CS 4373/6373 High Performance Computing
Asynchronous Traveling Salesperson Programming Assignment

Author: Connor Coleman
Date: 2025-05-15

Asynchronous Traveling Salesperson Problem

Description and discussion of the parallel algorithm

The asynchronous traveling salesperson (ATSP) problem is a variation of the classical traveling salesperson problem (TSP) where the travel costs to get from one city to another are not necessarily the same value both ways. TSP is a problem where a traveler is trying to determine the most efficient route (or tour) to visit every city and end in the city they began in. For the purposes of this assignment, there were 1000 cities, each of which had some travel distance to every other city, making it a complete graph (which simplified the algorithm). For this project, the goal was to create and parallelize an algorithm which found the best tour within a time constraint of 60 seconds.

My approach to this problem was to have the threads use two main functions, one which randomly chose a starting city and found a tour based on that city, and another which called the previous function in a loop until 60 seconds elapsed which tested the tours to find a local best tour. I chose to utilize a greedy algorithm when creating tours since it would be quick and allow many attempts to find an efficient tour, albeit unlikely to provide an optimal solution for any given starting city. Since the problem was constrained to 60 seconds, I figured the more tests I was able to perform, the more chance I had of finding a tour more efficient than the best I had already found.

To accomplish this, I used the pthreads API. I decided on pthreads because I thought it would have the least-overhead for communicating the city connection matrix and accessing it in a parallel manner. Even though all editing of the matrix was performed before threads were created, I used a read write lock for writing the travel matrix and whenever a thread accessed the matrix to figure out a tour, and another write lock for communicating the best tour back to global.

The pseudocode walkthrough of my program is that I start timing by grabbing the wall time (ln 46), grab the executed arguments (ln 48), then allocate all my global arrays (ln 50-78) and open the file to read in the travel matrix from (ln 81-86). I activate the write lock (ln 89), then read in from the file by looping through the total number of rows (ln 90), getting a line at a time (ln 92), tokenizing it (ln 96, 100, 109), and looping through each column to set the relevant element (ln 98-110) and unlock when I’m finished (ln 112).

After reading in everything from the file, I loop through the thread count, setting the relevant argument structure so I can pass thread rank and start time to the thread (ln 116-117) and creating a new thread (ln 118-119).

The threads run Find_best_tour(), which starts by setting up some local variables and arrays (ln 152-158), then enters a do while loop to randomly find tours (ln 160-165), then test if it's better than the current local best tour (ln 167-174). The end of the loop gets the wall time, then breaks out of the loop if the difference is 60 seconds or greater (ln 176-177). After breaking out of the loop, the thread grabs the write lock for the global best tour (ln 182), tests if the local best tour is better than the global (ln 183), sets the global variables if so (ln 185-187), and unlocks the global best tour lock (ln 193).

Find_tour() is a helper function used to find tours in a greedy manner and is used by Find_best_tour() (ln 165). Find_tour() takes as arguments: an array to place the tour in, a variable to place the total tour value in, and the city to start with. Find_tour() after initializing some local variables (ln 200-205), it loops for the number of cities there are (ln 207), resetting the local best tour value and smallest distance to an unvisited city (ln 110-111), then grabs a read lock for the travel matrix (ln 213) and loops through all connected cities to the current city, testing non-visited cities' travel times to see which has the smallest travel time before releasing the read lock (ln 214-222). After going through every connecting non-visited city, the tour is updated with the best next city to go to (ln 224-226). After all cities have been visited, the travel matrix read lock is taken, the first city is added to the end of the tour, and the travel value from the last city to the first city is added to the value before the read lock is relinquished (ln 228-231).

Once all threads are finished, I join all threads back (ln 122-123), calculate the total elapsed time (ln 125-126), print out the relevant data for output (ln 128-142), then free up any remaining memory (ln 144-146).

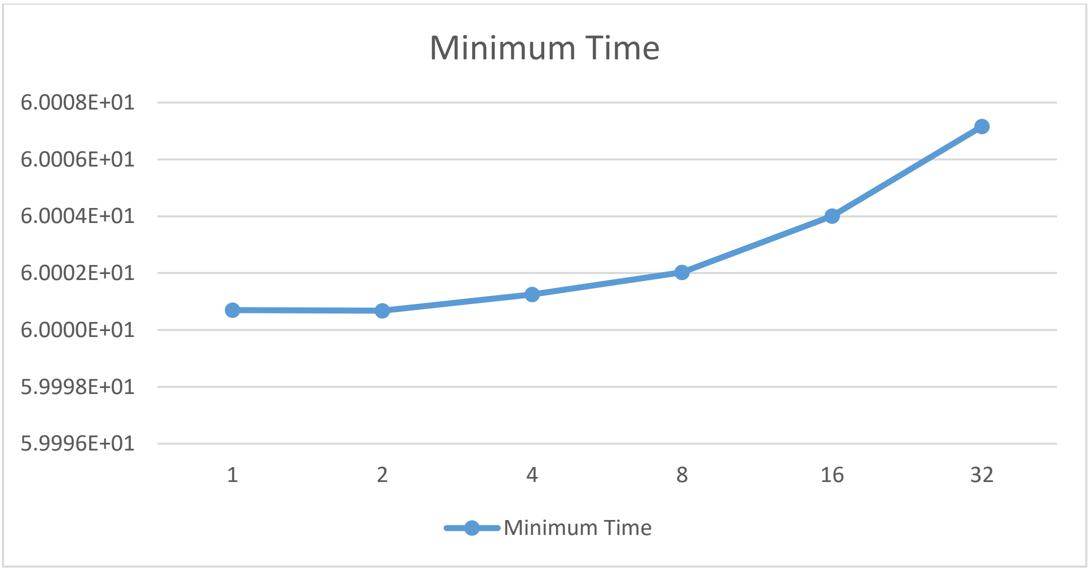
Timing methodology

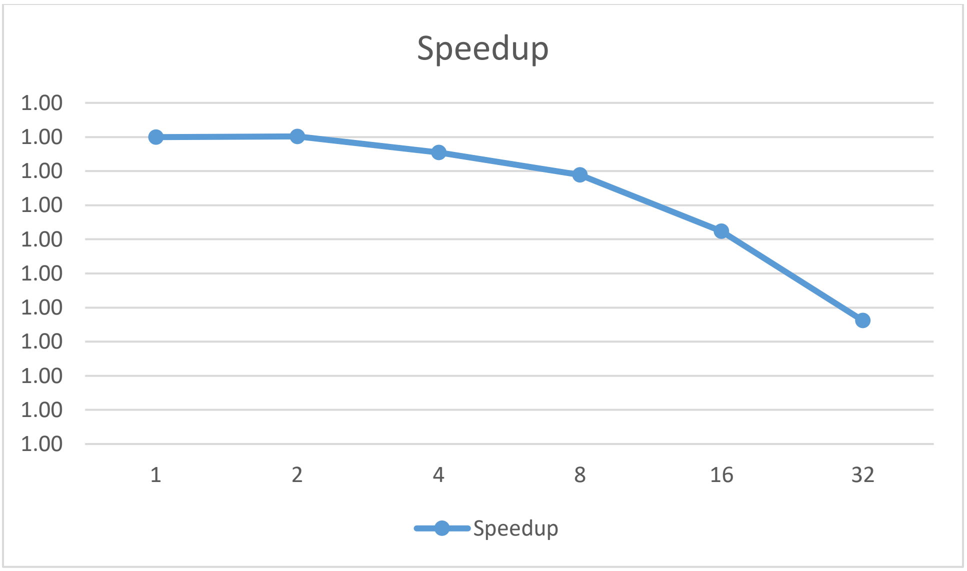
My methodology for the timing the asynchronous traveling salesperson program was to start timing before creating threads, finish timing after the threads are joined back, and calculate the difference between those times. I ran the program five times with thread counts of 1, 2, 4, 8, 16, and 32 and used the minimum time of those five to calculate speedup and efficiency. A limitation of timing this way is that I am unable to tell why any thread was slower than another and I'm not tracking the timing for every thread except for the few runs where I added that functionality for debugging purposes.

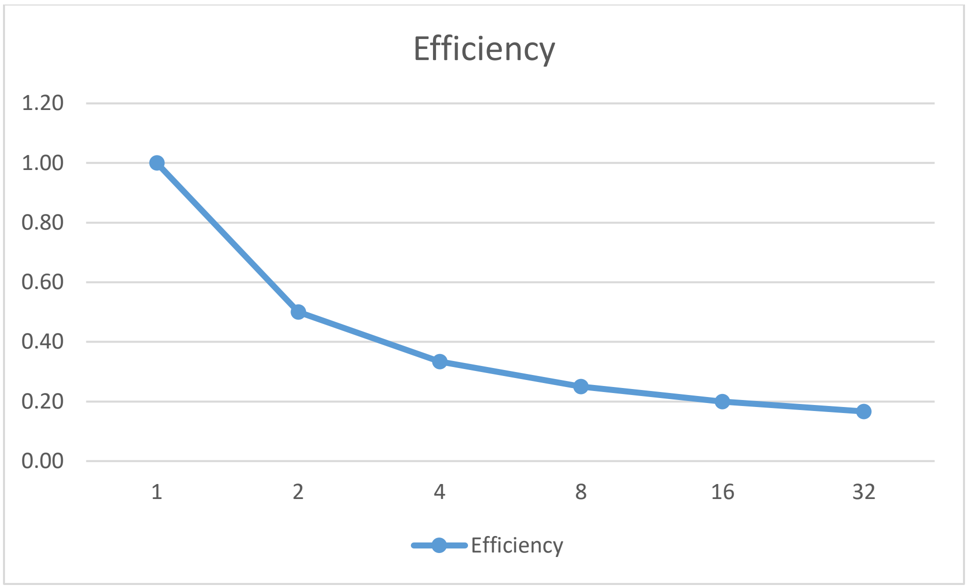
A limitation to timing in this manner is that I'm unable to determine where the slowdown caused by increasing the number of threads. This caused me to run two debugging tests to determine where the time increase due to communication happens. For these tests, I created an array of times which the threads wrote to when sections of their work were finished. It seemed like the write-lock section and thread joining had a negligible impact and the slowdown was occurring inside the timed loop, and after making the read lock section in the tour finder helper function more efficient, the slowdown was essentially eliminated.

Graph of results

Column1	1	2	4	8	16	32
Run 1	6.0005E+01	6.0001E+01	6.0003E+01	6.0003E+01	6.0006E+01	6.0007E+01
Run 2	6.0002E+01	6.0003E+01	6.0002E+01	6.0003E+01	6.0008E+01	6.0007E+01
Run 3	6.0001E+01	6.0001E+01	6.0001E+01	6.0006E+01	6.0004E+01	6.0008E+01
Run 4	6.0001E+01	6.0001E+01	6.0002E+01	6.0002E+01	6.0004E+01	6.0012E+01
Run 5	6.0002E+01	6.0002E+01	6.0003E+01	6.0006E+01	6.0008E+01	6.0008E+01
Min Val	6.0001E+01	6.0001E+01	6.0001E+01	6.0002E+01	6.0004E+01	6.0007E+01
Speedup	1.00	1.00	1.00	1.00	1.00	1.00
Efficiency	1.00	0.50	0.33	0.25	0.20	0.17







Example output for thread timings immediately after tour-finding loop

```
Number of threads: 8
Elapsed time = 6.000299e+01 seconds
Thread 0 post-loop time 6.000091e+01
Thread 1 post-loop time 6.000172e+01
Thread 2 post-loop time 6.000155e+01
Thread 3 post-loop time 6.000003e+01
Thread 4 post-loop time 6.000292e+01
Thread 5 post-loop time 6.000155e+01
Thread 6 post-loop time 6.000234e+01
Thread 7 post-loop time 6.000196e+01

Number of threads: 16
Elapsed time = 6.000407e+01 seconds
Thread 0 post-loop time 6.000021e+01
Thread 1 post-loop time 6.000023e+01
Thread 2 post-loop time 6.000305e+01
Thread 3 post-loop time 6.000261e+01
Thread 4 post-loop time 6.000225e+01
Thread 5 post-loop time 6.000282e+01
Thread 6 post-loop time 6.000259e+01
Thread 7 post-loop time 6.000223e+01
Thread 8 post-loop time 6.000205e+01
Thread 9 post-loop time 6.000147e+01
Thread 10 post-loop time 6.000079e+01
Thread 11 post-loop time 6.000102e+01
Thread 12 post-loop time 6.000333e+01
Thread 13 post-loop time 6.000397e+01
Thread 14 post-loop time 6.000238e+01
Thread 15 post-loop time 6.000237e+01
```

Analysis and speculations based on the graphs

Initially, my data showed that there was a slow but noticeable and consistent increase in time as thread count doubled. After analyzing where this slowdown might occur, I discovered the location of read locks and unlocks inside the tour finding function was inefficient and was causing a massive number of locks and unlocks. I had placed the read lock and unlock inside of a for loop that iterated through every column of the travel matrix instead of locking once for the entire loop and unlocking when finished. Since I used read locks for this section and there was a small amount of code which did not read from the matrix, I do not think it will cause issues since multiple read locks can be active at once. I'm still not exactly sure why the time increased so dramatically, but that's fixed now so it's something of a moot issue.

The data gathered after fixing that inefficiency is much closer to what I expected, which is a fairly flat speedup and time increase, with time slightly increasing as threads are added due to overhead. I included an example output of each thread's time post tour finding loop to show the time between finishing the main work loop and the rest of the communication. The efficiency curve is expected to dramatically decrease since the number of threads being used is increasing while the speedup is staying essentially static.

Weak and strong scalability

I don't think this algorithm can accurately be labeled weakly or strongly scalable since the algorithm has a fixed length of time it runs for. With the modifications to remove some inefficiencies when read locking or unlocking, the time to run stays very close to the intended 60 seconds no matter how many threads it was tested with. There might be an argument to say this means it is highly scalable, perhaps even weakly scalable since the run time should stay the same no matter how many cities were added.

Output to substantiate results

Example output of program run with 16 threads and a seed of 256

```
$./main 16 256
Cities of best tour found: 165 18 158 46 110 21 195 43 101 24 64 28 87 285 20 159 23 17 197
146 170 310 94 122 50 149 171 25 80 65 194 105 229 72 1 131 11 88 69 54 133 130 45 627 8 15
92 62 99 73 98 77 127 0 157 6 198 191 40 193 78 95 91 312 125 221 177 166 161 81 48 3 96
249 304 150 280 32 123 126 86 406 41 147 37 548 63 93 151 31 347 106 52 175 306 115 181
209 44 67 168 119 140 124 251 172 22 26 10 2 210 5 262 129 207 269 47 141 97 29 16 137 184
272 173 83 109 33 350 309 252 388 238 224 49 58 36 144 113 35 513 240 19 42 134 205 228 68
120 70 316 200 203 279 153 51 265 117 223 143 187 206 155 56 135 373 256 222 271 192 132
38 13 85 487 163 353 61 76 116 281 255 79 530 293 148 247 237 34 182 543 518 9 375 273 241
```

```
53 108 219 233 361 302 396 400 215 128 526 349 100 104 102 218 322 202 160 14 136 330 512
263 331 430 27 7 242 220 60 114 712 12 139 239 145 419 250 196 383 378 152 154 294 266 82
244 370 307 226 84 314 211 235 354 308 282 4 278 582 288 298 555 699 112 345 55 90 568
270 363 248 463 246 485 346 326 201 469 325 30 371 364 520 176 499 756 517 597 59 320 188
470 199 214 656 392 501 71 299 180 204 324 174 479 74 260 287 410 457 190 428 274 185 75
318 303 727 418 183 333 258 236 792 178 283 459 39 296 234 444 369 107 311 433 468 212
436 448 334 164 765 413 328 344 225 291 422 366 103 477 243 411 404 639 254 227 440 257
321 253 577 431 232 179 420 372 397 217 417 315 427 600 337 89 327 300 284 421 642 374
348 423 550 156 277 357 529 343 365 111 407 462 495 435 319 482 514 611 464 426 382 405
275 352 208 189 532 169 305 387 628 500 259 706 657 502 295 429 332 599 362 415 562 471
632 583 554 726 445 467 261 563 213 66 286 556 745 565 617 474 380 231 409 566 289 450
356 339 452 442 449 560 379 267 451 478 437 402 547 384 441 453 301 439 576 167 391 733
142 313 412 519 581 655 509 432 558 360 505 118 340 800 342 377 336 138 186 606 636 591
603 749 497 381 358 389 162 686 57 510 564 484 121 424 539 768 542 456 473 761 443 434
268 486 297 516 368 672 620 663 506 717 541 507 535 614 649 645 572 376 666 533 490 491
492 264 705 559 414 472 650 498 897 774 623 669 585 785 398 725 766 608 701 644 528 546
653 544 571 625 496 570 688 393 624 764 588 848 508 728 489 646 660 351 408 594 677 276
458 803 695 522 982 216 578 692 359 395 524 480 483 734 569 638 651 811 367 704 615 847
954 703 621 292 810 613 770 416 494 922 580 670 721 525 844 504 475 590 425 335 671 447
549 658 329 665 537 652 604 586 744 598 682 399 691 551 552 536 454 790 290 731 523 476
584 561 618 493 527 791 830 673 622 481 675 931 876 771 654 696 690 730 401 538 787 804
908 323 961 715 386 946 640 601 858 534 851 707 806 515 553 773 631 317 747 698 573 455
711 593 942 936 762 681 714 338 807 635 694 702 668 846 742 575 579 674 592 975 637 819
718 934 446 385 540 595 877 738 754 758 716 836 713 230 760 587 882 778 784 679 685 735
662 684 687 902 683 355 634 612 466 795 488 545 782 245 616 641 863 403 750 767 710 917
743 979 955 780 341 777 610 904 797 763 697 869 841 913 708 959 460 880 989 874 798 859
878 732 838 700 693 900 815 886 719 724 567 888 629 981 884 589 680 461 752 856 664 939
739 812 984 820 783 557 840 438 531 835 793 809 834 891 873 781 633 511 709 937 753 786
394 881 832 843 938 648 736 676 962 788 833 935 813 826 605 659 808 630 741 822 821 772
944 737 928 910 901 860 805 802 779 827 911 825 872 862 849 829 929 890 775 845 824 968
965 746 609 521 503 950 816 893 967 596 966 974 914 465 958 885 972 949 755 751 879 667
905 941 607 898 801 799 866 916 769 837 678 899 978 720 887 906 933 896 661 729 987 842
921 960 995 776 850 947 867 889 990 927 925 951 789 814 823 855 871 969 722 923 857 970
390 796 988 926 903 999 870 976 892 912 794 723 626 985 895 996 868 817 619 983 956 920
948 930 998 759 971 940 602 994 852 953 977 574 864 973 997 839 875 964 689 853 740 957
945 748 963 647 919 861 818 643 909 854 865 831 992 757 943 980 828 918 991 952 894 915
907 993 883 932 986 924 165
Number of cities traversed: 1001
Best tour value: 5242
Number of threads: 16
Elapsed time = 6.000407e+01 seconds
```

Credit Statement

All work on coding and report writeup for the ATSP problem was done by Connor Coleman

Source code

```
// Async Traveling Salesperson with Pthreads

// Compile: gcc -g -Wall -o atsp_pth atsp_pth.c -lm -lpthread
Execute: ./atsp_pth <number of threads> <seed>

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include "timer.h"
#include <limits.h>
#include <string.h>

#define MAX_THREADS 1024
#define COLS 1000
#define ROWS 1000
#define MAX_LINE_LEN 12000

int travel_matrix[COLS][ROWS];
int *global_best_tour;
int global_best_tour_value;
long thread_count;
unsigned int seed = 256;
pthread_rwlock_t rwlock_travel_matrix = PTHREAD_RWLOCK_INITIALIZER;
pthread_rwlock_t rwlock_best_tour = PTHREAD_RWLOCK_INITIALIZER;

double *non_comm_end_time;

typedef struct
{
  long rank;
  double start_time;
} pth_arg;

void Get_args(int argc, char *argv[]);
void *Estimate_pi(void *rank);
void Usage(char *prog_name);
void *Find_best_tour(void *arguments);
void Find_tour(int *test_tour, int *tour_value, int city_start);

int main(int argc, char *argv[])
{
  long thread;
  double start, finish, elapsed;
  pthread_t *thread_handles;
```

```

GET_TIME(start);

Get_args(argc, argv);

non_comm_end_time = malloc(thread_count * sizeof(double));
if (non_comm_end_time == NULL)
{
  printf("non_comm_end_time failed to allocate\n");
  exit(1); // Handle memory allocation failure
}

thread_handles = (pthread_t *)malloc(thread_count * sizeof(pthread_t));
if (thread_handles == NULL)
{
  printf("thread_handles failed to allocate\n");
  exit(1); // Handle memory allocation failure
}

pth_arg *arguments = malloc(thread_count * sizeof(pth_arg));

if (arguments == NULL)
{
  printf("Pth argument failed to allocate\n");
  exit(1); // Handle memory allocation failure
}

global_best_tour_value = INT_MAX;
global_best_tour = malloc(ROWS * sizeof(int));
if (global_best_tour == NULL)
{
  printf("global_best_tour failed to allocate\n");
  exit(1); // Handle memory allocation failure
}

// Read in matrix
FILE *matrix_file = fopen("./DistanceMatrix1000_v2.csv", "r");
if (!matrix_file)
{
  printf("Failed to load matrix file.\n");
  exit(1);
}

char temp_line[MAX_LINE_LEN];
pthread_rwlock_wrlock(&rwlock_travel_matrix);
for (int row = 0; row < ROWS; row++)
```

```
{
  fgets(temp_line, sizeof(temp_line), matrix_file);

  char *token;
  int temp_entry;
  token = strtok(temp_line, ",");

  for (int col = 0; col < COLS; col++)
  {
    temp_entry = strtol(token, NULL, 10);
    if (temp_entry > 0 && temp_entry < 1000)
    {
      travel_matrix[row][col] = temp_entry;
    }
    else
      printf("Row %d Col %d: Error, entry = %d.\n",
             row, col, temp_entry);

    token = strtok(NULL, ",");
  }
}
pthread_rwlock_unlock(&rwlock_travel_matrix);

for (thread = 0; thread < thread_count; thread++)
{
  arguments[thread].rank = thread;
  arguments[thread].start_time = start;
  pthread_create(&thread_handles[thread], NULL,
                 Find_best_tour, (void *)&arguments[thread]);
}

for (thread = 0; thread < thread_count; thread++)
  pthread_join(thread_handles[thread], NULL);

GET_TIME(finish);
elapsed = finish - start;

printf("Cities of best tour found:");
for (int i = 0; i < ROWS + 1; i++)
{
  printf(" %d", global_best_tour[i]);
}
printf("\n");
printf("Number of cities traversed: %d\n", ROWS + 1);
printf("Best tour value: %d\n", global_best_tour_value);
printf("Number of threads: %ld\n", thread_count);
```

```
  printf("Elapsed time = %e seconds\n", elapsed);
  for (int i = 0; i < thread_count; i++)
  {
    printf("Thread %d post-loop time %e\n",
           i, non_comm_end_time[i] - start);
  }

  free(thread_handles);
  free(arguments);
  free(non_comm_end_time);
  return 0;
}

void *Find_best_tour(void *arguments)
{
  pth_arg *args = (pth_arg *)arguments;
  long my_rank = (long)args->rank;
  unsigned int my_seed = seed + my_rank;
  double my_working_time;

  int *my_best_tour = malloc(ROWS * sizeof(int));
  int my_best_tour_value = INT_MAX;

  do
  {
    int city_start = rand_r(&my_seed) % ROWS;
    int *test_tour = malloc((ROWS + 1) * sizeof(int));
    int tour_value = 0;
    Find_tour(test_tour, &tour_value, city_start);

    if (tour_value < my_best_tour_value)
    {
      free(my_best_tour);
      my_best_tour = test_tour;
      my_best_tour_value = tour_value;
    }
    else
      free(test_tour);

    GET_TIME(my_working_time);
  } while (my_working_time - args->start_time < 60.0);

  non_comm_end_time[my_rank] = my_working_time;

  // grab write lock
  pthread_rwlock_wrlock(&rwlock_best_tour);
  if (my_best_tour_value < global_best_tour_value)
  {
    free(global_best_tour);
    global_best_tour_value = my_best_tour_value;
    global_best_tour = my_best_tour;
  }
  else
  {
    free(my_best_tour);
  }
  pthread_rwlock_unlock(&rwlock_best_tour);

  return NULL;
} /* Find_best_tour */

void Find_tour(int *test_tour, int *tour_value, int city_start)
{
  test_tour[0] = city_start;

  int visited[COLS] = {0};
  visited[city_start] = 1;

  *tour_value = 0;

  for (int row = 1; row < ROWS; row++)
  {
    int previous = test_tour[row - 1];
    int best = -1;
    int min_dist = INT_MAX;

    pthread_rwlock_rdlock(&rwlock_travel_matrix);
    for (int col = 0; col < COLS; col++)
    {
      if (!visited[col] && travel_matrix[previous][col] < min_dist)
      {
        min_dist = travel_matrix[previous][col];
        best = col;
      }
    }
    pthread_rwlock_unlock(&rwlock_travel_matrix);

    test_tour[row] = best;
    visited[best] = 1;
    *tour_value += min_dist;
  }
  pthread_rwlock_rdlock(&rwlock_travel_matrix);
```

```
   *tour_value += travel_matrix[test_tour[ROWS - 1]][city_start];
   pthread_rwlock_unlock(&rwlock_travel_matrix);
   test_tour[ROWS] = city_start;

} /* Find_tour */

/*---------------------------------------------------------------
 * Function:    Get_args
 * Purpose:     Get the command line args
 * In args:     argc, argv
 * Globals out: thread_count, seed
 */
void Get_args(int argc, char *argv[])
{
   if (argc != 3)
     Usage(argv[0]);
   thread_count = strtol(argv[1], NULL, 10);
   if (thread_count <= 0 || thread_count > MAX_THREADS)
     Usage(argv[0]);
   seed = strtol(argv[2], NULL, 10);
   if (seed < 0)
     Usage(argv[0]);

} /* Get_args */

/*---------------------------------------------------------------
 * Function:  Usage
 * Purpose:   Print a message explaining how to run the program
 * In arg:    prog_name
 */
void Usage(char *prog_name)
{
   fprintf(stderr, "usage: %s <number of threads> <seed>\n", prog_name);
   fprintf(stderr, "   seed is the starting seed for randomness and should be >= 0\n");
   exit(0);
} /* Usage */
```