MPI_STA
}
```

Com todos processos tendo a instância inicial era necessário fazer uma busca em largura para dividir as possibilidades entre os processos antes de dividir pelas *threads*. Ou seja, a divisão inicial era feita pelo `número de processos` e para cada um desses processos ocorria uma divisão pelo `número de threads`.

```
void ThreadsSplit(int num_threads, queue* bfs_queue) {
  int error;
  stack* threads_stacks[num_threads];

  // Initialize threads stacks
  for(int i=0; i<NUM_THREADS; i++) {
    threads_stacks[i] = CreateStack((n_cities*n_cities)/2);
```

```
  }

  ShareQueue(num_threads, threads_stacks, bfs_queue);

  pthread_t* workers = (pthread_t*) calloc (NUM_THREADS, sizeof(pthread_
  if (!workers) { exit(-1); }

  for(int i=0; i < num_threads; i++) {
    error = pthread_create(&workers[i], NULL, &execute, (void*)threads_s
    if(error) { printf("Failed to create thread: %lu\n", (long)workers[i
  }

  for(int i=0; i < num_threads; i++) {
    error = pthread_join(workers[i], NULL);
    if(error) { printf("Failed to join thread: %lu\n", (long)workers[i])
  }

  for(int i=0; i < num_threads; i++) {
    FreeStack(threads_stacks[i]);
  }
}

void ProcessesSplit(int num_processes, int process_rank, queue* bfs_qu
eue, graph* graph_t) {
  stack* my_stack;
  queue* threads_bfs_queue = CreateQueue(MaxSizeQueue(bfs_queue));
  stack* processes_stacks[num_processes];

  for(int i=0; i < num_processes; i++) {
    processes_stacks[i] = CreateStack((n_cities*n_cities)/2);
  }

  ShareQueue(num_processes, processes_stacks, bfs_queue);
  my_stack = processes_stacks[process_rank];

  while(!Empty(my_stack)) {
    FillBFSQueue(NUM_THREADS, graph_t, threads_bfs_queue, Pop(my_stack))
  }

  ThreadsSplit(NUM_THREADS, threads_bfs_queue);

  for(int i=0; i < num_processes; i++) {
    FreeStack(processes_stacks[i]);
  }

  FreeQueue(threads_bfs_queue);
}
```

Com a divisão finalizada bastava executar o algoritmo TSP para cada uma das *stacks* de cada uma das *threads* dos processos. Entretanto, ainda era possível otimizar compartilhando o melhor *tour* entre os processos. Então, foi adicionado ao código da avaliação de *tours* métodos para receber e enviar novas melhores rotas.

```
void CheckNewBestTour(float* best_tour, int src) {
  int msg_available;
  float received_cost;
  MPI_Status status;
  MPI_Message msg;

  MPI_Improbe(src, 0, MPI_COMM_WORLD, &msg_available, &msg, &status);
  if(msg_available) {
    MPI_Mrecv(&received_cost, 1, MPI_FLOAT, &msg, MPI_STATUS_IGNORE);
    *best_tour = received_cost;
  }
}

void SendNewBestTour(float* best_tour, int num_processes, int process_
rank) {
  for(int dest = 0; dest < num_processes; dest++) {
    if(dest != process_rank)
      MPI_Send(best_tour, 1, MPI_FLOAT, dest, 0, MPI_COMM_WORLD);
  }
}
```

O método de envio manda para todas os processos diferentes dele o novo valor de melhor *tour*. Já o método de receber verifica se há algum novo melhor *tour*, caso exista, então ele atribui a variável de melhor *tour*. O código de avaliação de *tours* ficou da seguinte maneira:

```
void EvaluateTours(stack* stack_t, graph* graph_t, float* best_tour, p
thread_mutex_t evaluate_mutex, term* term_t, int n_cities, int hometow
n, int num_threads, int num_processes, int process_rank) {
  tour* current_tour;

  while(!Termination(stack_t, term_t, num_threads)) {
    current_tour = Pop(stack_t);

    if(GetTourCost(current_tour) > *best_tour && *best_tour != -1) {
      continue;
    }

    if(GetTourNumberCities(current_tour) == n_cities) {
      AddCity(current_tour, graph_t, hometown);

      for(int src = 0; src < num_processes; src++)
        CheckNewBestTour(best_tour, src);

      if(BestTour(current_tour, *best_tour)) {
        pthread_mutex_lock(&evaluate_mutex);
        *best_tour = GetTourCost(current_tour);
        SendNewBestTour(best_tour, num_processes, process_rank);
        pthread_mutex_unlock(&evaluate_mutex);
      }
    } else {
      for (int nbr=n_cities-1; nbr >= 0; nbr--) {
        if (nbr == hometown) { continue; }

        if(!TourContainCity(current_tour, nbr)) {
          AddCity(current_tour, graph_t, nbr);
          PushCopy(stack_t, current_tour);
          RemoveLastCity(current_tour, graph_t);
        }
      }
    }
    FreeTour(current_tour);
  }
}
```

Ao final da execução foi preciso fazer uma sincronização do melhor *tour*, para garantir que o processo zero mostre o melhor *tour* de fato. Isso teve que ser feito, pois pode acontecer de algum processo terminar antes e não pegar a atualização de melhor *tour* de outro processo ainda em execução, se isso acontecer com o processo zero (responsável por exibir o melhor *tour* no final), então ele vai exibir um melhor *tour* desatualizado.

```
// Final sync for best tour
if(process_rank != 0) {
MPI_Send(&best_tour, 1, MPI_FLOAT, 0, BEST_TOUR_FINAL_SYNC_TAG, MPI_CO
} else {
for(int dest=1; dest < num_processes; dest++) {
  int received_cost;
  MPI_Recv(&received_cost, 1, MPI_FLOAT, dest, BEST_TOUR_FINAL_SYNC_TA

  if(received_cost < best_tour) {
    best_tour = received_cost;
  }
}
```

## Testes

Foram feitos testes para verificar o tempo de execução para 1, 2, 4 e 8 processos e 2 ou 4 threads em cada processo. Para medir o tempo de execução foi utilizado o método `MPI_Wtime()` da biblioteca do MPI. Foi sugerido utilizar as instâncias de 12 e 15 cidades.
Abaixo segue os resultados para a instância de 12 cidades. Para a instância de 15 cidades tive algum problema de memória (eu acho...) na execução, onde o MPI aborta, abaixo deixarei a tentativa de execução e o erro reportado, tentei descobrir o que poderia ser, mas não encontrei solução.
Ambas instâncias se encontram na pasta `/instances`.

**Instância com 12 cidades:**

```
  0 300 352 466 217 238 431 336 451  47 415 515
300   0 638 180 595 190 138 271 229 236 214 393
352 638   0 251  88 401 189 386 565 206 292 349
466 180 251   0 139 371 169 316 180 284 206 198
217 595  88 139   0 310 211 295 474 130 133 165
238 190 401 371 310   0 202 122 378 157 362 542
431 138 189 169 211 202   0 183  67 268 117 369
336 271 386 316 295 122 183   0 483 155 448 108
451 229 565 180 474 378  67 483   0 299 246 418
 47 236 206 284 130 157 268 155 299   0 202 327
415 214 292 206 133 362 117 448 246 202   0 394
515 393 349 198 165 542 368 108 418 327 394   0
```

Podemos ver abaixo que quando executamos com 2 threads o desempenho é melhor. Inclusive, podemos ver que a execução com 4 e 8 processos para 2 threads tem, praticamente, o mesmo tempo de execução.

**1 processo**

```
2 threads

[inf2591-06@server parallel-tsp]$ mpirun -np 1 --hostfile host_file ./
main
Cities number:
12

BEST TOUR:
Best tour: 1733.00
Total execution time: 2.10s
-------------------------------------
4 threads

[inf2591-06@server parallel-tsp]$ mpirun -np 1 --hostfile host_file ./
main
Cities number:
12

BEST TOUR:
Best tour: 1733.00
Total execution time: 2.69s
```

### 2 processos

```
2 threads

[inf2591-06@server parallel-tsp]$ mpirun -np 2 --hostfile host_file ./
main
Cities number:
12

BEST TOUR:
Best tour: 1733.00
Total execution time: 1.35s
-------------------------------------
4 threads

[inf2591-06@server parallel-tsp]$ mpirun -np 2 --hostfile host_file ./
main
Cities number:
12

BEST TOUR:
Best tour: 1733.00
Total execution time: 1.51s
```

### 4 processos

```
2 threads

[inf2591-06@server parallel-tsp]$ mpirun -np 4 --hostfile host_file ./
main
Cities number:
12

BEST TOUR:
Best tour: 1733.00
Total execution time: 0.78s
-------------------------------------
4 threads

[inf2591-06@server parallel-tsp]$ mpirun -np 4 --hostfile host_file ./
main
Cities number:
12

BEST TOUR:
Best tour: 1733.00
Total execution time: 1.21s
```

### 8 processos

```
2 threads

[inf2591-06@server parallel-tsp]$ mpirun -np 8 --hostfile host_file ./
main
Cities number:
12

BEST TOUR:
Best tour: 1733.00
Total execution time: 0.77s
-------------------------------------
4 threads

[inf2591-06@server parallel-tsp]$ mpirun -np 8 --hostfile host_file ./
main
Cities number:
12

BEST TOUR:
Best tour: 1733.00
Total execution time: 1.06s
```

**Instância com 15 cidades:**

```
 0 29 82 46 68 52 72 42 51 55 29 74 23 72 46
29  0 55 46 42 43 43 23 23 31 41 51 11 52 21
82 55  0 68 46 55 23 43 41 29 79 21 64 31 51
46 46 68  0 82 15 72 31 62 42 21 51 51 43 64
68 42 46 82  0 74 23 52 21 46 82 58 46 65 23
52 43 55 15 74  0 61 23 55 31 33 37 51 29 59
72 43 23 72 23 61  0 42 23 31 77 37 51 46 33
42 23 43 31 52 23 42  0 33 15 37 33 33 31 37
51 23 41 62 21 55 23 33  0 29 62 46 29 51 11
55 31 29 42 46 31 31 15 29  0 51 21 41 23 37
29 41 79 21 82 33 77 37 62 51  0 65 42 59 61
74 51 21 51 58 37 37 33 46 21 65  0 61 11 55
23 11 64 51 46 51 51 33 29 41 42 61  0 62 23
72 52 31 43 65 29 46 31 51 23 59 11 62  0 59
46 21 51 64 23 59 33 37 11 37 61 55 23 59  0


[inf2591-06@server parallel-tsp]$ mpirun -np 8 --hostfile host_file ./
main
Cities number:
15

===================================================================
=============
=   BAD TERMINATION OF ONE OF YOUR APPLICATION PROCESSES
=   PID 10734 RUNNING AT n01.cluster.inf.puc-rio.br
=   EXIT CODE: 9
=   CLEANING UP REMAINING PROCESSES
=   YOU CAN IGNORE THE BELOW CLEANUP MESSAGES
===================================================================
=============
[proxy:0:0@n00.cluster.inf.puc-rio.br] HYD_pmcd_pmip_control_cmd_cb (p
m/pmiserv/pmip_cb.c:887): assert (!closed) failed
[proxy:0:0@n00.cluster.inf.puc-rio.br] HYDT_dmxu_poll_wait_for_event (
tools/demux/demux_poll.c:76): callback returned error status
[proxy:0:0@n00.cluster.inf.puc-rio.br] main (pm/pmiserv/pmip.c:202): d
emux engine error waiting for event
[mpiexec@server.cluster.inf.puc-rio.br] HYDT_bscu_wait_for_completion
(tools/bootstrap/utils/bscu_wait.c:76): one of the processes terminate
d badly; aborting
[mpiexec@server.cluster.inf.puc-rio.br] HYDT_bsci_wait_for_completion
(tools/bootstrap/src/bsci_wait.c:23): launcher returned error waiting
for completion
[mpiexec@server.cluster.inf.puc-rio.br] HYD_pmci_wait_for_completion (
pm/pmiserv/pmiserv_pmci.c:218): launcher returned error waiting for co
mpletion
[mpiexec@server.cluster.inf.puc-rio.br] main (ui/mpich/mpiexec.c:340):
 process manager error waiting for completion
```