Parallel Architecture and Distributed Programming
Lab Internals (16CS 71)

Prog 3.

Aim: Write a program for Cache unfriendly sieve of Eratosthenes and Cache friendly sieve of Eratosthenes for enumerating prime numbers upto N and prove the correctnes

```
#include <math.h>
# include <string.h>
# include <omp.h>
# include <iostream>
 using name spau std;
 double t = 0.0;
 inline long  Strike ( bool composite [], long i, long stride, long limit)
 {     for (; i<=limit ; i+=stride)
               composite [i] = true;

       return i;

 }
 long  CacheUnfriendly Sieve (long n)
 {     long count = 0;
       long m = (long) sqrt ((double)n );
       bool * composite = new bool [n+1];
       memset (composite, 0, n);
       t = omp_get_ wtime ();
       for ( long i=2; i <= m; ++i)
       {           if (! composite [i])
                   {     ++count;
                         Strike ( composite, 2*i, i, n);

                   }

       }
```

```
        for ( long i=m+1; i<=n; ++i)
            if (!composite[i])
                ++count;

        t = omp-get-wtime () - t;
        delete [] composite;
        return count;
}
long    CacheFriendly Sieve (long n)
{
      long count = 0;
      long m = (long) sqrt ((double)n);
      bool * composite = new bool [n+1];
      memset (composite, 0, n);
      long * factor = new long [m];
      long * striker = new long [m];
      long n_factor = 0;
      t = omp-get_wtime ();
      for (long i=2; i<=m; ++i)
          if (!composite[i])
          {     ++count;
                striker [n_factor] = strike (composite, 2*i, i,
                                                        m);
                factor [n_factor ++] = i;

          }
      // chop sieve into window of size sqrt(n)
      for (long window=m+1; window <=n; window+=m)
      {
            long limit = min(window + m -1, n);
            for (long k=0; k<n_factor; ++k)
            // strikes through window
                striker[k] = strike (composite, striker[k],
                                          factor[k], limit);

      }
```

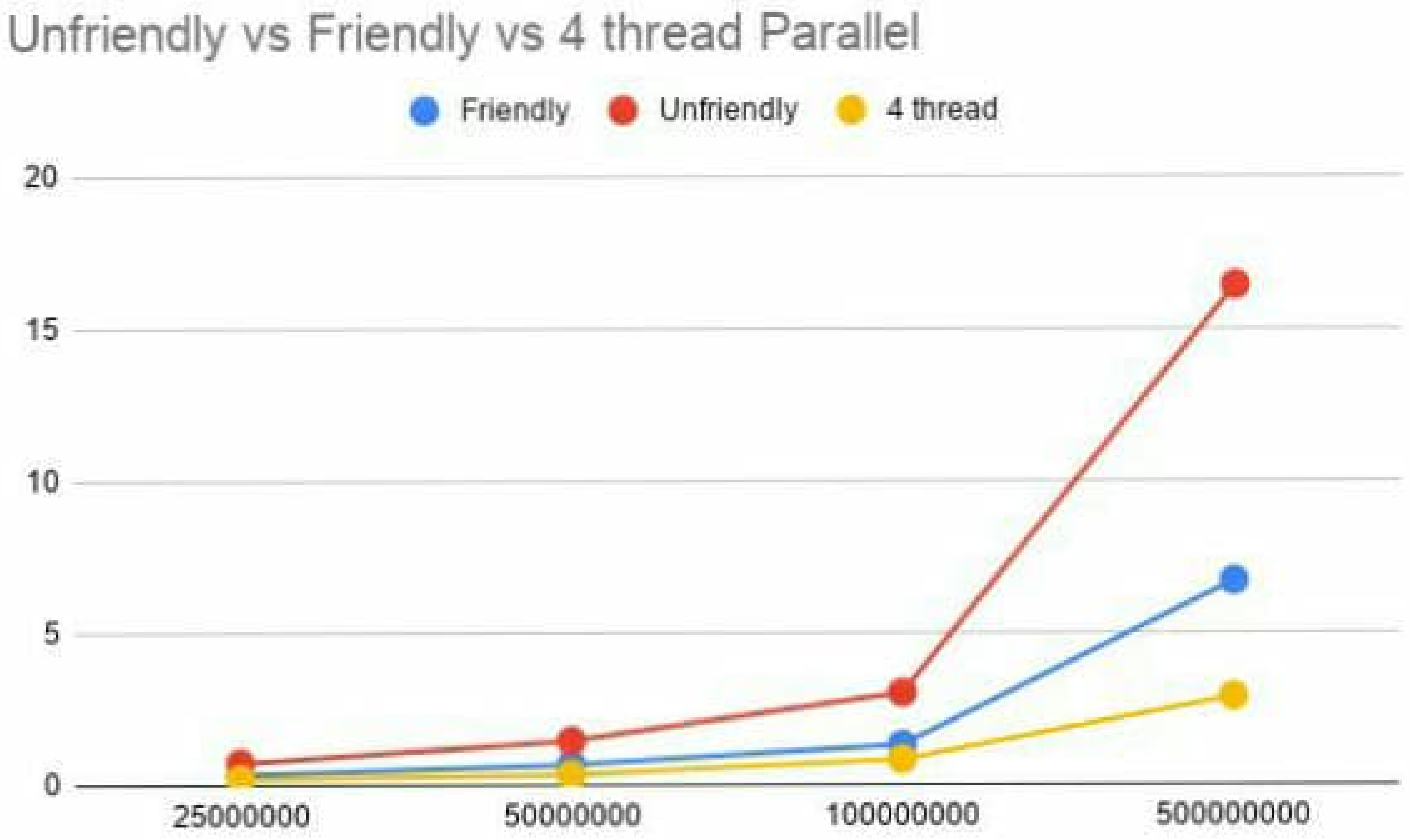
```
        for ( long i = window; i <= limit ; ++i )
        {         if ( ! composite [i])
                           ++ count;
        }
    }
    t = omp- get_wtime() - t;
    delete [] striker;
    delete [] factor;
    delete [] composite;
    retorn count;
}
long  Parallelsieve ( long n,  int numThreads)
{       long  count = 0;
        long m =  (long)sqrt((double)n);
        long n_factor  = 0;
        long * factor  =  new long [m];
         t = omp-get_wtime();
        omp-set_num_threads (num Threads);
        #pragma  omp parallel
        {      bool * composite  =  new bool [m+1];
               long * striker   =  now long [m];
               # pragma  omp single
               {     memset (composite, 0, m);
                     for (long i=2; i<=m ; ++i)
                     {       if (!composite [i])
                             {    ++ count;
                                  striker[n_factor]= Strike (composite,
                                                  2*i, i, m);
                                  factor [n_factor++] = i;
                     }       }
               }
               long  base  = -1;
```

```
// chop sieve into windows of size ≈ sqrt(n)
#pragma omp for reduction (+: count)
for (long window = m+1 ; window <= n ; window += m)
{     memset (composite, 0, m);
      if (base != window)
      {     base = window;
            for (long k = 0; k < n_factor ; ++k)
                 striker[k] = (base + factor[k] -1) * factor[k]
                                                      - base;
      }
      long limit = min (window + m -1, n) - base;
      for (long k = 0; k < n_factor; ++k)
           striker[k] = strike (composite, striker[k],
                                  factor[k], limit) - m;

      for (long i = 0; i <= limit; ++i)
           if (! composite[i])
                ++ count;

      base += m;
   }
   delete [] striker;
   delete [] composite;
  }
} t = omp_get_wtime - t;
  delete [] factor;
  return count;

}
```

```
int main()
{
        long N;
        cout <<" Enter value of N: ";
        cin >> N
        cout << "CACHE UNFRIENDLY SIEVE" << endl;
        cout << "Count =' << cache Unfriendly Sieve (N) << endl;
        cout << "Time =" << t << endl;
        cout << endl;
        cout << "CACHE FRIENDLY SIEVE" << endl;
        cout << "count =" << CacheFriendly Sieve (N) << endl;
        cout << "Time =" << t << endl;
        cout << endl;
        int numThreads;
        cout << "PARALLEL SIEVE" << endl;
        cout << "Enter no. of threads";
        cin >> numThreads;
        cout << "count =" << Parallel sieve (N, numThreads) << endl;
        cout << "Time =" << t << endl;
```

OUTPUT

			Parallel			
Size	Cache unfriendly	Cache Friendly	1	2	4	8
2.5×10^7	0.723615	0.339038	0.352405	0.238184	0.20902	0.154209
5×10^7	1.45348	0.664864	0.688958	0.577745	0.343001	0.298766
10^8	3.03144	1.30764	1.29965	0.71153	0.821015	0.630775
5×10^8	16.4208	6.70912	6.75891	3.45342	2.90193	2.90126


Unfriendly vs Friendly vs 4 thread Parallel
Friendly
Unfriendly
4 thread
20
15
10
5
0
25000000
50000000
100000000
500000000