The codes

Running the codes

Serial code

To run the serial code in this project, simply write the following in a Linux command line.

```
make serial
```

This will compile, link and execute the test program.

Parallel code

To run the parallel code of this project, simply write the following in a Linux command line.

```
make parallel
```

This will compile, link and execute the test program.

Code documentation

Extracting number of nodes and edges

To extract the number of nodes and edges from a given web-graph file, the code simply reads the first two lines to remove them. Then it uses the following code snippet to extract the number of nodes and edges:

```
fscanf(fp,  "%*s %*s %d %*s %d", N, &N_links);
```

read_graph_from_file1

To extract the values and fill the link matrix, I use the following code snippet:

```
//Fills the 2D-array with the values in the file.
int FromNodeId, ToNodeId;
for (int k = 0; k < N_links; k++){
  fscanf(fp, "%d %d", &FromNodeId, &ToNodeId);
  (*table2D)[ToNodeId][FromNodeId] = (char) 1;
}
```

read_graph_from_file2

In this function, we need to sort the data read from the web graph file in order to make the row_ptr and col_idx arrays. To this end, I've implemented a sorting algorithm known as shellsort in order to achieve a speedy implementation. First we store the data in temporary arrays as follows.

```
int FromNodeId, ToNodeId, self_link_counter;
self_link_counter = 0;
int l = 0;
for (int k = 0; k < *N_links; k++){
  fscanf(fp, "%d %d", &FromNodeId, &ToNodeId);
  if (FromNodeId != ToNodeId){
    tmp_col[l] = FromNodeId;
    tmp_row[l] = ToNodeId;
    row_count[ToNodeId] += 1;
    l++;
  }
  else{
    self_link_counter++;
  }
}
```

This part also keeps track of self-link occurrences such that these are removed from the final arrays. To create the row_ptr and prepare for creation of the sorted col_idx array, the following code is implemented:

```
//We copy FromNodeIds and ToNodeIds into arrays of the correct size to simplify sorting lat
*col_idx = (int*)calloc(*N_links, sizeof(int*));
int *row_elems = (int*)calloc(*N_links, sizeof(int));
for (int i = 0; i < *N_links; i++){
  (*col_idx)[i] = tmp_col[i];
  row_elems[i] = tmp_row[i];
}
//Now the values of interest are copied into arrays of the correct length, so we free up mem
free(tmp_col);
free(tmp_row);
```

The shellsort comes next:

```
int tmp1, tmp2, i, j, gap;
for (gap = *N_links/2; gap > 0; gap /= 2){
  for (i = gap; i < *N_links; i++){
    tmp1 = row_elems[i];
    tmp2 = (*col_idx)[i];
    for (j = i; j >= gap && row_elems[j-gap] > tmp1; j -= gap){
      row_elems[j] = row_elems[j-gap];
      (*col_idx)[j] = (*col_idx)[j-gap];
```

```
    }
    row_elems[j] = tmp1;
    (*col_idx)[j] = tmp2;
  }
}
```

count_mutual_links1

This function is split into two using the following code structure:

```
#if defined(_OPENMP)
{
  //parallelized code
}
#else
{
  //serial code
}
#endif
```

The algorithm in itself is simple, and the parallelized code is identical to the serial one except for a insertion of a pragma omp for with a private and reduction clause. The algorithm is as follows. We pick a row i, then we pick a column j. Then we traverse the row looking for instances where $A_{ij} = A_{ik} = 1$. If this is true, then we add to the number of links j and k is outbound. In the code, $i =$ inbound, $j =$ outbound1 and $k =$ outbound2, as follows.

```
for (inbound = 0; inbound < N; inbound++){
  for (outbound1 = 0; outbound1 < N; outbound1++){
    tmp = table2D[inbound][outbound1];
    for (outbound2 = outbound1 + 1; outbound2 < N; outbound2++){
      if (table2D[inbound][outbound2] == 1 && tmp == 1)
          {
            total_mutual_web_linkages++;
            num_involvements[outbound1]++;
            num_involvements[outbound2]++;
          }
    }
  }
}
```