```c
/*****************************************************************************
Authors  : Albert Owusu-Asare , Box 4497, <owusuasa@grinnell.edu>
         : Ezra Edgerton      , Box 3503, <edgerton@grinnell.edu>

This document contains answers to questions from CS213 Lab: Threads


Questions:
http://www.cs.grinnell.edu/~weinman/courses/CSC213/2014F/labs/threads.html

 modified code has the MODIFIED keyword
Original author is cited below :
 * Jerod Weinman
 * 21 May 2008
*****************************************************************************/

#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "matrix.h"
#include "matrixop.h"

#define SUB2IND(row,col,numcols) (row) * (numcols) + (col)

/* Check the dimensions of three matrices for the sum C=A+B
 *
 * Returns -1 if there is a dimension mismatch, and zero otherwise. */

int mtxCheckAddDim(const struct matrix_t *a, const struct matrix_t *b,
                        const struct matrix_t *c)
{

     if (a->rows != b->rows || a->rows != c->rows  ||
         a->cols != b->cols || a->cols != c->cols ) {
          return -1;
     }

     return 0;

}


/* Matrix addition C = A + B
 *
 * Preconditions:
 *   Matrix parameters a,b and c all have the same dimension
 *
 * Postconditions:
 *   The resulting matrix sum is stored in parameter c
 *   Return value of 0 indicates successful completion of the addition
 */
int mtxAdd( const struct matrix_t *a, const struct matrix_t *b,
                  struct matrix_t *c) {
     /* Make sure these matrices are "addable" */
     int res = mtxCheckAddDim(a,b,c);

     if (res<0) {
          fprintf(stderr,"Dimension mismatch for matrix add.\n");
          return res;
     }

     int m = a->rows;        /* Dimensions of input matrices */
     int n = a->cols;

     MTX_TYPE *pa, *pb, *pc; /* Pointers to matrix data */

     pa = a->data;           /* Initialize pointers to beginning of data */
     pb = b->data;
     pc = c->data;

     /* Iterate over every entry in the matrices, adding and storing the result */
     int i,j;
     for (i=0 ; i<m ; i++)
          for (j=0 ; j<n ; j++, pa++,pb++,pc++)
               *pc = *pa + *pb;

     return 0; /* Indicate successful completion */
}



/* MODIFIED : This is an additional piece to the original
 * Check the dimensions of three matrices for the sum C=AB
 *
 * Returns -1 if there is a dimension mismatch, and zero otherwise. */

int mtxCheckMultiplyDim(const struct matrix_t *a, const struct matrix_t *b,
                        const struct matrix_t *c)
{

     if (a->cols != b->rows || a->rows != c->rows
         || b->cols != c->cols ) {
          return -1;
     }

     return 0;

}



/*
 * MODIFIED
 * Matrix multiplication C = AB
 *
 * Preconditions:
 *   Matrix parameters a,b,c have the required dimension.
    mtxCheckMultiplyDim(a,b,c) returns >0
 *
 * Postconditions:
 *   The resulting matrix sum is stored in parameter c
 *   Return value of 0 indicates successful completion of the multiplication
 */

int mtxMultiplyMin( const struct matrix_t *a, const struct matrix_t *b,
                    struct matrix_t *c)
{
     /* Make sure these matrices are "multiply-able" */
     int res = mtxCheckMultiplyDim(a,b,c);

     if (res<0) {
          fprintf(stderr,"Dimension mismatch for matrix add.\n");
          return res;
     }

     int m = a->rows;        /* Dimensions of input matrices */
     int n = a->cols;
     int p = b->cols;


     MTX_TYPE *pa, *pb, *pc; /* Pointers to matrix data */

     pa = a->data;           /* Initialize pointers to beginning of data */
     pb = b->data;
```

```
    pc = c->data;



    /* Iterate over every entry in the matrices, adding and storing the result */
    int i,j,k, l;

    /* initialise elements of matrix c to 0*/

    for (l = 0; l < (m*p); l++, pc++)
      {
        *pc = 0;
      }
    pc = c->data; // point back to start of c matrix


    for (i = 0; i < m; i++,  pc += SUB2IND(1,0,p), pb = b->data)
      //i loops through the rows of a and the rows of c
      {
        for (k = 0; k < n; k++, pa++, pc = pc - p)
          //k loops through the columns of a and the rows of b
          {
            for (j = 0; j < p; j++,pb++,pc++)
              //the columns of b and the columns of c
              {
                *pc += *pa * *pb;

              }
          }
      }

    return 0; /* Indicate successful completion */
}
/*
 * MODIFIED
 * Does the matrix multiplication work for one thread
 * preconditions:  none
 * postconditions : matrix c is modified such that each of its values agrees with m
atrix
 * multiplication of c = a * b.
 *
 * much of the code involving the creation and running of threads was taken from th
e POSIX
 * Threads Programming tutorial:
 * https://chttps://computing.llnl.gov/tutorials/pthreads/#Joining
 * computing.llnl.gov/tutorials/pthreads/#PassingArguments
 */

void* threadMtxMultiplyMin(void *multParam)
{
  struct matrixThreadParam_t  *data;
  data =  ( struct matrixThreadParam_t *) multParam ;

  /* Dimensions of input matrices */
  int m = data->a->rows;
  int n = data->a->cols;
  int p = data->b->cols;


  int stride = data->numThreads;
  int ID = data->threadId;

/* Pointers to matrix data */
  MTX_TYPE *pa, *pb, *pc;

/* Initialize pointers to beginning of data */
  pa = data->a->data;
  pb = data->b->data;
  pc = data->c->data;

  /* Iterate over every entry in the matrices, adding and storing the result */
  int i,j,k;

  pc += SUB2IND(ID, 0, p);
  pa += SUB2IND(ID, 0, n);


  for (i = ID; i < m;
       i= i + stride, pc += SUB2IND(stride, 0, p), pb = data->b->data,
         pa += SUB2IND(stride-1, 0 , n))
    //i loops through the rows of a and the rows of c
    {

      for (k = 0; k < n; k++, pa++, pc = pc - p)
        //k loops through the columns of a and the rows of b
        {
          for (j = 0; j < p; j++,pb++,pc++)
            //the columns of b and the columns of c
            {
              *pc += *pa * *pb;

            }
        }
    }

  return NULL; /* Indicate successful completion */
}

/*
 * MODIFIED
 * Computes the threaded concurrent matrix product.
 * preconditions:  none
 * postconditions : matrix c is modified such that each of its values agrees with m
atrix
 * multiplication of c = a * b.
 *
 * much of the code involving the creation and running of threads was taken from th
e POSIX
 * Threads Programming tutorial:
 * https://chttps://computing.llnl.gov/tutorials/pthreads/#Joining
 * computing.llnl.gov/tutorials/pthreads/#PassingArguments
 */

int parMtxMultiply( const struct matrix_t *a,
                    const struct matrix_t *b,
                    struct matrix_t *c,
                    int numThreads)
{
  //checks to see if matrixes are multipliable
 int res = mtxCheckMultiplyDim(a,b,c);

     if (res<0) {
          fprintf(stderr,"Dimension mismatch for matrix add.\n");
          return res;
     }

     /* initialise elements of matrix c to 0*/
     int l;
     MTX_TYPE *pc;
     pc = c->data;

     for (l = 0; l < (a->rows * b->cols); l++, pc++)
       {
         *pc = 0;
       }
     pc = c->data; // point back to start of c matrix
```

```
    //creates threads according to numThreads
    int t;
    int rc; //gets return value when thread is created and when thread is joined
    struct matrixThreadParam_t thread_matrix_array[numThreads];
    pthread_t threads[numThreads];

    pthread_attr_t attr;
    void *status;

    //initialization of attr to make threads joinable
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

    for (t = 0; t < numThreads; t++)
      {
        //set elements of thread_array[t]
        thread_matrix_array[t].a = a;
        thread_matrix_array[t].b = b;
        thread_matrix_array[t].c = c;
        thread_matrix_array[t].numThreads = numThreads;
        thread_matrix_array[t].threadId = t;
        rc = pthread_create(&threads[t], &attr, threadMtxMultiplyMin,
                            (void *) &thread_matrix_array[t]);
        if (rc) {
          fprintf (stderr, "Could not create thread %d\n", t);
          return EXIT_FAILURE;
        }
      }
    //join threads
    pthread_attr_destroy(&attr);
    for (t = 0; t < numThreads; t++)
      {
        rc =  pthread_join(threads[t], &status);
        if (rc)
          {
            fprintf(stderr, "Thread %d could not join\n", t);
            return EXIT_FAILURE;
          }
      }
    return 0;
}
```

```
/*****************************************************************************
Authors  : Albert Owusu-Asare , Box 4497, <owusuasa@grinnell.edu>
         : Ezra Edgerton      , Box 3503, <edgerton@grinnell.edu>

This document contains answers to questions from CS213 Lab: Threads


Questions:
http://www.cs.grinnell.edu/~weinman/courses/CSC213/2014F/labs/threads.html

 modified code has the MODIFIED keyword

 * Created by Jerod Weinman, 21 May 2008



*****************************************************************************/

#ifndef __MATRIXOP_H__
#define __MATRIXOP_H__

#include "matrix.h"

/* Matrix addition C = A + B
 *
 * Preconditions:
 *   Matrix parameters a,b and c all have the same dimension
 *
 * Postconditions:
 *   The resulting matrix sum is stored in parameter c
 *   Return value of 0 indicates successful completion of the addition
 */
int mtxAdd( const struct matrix_t *a, const struct matrix_t *b,
                  struct matrix_t *c );



/*
 * MODIFIED
 * Matrix multiplication C = AB
 *
 * Preconditions:
 *   Matrix parameters a,b,c have the required dimenetion.
    mtxCheckMultiplyDim(a,b,c) returns >0
 *
 * Postconditions:
 *   The resulting matrix sum is stored in parameter c
 *   Return value of 0 indicates successful completion of the multiplication
 */

int mtxMultiplyMax( const struct matrix_t *a, const struct matrix_t *b,
                 struct matrix_t *c);

int mtxMultiplyMin( const struct matrix_t *a, const struct matrix_t *b,
                 struct matrix_t *c);


/*
 *MODIFIED
 * struct code taken from Jerod Weinman's CSC213 Threads lab:
 * http://www.cs.grinnell.edu/~weinman/courses/CSC213/2014F/labs/threads.html

 * matrixThreadParam_t contains the parameters needed for a threaded matrix
 * product.
 */
struct matrixThreadParam_t {
  const struct matrix_t *a;
  const struct matrix_t *b;
  struct matrix_t *c;
  int numThreads;
  int threadId;
};

/*
 * MODIFIED
 * Computes the threaded concurrent matrix product.
 * preconditions:  none
 * postconditions : matrix c is modified such that each of its values agrees with m
atrix
 * multiplication of c = a * b.
 *
 * much of the code involving the creation and running of threads was taken from th
e POSIX
 * Threads Programming tutorial:
 * https://chttps://computing.llnl.gov/tutorials/pthreads/#Joining
 * computing.llnl.gov/tutorials/pthreads/#PassingArguments
 */

int parMtxMultiply( const struct matrix_t *a,
                    const struct matrix_t *b,
                    struct matrix_t *c,
                    int numThreads);

/*
 * MODIFIED
 * Does the matrix multiplication work for one thread
 * preconditions:  none
 * postconditions : matrix c is modified such that each of its values agrees with m
atrix
 * multiplication of c = a * b.
 *
 * much of the code involving the creation and running of threads was taken from th
e POSIX
 * Threads Programming tutorial:
 * https://chttps://computing.llnl.gov/tutorials/pthreads/#Joining
 * computing.llnl.gov/tutorials/pthreads/#PassingArguments
 */

  void* threadMtxMultiplyMin(void *multParam);

#endif
```

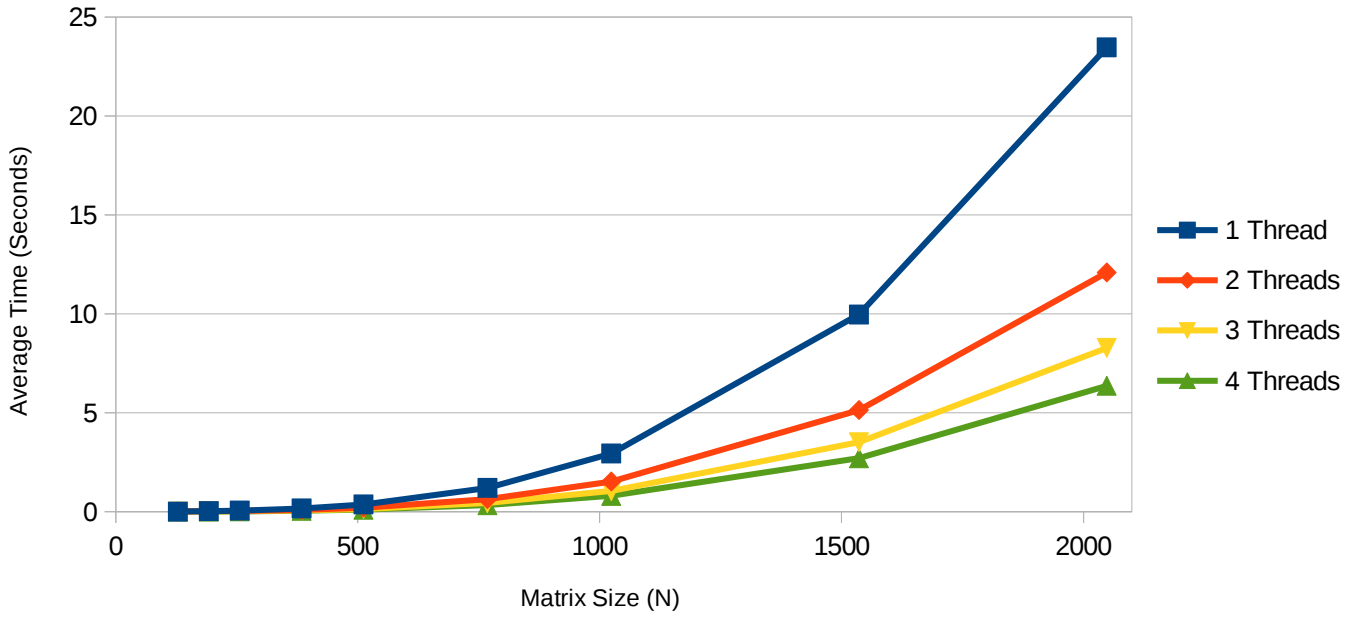
```
Script started on Tue 14 Oct 2014 06:45:13 PM CDT
burroughs$ make
gcc -g -Wall -D_GNU_SOURCE -c matrixop.c
burroughs$ make pt\007imemultiply
gcc -g -Wall -D_GNU_SOURCE -c matrixop.c
gcc -g -Wall -D_GNU_SOURCE -c matrixutil.c
gcc -g -Wall -D_GNU_SOURCE -pthread -o ptimemultiply \
        matrixop.o matrixutil.o ptimemultiply.c
burroughs$ pt\033[K033[K\007\007\007\007exit

Script done on Tue 14 Oct 2014 06:45:28 PM CDT
```

Data from running :

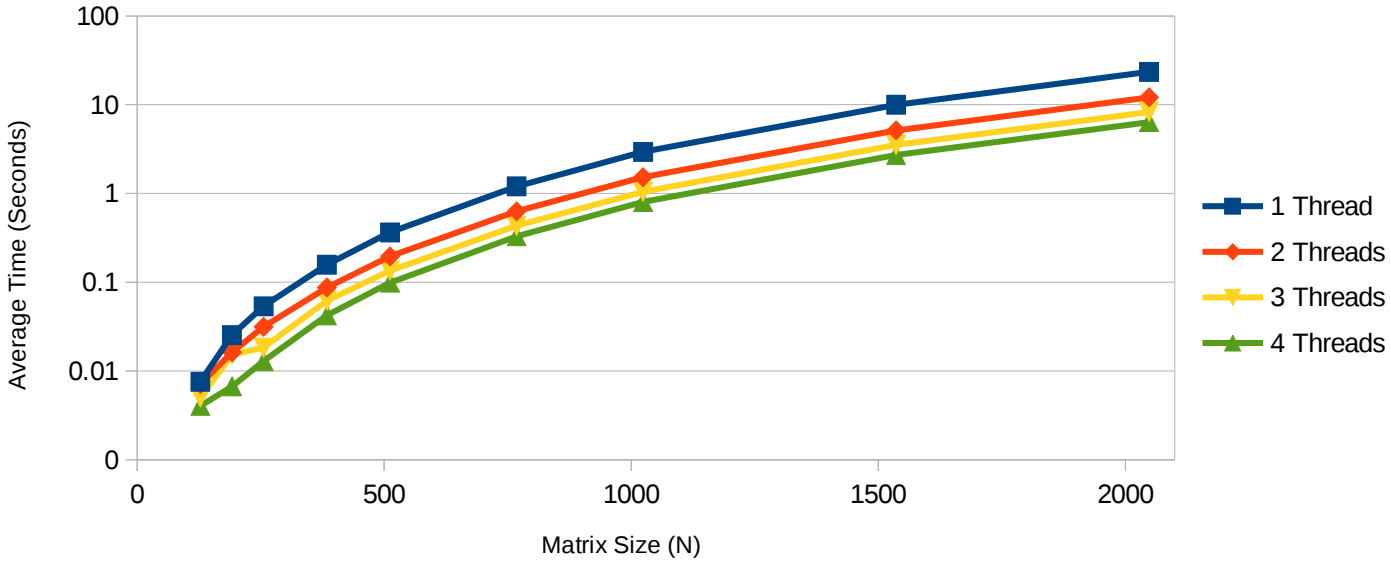
```
for n in 128 192 256 384 512 768 1024 1536 2048;
 do     echo -ne $nt >> mtxMultiplyProfile;
 ./ptimemultiply $n 1 >> mtxMultiplyProfile; done

for n in 128 192 256 384 512 768 1024 1536 2048;
  do     echo -ne $nt >> mtxMultiplyProfile;
  ./ptimemultiply $n 2 >> mtxMultiplyProfile; done

for n in 128 192 256 384 512 768 1024 1536 2048;
  do     echo -ne $nt >> mtxMultiplyProfile;
  ./ptimemultiply $n 3 >>  mtxMultiplyProfile; done

for n in 128 192 256 384 512 768 1024 1536 2048;
 do     echo -ne $nt >> mtxMultiplyProfile;
 ./ptimemultiply $n 4 >> mtxMultiplyProfile; done
```

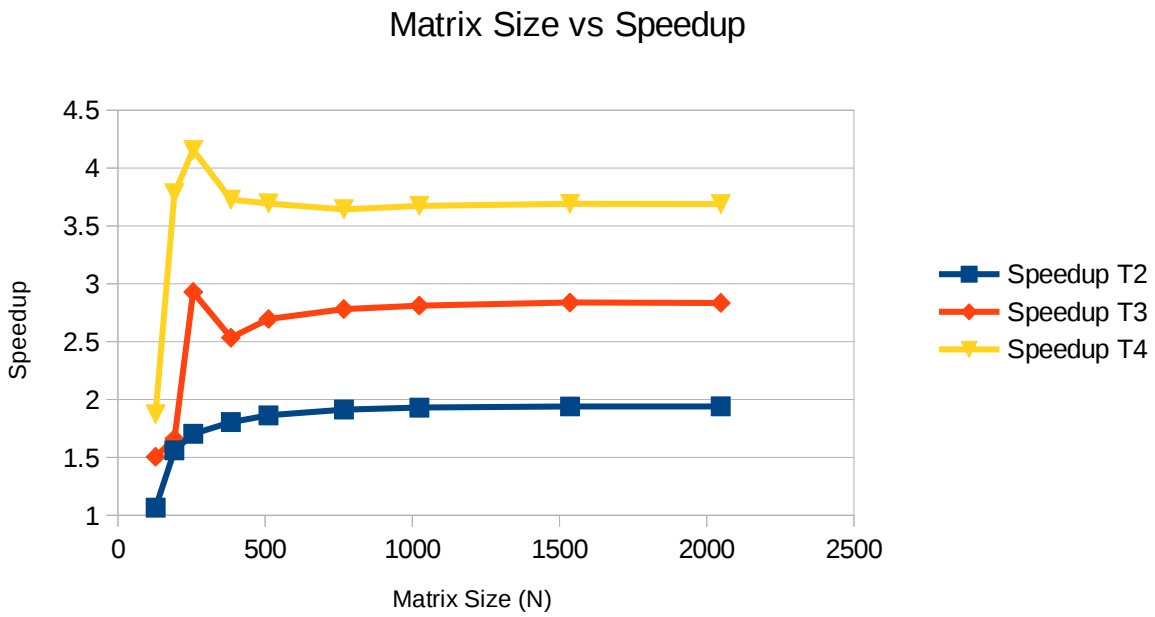
```
Threads: 1
  0.010495        0.005843        0.006341
  0.028220        0.022967        0.024868
  0.053346        0.056714        0.050666
  0.161465        0.152446        0.159345
  0.366919        0.357090        0.368407
  1.204320        1.196387        1.202859
  2.934068        2.922738        2.968047
  9.922737        9.914455       10.058668
 23.468226       23.461429       23.460662

Threads: 2
  0.007192        0.007133        0.006954
  0.019720        0.015568        0.013420
  0.035314        0.031007        0.027983
  0.087355        0.085374        0.089545
  0.190058        0.197384        0.198516
  0.625959        0.629656        0.628030
  1.517283        1.515384        1.538852
  5.111999        5.110251        5.184066
 12.090712       12.086609       12.089291

Threads: 3
   0.005107       0.004952        0.005003
  0.015987        0.016010        0.013790
  0.016739        0.016980        0.021162
  0.064651        0.061131        0.060980
  0.136234        0.134229        0.134733
  0.428568        0.434557        0.432197
  1.040933        1.043946        1.053486
  3.497636        3.498054        3.537673
  8.278215        8.275488        8.278231

Thread: 4
  0.004011        0.004247        0.003847
  0.008683        0.005844        0.005583
  0.012906        0.012855        0.012919
  0.042088        0.042247        0.042629
  0.098370        0.097934        0.099495
  0.329105        0.329907        0.330154
  0.798945        0.795852        0.808042
  2.688160        2.687280        2.725116
  6.355615        6.373270        6.355293
```

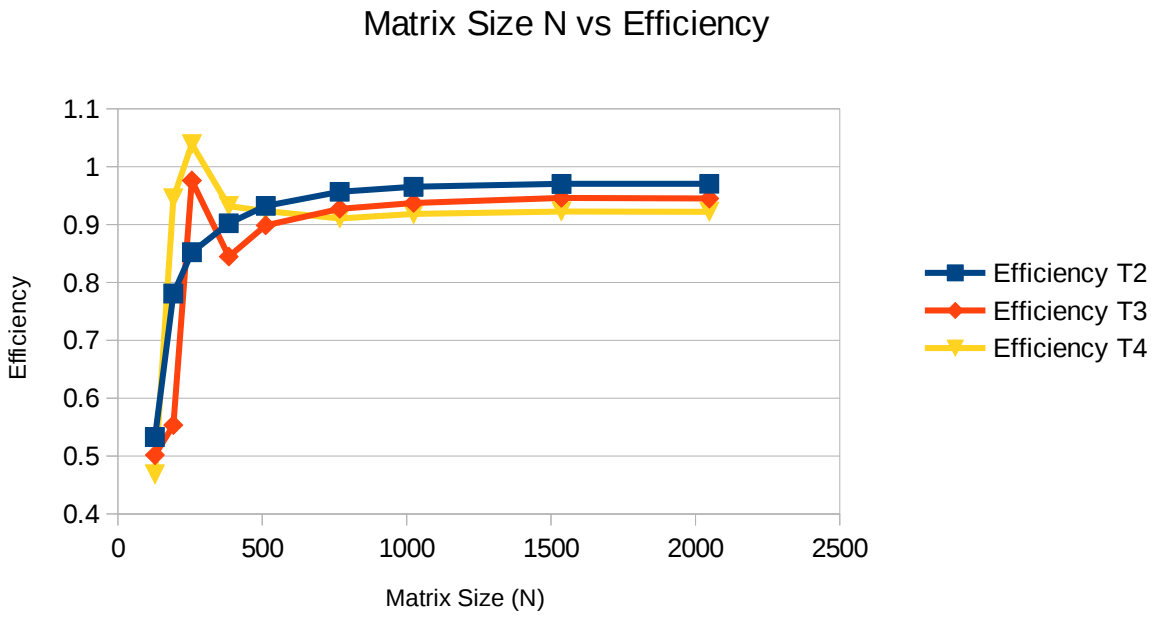
Graph of Average Times vs Matrix Size

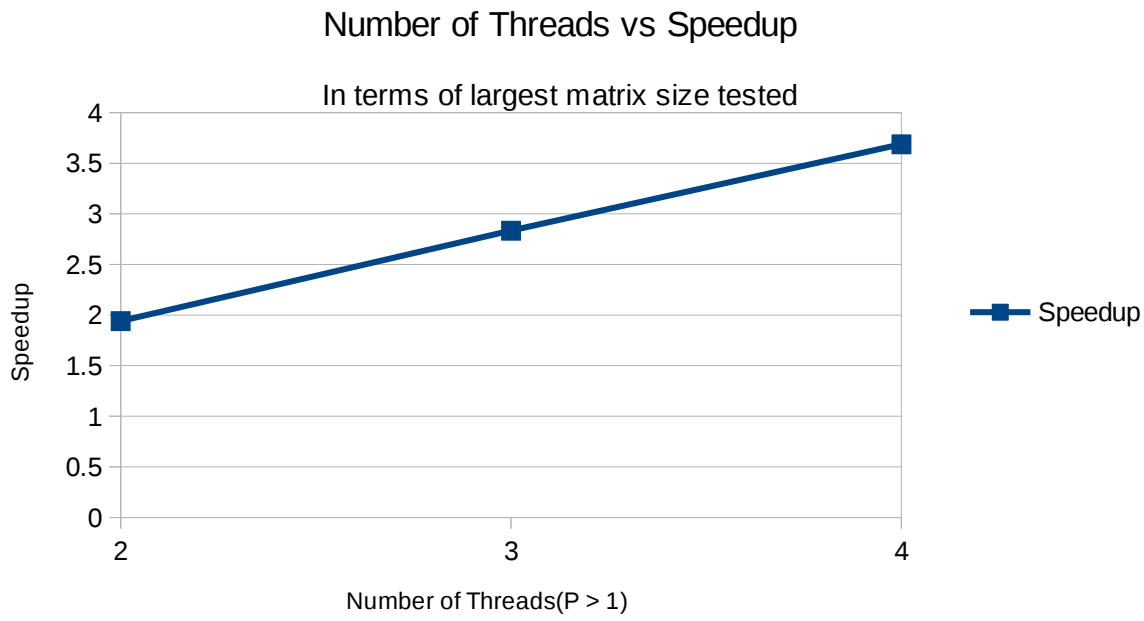


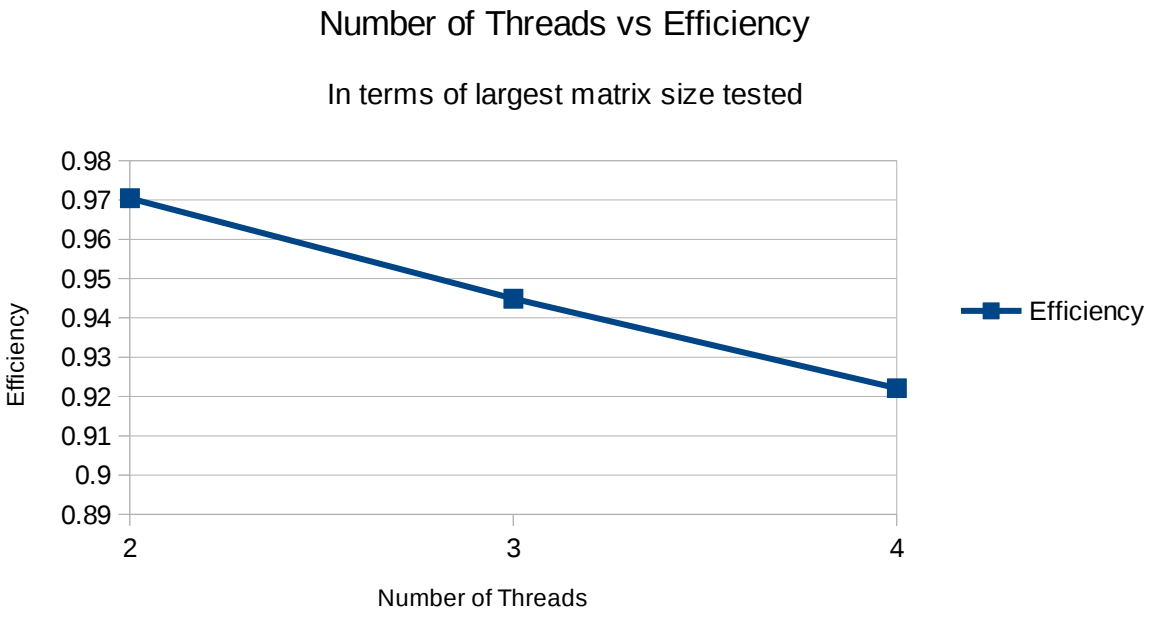
Graph of Average Times vs Matrix Size on Logarithmic Scale




Matrix Size vs Speedup
4.5
4
3.5
3
2.5
2
1.5
1
Speedup
0
500
1000
1500
2000
2500
Matrix Size (N)
Speedup T2
Speedup T3
Speedup T4


Matrix Size N vs Efficiency
1.1
1
0.9
0.8
0.7
0.6
0.5
0.4
Efficiency
0
500
1000
1500
2000
2500
Matrix Size (N)
Efficiency T2
Efficiency T3
Efficiency T4


Number of Threads vs Speedup
In terms of largest matrix size tested
Speedup
4
3.5
3
2.5
2
1.5
1
0.5
0
2
3
4
Number of Threads(P > 1)
Speedup


Number of Threads vs Efficiency
In terms of largest matrix size tested
Efficiency
0.98
0.97
0.96
0.95
0.94
0.93
0.92
0.91
0.9
0.89
2
3
4
Number of Threads
Efficiency

```
*****************************************************************************
```
Authors : Albert Owusu-Asare , Box 4497, <owusuasa@grinnell.edu>
: Ezra Edgerton , Box 3503, <edgerton@grinnell.edu>

This document contains answers to questions from CS213 Lab: Threads

Questions:
http://www.cs.grinnell.edu/~weinman/courses/CSC213/2014F/labs/threads.html

```
*****************************************************************************
```

Answers

Relevant information:

Page size: 4096

Number of processors available (nproc) = 4;
processor 0:
cpu MHz = 1600.000
cache size = 3072 KB
cpu cores = 4
address sizes : 36 bits physical, 48 bits virtual

processor 1:
cpu MHz = 1600.000
cache size = 3072 KB
cpu cores = 4
address sizes : 36 bits physical, 48 bits virtual

processor 2:
cpu MHz = 1600.000
cache size = 3072 KB
cpu cores = 4
address sizes : 36 bits physical, 48 bits virtual

processor 3:
cpu MHz = 1600.000
cache size = 3072 KB
cpu cores = 4
address sizes : 36 bits physical, 48 bits virtual

Part B
---

3)

a.

The average times taken for a set of threaded processes to run a matrix of
size N is the average time for 1 thread divided by the number of threads in the pro
cess.
As a result the plots of the different threaded processes showed less
growth in average time(seconds) vs matrix size as the number of threads rose.

As we increase the number of threads, we are able to obtain more
concurrency because as we saw from nproc, we have 4 available processors on
the computer. The more threads we use, the less number of rows the thread
is responsible for calculating in the matrix and the greater number of rows
can be computed at a time; thus we have less overall time calculating
the matrix. In the case of 4 threads being used, each thread uses a
processor to calculate the rows that it is responsible for so the average
run time is 1/4 of the one threaded average run time. This holds for the 2
and 3 threaded runs being 1/2 and 1/3 respectively of the run time of one thread as
well.

b. The trend in speedup as N increases is that the differently threaded
speedups hold more or less constant. Two Threads holds at just below 2,
Three threads holds at a little more below 3 than 2 threads held below 2,
and four threads holds at a little more below 4 than 3 threads held below
3.The difference from the numbers that the speedups were just below is
caused by the overhead from creating and running the threads, the overhead
will increase when we create more threads, bringing the speedup time below
the round number that just running the program with overheadless threads
would be. 4 threads is the farthest below 4 because the 4 threads would
have the most overhead.

The best answer we could come up with is that the deviations at the
beginning must have something to do with the page or cache size. That at an
N of between 256 and 384, each row would pass the limit of the
page size this would cause a larger number of tlb misses and slow down the
process significantly. Before this, the tlb misses of each thread would
decrease as the number of threads increased, so the speedup time for runs
with greater threads would increase more quickly.

c.
Efficiency seems to remove the advantage that the speedup metric measures
by dividing by the number of threads, it seems to measure the differences
in overhead, because with one thread, the efficiency is one. As N
increases, we see that 2 threads is the most efficient, holding at a little
below 1, while 3 and 4 threads are less than the smaller number of
threads. This is because, as we said in b, the overheads for more threads
increases as we build more threads.

At smaller values of N, the deviation from the trend is the same as our
answer for part b. However, we see that 4 threads goes above 1 in
efficiency at N = 256. This must be due to a reason similar to what was
mentioned in b.

d.
As P increases the speedup increases linearly at a rate of slightly less
than 1. As we said previously, the overhead causes the slight decrease in
growth below 1. Without the overhead of thread creation and joining, the growth
would be at a constant increase of one because the number of threads decreases
the run time of the matrix multiply at that rate, as we explained in 3a
above. More threads mean more concurrent processing of rows, and as a
result a faster overall calculation of the matrix.

e. As P increases the efficiency decreases linearly at a rate of roughly -.025. Thi
s measures the
amount of time that is spend in overhead for each increase in threads. The
more threads we have, the more time we spend creating and joining them, and
that cancels out a small amount of the speedup effect.