Parallel Algorithms: Eratosthenes' Sieve

Paul van der Walt
paul@denknerd.org
http://github.com/toothbrush/eratosthenes*

November 2, 2010

Sift the Twos and sift the Threes,
The Sieve of Eratosthenes.
When the multiples sublime,
The numbers that remain are Prime.

Abstract

In this report the findings are presented after benchmarking the Dutch supercomputer Huygens and a personal computer. The Sieve of Eratosthenes is then implemented in sequence and parallel (including a number of performance improvements), and tested on Huygens and the personal computer.

Contents

*The complete working code and project can be found at this location, so all the tests and results presented can be verified.

1 Introduction

This report documents the use of BSP-style[BSP] parallel programming to find primes in parallel, using Eratosthenes' method.

The so-called sieve of Eratosthenes is an old and unsophisticated method for finding all primes up to a given number N, which lends itself quite nicely to parallelisation. The idea is roughly as follows: start with the lowest known prime i (given: $i \leftarrow 2$ in the first iteration) and cross all multiples of i out of your list of potential primes, which starts off with all natural numbers $1 \ldots N$. While i is less than N, repeat. The next lowest known prime is the smallest number which hasn't been crossed off the list yet. More information about Eratosthenes' prime sieve can be found in Section 2.

Secondly, we take a look at benchmarking BSP computers and the performance measured when testing on Huygens[SHuy], the Dutch national supercomputer, and on a recent MacBook. Specifically, the benchmark is aimed at parallel computers, measuring not only computation speed, but also synchronisation time and communication speed.

2 Prime Sieve

2.1 Description

To quickly recap, Eratosthenes' method finds all prime numbers less than or equal to n. There are a number of obvious and less obvious improvements which can be made on this algorithm, but at its most basic form it is as follows.

1. Create a list of consecutive integers from two to n: (2, 3, 4, ..., n).
2. Initially, let p equal 2, the first prime number.
3. Strike from the list all multiples of p less than or equal to n. ($2p, 3p, 4p$, etc.)
4. Find the first number remaining on the list after p (this number is the next prime); replace p with this number.
5. Repeat steps 3 and 4 until p^2 is greater than n.
6. All the remaining numbers in the list are prime.

This algorithm will be assumed correct (proof is outside the scope of this report). One of the first optimisations is that crossing off can begin at p^2 since all smaller multiples of p will already have been crossed off in earlier iterations. The time complexity after this optimisation is $O(n \log \log n)$ [Prit87]. Another optimisation one could implement is to not make a list of all n numbers, but leave out all even numbers, since these are known to be non-prime. This will save half the necessary memory, but asymptotically this makes no difference for memory requirements in the RAM model.

The sequential implementation can be found in Appendix A, and follows this algorithm closely.

2.2 Data distribution

As a result of the BSP model's non-shared memory assumption, finding a parallel implementation of our algorithm starts with deciding on a good data distribution. The consideration is that most of the work the algorithm does is crossing off numbers, therefore, we must distribute such that the crossing off of numbers is as evenly distributed over processors as possible. The first idea is then to use a so-called block distribution, where given P processors and N values, we give the each processor as close as possible to N/P values. This way, we can broadcast the smallest prime found, each processor can independently cross multiples off and find it's local minimum prime, and let $P(0)$ calculate the global minimum prime, and repeat. For more on the BSP model and data distributions, see [Biss04].

The data distribution we used is therefore slightly different than the "naive" block distribution, where the last processor potentially has only 1 element.

- First element controlled by processor s: $\lfloor \frac{s \cdot N}{P} \rfloor$
- Last element controlled by processor s: $\lfloor \frac{(i+1)N}{P} \rfloor - 1$
- Processor controlling element j: $\lfloor \frac{P(j+1)-1}{N} \rfloor$

This distribution causes the number of elements per processor to differ by at most 1.

2.3 The parallel algorithm

More formally, the algorithm becomes as follows. In common BSP style, the algorithm is parameterised by the current processor ID s. The functions minidx(), maxidx() and local() are used for keeping track of indices in the array, since it has been split into blocks over the processors. Min and maxidx give the minimum and maximum number stored on a given processor, and local gives the local index of a number on a processor (this is found by calculating the argument modulo $\lceil N/P \rceil$, roughly). See Appendix B for implementational details.

1. Initialise a list A minidx(s) .. maxidx(s), all `true`.
2. If $s = 0$ then $A_1 \leftarrow$`false` (we know 1 isn't prime).
3. Set $k \leftarrow 2$ (2 is our largest prime to start off with).
4. While $k^2 \leq n$ **do**
   (a) For $i \in k, 2k, \cdots$ set $A_{\text{local}(i)} \leftarrow$`false` (eliminate multiples of k locally).
   (b) Find $m_s \leftarrow \text{argmin}_i(A_i =$ `false`) (local next prime).
   (c) If $s = 0$ then read all m_i and select minimum (collect local next primes).
   (d) Broadcast as k (global next prime).
5. **od**

2.4 Complexity

Before running our experiments, we proceed to analyse the theoretical complexity of the parallel algorithm. This will give an idea of what to expect in the following section.

First we look at the main loop, which runs for $k \in [1..\sqrt{n}]$. The step size can be assumed to be such that at most the main loop will be done once for each prime in the interval $[1..\sqrt{n}]$. We use Gauss' approximation (1) of the prime distribution, which is

$$\int_2^N \frac{1}{\log t} \mathrm{t}. \tag{1}$$

Next we look at the amount of work done in each loop iteration. Roughly $\frac{n}{pk}$ numbers are crossed out, n/p flops are required to find local smallest primes, and p flops are required to find the global prime. This, along with Gauss' approximation, leads to total (discretised) computation costs of

$$\sum_{k=2}^{\sqrt{n}} \left(\frac{1}{\log k} \left(\frac{n}{pk} + p + \frac{n}{p} \right) \right). \tag{2}$$

We assume that the integral over $\frac{1}{\log t}$ is $O(n \log\log n)$, which leads to computation costs of $O(n \log\log n + p + \frac{n}{p})$, which is still dominated by the $n \log\log n$ term, which is the number of cross-offs that need to be done.

The communication required is also $2(p-1)$ for each loop iteration, as the global and local next primes need to be broadcast (in other words, $b = O(pn \log\log n)$ in the total cost equation).

Finally, there are 4 supersteps, namely the local work of crossing off non-primes, then the broadcast of local next primes, then the computation of global minimum prime, then the broadcast of that value.

Ignoring the non-dominating initial setup costs (initialising arrays etc.) this leads to costs of the form 3.

$$O(n \log\log n) + O(pn \log\log n)g + 4l \tag{3}$$

3 Performance

3.1 Timed results

Next we put our algorithms to the test. We hope to see a marked improvement over the sequential (read: $P = 1$) implementation, since the complexity is much better when P increases.

The obtained results can be seen in Table 1.

In comparison, the results obtained using the simpler sequential algorithm, also using $N = 10^8$, are more interesting.

Due to memory limitations (as a result of not fully optimising the algorithm) N isn't set higher than 10^8. We see that in this case, Huygens is a little faster than the laptop, which isn't what we would expect given the benchmark results, but is what we would expect given that the laptop is running at 2.4Ghz and Huygens' cores run at 4.7Ghz.

Computer	$P = 1$	$P = 2$	$P = 4$	$P = 8$	$P = 16$	$P = 32$
Huygens	20.36	10.87	5.54	2.96	1.91	11.28
MacBook	12.14	7.60	7.88	8.40	11.16	30.04

Table 1: The results when running the parallel implementation of the Sieve of Eratosthenes. In all cases, $N = 10^8$ and unit of time is seconds.

Computer	time
Huygens	6
MacBook	7

Table 2: The results when running the sequential implementation of the Sieve of Eratosthenes. Unit of time is seconds.

In all cases the output of the algorithm was verified to produce the correct primes. Mathematica was used as the authority on the number of primes below a given number.

Finally, it is interesting to look at the so-called speedup plot. This is the time for each run, divided by the time when $P = 1$. This gives an idea how well the algorithm scales with number of processors.

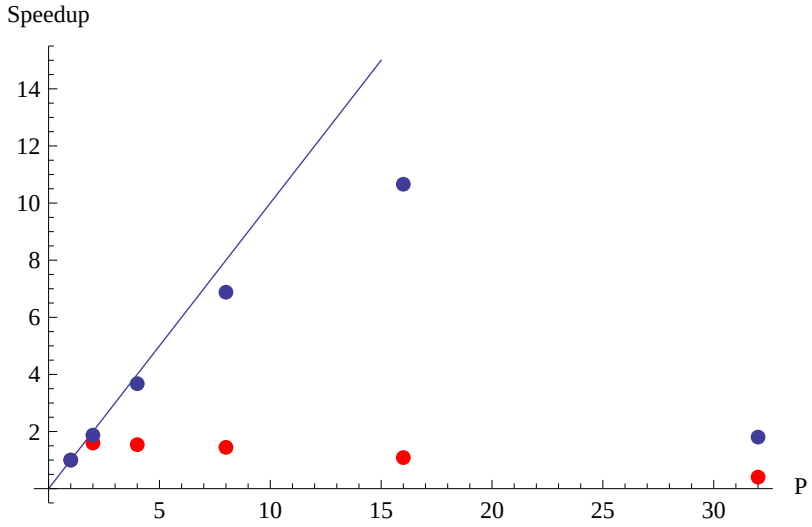

Figure 1: The speedup plot for the parallel implementation. The red points are the MacBook, and the blue points represent Huygens. The solid line is ideal parallel scaling.

We see quite reasonable behaviour in the speedup plot (Figure 1), since on Huygens, the speedup is close to the ideal parallelisation expectation. Evidently there is still some room for improvement though. The speedup drops off dramatically after $P = 16$, this is explained in Section 3.2. The results for the MacBook are rather poor, the reason for this being that it only has 2 cores, so the incurred overhead for $P > 2$ dominates.

3.2 Analysis of results

A number of things are clear when we look at the results of Table 1. First of all, we see that the laptop indeed outperforms Huygens when using a single core, which is also seen in Section 4 where the raw computational rate r is measured to be higher on the laptop. The most likely cause for this is that when running on the laptop, all other processes can be given lower priority, enabling the sieve to use all available processor power. On Huygens, however, nodes are shared, and therefore we are not the sole users of a given node.

When we start increasing P we see the profit from parallelising the algorithm. The running time roughly halves, as expected, when running on 2 cores, and for the MacBook, this is when maximal performance is achieved. This is because the MacBook has a dual-core processor, so setting $P = 2$ is making optimal use of the resources. On Huygens, however, nodes have 16 cores each, which is why we see a continuing decrease in running time until $P = 16$. The sudden increase at $P = 32$ is as a result of using 2 nodes with 16 cores each; now the communication has to cross an intermediary network and isn't local any more.

On the MacBook, the reason for the time increase is different: all threads are still local, but since there are only 2 cores, creating more threads only adds to the overhead which BSP has to administer, and doesn't off a performance improvement.

These results are all consistent with our theoretical analysis of the algorithm, namely that the performance will increase linearly with the number of processors available, assuming communication cost doesn't dominate, which is what we see in the case of $P = 32$ on Huygens.

4 Benchmarks

The two computers tested in this section, as already mentioned, are a recent MacBook[1] and Huygens, the national supercomputer. Of course we didn't do the benchmark on all the processors of Huygens, as there are in the order of three thousand of them, but only on 1 and 2 nodes (each node has 16 cores).

The benchmark that has been done is the one included in BSPedupack which can be found at [BSPep], referred to as `bspbench.c`. The results are presented in Figures 2 to 8. In all these figures, the horizontal axis represents the value of the h-relation, in other words, how many data words are communicated in the communication superstep, and the vertical axis represents the time (in flops) taken for the communication superstep. Furthermore the well-known meanings of r, g and l hold, namely computation rate, communication cost per data word in flops and sync time in flops, respectively.

4.1 Original benchmark

In figures 2 and 3 we see the results of running the plain benchmark on Huygens and the laptop. Interestingly enough, the laptop seems to perform better than Huygens in all respects. The only reason we can give for this is that the shared cores on Huygens are a lot more overworked than the laptop when it is doing

[1] Core2Duo 2.4Ghz with 4GB memory.

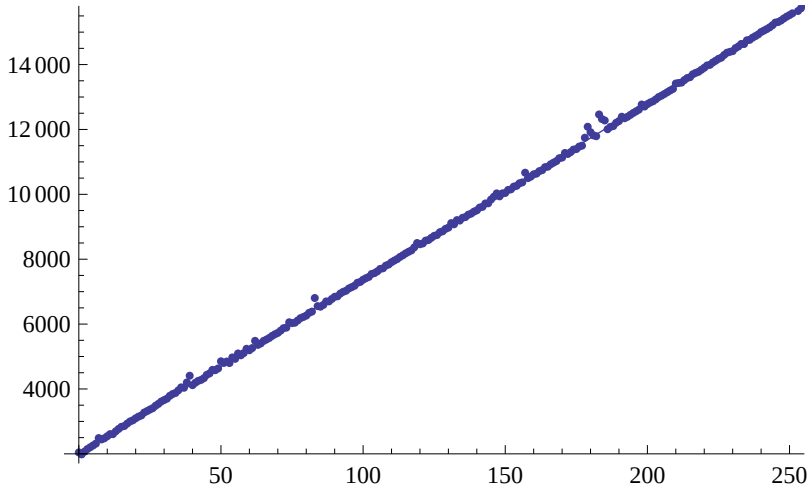

Figure 2: Benchmark results on Huygens with P=2; the benchmark bottom line was $r = 194.357$ Mflop/s, $g = 54.2$, $l = 1986.4$

nothing else other than running the benchmark. In a sense this is therefore an unfair comparison.

Note that on Huygens we used 1 node, which means the communication would probably be cheaper than if we were to do the same benchmark, but distribute it over 2 nodes, so the communication has to leave the node as opposed to being local, as it would have been in Figure 2.

We see predictable and expected linear behaviour on Huygens, but the laptop results are a lot more noisy. We attribute this to the fact that other processes such as the graphical user interface are running which generate unpredictable performance hits. These observations also hold on the get-versions of these 2 tests.

The results shown in Figure 4 are when we select 2 distinct nodes on Huygens. We indeed see a marked increase in communication cost g and synchronisation cost l. As we would expect, the computational rate r is basically unaffected by this change, as the work needing to be done locally is the same as before.

Since here we are forcing network communication, the performance becomes more interesting to analyse. Where the local version (Figure 2) has a very straight performance "curve", we now observe a peak and some waviness. The waviness is attributed to other network communication outside of our control which will affect the speed with which our packets reach their destination; the peak is possibly the point at which the underlying layer decides to switch to another, more high-volume, communication scheme. The latter point is, however, guesswork, without knowing more about our platform.

4.2 Changing to `gets`

Now we change puts into gets in the benchmark algorithm. The hypothesis is that getting is more expensive than putting, since a put is passive with respect to the receiver, whereas when getting both processors have to work. See figures

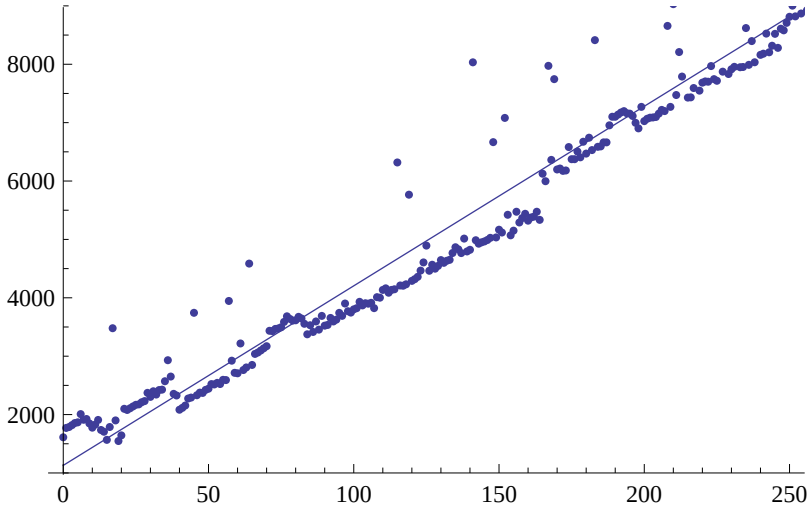

Figure 3: Benchmark results on MacBook with P=2; bottom line was r = 399.853 Mflop/s, $g = 30.9$, $l = 1112.4$

5 to 7.

Comparing 5 and 6 to the corresponding figures in the previous section, we indeed see a marked increase in communication cost (note: not in synchronisation cost) on Huygens. Surprisingly enough, on the laptop, this increase is not so great on the laptop. We expect and observe the synchronisation time to be comparable with puts and gets, since this change shouldn't affect the synchronisation steps.

Once again, we try forcing the benchmark to run on 2 distinct nodes on Huygens (see Figure 7). Again the computation rate r is the same, communication cost g is significantly increased, but in this case, also l suffers, increasing by nearly a factor of 2. We cannot give an explanation for this, and attribute it to incidental load on the shared node being used. Note that this is plausible seeing the entire benchmark runs in about 20 seconds, which will never give a very accurate picture of a computers average performance.

We also see not 1, but at least 2 discernible peaks in the curve. Once again the conjecture is that the communication schemes used switch at these points.

As a last effort at producing more interesting results, we run the benchmark on the laptop with $P = 56$. This isn't done on Huygens since it will be a waste of resources and will probably have to wait a long time in the queue. See Figure 8. The results here speak for themselves: the computation rate r is still independent of the configuration, but the communication cost and specifically synchronisation time skyrocket. The reasons for this are obvious: namely, communication becomes more costly when the underlying BSP layer has to manage more and more processors, and the synchronisation cost is hypothesised to be $\omega(n)$ (i.e. at least linearly increasing in the number of processors used, N).

Interestingly we also see a sharp increase in communication time before $h = 56$, where after that point the time levels out to become linear. This is logical, since before $P = 56$, not all processors are being communicated with, so

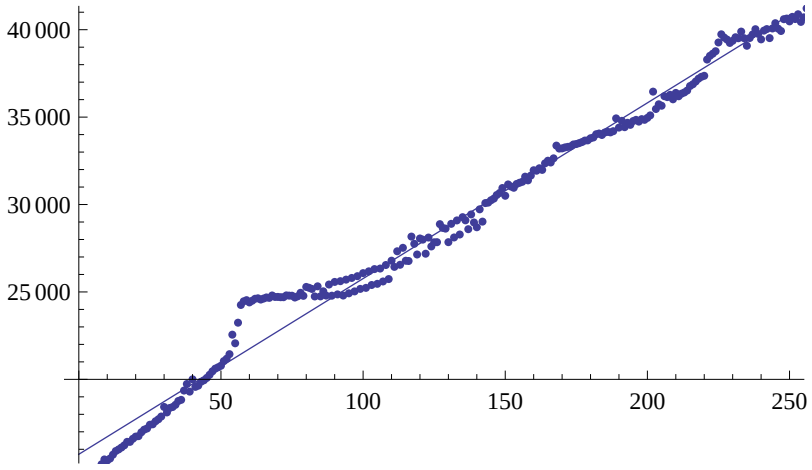

Figure 4: The same test as in Figure 2, but then instead of 2 processes on one node, we tried 2 processes on 2 nodes. The bottom line becomes $r = 195.648$ Mflop/s, $g = 100.2$, $l = 15770.4$

the communication superstep takes much longer each time an extra processor is talked to. After all the processors have been communicated with (i.e. $h > 56$) BSP is smart enough to bundle communication to a given processor, so that the communication time increases linearly in h.

5 Conclusion

Evidently a lot of performance can be gained if an algorithm is parallelised. Unfortunately this isn't always trivial; the case of the sieve of Eratosthenes happens to be relatively simple, but many other algorithms are a lot more of a challenge to parallelise and still see a performance increase.

The conceptual complexity of an algorithm also dramatically increases when we have to reason about concurrently working processors, which complicates the task even more. It is, however, fruitful to assess whether a given algorithm can be parallelised, as we have seen the gains to be achieved in the results above.

A Sequential sieve code

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

#define ulong long

enum bool { true, false };
ulong nextPrime(ulong, enum bool*) ;
void printPrimes(ulong, enum bool*) ;                                    10
```

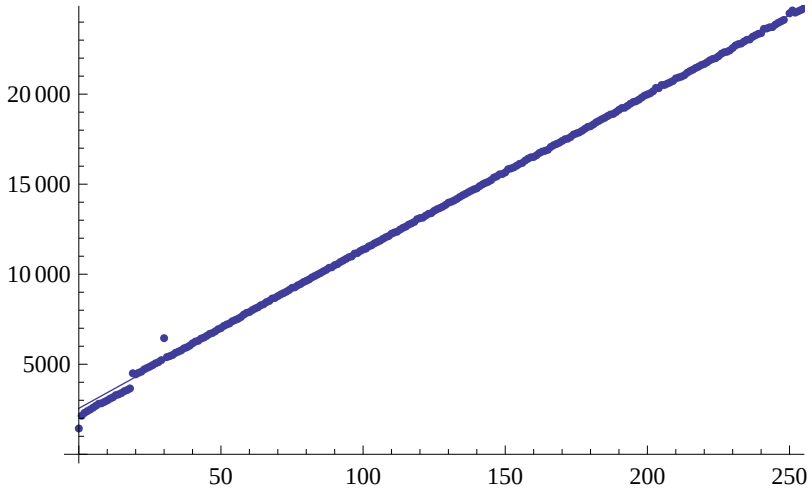

Figure 5: Having changed `puts` into `gets` we measure again on Huygens with $P = 2$. Bottom line: $r = 191.062$ Mflop/s, $g = 87.2$, $l = 2574.6$

```
int main(int argc, char** argv)
{
    ulong N, i;
    printf("Hi! Welcome to sequential sieve.\n");
    if(argc != 2) {
        printf("Incorrect arguments idiot.\n");
        exit(1);
    }
    else                                                          20
    {
        sscanf(argv[1], "%lu", &N);
        printf("you entered N== %lu\n", N);
    }

    // --- begin

    time_t t1,t2;
    time(&t1);
                                                                  30
    printf("Trying to alloc %lu Mb. . . \n",N*sizeof(enum bool)/1024/1024);
    enum bool *A = malloc(sizeof(enum bool)*N);
    if(A == NULL)
    { printf("Bad news, malloc failed. Try lower N.\n"); exit(1); }

    // assume all are prime to start with,
    // except 0 and 1
    A[0] = false;
    A[1] = false;
    for(i = 2; i < N; i++)                                        40
    {
        A[i] = true;
    }

    // ... and start with k <- 2
    ulong current = 2;
```

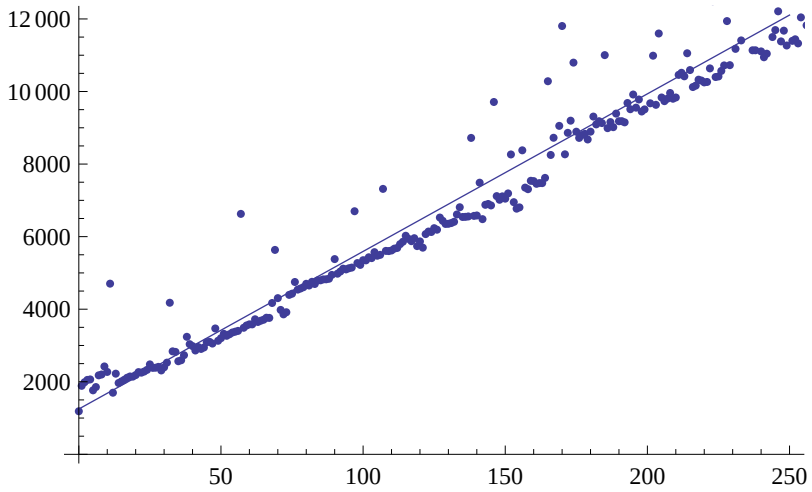

Figure 6: MacBook with $P = 2$. Bottom line: $r = 389.348$ Mflop/s, $g = 43.5$, $l = 1240.1$

```
    while (current*current <= N)
    {
        // start at current^2, optimisation
        for(i = current; i <= N/current; i++)
        {
            A[i*current] = false;
        }
        current = nextPrime(current, A);
    }

    printPrimes(N, A);
    time(&t2);
    printf("And it took: %lf sec\n", difftime(t2,t1));
    exit(0);
}

ulong nextPrime(ulong c, enum bool*A)
{
    ulong next = c+1;
    while ( A[next] != true)
        next++;

    return next;
}

void printPrimes(ulong N, enum bool* A)
{
    ulong i;
    ulong nPrimes=0;
    for(i = 0; i < N; i++)
        if(A[i] == true)
        {
            // do not print primes, just count them.
            nPrimes++;
        }

    printf("==> %lu primes.\n", nPrimes);
```

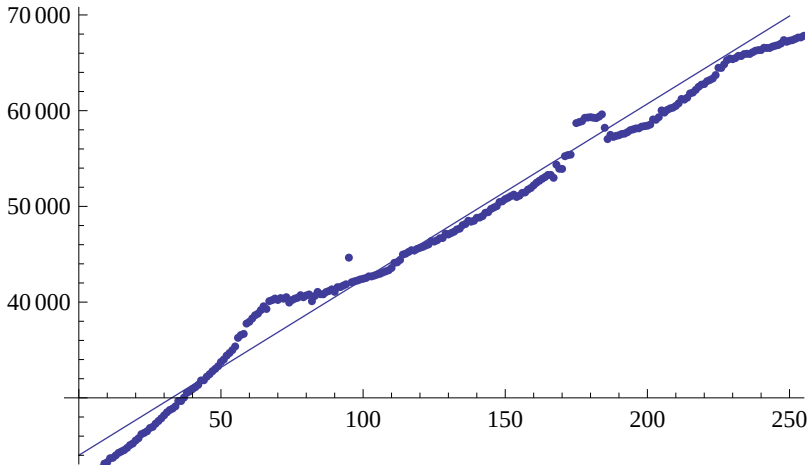

Figure 7: The same test as in Figure 5, but then instead of 2 processes on one node, we tried 2 processes on 2 nodes. The bottom line becomes $r = 194.498$ Mflop/s, $g = 182.3$, $l = 24222.0$

}

B Parallel sieve code

```
#include "../bspedupack.h"
#include <limits.h>
#include <stdlib.h>

/*
 * Author: Paul van der Walt
 * october 2010
 *
 * This program computes all prime numbers <= a given integer N.
 */

// prototypes:
ulong blockLow(int, int, ulong);
/*
ulong blockSize(ulong, ulong, ulong);
ulong blockHigh(ulong, ulong, ulong);
ulong blockOwner(ulong, ulong, ulong);
ulong globalIdx(ulong, ulong, ulong, ulong);
ulong localIdx(ulong, ulong, ulong, ulong);
*/

//globals:
int P; /* number of processors requested */
ulong N; /* requested max prime */

ulong findMinimum(int p, ulong* ks)
{
```

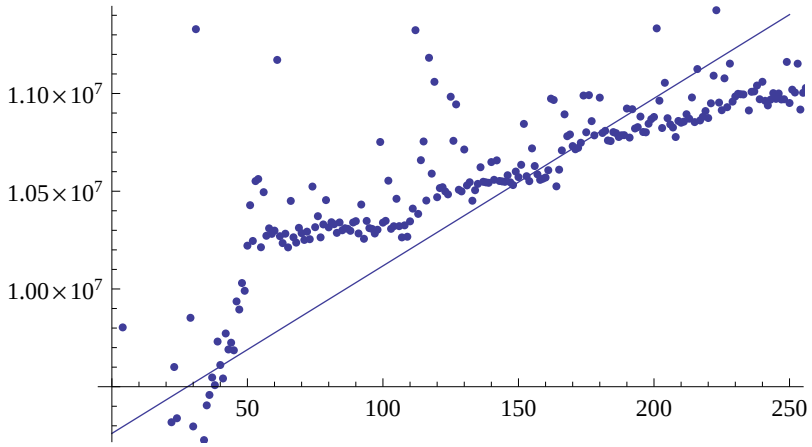

Figure 8: Test on the MacBook with $P = 56$, using `gets`. Bottom line now becomes $r = 445.799$ Mflop/s, $g = 4201.9$, $l = 10037639.0$

```
    ulong min = LLONG_MAX;
    ulong i;
    for(i = 0; i<p; i++)
    {
        min = MIN(ks[i], min);
    }

    return min;
}
ulong globalIdx(int p, int s, ulong n, ulong local)
{
    return local+blockLow(p,s,n);
}
ulong localIdx(int p, int s, ulong n, ulong global)
{
    return global-blockLow(p,s,n);
}
ulong blockLow(int p, int s, ulong n)
{
    // this means we have overflowed our max_value
    // should not happen using LLONGs
    if(s*n < 0)
        printf("Hm. s*n < 0: s=%d, n=%lld, s*n=%lld\n", s,n,s*n);
    return (s*n)/p; //implicit floor
}

ulong blockHigh(int p, int s, ulong n)
{
    return blockLow(p, s+1, n) - 1;
}

int blockOwner(int p, ulong index, ulong n)
{
    if(p*(index+1) < 0)
        printf("Hm. p*index < 0: p = %d, i = %lld, p*i = %lld\n", p, index, p*index);
    return ((p*(index+1))-1)/n; //implicit floor
}
```

```
ulong blockSize(int p, int s, ulong n){
    /* Compute number of local components of processor s for vector
       of length n distributed over p processors with balanced
       block distribution (see paper). */

    return blockLow(p,s+1,n)−blockLow(p,s,n) ;

} /* end blockSize */

void bspmarkmultiples(int p, int s, ulong n, ulong k, ulong *x)
{
    // mark all multiples of k as non−prime in x
    /*
     * if (!(low_value % prime)) first = 0;
     * else first = prime - (low_value % prime);
     */

    ulong i;
    ulong first;

    if(k * k > blockLow(p,s,n)) // k is in our block, or beyond
        first = k * k − blockLow(p,s,n);
    else // k falls before our block;
    {
        if(!(blockLow(p,s,n) % k)) // first element in our block is divisible by k
            first = 0;
        else
            first = k − (blockLow(p,s,n) % k); // start at first multiple of k in block
    }

    for (i=first; i < blockSize(p,s,n); i+= k)
    {
        x[i] = 0; //not a prime
    }


} /* end bspmarkmultiples */

ulong nextPrime(int p, int s, ulong n, ulong k, ulong *x)
{
    // find minimal i s.t. i > k and i unmarked

    ulong newK = k+1;
    ulong local = MAX(
                  localIdx(p,s,n,newK),
                  0); // do not consider primes outside our range

    while(local < blockSize(p,s,n)−1 && x[local] == 0)
    {
        local++;
    }

    if(x[local] == 0)
    {
        return LLONG_MAX; // no primes for this processor. This is possible.
    }
    else
    {
        // if we get here we assume we found a prime!

        return globalIdx(p,s,n,local);
    }
```

```
} /* end nextPrime */

void bspsieve(){

    double time0, time1;
    ulong *x; // local list of candidates
    ulong *ks; //place for proc0 to store intermediate ks
    ulong n,
          nl,
          i,
          iglob;
    int   s,
          p;
    ulong k;  // the current largest sure-prime

    n = N+1; // copy global N and increase by 1. (only proc 1 knows this)
             // this is so the maximum array idx == N

    bsp_begin(P);
    p= bsp_nprocs(); /* p = number of processors obtained */
    printf("Now we have %d processors.\n", p);
    s= bsp_pid();  /* s = processor number */
    if (s==0){
        if(n<0)
            bsp_abort("Error in input: n is negative");
        ks = vecalloculi(p);
    }

    bsp_push_reg(&n,SZULL);
    bsp_sync();

    bsp_get(0,&n,0,&n,SZULL); //everyone reads N from proc 0
    bsp_sync();
    bsp_pop_reg(&n);

    nl= blockSize(p,s,n); // how big must s block be?
    printf("P(%d) tries to alloc vec of %lld ulongs", s, nl);
    printf(", size would be = %lld Mb\n", nl*SZULL/1024/1024);
    x= vecalloculi(nl);

    for (i=0; i<nl; i++){
        // start by assuming everything is prime, except 1
        iglob= globalIdx(p,s,n,i);
        x[i]= iglob;
    }
    if(s==0)
        x[1]=0;
    bsp_sync();
    time0=bsp_time();
    k = 2;
    // begin work

    while( k*k <= n )
    {
        bspmarkmultiples(p,s,n,k,x);
        k = nextPrime(p,s,n,k,x);

        bsp_push_reg(&k, SZULL);
        bsp_sync();

        if(s==0)
```

```
        {
            ks[0] = k; // my k
            for(i=1;i<p; i++)
            {
                bsp_get(i, &k, 0, &ks[i], SZULL);
            }
        }

        bsp_sync();

        if(s==0)
        {
            k = findMinimum(p,ks);
        }
        bsp_sync();

        //broadcast minimum
        bsp_get(0,&k,0,&k,SZULL);
        bsp_sync();

        bsp_pop_reg(&k);
    }

    // end work
    bsp_sync();
    time1=bsp_time();

    ulong primes= 0;
    //printf("Processor %lld primes: \n", s);
    for(i = 0; i < blockSize(p,s,n); i++)
        if( x[i] != 0)
            primes++;
    //do not print primes, just count them.
    printf("proc %d finds %lld primes.\n", s, primes);

    fflush(stdout);
    if (s==0){
        printf("This took only %.6lf seconds.\n", time1-time0);
        fflush(stdout);
        vecfreeuli(ks);
    }

    vecfreeuli(x);
    bsp_end();

} /* end bspsieve */

int main(int argc, char **argv){

    bsp_init(bspsieve, argc, argv);

    /* sequential part */
    if (argc != 2)
    {
        printf("Usage: %s N\n", argv[0]);
        bsp_abort("Incorrect invocation.\n");
    }
    sscanf(argv[1], "%lld", &N);

    printf("max prime requested = %lld\n", N);
    P = bsp_nprocs(); // maximum amount of procs
```

```
    if ( blockSize(P, 0, N) < sqrt(N))
        printf("WARNING: such a large P (%d) with relatively small N (%lld) is inefficient. \n Choosing a lower P is r

    printf("Using %d processors. \n", P);

    /* SPMD part */
    bspsieve();
                                                                                                        260
    /* sequential part */
    exit(0);

} /* end main */
```

References

[BSP] `http://www.bsp-worldwide.org`, homepage of the BSP association.

[SHuy] `https://subtrac.sara.nl/userdoc/wiki/huygens/description`, information page on the Huygens supercomputer.

[BSPep] `http://www.staff.science.uu.nl/~bisse101/Software/software.html`, homepage of BSPedupack, an education set of sample code for the BSP library.

[Prit87] Pritchard, Paul, "Linear prime-number sieves: a family tree," Sci. Comput. Programming 9:1 (1987), pp. 1735.

[Biss04] Rob H. Bisseling, Parallel Scientific Computation: A Structured Approach using BSP and MPI, Oxford University Press, March 2004. ISBN 978-0-19-852939-2