TOP RED COMP/SIST DISTRIB II
Travelling Salesman Problemem com MPI e threads
Marcelo Paulon, Rayanne Souza
12 de Dezembro de 2018

1 Implementação

A solução do problema do caixeiro viajante consistiu em adaptar a implementação com paralelização dinâmica usando pthreads apresentada pelo Peter Pacheco no livro *An Introduction to Parallel Programming*.

Em cada nó é requerido rodar um processo MPI com várias threads portanto a primeira adaptação foi incluir na função principal os processos MPI. O trecho de código 1 mostra essas adaptações. Dado que cada nó MPI executa um conjunto de threads, foi necessário usar a função *MPI_Init_thread()* com o parâmetro MPI_THREAD_MULTIPLE (linha 5). Cada processo MPI lê o dígrafo, inicializa o seu *local_best_tour* como nulo e o melhor custo com um valor bastante alto (linhas 30-37).

Nas linhas 53-54, a rotina *Par_tree_search* vai distribuir para cada thread uma subárvore através da chamada da função *Partition_tree*. Uma vez que o número de nós MPI e a quantidade de threads para cada nó é passado por parâmetro, essa função foi alterada (código 2) acrescentando a variável *thread_global_rank* (linha 3), que através do uso da *cluster_rank* computa para qual thread dentre todas existentes em todo os nós a árvore está sendo repartida. Para possibilitar essa divisão, a função *Set_init_tours* do código 3 teve seus parâmetros de cáculo *quotient* (linha 5) e *remainder* (linha 6) alterados para considerar o número threads total.

```
1  int main(int argc, char* argv[]) {
2
3      ...
4
5      MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &provided);
6      comm = MPI_COMM_WORLD;
7      MPI_Comm_size(comm, &cluster_count);
8      MPI_Comm_rank(comm, &cluster_rank);
9
10     if (argc != 5) {
11         Usage(argv[0]);
12         return -1;
13     }
14
15     char *node_count_arg = argv[1];
16     char *threads_per_node_count_arg = argv[2];
17     char *digraph_file_path_arg = argv[3];
18     char *min_split_size_arg = argv[4];
19
20     ...
21
22     digraph_file = fopen(digraph_file_path_arg, "r");
23
24     if (digraph_file == NULL) {
25         fprintf(stderr, "Can't open %s\n", digraph_file_path_arg);
26         Usage(argv[0]);
27         return -1;
```

```
28      }
29
30      Read_digraph(digraph_file);
31      fclose(digraph_file);
32
33      ...
34
35      loc_best_tour = Alloc_tour(NULL);
36      Init_tour(loc_best_tour, INFINITY);
37      global_best_tour_cost = INFINITY;
38
39      MPI_Type_contiguous(n+1, MPI_INT, &tour_arr_mpi_t);
40      MPI_Type_commit(&tour_arr_mpi_t);
41
42      MPI_Pack_size(1, MPI_INT, comm, &one_msg_sz);
43      mpi_buffer =
44              malloc(100*cluster_count*(one_msg_sz + MPI_BSEND_OVERHEAD)*
    sizeof(char));
45      MPI_Buffer_attach(mpi_buffer,
46                        100*cluster_count*(one_msg_sz + MPI_BSEND_OVERHEAD));
47
48
49      ...
50
51      start = MPI_Wtime();
52      for (thread = 0; thread < thread_count; thread++)
53          pthread_create(&thread_handles[thread], NULL,
54                         Par_tree_search, (void*) thread);
55
56      for (thread = 0; thread < thread_count; thread++)
57          pthread_join(thread_handles[thread], NULL);
58
59      finish = MPI_Wtime();
60      Cleanup_msg_queue();
61      MPI_Barrier(comm);
62      Get_global_best_tour();
63
64      MPI_Buffer_detach(&ret_buf, &one_msg_sz);
65
66      if (cluster_rank == 0) {
67          if(loc_best_tour == NULL) {
68              printf("Tour not found");
69          }
70          else {
71              Print_tour(cluster_rank, loc_best_tour, "Best tour");
72              printf("Cost = %d\n", loc_best_tour->cost);
73              printf("Elapsed time = %e seconds\n", finish - start);
74          }
75      }
76
77      ...
78      return 0;
79
80  }
```

Listing 1: Função principal

```
1  void Partition_tree(long my_rank, my_stack_t stack) {
2      int my_first_tour, my_last_tour, i;
3      int thread_global_rank = (int) (my_rank + (thread_count * cluster_rank)
       );
4
5      if (my_rank == 0) queue_size = Get_upper_bd_queue_sz();
6      My_barrier(bar_str);
7
8      ...
9
10     if (my_rank == 0) Build_initial_queue();
11     My_barrier(bar_str);
12     Set_init_tours(thread_global_rank, &my_first_tour, &my_last_tour);
13
14     ...
15
16 } /* Partition_tree */
```

Listing 2: Função de partição da árvore para todos as threads do programa

```
1  void Set_init_tours(long thread_global_rank, int* my_first_tour_p,
2                      int* my_last_tour_p) {
3      int quotient, remainder, my_count;
4
5      quotient = init_tour_count/(thread_count * cluster_count);
6      remainder = init_tour_count % (thread_count * cluster_count);
7      if (thread_global_rank < remainder) {
8          my_count = quotient+1;
9          *my_first_tour_p = thread_global_rank*my_count;
10     } else {
11         my_count = quotient;
12         *my_first_tour_p = thread_global_rank*my_count + remainder;
13     }
14     *my_last_tour_p = *my_first_tour_p + my_count - 1;
15 }   /* Set_init_tours */
```

Listing 3: Função Set_init_tours: determina que tours podem ser atribuídos a uma thread específica.

Durante a execução da rotina *Par_tree_search* (Código 4), cada thread examina se a adição de uma nova cidade é viável se comparado ao valor atual do *local_best_tour* (linha 26). Se uma thread consegue visitar todas as cidades, ela verifica se aquele é o melhor tour entre todos os nós MPI através da chamada da função *Best_tour* (linhas 17-22). A *Best_tour* por sua vez chama *Look_for_best_tours* (Código 5) para examinar as mensagens sobre o melhor custo dos outros processos para atualizar o seu melhor custo local.

A função *Update_best_tour* (linha 22) do código 4 é mostrada no Códido 6. Esta é responsável por modificar a variável global do melhor tour (linha 7). Foi necessário alterá-la de forma a permitir que quando uma thread em um dos nós MPI encontrasse um novo melhor tour, este pudesse ser comunicado aos outros nós MPI. Para isso acrescentou-se a função *Bcast_tour_cost* (linha 9) que faz o broadcast desse valor para os outros nós MPI como mostrado no Código 7.

```
1  void* Par_tree_search(void* rank) {
2      long my_rank = (long) rank;
3      city_t nbr;
4      my_stack_t stack;  // Stack for searching
5      my_stack_t avail;  // Stack for unused tours
6      tour_t curr_tour;
7
8      avail = Init_stack();
9      stack = Init_stack();
10     Partition_tree(my_rank, stack);
11
12     while (!Terminated(&stack, my_rank)) {
13         curr_tour = Pop(stack);
14 #      ifdef PTSDEBUG
15         Print_tour(my_rank, curr_tour, "Popped");
16 #      endif
17         if (City_count(curr_tour) == n) {
18             if (Best_tour(curr_tour)) {
19 #               ifdef PTSDEBUG
20                 Print_tour(my_rank, curr_tour, "Best tour");
21 #              endif
22                 Update_best_tour(curr_tour);
23             }
24         } else {
25             for (nbr = n-1; nbr >= 1; nbr--)
26                 if (Feasible(curr_tour, nbr)) {
27                     Add_city(curr_tour, nbr);
28                     Push_copy(stack, curr_tour, avail);
29                     Remove_last_city(curr_tour);
30                 }
31         }
32         Free_tour(curr_tour, avail);
33     }
34
35     Free_stack(avail);
36     if (my_rank == 0) {
37         Free_queue(queue);
38     }
39
40     return NULL;
41 } /* Par_tree_search */
```

Listing 4: Rotina de busca na árvore para todas as threads do programa

```
1  void Look_for_best_tours(void) {
2      int done = FALSE, msg_avail, tour_cost;
3      MPI_Status status;
4
5      while(!done) {
6          MPI_Iprobe(MPI_ANY_SOURCE, TOUR_TAG, comm, &msg_avail,
7                     &status);
8          if (msg_avail) {
9              MPI_Recv(&tour_cost, 1, MPI_INT, status.MPI_SOURCE, TOUR_TAG,
10                     comm, MPI_STATUS_IGNORE);
11 #         ifdef STATS
12             best_costs_received++;
13 #         endif
14 #         ifdef VERBOSE_STATS
15             printf("Proc %d > received cost %d\n", my_rank, tour_cost);
16 #         endif
17             if (tour_cost < global_best_tour_cost) global_best_tour_cost =
       tour_cost;
18         } else {
19             done = TRUE;
20         }
21     } /* while */
22 } /* Look_for_best_tours */
```

Listing 5: Função que atualiza a variável global de melhor tour e a comunica entre os outros nós MPI

```
1  void Update_best_tour(tour_t tour) {
2      pthread_mutex_lock(&best_tour_mutex);
3      if (Best_tour(tour)) {
4          Copy_tour(tour, loc_best_tour);
5          Add_city(loc_best_tour, home_town);
6
7          global_best_tour_cost = Tour_cost(loc_best_tour);
8
9          Bcast_tour_cost(global_best_tour_cost);
10     }
11     pthread_mutex_unlock(&best_tour_mutex);
12 } /* Update_best_tour */
```

Listing 6: Função que atualiza a variável global de melhor tour e a comunica entre os outros nós MPI

```
1  void Bcast_tour_cost(int tour_cost) {
2      int dest;
3
4      for (dest = 0; dest < cluster_count; dest++)
5          if (dest != cluster_rank)
6              MPI_Bsend(&tour_cost, 1, MPI_INT, dest, TOUR_TAG, comm);
7  #   ifdef STATS
8      best_costs_bcast++;
9  #   endif
10 } /
```

Listing 7: Broadcast do melhor tour para os outros nós MPI

2 Resultados

A tabela 1 mostra o tempo de execução para a solução proposta nesse trabalho. A execução com 2 processos MPI e 4 threads obteve o melhor desempenho dentre as demais execuções. O aumento do número de nós MPI de 2 para 4 melhorou o desempenho quando uma thread foi usada, reduzindo o tempo de execução em 40.11%. Entretanto com o uso de 4 threads o tempo de execução quadruplicou. Esta piora pode estar associada ao aumento da troca de mensagens entre os nós MPI no processo de comunicação do melhor tour.

N processos MPI	N threads	Execução 1	Execução 2	Execução 3	Valor Médio
1	1	190.7643 s	191.0191 s	190.6337 s	190.8057 s
	4	55.52768 s	55.6296 s	55.65208 s	55.60312 s
2	1	112.3689 s	112.4603 s	112.5057 s	112.44496 s
	4	32.29091 s	32.33823 s	32.39729 s	32.34214 s
4	1	67.21208 s	67.29646 s	67.52202 s	67.34352 s
	4	177.0083 s	177.9700 s	177.7623 s	177.5802 s

Tabela 1: Tempo de execução para 1, 2 e 4 processos MPI e 1 e 4 threads

As Tabelas 2 e 3 tratam-se, respectivamente, do desempenho da implementação do autor para a paralelização MPI com divisão de trabalho estática e paralelização com thread empregando a divisão de trabalho dinâmica. O melhor resultado para a paralelização MPI foi com o uso de 4 processadores, representando uma melhora de 37.19% no tempo de execução. O melhor resultado para a paralelização com threads foi para o uso de 4 threads reduzindo o tempo de execução em 71.00%.

N processadores	Execução 1	Execução 2	Execução 3	Valor Médio
1	171.9633 s	166.9757 s	168.3589 s	169.0993 s
2	108.8624 s	106.7728 s	115.6918 s	110.44233 s
4	105.0575 s	105.4199 s	108.1150 s	106.19746 s

Tabela 2: Tempo de execução do programa para paralelização com MPI com divisão estática de trabalho para 2, 4 e 8 processadores

N threads	Execução 1	Execução 2	Execução 3	Valor Médio
1	425.0903 s	424.9780 s	425.4499 s	425.17273 s
4	123.3050 s	123.4761 s	123.1057 s	123.2956 s

Tabela 3: Tempo de execução do programa para paralelização com threads com divisão dinâmica de tarefas para 1 e 4 threads

Comparando os resultados das Tabelas 1, 2 e 3 pode-se inferir que a implementação apresentada neste trabalho obteve o melhor desempenho atingindo um tempo de execução de 32.34214 s na obtenção do melhor tour. Para todos os 3 programas o melhor trajeto encontrado foi : 0 1 3 5 7 9 11 13 15 2 4 6 8 10 12 14 0.