```
Yang zheng
CSCI 347 Spring23
Assignment 6
```

```c
/* CSCI347 Spring23
 * Assignment 6
 * Modified May 27, 2023 Yang zheng
 */
#define _XOPEN_SOURCE 600
#include <stdio.h>
#include <math.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <time.h>
#include <pthread.h>
#include <sys/time.h>

typedef struct {
  int y;
  int z;
  int start_index;
  int work;
  int tNumber;
  double *a;
  double *b;
  double *c;
  pthread_barrier_t* barr;
  int times;
} Multiplier;

/* idx macro calculates the correct 2-d based 1-d index
 * of a location (x,y) in an array that has col columns.
 * This is calculated in row major order.
 */

#define idx(x,y,col)  ((x)*(col) + (y))


/* Matrix Multiply:
 *  C (x by z)  =  A ( x by y ) times B (y by z)
 *  This is the slow n^3 algorithm
 *  A and B are not be modified
 */
void fatal (long n) {
  printf ("Fatal error, lock or unlock error, thread %ld.\n", n);
  exit(n);
}

/* Print a matrix: */
void MatPrint (double *A, int x, int y)
{
  int ix, iy;

  for (ix = 0; ix < x; ix++) {
    printf ("Row %d: ", ix);
    for (iy = 0; iy < y; iy++)
      printf (" %10.5G", A[idx(ix,iy,y)]);
    printf ("\n");
  }
}
```

```c
void dot_product(Multiplier *m) {
  int index = m−>start_index;
  for (int j = 0; j < m−>work; j++) { // do work of dot product
    double tval = 0;
    int row = index / m−>z;
    int col = index % m−>z;
    for (int i = 0; i < m−>y; i++) { //  dot product
      double a = m−>a[idx(row, i, m−>y)];
      double b = m−>b[idx(i, col, m−>z)];
      tval += (a * b);
    }
    m−>c[idx(row, col, m−>z)] = tval;
    index++;
  }
}

void * mul_main(void *mm) {
  Multiplier* m = (Multiplier*)mm;
  dot_product(m);
  pthread_exit(NULL);
}

void * square_main(void *mm) {;
  Multiplier* m = (Multiplier*)mm;
  double *T;
  T = m−>b;
  m−>b = m−>a;
  dot_product(m);

  if (m−>times > 1) {
    pthread_barrier_wait(m−>barr);
    m−>a = m−>c;
    m−>b = m−>c;
    m−>c = T;
    for (int i = 1; i < m−>times; i++) {
      dot_product(m);
      pthread_barrier_wait(m−>barr);
      m−>a = m−>c;
      m−>c = m−>b;
      m−>b = m−>a;
    }
  }
  pthread_exit(NULL);
}

void create_thread(Multiplier *m, pthread_t *threads, double *A, double *B, doub
le *C, int x, int y, int z, int nThread, void *(*start_routine)(void *)) {
  int work = (x * z) / nThread; //  the # of values that each thread does
  int do_extra_work = (x * z) % nThread; //  first # of threads that compute one
 extra value
  if (B == NULL) {
    B = (double*)malloc(sizeof(double) * y * z);
  }
  for (int i = 0; i < nThread; i++) {
    m[i].a = A;
    m[i].b = B;
    m[i].c = C;
    m[i].y = y;
    m[i].z = z; //  new matrix is x by z
    m[i].tNumber = i;
    m[i].start_index = i * work + (i < do_extra_work ? i : do_extra_work);
    m[i].work = work + (i < do_extra_work ? 1 : 0);
    pthread_create(&threads[i], NULL, start_routine, (void*)&m[i]);
  }
}

void MatMul (double *A, double *B, double *C, int x, int y, int z, int nThread)
{
  pthread_t mThreads[nThread];
  Multiplier mm[nThread];
  create_thread(mm, mThreads, A, B, C, x, y, z, nThread, mul_main);
  for (int i = 0; i < nThread; i++) {
    pthread_join(mThreads[i], NULL);
  }
}

/* Matrix Square:
 *  B = A ^ 2*times
 *
 *    A are not be modified.
 */

void MatSquare (double *A, double *B, int x, int times, int nThread)
{
  pthread_barrier_t barr;
  pthread_t sThreads[nThread];
  Multiplier ms[nThread];
  pthread_barrier_init(&barr, NULL, nThread);

  for (int i = 0; i < nThread; i++) {
    ms[i].barr = &barr;
    ms[i].times = times;
  }
  create_thread(ms, sThreads, A, NULL, B, x, x, x, nThread, square_main);
  for (int i = 0; i < nThread; i++) {
    pthread_join(sThreads[i], NULL);
  }
  if (times % 2 == 0) {
    memcpy(B, ms[0].b, sizeof(double)*x*x);
  }
  pthread_barrier_destroy(&barr);
}

/* Generate data for a matrix: */
void MatGen (double *A, int x, int y, int rand)
{
  int ix, iy;

  for (ix = 0; ix < x ; ix++) {
    for (iy = 0; iy < y ; iy++) {
      A[idx(ix,iy,y)] = ( rand ?
                          ((double)(random() % 200000000))/2000000000.0 :
                          (1.0 + (((double)ix)/100.0)
                           + (((double)iy/1000.0))));
    }
  }
}
```

```c
/* Print a help message on how to run the program */

void usage(char *prog)
{
  fprintf (stderr, "%s: [−dr] −x val −y val −z val −n num of threads\n", prog);
  fprintf (stderr, "%s: [−dr] −s num −x val −n num of threads\n", prog);
  exit(1);
}

/* Main function
 *
 *  args:  −d   −− debug and print results
 *         −r   −− use random data between 0 and 1
 *         −s t −− square the matrix t times
 *         −x   −− rows of the first matrix, r & c for squaring
 *         −y   −− cols of A, rows of B
 *         −z   −− cols of B
 *         −n   −− # of threads
 *
 */

int main (int argc, char ** argv)
{
  extern char *optarg;   /* defined by getopt(3) */
  int ch;                /* for use with getopt(3) */

  /* option data */
  int x = 0, y = 0, z = 0;
  int debug = 0;
  int square = 0;
  int useRand = 0;
  int sTimes = 0;
  int num_threads = 8;
  int reportTime = 0;

  while ((ch = getopt(argc, argv, "drs:x:y:z:n:T")) != −1) {
    switch (ch) {
    case ’d’:  /* debug */
      debug = 1;
      break;
    case ’r’:  /* debug */
      useRand = 1;
      srandom(time(NULL));
      break;
    case ’s’:  /* s times */
      sTimes = atoi(optarg);
      square = 1;
      break;
    case ’x’:  /* x size */
      x = atoi(optarg);
      break;
    case ’y’:  /* y size */
      y = atoi(optarg);
      break;
    case ’z’:  /* z size */
      z = atoi(optarg);
      break;
    case ’n’:
      num_threads = atoi(optarg);
      break;
    case 'T':
      reportTime = 1;
      break;
    case '?': /* help */
    default:
      usage(argv[0]);
    }
  }

  /* verify options are correct. */
  if (square) {
    if (y != 0 || z != 0 || x <= 0 || sTimes < 1) {
      fprintf (stderr, "Inconsistent options\n");
      usage(argv[0]);
    }
  } else if (x <= 0 || y <= 0 || z <= 0 || num_threads <= 0) {
    fprintf (stderr, "-x, -y, -z, -n all need"
             " to be specified or -s and -x.\n");
    usage(argv[0]);
  }

  /* Matrix storage */
  double *A;
  double *B;
  double *C;
  double cpu_time;
  double elapsed_time;
  struct timeval start, end;
  clock_t cStart, cEnd;

  if (square) {
    A = (double *) malloc (sizeof(double) * x * x);
    B = (double *) malloc (sizeof(double) * x * x);
    MatGen(A,x,x,useRand);
    cStart = clock();
    gettimeofday(&start, NULL);
    MatSquare(A, B, x, sTimes, num_threads);
    gettimeofday(&end, NULL);
    cEnd = clock();
    elapsed_time = (end.tv_sec - start.tv_sec) * 1.0;
    elapsed_time += (end.tv_usec - start.tv_usec) / 1000000.0;
    cpu_time = ((double) cEnd - cStart) / CLOCKS_PER_SEC;

    if (debug) {
      printf ("-------------- orignal matrix ------------------\n");
      MatPrint(A,x,x);
      printf ("--------------  result matrix ------------------\n");
      MatPrint(B,x,x);
    }
  } else {
    A = (double *) malloc (sizeof(double) * x * y);
    B = (double *) malloc (sizeof(double) * y * z);
    C = (double *) malloc (sizeof(double) * x * z);
    MatGen(A,x,y,useRand);
    MatGen(B,y,z,useRand);
    cStart = clock();
    gettimeofday(&start, NULL);
    MatMul(A, B, C, x, y, z, num_threads);
    gettimeofday(&end, NULL);
    cEnd = clock();
    elapsed_time = (end.tv_sec - start.tv_sec) * 1.0;
    elapsed_time += (end.tv_usec - start.tv_usec) / 1000000.0;
    cpu_time = ((double) cEnd - cStart) / CLOCKS_PER_SEC;

    if (debug) {
      printf ("−−−−−−−−−−−−−− orignal A matrix −−−−−−−−−−−−−−−−−−\n");
      MatPrint(A,x,y);
      printf ("−−−−−−−−−−−−−− orignal B matrix −−−−−−−−−−−−−−−−−−\n");
      MatPrint(B,y,z);
      printf ("−−−−−−−−−−−−−−  result C matrix −−−−−−−−−−−−−−−−−−\n");
      MatPrint(C,x,z);
    }
  }
  if (reportTime) {
    printf("CPU time: %.5f seconds, elapsed time: %.5f seconds\n", cpu_time, elapsed_time);
  }
  return 0;
}
```

```c
/* Simple matrix multiply program
 *
 * Phil Nelson, March 5, 2019
 *
 */

#include <stdio.h>
#include <math.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <sys/time.h>
#include <time.h>

/* idx macro calculates the correct 2-d based 1-d index
 * of a location (x,y) in an array that has col columns.
 * This is calculated in row major order.
 */

#define idx(x,y,col)  ((x)*(col) + (y))

/* Matrix Multiply:
 *  C (x by z)  =  A ( x by y ) times B (y by z)
 *  This is the slow n^3 algorithm
 *  A and B are not be modified
 */

void MatMul (double *A, double *B, double *C, int x, int y, int z)
{
  int ix, jx, kx;

  for (ix = 0; ix < x; ix++) {
    // Rows of solution
    for (jx = 0; jx < z; jx++) {
      // Columns of solution
      double tval = 0;
      for (kx = 0; kx < y; kx++) {
        // Sum the A row time B column
        tval += A[idx(ix,kx,y)] * B[idx(kx,jx,z)];
      }
      C[idx(ix,jx,z)] = tval;
    }
  }
}

/* Matrix Square:
 *  B = A ^ 2*times
 *
 *    A are not be modified.
 */

void MatSquare (double *A, double *B, int x, int times)
{
  int i;

  MatMul (A, A, B, x, x, x);
  if (times > 1) {
    /* Need a Temporary for the computation */
    double *T = (double *)malloc(sizeof(double)*x*x);
    for (i = 1; i < times; i+= 2) {
      MatMul (B, B, T, x, x, x);
      if (i == times - 1)
        memcpy(B, T, sizeof(double)*x*x);
      else
        MatMul (T, T, B, x, x, x);
    }
    free(T);
  }
}

/* Print a matrix: */
void MatPrint (double *A, int x, int y)
{
  int ix, iy;

  for (ix = 0; ix < x; ix++) {
    printf ("Row %d: ", ix);
    for (iy = 0; iy < y; iy++)
      printf (" %10.5G", A[idx(ix,iy,y)]);
    printf ("\n");
  }
}


/* Generate data for a matrix: */
void MatGen (double *A, int x, int y, int rand)
{
  int ix, iy;

  for (ix = 0; ix < x ; ix++) {
    for (iy = 0; iy < y ; iy++) {
      A[idx(ix,iy,y)] = ( rand ?
                         ((double)(random() % 200000000))/2000000000.0 :
                         (1.0 + (((double)ix)/100.0)
                          + (((double)iy/1000.0))));
    }
  }
}

/* Print a help message on how to run the program */

void usage(char *prog)
{
  fprintf (stderr, "%s: [-dr] -x val -y val -z val\n", prog);
  fprintf (stderr, "%s: [-dr] -s num -x val\n", prog);
  exit(1);
}


/* Main function
 *
 *  args:  -d   -- debug and print results
 *         -r   -- use random data between 0 and 1
 *         -s t -- square the matrix t times
 *         -x   -- rows of the first matrix, r & c for squaring
 *         -y   -- cols of A, rows of B
 *         -z   -- cols of B
 *
 */

int main (int argc, char ** argv)
{
  extern char *optarg;   /* defined by getopt(3) */
  int ch;                /* for use with getopt(3) */

  /* option data */
  int x = 0, y = 0, z = 0;
  int debug = 0;
  int square = 0;
  int useRand = 0;
  int sTimes = 0;
  int reportTime = 0;

  while ((ch = getopt(argc, argv, "drs:x:y:z:T")) != -1) {
    switch (ch) {
    case 'd':  /* debug */
      debug = 1;
      break;
    case 'r':  /* debug */
      useRand = 1;
      srandom(time(NULL));
      break;
    case 's':  /* s times */
      sTimes = atoi(optarg);
      square = 1;
      break;
    case 'x':  /* x size */
      x = atoi(optarg);
      break;
    case 'y':  /* y size */
      y = atoi(optarg);
      break;
    case 'z':  /* z size */
      z = atoi(optarg);
      break;
    case 'T':
      reportTime = 1;
      break;
    case '?': /* help */
    default:
      usage(argv[0]);
    }
  }

  /* verify options are correct. */
  if (square) {
    if (y != 0 || z != 0 || x <= 0 || sTimes < 1) {
      fprintf (stderr, "Inconsistent options\n");
      usage(argv[0]);
    }
  } else if (x <= 0 || y <= 0 || z <= 0) {
    fprintf (stderr, "-x, -y, and -z all need"
             " to be specified or -s and -x.\n");
    usage(argv[0]);
  }
```

```c
  /* Matrix storage */
  double *A;
  double *B;
  double *C;
  double cpu_time;
  double elapsed_time;
  struct timeval start, end;
  clock_t cStart, cEnd;

  if (square) {
    A = (double *) malloc (sizeof(double) * x * x);
    B = (double *) malloc (sizeof(double) * x * x);
    MatGen(A,x,x,useRand);
    cStart = clock();
    gettimeofday(&start, NULL);
    MatSquare(A, B, x, sTimes);
    gettimeofday(&end, NULL);
    cEnd = clock();
    MatSquare(A, B, x, sTimes);
    elapsed_time = (end.tv_sec - start.tv_sec) * 1.0;
    elapsed_time += (end.tv_usec - start.tv_usec) / 1000000.0;
    cpu_time = ((double) cEnd - cStart) / CLOCKS_PER_SEC;
    if (debug) {
      printf ("−−−−−−−−−−−−−− orignal matrix −−−−−−−−−−−−−−−−−−\n");
      MatPrint(A,x,x);
      printf ("−−−−−−−−−−−−−−  result matrix −−−−−−−−−−−−−−−−−−\n");
      MatPrint(B,x,x);
    }
  } else {
    A = (double *) malloc (sizeof(double) * x * y);
    B = (double *) malloc (sizeof(double) * y * z);
    C = (double *) malloc (sizeof(double) * x * z);
    MatGen(A,x,y,useRand);
    MatGen(B,y,z,useRand);
    cStart = clock();
    gettimeofday(&start, NULL);
    MatMul(A, B, C, x, y, z);
    gettimeofday(&end, NULL);
    cEnd = clock();
    elapsed_time = (end.tv_sec - start.tv_sec) * 1.0;
    elapsed_time += (end.tv_usec - start.tv_usec) / 1000000.0;
    cpu_time = ((double) cEnd - cStart) / CLOCKS_PER_SEC;
    if (debug) {
      printf ("−−−−−−−−−−−−−− orignal A matrix −−−−−−−−−−−−−−−−−−\n");
      MatPrint(A,x,y);
      printf ("−−−−−−−−−−−−−− orignal B matrix −−−−−−−−−−−−−−−−−−\n");
      MatPrint(B,y,z);
      printf ("−−−−−−−−−−−−−−  result C matrix −−−−−−−−−−−−−−−−−−\n");
      MatPrint(C,x,z);
    }
  }
  if (reportTime) {
    printf("CPU time: %.5f seconds, elapsed time: %.5f\n", cpu_time, elapsed_time);
  }
  return 0;
}
```

Assignment 6 Report
Author: Yang Zheng

--Pthread Implementation & Speed up--
The threaded versions of matrix multiplication and matrix squaring accept the -n flag, which denotes the number of threads that will be employed. Work for each thread is divided based on the dimensions of the resultant matrix.

A Multiplier struct has been developed to hold all the necessary information for multiplication and squaring: the matrix dimensions, the matrix to be multiplied, the resultant matrix, the task each thread executes, the starting index of each thread, the barrier variable, and the squaring times.

Matrix multiplication and squaring both use the same function to create threads, however, they call two distinct thread main functions. The thread main function of matrix multiplication invokes the dot_product function, which calculates the dot product of rows and columns to populate each entry in the resultant matrix. The dot_product function, through the index of the entry it is to fill, discerns which row and column to utilize.

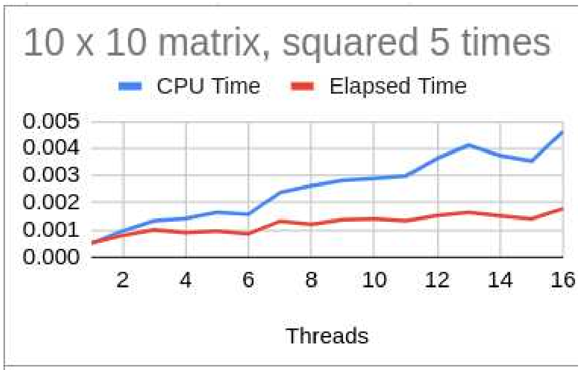
Matrix squaring, similar to matrix multiplication, calls the square thread main function. In square main, we square matrix A a predetermined number of times (s) and store the result in matrix B. To reuse the dot_product function for multiplication, we employ three matrices for the squaring process in dot_product: A(a field in struct), A(b field in struct), and B(c field in struct). The resultant matrix should ideally be stored in the c field of the Multiplier struct.

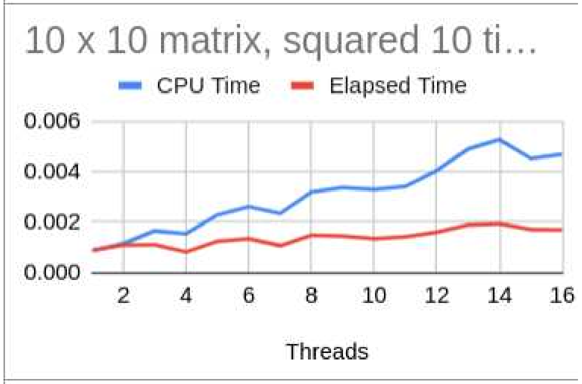
In theory, by leveraging multi-threading, the matrix multiplication (or squaring) time should decrease nearly linearly with the increase in threads, subject to hardware limitations.

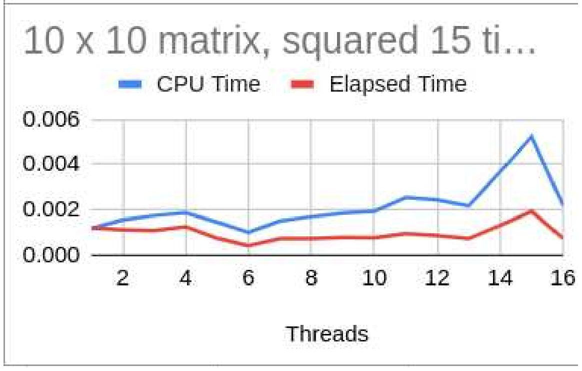
--Results & Analysis--
For matrix squaring analysis, I began with small matrices. I divided the work into three groups, each tasked with squaring 5, 10, and 15 times. For each group, the task was run using 1 to 16 threads, and each thread count was executed 10 times to compute the average CPU time and elapsed time. The results were recorded in a txt file.

The graph depicts that for small matrices, the process speed doesn't increase proportionally with the number of threads. For instance, using two threads doesn't halve the computation time. However, there's still a discernible speed increase, which becomes more pronounced as the thread count rises.

The potential reason could be that the time required to create, manage, and eventually join threads may overshadow the actual computation time for small matrices. This overhead includes time taken to synchronize between threads, which may make the parallel version slower than the sequential one if the actual computation time is minimal.

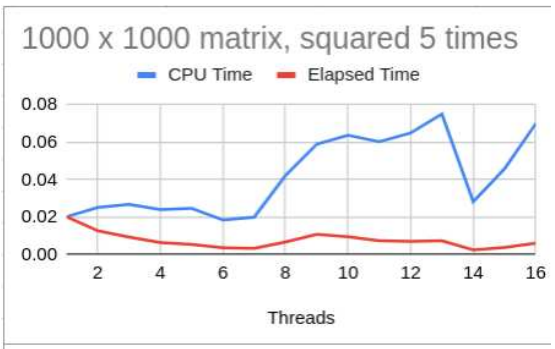


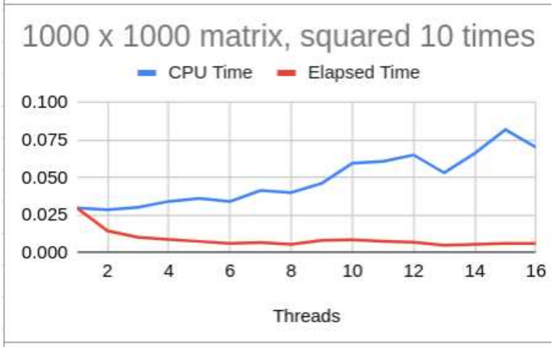


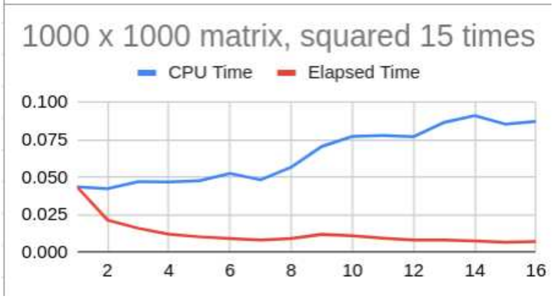


For larger matrices, like 1000 x 1000, the results aligned more closely with expectations. The process speed appears proportional to the number of threads initially, but the increase in speed starts diminishing after 8 or 9 threads. Nonetheless, there's still a general trend of increasing speed with more threads, albeit not strictly proportional.

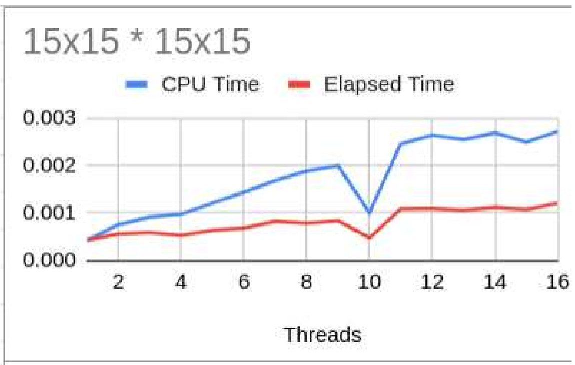
This could be due to the time taken to manage threads outweighing the benefits of parallelism when the thread count is high. Alternatively, when the number of threads exceeds the number of cores, cores must be shared among threads, potentially diminishing performance.

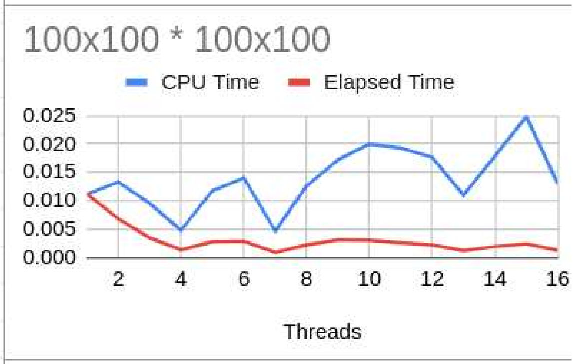


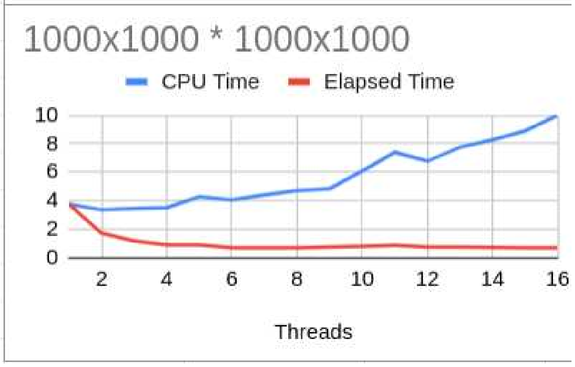




For matrix multiplication analysis, I used three different matrix sizes: 15x15, 100x100, and 1000x1000. Each matrix multiplication was executed 10 times to compute average CPU and elapsed times, enhancing data accuracy. The results show a general trend of decreasing elapsed time with increasing thread counts up to 4 threads, beyond which the elapsed time change becomes negligible. However, CPU time consistently increases, with the rate of change also rising with more threads.







-Conclusion--

This study demonstrates the potential of multi-threading to significantly speed up computationally intensive tasks, such as matrix multiplication and squaring, especially when dealing with larger data structures. The trend observed indicates that as the number of threads increases, there is a substantial decrease in processing time.

However, it's important to note that the efficiency gain is not always proportional to the number of threads used, and there are diminishing returns after a certain point. This could be attributed to the overhead associated with managing additional threads and the hardware constraints of the system, particularly when the number of threads exceeds the number of processor cores.

Additionally, for smaller matrices, the overhead of creating and managing multiple threads can even result in slower performance than sequential processing. Thus, multi-threading provides the most benefit when applied to larger, more complex computations, where the time spent on calculations significantly exceeds the overhead of thread management.

Therefore, while multi-threading is a powerful tool for improving performance, it requires a nuanced understanding of both the computational task at hand and the system's hardware capabilities to fully leverage its potential. Future work could further explore optimizing the trade-off between thread creation overhead and computational speed-up, potentially making multi-threading more beneficial across a wider range of tasks and matrix sizes.