TSP com workstealing

Como executar

O projeto tem um `Makefile`. Para compilar o programa basta executar `make build`.
Para executar o código utilizar `make run`.
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.

Workstealing

O Workstealing consiste em "roubar" da fila de outra thread uma *task* caso a sua fila de *task* esteja vazia. No cenário do TSP vamos enfileirando e processando na estrutura de cada uma das threads as rotas. Quando a fila de uma das threads fica vazia, então essa thread tenta "roubar" uma rota da fila de outra thread, e para isso funcionar, o "roubo" acontece na outra ponta da fila (por isso a estrutura usada é uma DEque, double ended queue).

Desenvolvimento

Foi utilizado como base o trabalho anterior, onde foi desenvolvido o TSP com MPI e Threads. Para desenvolvimento do cenário atual, foram feitar modificações no código original em duas etapas. A primeira para garantir o funcionamento do TSP sem o MPI e com balanceamento de threads por divisão das stack (como era a proposta do trabalho anterior). A segunda com objetivo de modificar a estrutura de dados de stack para deque, e adicionar o workstealing em vez do stack split para balanceamento das threads.
O método para verificar a finalização do TSP esta abaixo. Ele funciona de maneira que continue o processamento caso a fila da thread não esteja vazia, e caso contrário tenta "roubar" a rota de uma das outras threads.

```
int Termination(deque** deques, int my_id, int num_threads, pthread_mu
tex_t top_mutex) {
    deque* my_deque = deques[my_id];
    tour* top_tour = NULL;

    if (!EmptyDeque(my_deque)) {
        return 0;
    } else {
        pthread_mutex_lock(&top_mutex);

        for(int i=0; i < num_threads; i++) {
            if(i == my_id) { continue; }

            deque* current_deque = deques[i];
            top_tour = PopTopDeque(current_deque);

            if(top_tour != NULL) {
            PushBottomDeque(my_deque, top_tour);
                pthread_mutex_unlock(&top_mutex);
                return 0;
            }
        }
    }

    if(top_tour == NULL && AllDequesEmpty(deques, num_threads)) {
        pthread_mutex_unlock(&top_mutex);
        return 1;
    }

    pthread_mutex_unlock(&top_mutex);
    return 0;
}
```

Testes

Foram feitos testes para instância de 12 e 15 cidades e verificado o tempo de execução para 2 ou 4 threads. Podemos notar que a execução para 4 threads tem um tempo de execução menor. Vou colocar abaixo a execução do TSP com MPI e stack split para 1 processo com o objetivo de comparação.

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
```

Resultado para TSP com Workstealing

```
2 threads

Cities number:
12

BEST TOUR:
Best tour: 1733.00
2 threads, execution time: 365288us (0.36s)
-------------------------------------
4 threads

Cities number:
12

BEST TOUR:
Best tour: 1733.00
4 threads, execution time: 224461us (0.22s)
```

Resultado para TSP com MPI e stack split

```
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
```

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
```

Resultado para TSP com Workstealing

```
2 threads

Cities number:
15

BEST TOUR:
Best tour: 291.00
4 threads, execution time: 34324502us (34.32s)

-------------------------------------
4 threads

Cities number:
15

BEST TOUR:
Best tour: 291.00
4 threads, execution time: 24054795us (24.05s)
```

Resultado para TSP com MPI e stack split

```
2 threads

[inf2591-06@server parallel-tsp]$ mpirun -np 1 --hostfile host_file ./
main
Cities number:
15

BEST TOUR:
Best tour: 291.00
Total execution time: 52.13s

-------------------------------------
4 threads

[inf2591-06@server parallel-tsp]$ mpirun -np 1 --hostfile host_file ./
main
Cities number:
15

BEST TOUR:
Best tour: 291.00
Total execution time: 32.91s
```

Podemos ver que para essa nova estratégia de balanceamento de threads tivemos um tempo de resposta melhor para ambas instâncias. Isso pode ser pelo fato de não utilizarmos `cond_wait` e/ou por não ficar gerando novas estruturas como era feito anteriormente

gerando uma nova stack.