

CS 442 Introduction to Parallel Processing Project 2: `pthread`s Sieve of Eratosthenes

Erik Barry Erhardt and Daniel Roland Llamocca Obregon

May 2, 2006

Contents

1	Executive Summary	3
2	Problem Description	4
3	Data Generation	5
4	Results	6
5	Conclusions	11
A	Appendix	12

List of Figures

1	Time to complete for small problem sizes n and threads requested t	7
2	Time to complete for medium problem sizes n and threads requested t	8
3	Time to complete for large problem sizes n and threads requested t , and useful threads.	9
4	Scalability for selected problem sizes $n = 1000, 100000, 10000000, 500000000$	10

List of Tables

1	Design space explored.	5
2	Time to complete for small problem sizes n and threads requested t	17
3	Time to complete for medium problem sizes n and threads requested t	18
4	Time to complete for large problem sizes n and threads requested t	19
5	Time to complete for huge problem sizes n and threads requested t	20

1 Executive Summary

For this project, an algorithm to find prime numbers based on the Sieve of Eratosthenes was parallelized with pthreads. The parallel threads-based algorithm was implemented in C and pthreads. The program was run on up to four processors of one node of Linux cluster AZUL. Using this code, many different parallel runs were conducted, and program speedup, efficiency, and isoefficiency were explored.

The rest of this paper describes the sieving problem and our implementation in greater detail, gives an overview of the parallel machine we ran on, discusses our experiment, presents the data we collected in various speedup and efficiency plots, and makes some observations about the data. Finally, conclusions are made and our code is presented in the appendix.

2 Problem Description

The Sieve of Eratosthenes is a simple, ancient algorithm for finding all prime numbers up to a specified integer. The basic algorithm starts with a list of all integers from two to the maximum integer specified, n . The algorithm places the lowest number remaining on the list of numbers on the list of primes, deletes all the numbers that are multiples of the latest prime discovered, and repeats.

Our parallel implementation has threads zero multiples in the array of numbers from 1 to n . The master program initializes an array `nums` with integers from 1 to n , then sets element 1 to 0 (reason to become clear in a moment). Then the master spawns threads. Each thread reads through the `nums` array until finding a positive number. This number will be a prime number. The thread negates the number (eg. changes 3 to -3), then sets all multiples to 0 by skipping through the array and writing 0s — no need to read the array. After the thread has zeroed all multiples, it reads through the `nums` array for the next positive number. If the thread reads beyond \sqrt{n} , then it terminates, setting a switch to indicate it is complete for the master program to use. The master knows when all threads have completed when all the done switches are set. The master can then read all the nonzero numbers in the `nums` array, their absolute values are the prime numbers.

Note that thinking of the problem in this way avoids the need for any locks. There is never contention for reads or writes. If a number is a multiple of more than one prime, it is simply set to 0 more than once. If a prime (eg. 2) is chosen by one thread, and a multiple (eg. 4) is chosen as a prime by another thread before it is zeroed, then some duplicate work is done by the second thread, but all multiples are eventually zeroed.

3 Data Generation

Our data were generated on Linux cluster AZUL, each node having four 550 MHz Intel Xeon CPUs with 512 kB L2 Cache and 2 GiB of RAM. We used one node as a shared memory machine for pthreads. Parallel jobs were submitted through a scheduler and executed in batch.

Table 1 on page 5 summarizes the design space we explored. We searched integers for primes up to the values of n , where n took the form of 1, 2, or 5 times 10 to the powers 3 to 8, that is from 1,000 to 500,000,000. We invoked the number of threads t from 1 to 12, then 14 to 20 by 2, then 25 to 60 by 5. We used all combinations of $n \times t$.

<u>problem size n</u>	<u>threads t</u>
1000	1
2000	2
5000	3
10000	4
20000	5
50000	6
100000	7
200000	8
500000	9
1000000	10
2000000	11
5000000	12
10000000	14
20000000	16
50000000	18
100000000	20
200000000	25
500000000	30
	35
	40
	45
	50
	55
	60

Table 1: Design space explored.

4 Results

The plots in Figure 1, 2, and 1 on pages 7 through 9 gives the length of time for different problem sizes n and number of threads requested t , where n is the maximum integer checked, (refer to Table 1 on page 5). Tables 2 through 5 in the appendix also give the times. These two plots show two features: (1) There is a performancy penalty for thread creation, and (2) using multiple threads benefits only the largest of problem sizes. As threads requested increases the running times strictly increase for $n = 1000, 2000, 5000, 10000, 20000$. It is at $n = 50000$ when there is a slight time benefit from using a second thread but time increases beyond 2 threads, and at $n = 200000$ when a third thread helps by time increases beyond 3 threads. At $n = 500000$ and beyond, additional threads do not pose great time penalties, but do not improve performance beyond a few threads. At the largest problem size $n = 500000000$, up to 17 threads improve the time, but there is no improvement beyond that.

The second two plots in Figure 3 on page 9 give the number of useful threads and the time when the number of requested threads were all used. Our program tracks whether a thread that was created was used in the process of finding primes. In many cases, the first few threads created complete all the work and the addition threads are created and do no work. For many small problem sizes, one or two processes do all the work. It is only for the few largest problem sizes when all 60 threads were used. Plotting only those times when all threads were used, these are the best cases, the time remains short for all but the largest of problem sizes. In the case when for a single n multiple sets of requested threads used all the threads, the largest number of threads was plotted.

The plots in Figure 4 on page 10 give the scalability for selected problem sizes. Ideal scalability follows a line of slope 1. For the smallest two problem size $n = 1000$, there is a severe penalty for using more than a single thread. For the second problem size $n = 100000$, there is value is creating a second thread (increase in scalability) but major penalties for additional threads. For the third problem size $n = 100000000$, there is benefit for a few threads, but neither benefit nor penalty for additional threads. For the largest problem size $n = 500000000$, there is great benefit for a few threads, then minor benefit up to about 17 threads, then neither benefit nor penalty for additional threads.

Had the problem been more computationally intensive (more work), we would expect to see better scalability. In this case, the first few threads end up completing so much of the total work that the additional threads add little to the performance.

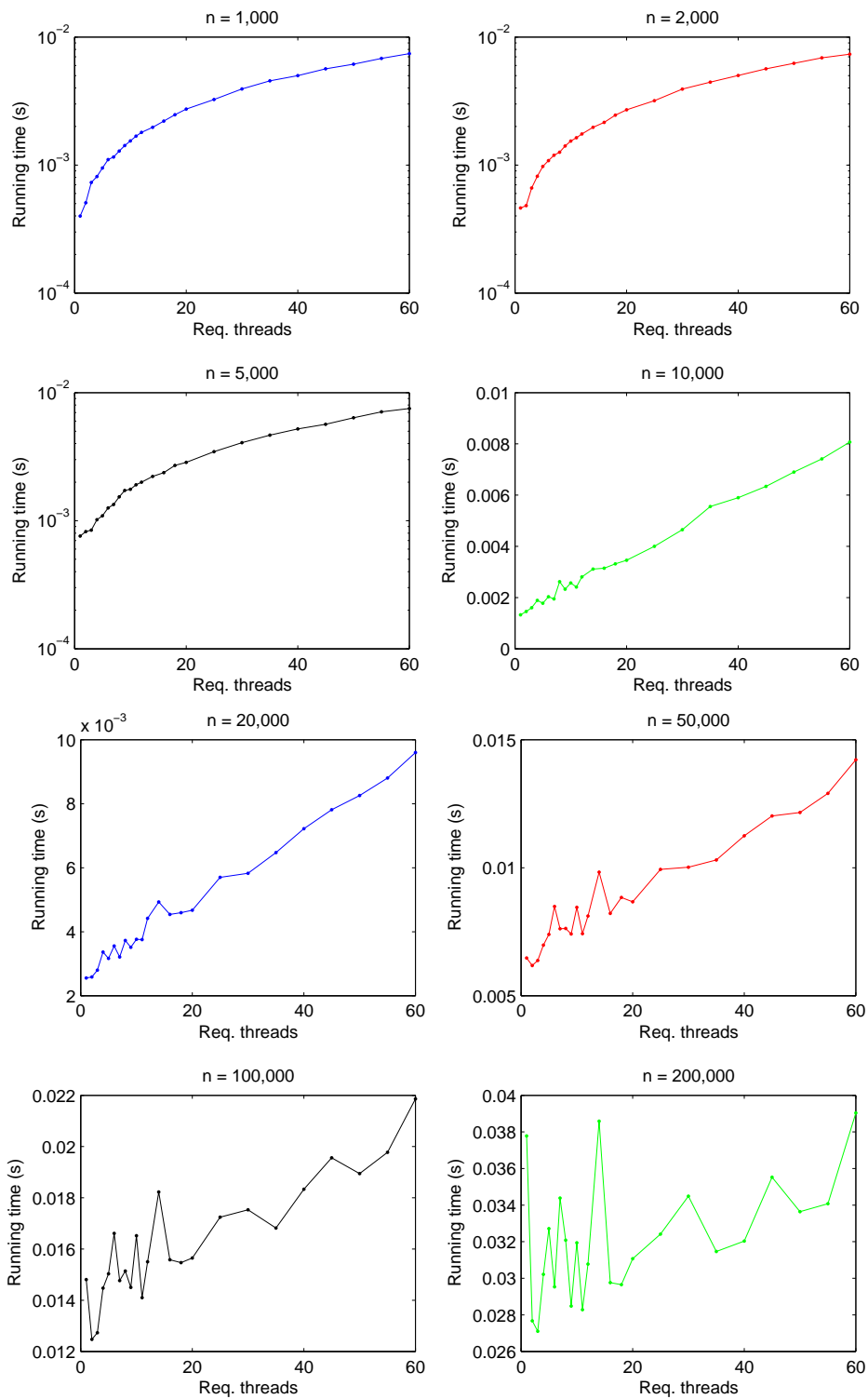


Figure 1: Time to complete for small problem sizes n and threads requested t .

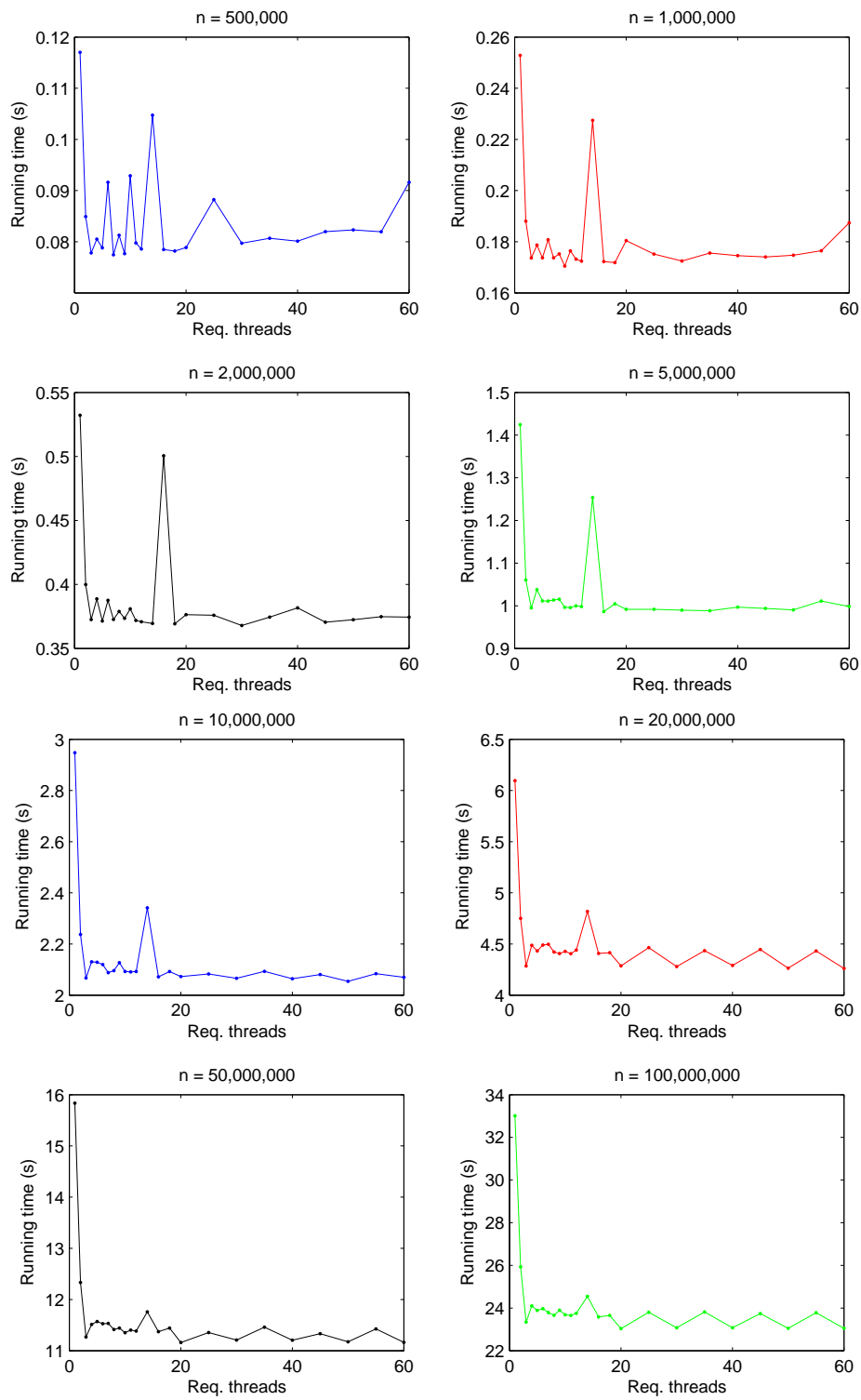


Figure 2: Time to complete for medium problem sizes n and threads requested t .

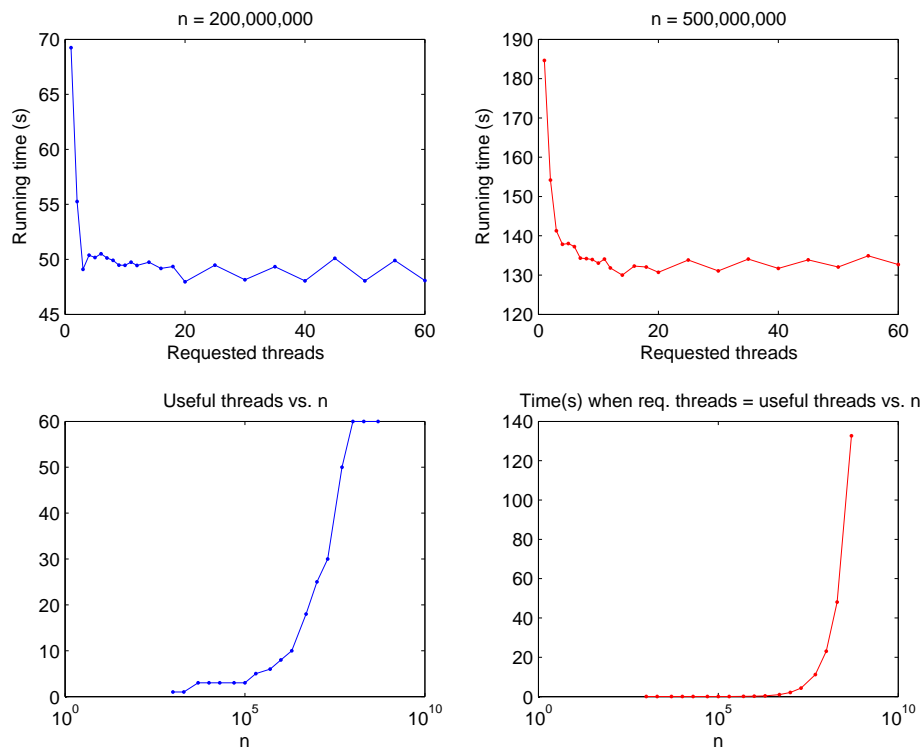


Figure 3: Time to complete for large problem sizes n and threads requested t , and useful threads.

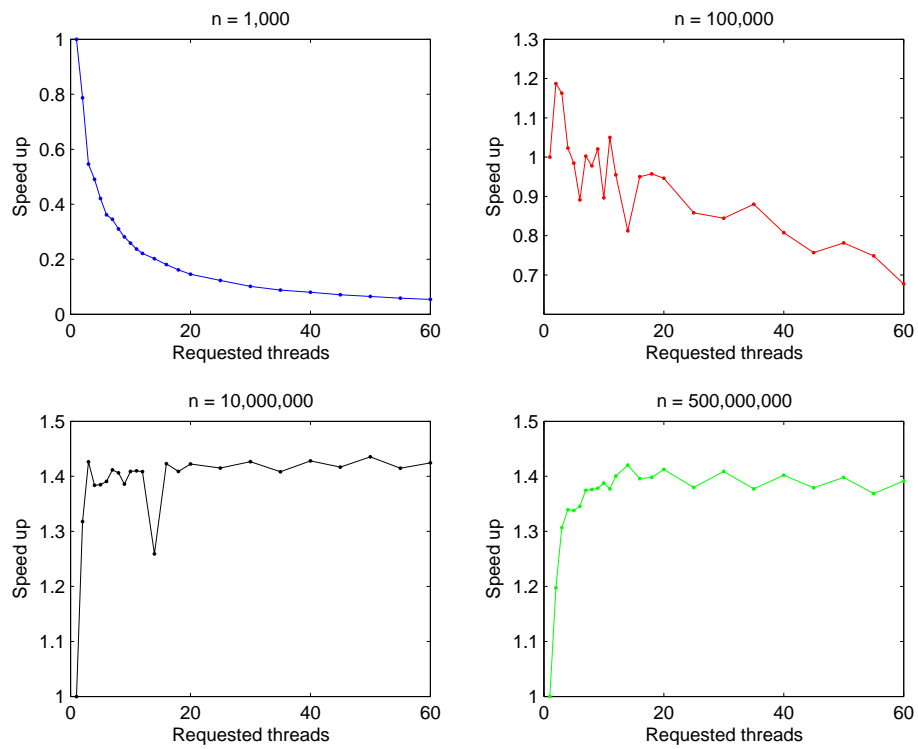


Figure 4: Scalability for selected problem sizes $n = 1000, 100000, 10000000, 500000000$.

5 Conclusions

In this project, we successfully used pthreads to implement a parallel version of the Sieve of Eratosthenes. In doing so, we came to understand better parallelism with threads and how it can be used on shared memory systems. We were disappointed to fail to see much scalability in this problem, however pleased since it suggests our approach to solving the problem was very efficient.

A Appendix

Listing 1: pthreads implementation: primes_thread.c.

```

/*
   Sieve of Eratosthenes — pthreads implementation

   Authors:
   5     Erik Barry Erhardt
        Daniel Roland Llamocca Obregon

   4/26/2006
*/
10 /*
   /*
   Using cluster azul

   compile/run:
15     gcc -pthread primes_thread.c -o primes_thread
        ./primes_thread -n 5000 -t 100 -v -v

   request an hour on a node:
20     qsub -I -l nodes=1,walltime=3600

   submit a batchjob:
        qsub < batch1.bat

   where batch1.bat looks like:
25     #!/bin/tcsh
        #PBS -N e_d_th1
        #PBS -l nodes=1
        #PBS -l walltime=3600
        cd /home/erike/proj2_pthreads_primes
30     pwd
        rm -f results1.txt
        touch results1.txt
        ./primes_thread -n 1000 -t 1 >> results1.txt
        ./primes_thread -n 2000 -t 1 >> results1.txt
35     ./primes_thread -n 5000 -t 1 >> results1.txt
        ./primes_thread -n 10000 -t 1 >> results1.txt
        ./primes_thread -n 20000 -t 1 >> results1.txt
        ./primes_thread -n 50000 -t 1 >> results1.txt
        ./primes_thread -n 100000 -t 1 >> results1.txt
40     ./primes_thread -n 200000 -t 1 >> results1.txt
        ./primes_thread -n 500000 -t 1 >> results1.txt
        ./primes_thread -n 1000000 -t 1 >> results1.txt
        ./primes_thread -n 2000000 -t 1 >> results1.txt
45     ./primes_thread -n 5000000 -t 1 >> results1.txt
        ./primes_thread -n 10000000 -t 1 >> results1.txt
        ./primes_thread -n 20000000 -t 1 >> results1.txt
        ./primes_thread -n 50000000 -t 1 >> results1.txt
        ./primes_thread -n 100000000 -t 1 >> results1.txt
50     ./primes_thread -n 200000000 -t 1 >> results1.txt
        ./primes_thread -n 500000000 -t 1 >> results1.txt
*/

#include <stdio.h>
#include <stdlib.h>
55 #include <unistd.h>
#include <pthread.h>

/* function declarations */
int timeval_subtract(struct timeval *result, struct timeval *x, struct timeval *y);
60 void *zero_multiples(void *threadid);

/* Global variables for shared memory */
int n = 0, t = 0; /* n = nums[] array size, t = number of threads, from command line */
long int *nums = NULL; /* nums[] is the array of numbers */
65 int *done_sw = NULL; /* indicates that all threads are complete */
int *did_work_sw = NULL; /* indicates which threads actually did work */

int main (int argc, char *argv[])
{
70     pthread_t *threads = NULL; /* threads structure */
    int rc; /* return code for pthread_create() */

    int c = 0; /* command line arguments */
    int i = 0, k = 0; /* for loops */
75     struct timeval tv_1; /* time before */
    struct timeval tv_2; /* time after */
    struct timeval result; /* time result */
    int status; /* time of day */

80     int debug_level = 0; /* used for verbose messaging to screen */
    int done_sw_main = 0; /* used to check if all the threads have finished */
    int did_work_howmany = 0; /* tallies how many threads actually did work */

```

```

85     int n_primes = 0;          /* tallies the number of primes found */

    /** Parse command line arguments *****/
    while( c = getopt( argc, argv, "n:t:v" ) != EOF )
    {
90         switch( c )
        {
            case 'n': n = strtol( optarg, (char **)NULL, 10 );
            break;

            case 't': t = strtol( optarg, (char **)NULL, 10 );
95             break;

            case 'v': debug_level++;
            break;
        }
100    }

    if(debug_level) printf("n=%d, t=%d\n", n, t); /* print n and t */

    status=gettimeofday(&tv_1, 0);          /* start timer */
105
    nums = (long int *)malloc(n * sizeof(long int)); /* allocate nums[] array */
    if(nums == NULL){fprintf(stderr, "nums Out of memory!\n");exit( EXIT_FAILURE );}

    /* population nums[] array from 1 to n */
110    for(i=1; i<n; i++) {nums[i]=i+1;} /* start at index 1 so that number 1 starts zeroed out */

    threads = (pthread_t *)malloc(t * sizeof(pthread_t)); /* allocate threads[] structure array */
    if(threads == NULL){fprintf(stderr, "threads Out of memory!\n");exit( EXIT_FAILURE );}

115    done_sw = (int *)malloc(t * sizeof(int)); /* allocate done_sw[] array */
    if(done_sw == NULL){fprintf(stderr, "done_sw Out of memory!\n");exit( EXIT_FAILURE );}

    did_work_sw = (int *)malloc(t * sizeof(int)); /* allocate did_work_sw[] array */
    if(did_work_sw == NULL){fprintf(stderr, "did_work_sw Out of memory!\n");exit( EXIT_FAILURE );}
120

    /* create threads and run zero_multiples */
    for(i=0; i<t; i++){
        if(debug_level>1) printf("Creating thread %d\n", i);
        rc = pthread_create(&threads[i], NULL, zero_multiples, (void *)i); /* create thread to run zero_multiples() */
125        if (rc){printf("ERROR; return code from pthread_create() is %d\n", rc);exit(-1);}
    }

    /* main program wait until all threads are complete */
    while(done_sw_main < t) /* exit only when all threads have set their done_sw */
130    {
        done_sw_main = 0;
        for(i=0; i<t; i++)
        {
            done_sw_main = done_sw_main + done_sw[i]; /* count how many threads are done */
135        }
    }

    /* count number of threads that did work */
    did_work_howmany = 0;
140    for(i=0; i<t; i++){did_work_howmany = did_work_howmany + did_work_sw[i];}

    /* count the number of primes found */
    if(debug_level)
    {
145        n_primes = 0;
        for(k=0; k < n; k++)
        {
            if(nums[k] != 0){n_primes++;} /* primes are all the non 0 numbers remaining in nums[] */
        }
150        printf("n_primes=%d\n", n_primes);
    }

    status=gettimeofday(&tv_2,0);          /* stop timer */
    timeval_subtract(&result,&tv_2,&tv_1); /* calculate elapsed time */
155

    /* report results */
    printf("%d %d %d %d %d.%06d\n", n, t, did_work_howmany, n_primes, result.tv_sec, result.tv_usec);

    pthread_exit(NULL); /* all done */
160 }

    /* Function that each thread executes to zero out multiples of primes they find */
    void *zero_multiples(void *threadid)
165    {
        int prime = 0; /* current prime */
        int i_prime= 0; /* current prime index in nums[] */
        int i, k; /* for looping */

        /* Each thread reads nums[] up to sqrt(n)

```

```
170     If a positive number is encountered, it is prime:
        the number is negated, and all multiples are zeroed
        then continues looping looking for positive (prime) numbers */
for(i = 0; i*i <= n; i++) /* read nums[] until locate a positive number or until sqrt(n) */
{
175     prime = nums[i];
        i_prime = i;

        if(prime>0) /* if locate a positive number, it must be prime */
        {
180             did_work_sw[(int) threadid]=1; /* indicates that this thread did some work */
                nums[i_prime] = -nums[i_prime]; /* set current prime to negative */
                for (k = i_prime + prime; k < n; k = k + prime) /* mark multiples to 0 */
                {
185                     nums[k] = 0;
                }
        }
}

done_sw[(int) threadid]=1; /* indicate that this thread is complete — no more primes left */
190 pthread_exit(NULL);
}

/* used for time elapsed */
195 /* Subtract the 'struct timeval' values X and Y,
        storing the result in RESULT.
        Return 1 if the difference is negative, otherwise 0. */
int timeval_subtract (struct timeval *result, struct timeval *x, struct timeval *y)
{
200     /* Perform the carry for the later subtraction by updating y. */
        if (x->tv_usec < y->tv_usec) {
            int nsec = (y->tv_usec - x->tv_usec) / 1000000 + 1;
            y->tv_usec -= 1000000 * nsec;
            y->tv_sec += nsec;
205     }
        if (x->tv_usec - y->tv_usec > 1000000) {
            int nsec = (x->tv_usec - y->tv_usec) / 1000000;
            y->tv_usec += 1000000 * nsec;
            y->tv_sec -= nsec;
210     }

        /* Compute the time remaining to wait.
            tv_usec is certainly positive. */
        result->tv_sec = x->tv_sec - y->tv_sec;
215     result->tv_usec = x->tv_usec - y->tv_usec;

        /* Return 1 if result is negative. */
        return x->tv_sec < y->tv_sec;
}
220 /* EOF */
```

Listing 2: Matlab script `run_script.m` used to generate simulation batches.

```

t=[1:12 14:2:20 25:5:60];      % t = number of possible threads
n=[];                          % n = size of nums[] array
for tens=3:8;
    for num=[1 2 5];
5         n=[n num*10^tens];
        end
    end

    %% create 1 file with all jobs in it
10    % fn='results.txt';      % filename for results to go to
    % fid = fopen('run.bat','w'); % run.bat is script file
    % fprintf(fid,'rm -f results.txt\n');% delete the output file
    % fprintf(fid,'touch results.txt\n');% create an empty output file to append to
    % for i_t = t
15    %     for i_n = n
    %         fprintf(fid,'./primes_pthread -n %d -t %d %s %s\n', i_n, i_t, '>>>', fn);
    %     end
    % end
    % fclose(fid);
20

    %% create a file for each i_t
    fid2 = fopen('run.bat','w'); % batch#.bat is script file
    for i_t = t
        fnout=strcat('results',num2str(i_t),'.txt'); % filename for results to go to
25        fnbat=strcat('batch',num2str(i_t),'.bat'); % filename for results to go to
        fid = fopen(fnbat,'w'); % batch#.bat is script file
        fprintf(fid,'#!/bin/tcsh\n');
        fprintf(fid,'#PBS -N e_d_th%d\n', i_t);
        fprintf(fid,'#PBS -l nodes=1\n');
30        fprintf(fid,'#PBS -l walltime=3600\n');
        fprintf(fid,'cd /home/erike/proj2_pthreads_primes \n');
        fprintf(fid,'pwd\n');
        fprintf(fid,'rm -f %s\n', fnout);% delete the output file
        fprintf(fid,'touch %s\n', fnout);% create an empty output file to append to
35        for i_n = n
            fprintf(fid,'./primes_pthread -n %d -t %d %s %s\n', i_n, i_t, '>>>', fnout);
        end
        fclose(fid);
        fprintf(fid2,'qsub < %s\n',fnbat);
40    end
    fclose(fid2);

```

Listing 3: Matlab script `plots.m` used to create plots.

```

clear all; close all; clc

req_t =[1:12 14:2:20 25:5:60];
n=[]; % n = size of nums[] array
5
for tens=3:8;
    for num=[1 2 5];
        n=[n num*10^tens];
    end
10 end

load time;
% time{1} n = 1000
% time{2} n = 2000
15 % time{3} n = 5000
% time{4} n = 10,000
% time{5} n = 20,000
% time{6} n = 50,000
% time{7} n = 100,000
20 % time{8} n = 200,000
% time{9} n = 500,000
% time{10} n = 1,000,000
% time{11} n = 2,000,000
% time{12} n = 5,000,000
25 % time{13} n = 10,000,000
% time{14} n = 20,000,000
% time{15} n = 50,000,000
% time{16} n = 100,000,000
% time{17} n = 200,000,000
30 % time{18} n = 500,000,000

% Graphs:
for i = 1:18
35 end
figure;
subplot (2, 2, 1); semilogy (req_t,timi(1,:), 'b.-'); xlabel ('Req. threads'); ylabel ('Running time (s)'); title ('n = 1,000');
subplot (2, 2, 2); semilogy (req_t,timi(2,:), 'r.-'); xlabel ('Req. threads'); ylabel ('Running time (s)'); title ('n = 2,000');
subplot (2, 2, 3); semilogy (req_t,timi(3,:), 'k.-'); xlabel ('Req. threads'); ylabel ('Running time (s)'); title ('n = 5,000');

```

```

40 subplot (2, 2, 4); plot (req_t,timi(4,:), 'g.-'); xlabel ('Req. threads'); ylabel ('Running time (s)'); title ('n = 10,000')
    fig1 = strcat('fig1.eps'); % plot name
    print(gcf, '-depsc2', fig1); % print plot

    figure;
45 subplot (2, 2, 1); plot (req_t,timi(5,:), 'b.-'); xlabel ('Req. threads'); ylabel ('Running time (s)'); title ('n = 20,000')
    subplot (2, 2, 2); plot (req_t,timi(6,:), 'r.-'); xlabel ('Req. threads'); ylabel ('Running time (s)'); title ('n = 50,000')
    subplot (2, 2, 3); plot (req_t,timi(7,:), 'k.-'); xlabel ('Req. threads'); ylabel ('Running time (s)'); title ('n = 100,000')
    subplot (2, 2, 4); plot (req_t,timi(8,:), 'g.-'); xlabel ('Req. threads'); ylabel ('Running time (s)'); title ('n = 200,000')
    fig2 = strcat('fig2.eps'); % plot name
50 print(gcf, '-depsc2', fig2); % print plot

    figure;
    subplot (2, 2, 1); plot (req_t,timi(9,:), 'b.-'); xlabel ('Req. threads'); ylabel ('Running time (s)'); title ('n = 500,000')
    subplot (2, 2, 2); plot (req_t,timi(10,:), 'r.-'); xlabel ('Req. threads'); ylabel ('Running time (s)'); title ('n = 1,000,000')
55 subplot (2, 2, 3); plot (req_t,timi(11,:), 'k.-'); xlabel ('Req. threads'); ylabel ('Running time (s)'); title ('n = 2,000,000')
    subplot (2, 2, 4); plot (req_t,timi(12,:), 'g.-'); xlabel ('Req. threads'); ylabel ('Running time (s)'); title ('n = 5,000,000')
    fig3 = strcat('fig3.eps'); % plot name
    print(gcf, '-depsc2', fig3); % print plot

60 figure;
    subplot (2, 2, 1); plot (req_t,timi(13,:), 'b.-'); xlabel ('Req. threads'); ylabel ('Running time (s)'); title ('n = 10,000,000')
    subplot (2, 2, 2); plot (req_t,timi(14,:), 'r.-'); xlabel ('Req. threads'); ylabel ('Running time (s)'); title ('n = 20,000,000')
    subplot (2, 2, 3); plot (req_t,timi(15,:), 'k.-'); xlabel ('Req. threads'); ylabel ('Running time (s)'); title ('n = 50,000,000')
    subplot (2, 2, 4); plot (req_t,timi(16,:), 'g.-'); xlabel ('Req. threads'); ylabel ('Running time (s)'); title ('n = 100,000,000')
65 fig4 = strcat('fig4.eps'); % plot name
    print(gcf, '-depsc2', fig4); % print plot

    figure;
70 subplot (2, 2, 1); plot (req_t,timi(17,:), 'b.-'); xlabel ('Requested threads'); ylabel ('Running time (s)'); title ('n = 20,000,000')
    subplot (2, 2, 2); plot (req_t,timi(18,:), 'r.-'); xlabel ('Requested threads'); ylabel ('Running time (s)'); title ('n = 50,000,000')

    % Only when useful threads = required threads: (the maximum)
    % Note that it does not necessarily mean the best time!!
    time_op(1) = timi(1,1); % n = 1,000, t = 1
75 time_op(2) = timi(2,1); % n = 2,000, t = 1
    time_op(3) = timi(3,3); % n = 5,000, t = 3
    time_op(4) = timi(4,3); % n = 10,000, t = 3
    time_op(5) = timi(5,3); % n = 20,000, t = 3
    time_op(6) = timi(6,3); % n = 50,000, t = 3
80 time_op(7) = timi(7,3); % n = 100,000, t = 3
    time_op(8) = timi(8,5); % n = 200,000, t = 5
    time_op(9) = timi(9,6); % n = 500,000, t = 6
    time_op(10) = timi(10,8); % n = 1,000,000, t = 8
    time_op(11) = timi(11,10); % n = 2,000,000, t = 10
85 time_op(12) = timi(12,15); % n = 5,000,000, t = 18
    time_op(13) = timi(13,17); % n = 10,000,000, t = 25
    time_op(14) = timi(14,18); % n = 20,000,000, t = 30
    time_op(15) = timi(15,22); % n = 50,000,000, t = 50
    time_op(16) = timi(16,24); % n = 100,000,000, t = 60
90 time_op(17) = timi(17,24); % n = 200,000,000, t = 60
    time_op(18) = timi(18,24); % n = 500,000,000, t = 60
    useful_t = [1 1 3 3 3 3 3 5 6 8 10 18 25 30 50 60 60 60];

    subplot (2,2,3); semilogx (n, useful_t, 'b.-'); xlabel ('n'); title ('Useful threads vs. n');
95 subplot (2,2,4); semilogx (n, time_op, 'r.-'); xlabel ('n'); title ('Time(s) when req. threads = useful threads vs. n');
    fig5 = strcat('fig5.eps'); % plot name
    print(gcf, '-depsc2', fig5); % print plot

    % Speed-up
100 % n = 1,000 time{1}
    for i = 1:24
        S_a(i) = time{1}(1)/time{1}(i);
    end

105 % n = 100,000 time{7}
    for i = 1:24
        S_b(i) = time{7}(1)/time{7}(i);
    end

110 % n = 10,000,000 time{13}
    for i = 1:24
        S_c(i) = time{13}(1)/time{13}(i);
    end

115 % n = 500,000,000 time{18}
    for i = 1:24
        S_d(i) = time{18}(1)/time{18}(i);
    end

120 figure;
    subplot (2,2,1); plot (req_t, S_a, 'b.-'); xlabel ('Requested threads'); ylabel ('Speed up'); title ('n = 1,000');
    subplot (2,2,2); plot (req_t, S_b, 'r.-'); xlabel ('Requested threads'); ylabel ('Speed up'); title ('n = 100,000');
    subplot (2,2,3); plot (req_t, S_c, 'k.-'); xlabel ('Requested threads'); ylabel ('Speed up'); title ('n = 10,000,000');
    subplot (2,2,4); plot (req_t, S_d, 'g.-'); xlabel ('Requested threads'); ylabel ('Speed up'); title ('n = 500,000,000');
125 fig6 = strcat('fig6.eps'); % plot name
    print(gcf, '-depsc2', fig6); % print plot

```

n (primes)	Requested threads t (useful threads)					
	Time to complete					
1000 (168)	1 (1)	2 (1)	3 (1)	4 (1)	5 (1)	6 (1)
	0.000399	0.000507	0.000731	0.000813	0.000948	0.001103
	7 (1)	8 (1)	9 (1)	10 (1)	11 (1)	12 (1)
	0.001156	0.001287	0.001418	0.001542	0.001682	0.001800
	14 (1)	16 (1)	18 (1)	20 (1)	25 (1)	30 (1)
	0.001972	0.002205	0.002473	0.002737	0.003246	0.003931
	35 (1)	40 (1)	45 (1)	50 (1)	55 (1)	60 (1)
0.004550	0.004989	0.005639	0.006144	0.006811	0.007415	
2000 (303)	1 (1)	2 (1)	3 (1)	4 (1)	5 (1)	6 (1)
	0.000462	0.000481	0.000662	0.000817	0.000976	0.001083
	7 (1)	8 (1)	9 (1)	10 (1)	11 (1)	12 (1)
	0.001191	0.001258	0.001409	0.001538	0.001632	0.001749
	14 (1)	16 (1)	18 (1)	20 (1)	25 (1)	30 (1)
	0.001970	0.002151	0.002453	0.002704	0.003187	0.003925
	35 (1)	40 (1)	45 (1)	50 (1)	55 (1)	60 (1)
0.004442	0.005008	0.005646	0.006231	0.006890	0.007346	
5000 (669)	1 (1)	2 (2)	3 (3)	4 (3)	5 (3)	6 (3)
	0.000758	0.000822	0.000844	0.001018	0.001092	0.001259
	7 (3)	8 (3)	9 (3)	10 (3)	11 (3)	12 (3)
	0.001340	0.001536	0.001724	0.001761	0.001908	0.001998
	14 (3)	16 (3)	18 (3)	20 (3)	25 (3)	30 (3)
	0.002220	0.002372	0.002704	0.002855	0.003458	0.004072
	35 (3)	40 (3)	45 (3)	50 (3)	55 (3)	60 (3)
0.004654	0.005209	0.005667	0.006368	0.007082	0.007521	
10000 (1229)	1 (1)	2 (2)	3 (3)	4 (3)	5 (4)	6 (4)
	0.001326	0.001454	0.001603	0.001891	0.001776	0.002032
	7 (4)	8 (4)	9 (4)	10 (4)	11 (4)	12 (4)
	0.001947	0.002618	0.002328	0.002572	0.002411	0.002808
	14 (4)	16 (4)	18 (4)	20 (4)	25 (4)	30 (4)
	0.003111	0.003147	0.003313	0.003459	0.003998	0.004649
	35 (4)	40 (4)	45 (4)	50 (4)	55 (4)	60 (4)
0.005552	0.005899	0.006341	0.006900	0.007413	0.008072	
20000 (2262)	1 (1)	2 (2)	3 (3)	4 (3)	5 (4)	6 (4)
	0.002553	0.002584	0.002798	0.003362	0.003164	0.003554
	7 (4)	8 (4)	9 (4)	10 (4)	11 (4)	12 (4)
	0.003210	0.003724	0.003514	0.003762	0.003753	0.004417
	14 (4)	16 (4)	18 (4)	20 (4)	25 (4)	30 (4)
	0.004929	0.004539	0.004596	0.004672	0.005698	0.005828
	35 (4)	40 (4)	45 (4)	50 (4)	55 (4)	60 (4)
0.006473	0.007220	0.007812	0.008257	0.008801	0.009599	

Table 2: Time to complete for small problem sizes n and threads requested t .

n (primes)	Requested threads t (useful threads)					
	Time to complete					
50000 (5133)	1 (1)	2 (2)	3 (3)	4 (3)	5 (4)	6 (4)
	0.006472	0.006177	0.006375	0.006974	0.007393	0.008489
	7 (4)	8 (4)	9 (4)	10 (4)	11 (4)	12 (4)
	0.007616	0.007628	0.007413	0.008453	0.007423	0.008116
	14 (4)	16 (4)	18 (4)	20 (4)	25 (4)	30 (4)
	0.009826	0.008214	0.008839	0.008670	0.009934	0.010021
	35 (4)	40 (4)	45 (4)	50 (4)	55 (4)	60 (4)
0.010301	0.011245	0.012020	0.012156	0.012905	0.014215	
100000 (9592)	1 (1)	2 (2)	3 (3)	4 (3)	5 (4)	6 (4)
	0.014804	0.012471	0.012732	0.014472	0.015033	0.016610
	7 (4)	8 (4)	9 (4)	10 (4)	11 (4)	12 (4)
	0.014766	0.015138	0.014502	0.016517	0.014101	0.015503
	14 (4)	16 (4)	18 (4)	20 (4)	25 (4)	30 (4)
	0.018224	0.015583	0.015466	0.015648	0.017244	0.017534
	35 (4)	40 (4)	45 (4)	50 (4)	55 (4)	60 (4)
0.016819	0.018333	0.019560	0.018944	0.019778	0.021861	
200000 (17984)	1 (1)	2 (2)	3 (3)	4 (3)	5 (5)	6 (5)
	0.037784	0.027672	0.027107	0.030217	0.032718	0.034382
	7 (4)	8 (4)	9 (5)	10 (4)	11 (4)	12 (4)
	0.029534	0.032079	0.028481	0.031938	0.028280	0.030773
	14 (5)	16 (4)	18 (4)	20 (5)	25 (4)	30 (4)
	0.038588	0.029763	0.029649	0.031066	0.032412	0.034486
	35 (4)	40 (4)	45 (4)	50 (4)	55 (4)	60 (4)
0.031461	0.032036	0.035523	0.033640	0.034074	0.039042	
500000 (41538)	1 (1)	2 (2)	3 (3)	4 (3)	5 (5)	6 (6)
	0.117006	0.084923	0.077818	0.080502	0.078806	0.091633
	7 (6)	8 (6)	9 (6)	10 (6)	11 (6)	12 (6)
	0.077450	0.081279	0.077690	0.092886	0.079780	0.078590
	14 (6)	16 (6)	18 (6)	20 (6)	25 (6)	30 (6)
	0.104752	0.078467	0.078200	0.078868	0.088215	0.079702
	35 (6)	40 (6)	45 (6)	50 (6)	55 (6)	60 (6)
0.080681	0.080124	0.081990	0.082318	0.081957	0.091619	
1000000 (78498)	1 (1)	2 (2)	3 (3)	4 (4)	5 (5)	6 (6)
	0.252872	0.188016	0.173659	0.178691	0.173696	0.180802
	7 (7)	8 (8)	9 (8)	10 (8)	11 (8)	12 (8)
	0.173690	0.175263	0.170488	0.176459	0.173192	0.172405
	14 (8)	16 (8)	18 (8)	20 (8)	25 (8)	30 (8)
	0.227436	0.172230	0.171850	0.180473	0.175208	0.172450
	35 (8)	40 (8)	45 (8)	50 (8)	55 (8)	60 (8)
0.175576	0.174560	0.174063	0.174745	0.176508	0.187405	

Table 3: Time to complete for medium problem sizes n and threads requested t .

n (primes)	Requested threads t (useful threads)					
	Time to complete					
2000000 (148933)	1 (1)	2 (2)	3 (3)	4 (4)	5 (5)	6 (6)
	0.532224	0.399892	0.372592	0.388680	0.371474	0.387636
	7 (7)	8 (8)	9 (9)	10 (10)	11 (10)	12 (10)
	0.372675	0.378897	0.373559	0.380909	0.371871	0.370893
	14 (13)	16 (10)	18 (10)	20 (10)	25 (12)	30 (10)
	0.500548	0.369513	0.369225	0.376344	0.376002	0.367903
	35 (12)	40 (10)	45 (10)	50 (12)	55 (12)	60 (10)
0.374427	0.381711	0.370551	0.372428	0.374830	0.374455	
5000000 (348513)	1 (1)	2 (2)	3 (3)	4 (4)	5 (5)	6 (6)
	1.424476	1.060750	0.995090	1.037742	1.011496	1.010956
	7 (7)	8 (8)	9 (9)	10 (10)	11 (11)	12 (12)
	1.013497	1.015443	0.996567	0.995944	0.999886	0.998278
	14 (14)	16 (16)	18 (18)	20 (19)	25 (19)	30 (19)
	1.253758	0.986469	1.004526	0.991738	0.991733	0.989739
	35 (17)	40 (19)	45 (18)	50 (18)	55 (19)	60 (18)
0.988248	0.996908	0.993764	0.990465	1.011001	0.998305	
1e7 (664579)	1 (1)	2 (2)	3 (3)	4 (4)	5 (5)	6 (6)
	2.947869	2.237036	2.066807	2.130625	2.128636	2.119406
	7 (7)	8 (8)	9 (9)	10 (10)	11 (11)	12 (12)
	2.088069	2.096087	2.126888	2.092307	2.090862	2.092737
	14 (14)	16 (16)	18 (18)	20 (20)	25 (25)	30 (26)
	2.341456	2.071571	2.092379	2.072495	2.083050	2.066162
	35 (26)	40 (26)	45 (26)	50 (26)	55 (26)	60 (26)
2.093317	2.064359	2.080665	2.053868	2.083751	2.069733	
2e7 (1270607)	1 (1)	2 (2)	3 (3)	4 (4)	5 (5)	6 (6)
	6.096491	4.750722	4.284478	4.486591	4.431566	4.489721
	7 (7)	8 (8)	9 (9)	10 (10)	11 (11)	12 (12)
	4.497476	4.421862	4.406341	4.428048	4.405715	4.440867
	14 (14)	16 (16)	18 (18)	20 (20)	25 (25)	30 (30)
	4.817708	4.408149	4.415023	4.285703	4.464103	4.279061
	35 (34)	40 (34)	45 (34)	50 (35)	55 (34)	60 (35)
4.435391	4.290525	4.446403	4.263221	4.432420	4.260772	
5e7 (3001134)	1 (1)	2 (2)	3 (3)	4 (4)	5 (5)	6 (6)
	15.835393	12.331544	11.268145	11.513501	11.568131	11.528276
	7 (7)	8 (8)	9 (9)	10 (10)	11 (11)	12 (12)
	11.533443	11.413640	11.442190	11.352906	11.402080	11.382500
	14 (14)	16 (16)	18 (18)	20 (20)	25 (25)	30 (30)
	11.757911	11.372443	11.443325	11.160945	11.353323	11.208956
	35 (35)	40 (40)	45 (45)	50 (50)	55 (52)	60 (52)
11.459338	11.203706	11.333071	11.174897	11.425767	11.161547	

Table 4: Time to complete for large problem sizes n and threads requested t .

n (primes)	Requested threads t (useful threads)					
	Time to complete					
1e8 (5761455)	1 (1)	2 (2)	3 (3)	4 (4)	5 (5)	6 (6)
	33.011163	25.929887	23.339192	24.101956	23.887196	23.968062
	7 (7)	8 (8)	9 (9)	10 (10)	11 (11)	12 (12)
	23.786406	23.657695	23.897440	23.688316	23.654360	23.754729
	14 (14)	16 (16)	18 (18)	20 (20)	25 (25)	30 (30)
	24.546775	23.587154	23.653213	23.033871	23.803751	23.078008
	35 (35)	40 (40)	45 (45)	50 (50)	55 (55)	60 (60)
23.814698	23.077176	23.737852	23.048788	23.779899	23.047622	
2e8 (11078937)	1 (1)	2 (2)	3 (3)	4 (4)	5 (5)	6 (6)
	69.241135	55.254652	49.098646	50.387292	50.173795	50.504890
	7 (7)	8 (8)	9 (9)	10 (10)	11 (11)	12 (12)
	50.130417	49.914960	49.470869	49.454488	49.737794	49.447833
	14 (14)	16 (16)	18 (18)	20 (20)	25 (25)	30 (30)
	49.749970	49.179935	49.344618	47.955956	49.470626	48.151827
	35 (35)	40 (40)	45 (45)	50 (50)	55 (55)	60 (60)
49.338160	48.053016	50.096202	48.042824	49.903375	48.072644	
5e8 (26355867)	1 (1)	2 (2)	3 (3)	4 (4)	5 (5)	6 (6)
	184.633938	154.193454	141.288938	137.845428	138.015689	137.228580
	7 (7)	8 (8)	9 (9)	10 (10)	11 (11)	12 (12)
	134.316781	134.176361	133.960360	133.056746	134.059484	131.835291
	14 (14)	16 (16)	18 (18)	20 (20)	25 (25)	30 (30)
	129.977515	132.284299	132.057187	130.701120	133.822637	131.059783
	35 (35)	40 (40)	45 (45)	50 (50)	55 (55)	60 (60)
134.073412	131.694874	133.885167	132.057540	134.887862	132.674345	

Table 5: Time to complete for huge problem sizes n and threads requested t .