UNIVERSITY OF TEXAS AT AUSTIN

PCSE Final Project

Cameron Carter

July 3, 2016

1 INTRODUCTION

In chess, a knight's tour is a sequence of moves made by a knight on a chessboard such that the knight visits each square only once. This project develops a code to solve the knight's tour problem on mxn rectangular boards of various sizes.

2 METHODS AND RESULTS

2.1 BRUTE FORCE

My first strategy was to do a simple breadth first search of all the possible paths of a knight (A knight can move two spaces in one direction then one space in a perpendicular direction, an "L") starting from every possible square. In order to do this, I first generated an array of integers of length mxn, where each integer identified a specific square on the board. The "id's" increased first left to right across columns, and then down across rows. I then wrote a simple function "find tours" that took as input an id number, the dimensions of the board, the number of visited square (vis), the array of id's representing the board (board), and an array (hist) to keep track of previously visited squares. Given those inputs, the function generated a list of possible moves for a knight starting at the given square id, and then looped and over the possible moves and recursed like so

```
for (k=0;k<8;k++) {
  if (moves[k] != 0) {
    find_tours(moves[k],nrow,ncol,vis+1,board,hist);
  }
}
```

while keeping track of previously visited squares with "hist", and number of moves taken with "vis", for each potential tour. Only when a given knight had visited *m*x*n* squares was the path deemed a tour. After being run in serial, the code was parallelized using a simple parallel for loop with the number of threads set as the number of squares on the board. The variable tours was declared static and private to each thread.

```
static int tours = 0;
#pragma omp threadprivate(tours)
...
int main (int argc,char *argv[]) {
...
  #pragma omp parallel num_threads(m*n)
  {
    #pragma omp for private(q,p,vis,hist) schedule(static)
    for (q=1;q<(m*n+1);q++) {
      p = find_tours(q,m,n,vis,board,hist);
      tot = tot + p;
    }
  }
```

This program was run in serial and in parallel with boards ranging in size from 3 x 4 to 5 x 6, on 4, 8 and 16 cores. with speedup results shown below.

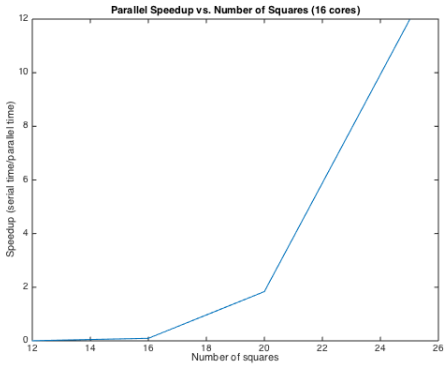

Figure 2.1: Parallel speedup vs number of squares for 16 cores. Interestingly, this was the trial with the least amount of parallel speedup, probably due to the relatively small numbers of squares that the program was able to handle.

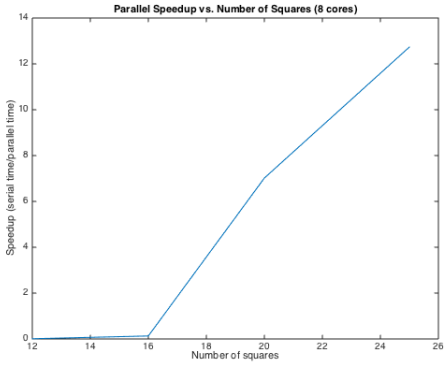

Figure 2.2: Parallel speedup vs number of squares for 8 cores. 8 cores was the sweet spot for this range of squares. It is clear that at larger m and n values, more cores started to perform better.

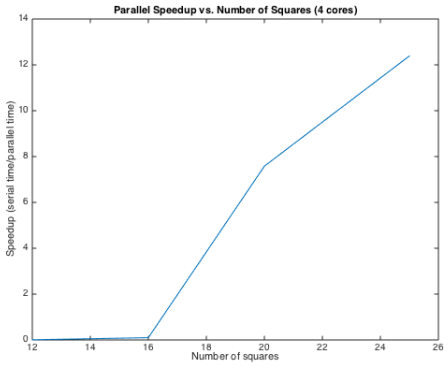

Figure 2.3: Parallel speedup vs number of squares for 4 cores.This trial vied with 8 cores for the most parallel speedup. 4 cores was more effective at smaller m and n values.

Interestingly, the greatest speedup came from 8 cores, and 16 cores was the slowest, although as the number of squares got larger, the trend favored more cores. The results are summarized below, where each time is the average of many. The parallelized code was able to solve 5 x 6 boards while the serial code was not, but neither could solve a 6 x 6 board.

	4 cores		8 cores		16 cores	
squares	serial	parallel	serial	parallel	serial	parallel
12	$1 * 10^{-5}$ s	$1.3 * 10^{-4}$ s		$1.1 * 10^{-4}$ s		$2.2 * 10^{-4}$ s
16	$4.7 * 10^{-5}$ s	$4.8 * 10^{-4}$ s		$3.6 * 10^{-4}$ s		$4.8 * 10^{-4}$ s
20	$1.1 * 10^{-3}$ s	$1.5 * 10^{-4}$ s		$1.6 * 10^{-4}$ s		$6.2 * 10^{-4}$ s
25	$8.3 * 10^{-2}$ s	$6.7 * 10^{-3}$ s		$6.5 * 10^{-3}$ s		$6.9 * 10^{-3}$ s
30		0.42 s		0.43 s		0.48 s

While this method is clearly inefficient and unsustainable, one advantage that it has is that it gives *all* possible knights tours, given that the board is sufficiently small.

2.2 Exploiting Symmetry

On boards large enough that brute force isn't an option, a more sophisticated approach must be taken. The basic idea here is to take advantage of the 90° rotational symmetry of the chess board. If we choose an arbitrary square and start moving, any path we create is identical to the three other rotated paths. At every move of the knight, not only the square that he lands on, but the three other rotationally symmetric squares can be removed from the list of available squares. When possible, the goal is to find four symmetrical paths that collectively either land on every square or all but one. This method is especially applicable to square boards (n x n) because all those with n even have a number of squares that is divisible by four, while all with n odd have (n x n)%4 = 1. I only went forward with the symmetry method for nxn boards with n odd, because to do the same with n even would have been tedious but would not require any significantly different code.

The code for the symmetry method differed from the brute force method in a few significant ways. Functions were included that take the id number of a square and convert it to a row and column index, and vice-versa, shown below

```
int iden(int row,int col,int ncol) {
  int i = (col+1) + (ncol*row);
  return i;
}
int col (int id, int ncol) {
  int c = (id-1) \%ncol;
  return c;
}
int row (int id, int ncol) {
  int r = (id-1)/ncol;
  return r;
}
```

On square boards with odd n values, there are an odd number of squares, so there is a perfectly central square with id $(n^2/2)+1$ (where / represents integer division). The "board" was created in the same way as before, in terms an array of id numbers, but then the id's were changed to rows and columns via the previously mentioned functions. The row's and column's indices were then shifted so that the central square was $(0,0)$, so that rotations about the central square could be performed using the Euclidean rotation matrix

$$R = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

which led to the following code, where $n_{col}/2$ is the value of the shift in the indices to center them about the center square.

```
int n_mid = ncol/2;
int r = row(id,ncol)-n_mid;
int c = col(id,ncol)-n_mid;
```

```
// 90 degree rotation
int r_1 = -1*c;
int c_1 = r;

// 180 degree rotation
int r_2 = -1*r;
int c_2 = -1*c;

 // -90 degree rotation
int r_3 = c;
int c_3 = -1*r;
```

After generating the symmetrical squares for each move, those squares were removed from contention before the recursive step took place. Once the program was able to generate all of the symmetric paths that were of length $(n^2/4)-1$ (the 1 representing the remaining square in nxn, n odd boards), a few more restrictions needed to be applied in order to find knight's tours. This program was to be run on 7 x 7 and larger boards. With so many possible moves, most starting squares could be expected to produce symmetric tours. It was then required that the one square that remain unvisited be the central one, and that each symmetric tour end one knight's move away from that square (In fact if one does, they all must, due to symmetry). Furthermore, each of the four symmetric tours were required to end one knight's move away from the start of a different symmetric tour. Then the four tours could be strung together, with the last one ending in the center, to create a tour of the entire board. Another way it can be done is with the central square serving as the intermediary between two of the paths, if each path is paired with another, and the movement isn't cyclic from path to path.

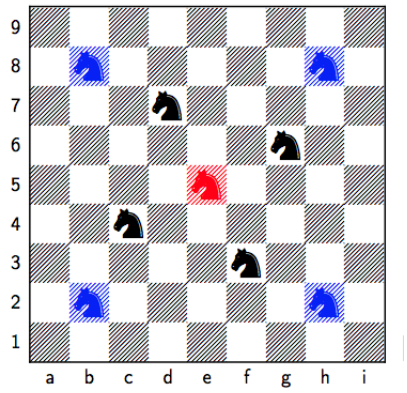

Figure 2.4: An example of the symmetry strategy for a 9x9 board. The blue squares represent the starting square and its symmetries. The central red square is untouched by any of the four symmetric paths. The intermediate white squares are the ending squares of the symmetric paths, where each symmetric path ends one knight's move away from another path's starting square.

The code was run in serial for $n = 7$ and $n = 9$. Below is a sample output for $n = 9$. Each of

the four columns represents one of the symmetric tours. Square 34 is adjacent to 17, 22 is adjacent to 11, 48 is adjacent to 65, and 60 is adjacent to 71. This is an example of a situation where the central square must come in between the four paths, because the path starting at 65 leads to the path starting at 17, which leads straight back to the original path.

```
Sym tour
65:71:17:11
76:54:6:28
69:35:13:47
80:18:2:64
63:7:19:75
70:26:12:56
81:9:1:73
62:16:20:66
51:33:31:49
44:14:38:68
27:3:55:79
8:10:74:72
15:29:67:53
4:46:78:36
21:57:61:25
40:50:42:32
59:43:23:39
52:24:30:58
45:5:37:77
34:22:48:60
```

2.3 Parallelization Strategies

I had a difficult time parallelizing the symmetry code. My first strategy was to exploit as much task parallelism as possible, as I had intended to do when I set out to write the second program, but I found that this just jammed things, making them much slower.

```
#pragma omp parallel num_threads(9)
{

  // 90 degree rotation
  #pragma omp task shared(r_1)
  r_1 = -1*c;
  #pragma omp task shared(c_1)
  c_1 = r;
  #pragma omp task shared(id_1)
  id_1 = iden(r_1+n_mid,c_1+n_mid,ncol);

  // 180 degree rotation
  #pragma omp task shared(r_2)
  r_2 = -1*r;
  #pragma omp task shared(c_2)
```

```
  c_2 = -1*c;
  #pragma omp task shared(id_2)
  id_2 = iden(r_2+n_mid,c_2+n_mid,ncol);

  // -90 degree rotation
  #pragma omp task shared(r_3)
  r_3 = c;
  #pragma omp task shared(c_3)
  c_3 = -1*r;
  #pragma omp task shared(id_3)
  id_3 = iden(r_3+n_mid,c_3+n_mid,ncol);

  #pragma omp taskwait
}
```

My code was not as amenable to tasking as I hoped it would be. I tried various other methods like parallel for loops and omp single and critical sections, but nothing worked out. The program was run in serial for $n = 7$ and $n = 9$ with 4, 8 and 16 cores to yield the following

	4 cores	8 cores	16 cores
n=7	$8.6 * 10^{-6}$ s	$8.6 * 10^{-6}$ s	$8.6 * 10^{-6}$ s
n=9	0.39 s	0.39 s	0.39 s

which is certainly not linear in n^2, probably due to an exponential buildup of overhead from my program. As one would expect, changing the number of cores has no effect. In conclusion, I was able to generate all the knight's tours for boards with up to 30 squares, and I was able to parallelize the code, and saw a greater than linear speedup. I was able to determine some knight's tours for larger boards up to 9 x 9, but was unable to see parallel speedup.

3 References

(1) I. Parberry. An Efficient Algorithm for the Knight's Tour Problem. Department of Computer Sciences, University of North Texas. Oct. 17, 1994.

(2) G. Koloskov. Intel Threading Challenge 2009 2.2 Knight's Tour (Interrupted). Intel Corporation. Sept. 25, 2009.

(3) M. Keen. The Knight's Tour. 2000.