Parallel Algorithms

Assigment one

Author: Rutger Kuyper
Student number: 0715204

Radboud University Nijmegen

Contents

1 BSP Benchmark

First we take a look at the results of the BSP benchmark, from BSPedupack, in an unmodified state. We use this benchmark to measure the BSP parameters g and l when using puts. The benchmark was run both on Huygens (in interactive mode) and on my own computer (a MacBook with a 1.83 GHz Intel Core 2 Duo processor[1]). The benchmark was run for $p = 1, 2, 4, 8, 16$. For each p the benchmark was run five times, after which the extremes were eliminated and the remaining results averaged. The results can be seen in the tables below.

p	g	l
1	53	555
2	56	2089
4	58	4015
8	58	8259
16	60	16716

Table 1: Benchmark on Huygens using puts

p	g	l
1	24	177
2	46	1828
4	79	17007
8	137	59393
16	663	328908

Table 2: Benchmark on MacBook using puts

There are a few things that are clear from these tables. First we see that, on Huygens, g is roughly constant. This is greatly different from the situation on the MacBook: here g clearly increases with the amount of processors (but is lower than on Huygens for $p = 1, 2$). However, remember that all processors on Huygens are *real* processors, whereas on the MacBook we run multiple threads on the same processor. Therefore, on the MacBook the communication is not *truly* parallel, since each thread has to wait for its turn.

This also explains why l quickly increases on the MacBook. On Huygens l roughly grows linearly with the amount of processors (except for the startup cost we get when moving from one to two processors). This effect is probably due to the locality when only using a relatively low amount of processors; for larger amounts of processors this linearity probably disappears.

Finally, we remark that the behaviour on the MacBook is quite erratic. This is probably because of the heavy machinery running in the background: it is impossible to stop all processes running in the background. One would also expect the step from one to two processors to be less extreme: after all, the MacBook is running on a dual core processor. However, the system probably mismanages the distribution over the two cores.

Next we change the puts used in the benchmark into gets. That is, we make the following modifications in bspbench.c:

- Add between line 65 and 66: `bsp_push_reg(src,MAXH*SZDBL);`
- Change line 143 into: `bsp_get(destproc[i],src,i*SZDBL,&dest[destindex[i]],SZDBL);`

We once again run the benchmark as described above, giving the results exhibited in the tables below.

p	g	l
1	86	779
2	92	2733
4	93	6421
8	88	12709
16	91	23406

Table 3: Benchmark on Huygens using gets

p	g	l
1	34	308
2	83	1917
4	107	25892
8	243	81154
16	1154	449414

Table 4: Benchmark on MacBook using gets

[1]However, the MacBook is crippled. It currently runs without battery since my battery showed heavy swelling. The system automatically downclocks while running without battery.

Roughly the same things can be said as above. However, we can also compare the behavioral difference between puts and gets. Observe that on both systems gets are more expensive than puts: we roughly get a 50% increase on Huygens and we also get a large increase on the MacBook. This is probably due to optimization: in the case of puts the traffic is known before the actual transfer starts and can thus be optimized beforehand; in the case of gets the sending processor does not yet know which data it has to send.

All benchmark results can be found in appendix A.

2 Sieve of Eratosthenes

2.1 Sequential algorithm

In this section we will take a look at the *sieve of Eratosthenes*. This is a method to generate all prime numbers up to a certain bound n. It works as follows: start with the integers from 2 to n. The number 2 is prime; cross out all larger multiples of 2. The smallest remaining number, 3, is a prime; cross out all lager multiples of 3. The smallest remaining number, 5, is a prime, etc.

Observe that we can stop crossing out multiples when the smallest remaining number is greater than $\sqrt{n}$. Since, assume that $k \leq n$ is not a prime number. Then k is the product of $r \geq 2$ primes, say $k = p_1 p_2 \cdots p_r$. But at least one of these primes is $\leq \sqrt{n}$, since otherwise $k > \sqrt{n}^p \geq \sqrt{n}^2 = n$, which contradicts $k \leq n$. Thus k is already crossed out before or when we reach $\sqrt{n}$ and we can thus stop after this point.

For exactly the same reason we have that for every prime number q we only need to start crossing out multiples at q^2.

We use this to write a sequential sieve program in C. This program can be found in appendix B. The input is an integer n, the output is an integer array numlist of length $n+1$ such that for all $0 \leq i \leq n$ we have numlist[i] $= 1$ if i is prime and numlist[i] $= 0$ if i is not prime (thus numlist can also be seen as a boolean array).

Next we analyze the cost of this sequential algorithm. Observe that, for every prime number $p \leq \sqrt{n}$ we cross out $\lfloor \frac{n}{p} \rfloor - p + 1$ numbers. From the prime number theorem we know that there are approximately $\lfloor \frac{\sqrt{n}}{\ln \sqrt{n}} \rfloor$ prime numbers $\leq \sqrt{n}$. Furthermore, if we write p_k for the k^{th} prime number, then from the same theorem we also know that $p_k \approx k \ln k$. Combining all this we get the following approximation for the number of operations:

$$\begin{aligned}
&\sum_{k=1}^{\lfloor \frac{\sqrt{n}}{\ln \sqrt{n}} \rfloor} \lfloor \frac{n}{p_k} \rfloor - p_k + 1 \approx \frac{n}{2} + \sum_{k=2}^{\lfloor \frac{2\sqrt{n}}{\ln n} \rfloor} \frac{n}{k \ln k} - k \ln k \approx \frac{n}{2} + \int_{x=2}^{\frac{2\sqrt{n}}{\ln n}} \frac{n}{x \ln x} - x \ln x \\
&= \frac{n}{2} + \left[n \ln(\ln x) - \frac{1}{4} x^2 (\ln(x) - 1) \right]_{x=2}^{\frac{2\sqrt{n}}{\ln n}} \\
&= n(\frac{1}{2} + \ln \ln \frac{2\sqrt{n}}{\ln n} - \ln \ln 2) - \frac{n}{(\ln n)^2} (\ln \frac{2\sqrt{n}}{\ln n} - 1) + \ln 2 - 1 \\
&\approx n(\frac{1}{2} + \ln \ln \frac{2\sqrt{n}}{\ln n} - \ln \ln 2) \\
&= n(\frac{1}{2} + \ln \frac{1}{2} \ln \frac{4n}{(\ln n)^2} - \ln \ln 2) \\
&= n(\frac{1}{2} + \ln \ln \frac{4n}{(\ln n)^2} - \ln 2 - \ln \ln 2) \\
&= n(\ln \ln \frac{4n}{(\ln n)^2} + \frac{1}{2} - \ln \ln 4)
\end{aligned}$$

We added an operation counter to the sequential sieve so we can compare our estimation with the actual number of operations. The results of this can be seen overhead.

n	Operations	Estimation
4	1	4
8	3	7
16	10	15
32	25	33
64	61	75
128	137	166
256	307	365
512	673	794
1024	1450	1705
2048	3086	3630
4096	6531	7668
8192	13735	16099
16384	28733	33622
32768	59771	69911
65536	123951	144832
131072	256223	299097
262144	528178	615998
524288	1086134	1265684
1048576	2228882	2595257

Table 5: Number of operations and our estimation for the sequential sieve.

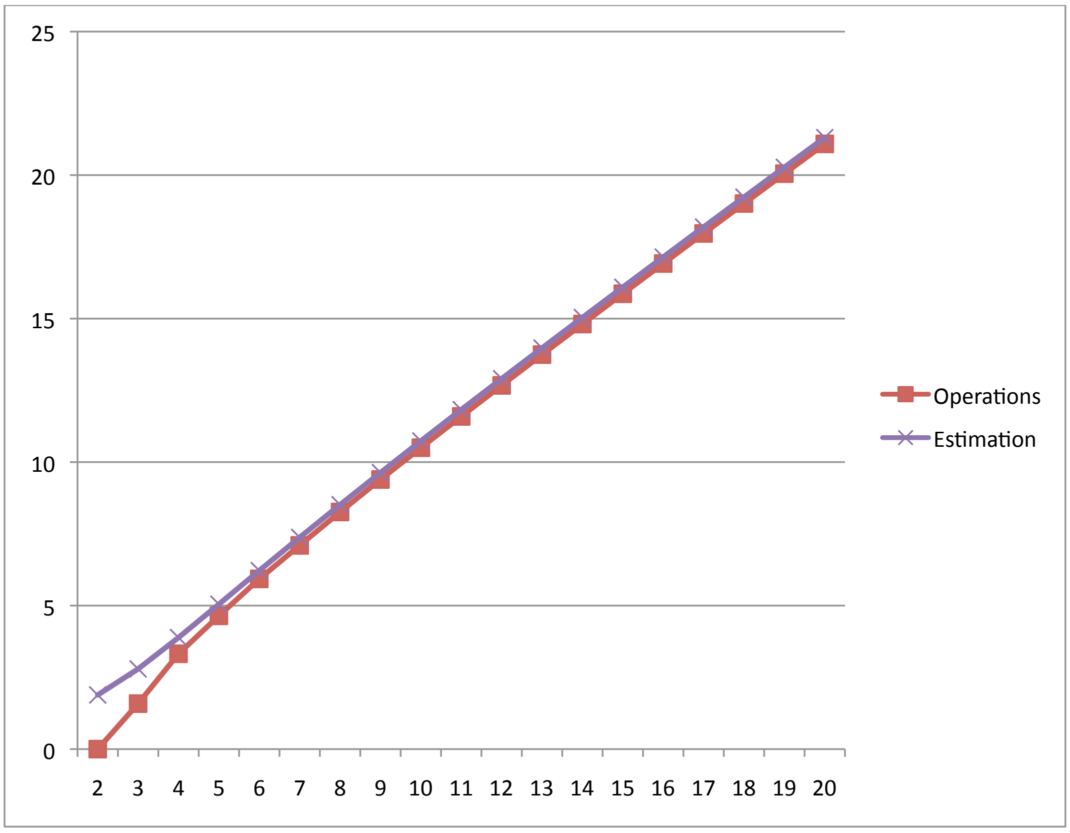

Figure 1: Graph of the number of operations and our estimation for the sequential sieve; both axes are Log_2.

As one can see, our estimation is not too far off (except for the initial estimations for very low n). Relative errors range around 16% and keep going down. The graph especially shows how good our estimation is when looking at orders of magnitude.

2.2 Parallel algorithm

We will now parallelize this algorithm. The first step is to analyze the best way to distribute the boolean array. We will distribute the array over the processor by blocks; that way, the crossing-out is distributed in an optimal fashion. Since, assume we are using the block distribution, let $2 \leq q \leq n$ and let n_i be the amount of multiples of q on processor i (for $0 \leq i < p$). If we take two fixed $0 \leq j, k < p$ then we see that there are at least $(n_i - 1)q + 1$ consecutive numbers on processor i. Thus, there are at least $(n_i - 1)q$ consecutive numbers on processor j, showing that $n_j \geq n_i - 1$. In the same way we find $n_i \geq n_j - 1$, thus $n_i + 1 \leq n_j \leq n_i - 1$. We thus see that for every two processors j, k we have $\mid n_i - n_j \mid \leq 1$.

But then every processor has at most $\lceil \frac{\lfloor \frac{n}{q} \rfloor}{p} \rceil$ multiples of q. Since assume that there is a processor which has at least $\lceil \frac{\lfloor \frac{n}{q} \rfloor}{p} \rceil + 1$ multiples of q. Then every other processor has, by the discussion above, at least $\lceil \frac{\lfloor \frac{n}{q} \rfloor}{p} \rceil$ multiples of q. But then the total amount of multiples of q is at least:

$$p\lceil \frac{\lfloor \frac{n}{q} \rfloor}{p} \rceil + 1 \geq p\frac{\lfloor \frac{n}{q} \rfloor}{p} + 1 = \lfloor \frac{n}{q} \rfloor + 1 > \lfloor \frac{n}{q} \rfloor$$

but the total amount of multiples of q is $\lfloor \frac{n}{q} \rfloor$, which leads to a contradiction. Thus, each processor has at most $\lceil \frac{\lfloor \frac{n}{q} \rfloor}{p} \rceil$ multiples of q.

However, by the pigeonhole principle we also see that there needs to be a processor p having at least $\lceil \frac{\lfloor \frac{n}{q} \rfloor}{p} \rceil$ multiples of q. We therefore see that the block distribution is optimal. So, we will use the block distribution.

As observed above, we know that we can stop when we reach $\sqrt{n}$. We can assume that $p \leq \sqrt{n}$; otherwise we are just far too rich and should use less processors. Under this assumption we see that $\sqrt{n} \leq \frac{n}{p}$. We can thus assume that we only need to look for new primes to cross out the multiples of on the first processor.

This gives us two possible methods to parallelize the sieve using the block distribution:

1. Find a new prime number on processor P(0), broadcast this number, cross out all multiples, find a new prime number on processor P(0), broadcast this number, cross out all multiples, etc.

2. First run the sequential sieve on processor P(0) up to $\sqrt{n}$, then broadcast all prime numbers found in one communication superstep and next cross out all multiples of all these prime numbers on the other processors.

To determine which method is better, we analyze the BSP costs of these two algorithms.

1. In this case we can derive the computation costs analogously to the sequential case and will find $\frac{n}{p}(\ln\ln \frac{4n}{(\ln n)^2} + \frac{1}{2} - \ln\ln 4) + \frac{2\sqrt{n}}{\ln n}l$. We also have to add the communication costs $\frac{2\sqrt{n}}{\ln n}(p-1)g + \frac{2\sqrt{n}}{\ln n}l$. This gives us total costs $\frac{n}{p}(\ln\ln \frac{4n}{(\ln n)^2} + \frac{1}{2} - \ln\ln 4) + \frac{2\sqrt{n}}{\ln n}(p-1)g + \frac{4\sqrt{n}}{\ln n}l$.

2. This gives us additional computation costs $\sqrt{n}(\ln\ln \frac{16\sqrt{n}}{(\ln n)^2} + \frac{1}{2} - \ln\ln 4) + l$ to perform the sequential sieve up to $\sqrt{n}$ (and saves us $(\frac{2\sqrt{n}}{\ln n} - 1)l$), but reduces the communication costs to $\frac{2\sqrt{n}}{\ln n}(p-1)g + l$. This gives a total cost of $\frac{n}{p}(\ln\ln \frac{4n}{(\ln n)^2} + \frac{1}{2} - \ln\ln 4) + \sqrt{n}(\ln\ln \frac{16\sqrt{n}}{(\ln n)^2} + \frac{1}{2} - \ln\ln 4) + \frac{2\sqrt{n}}{\ln n}(p-1)g + 3l$.

We thus see: the second algorithm is faster iff $\sqrt{n}(\ln\ln \frac{16\sqrt{n}}{(\ln n)^2} + \frac{1}{2} - \ln\ln 4) < (\frac{4\sqrt{n}}{\ln n} - 3)l$. We numerically evaluate this expression for a few values of n.

n	l
100	1.5
1000	2.1
10000	3.1
100000	4.3
1000000	5.8

Table 6: Minimal value of l for which algorithm 2 is faster.

Looking back to our benchmarks, it seems algorithm 2 is the better of the two. l is generally much larger than the values in the table above, showing that it is worthwhile to first locally sieve up to $\sqrt{n}$ on processor 0 before proceeding with the global sieve.

We have implemented this sieve as the function `bspsieve()`, the full code can be found in appendix C. We have timed the function in order to compare it with our BSP cost analysis above. We have iterated the algorithm 1000 times to obtain an accurate measurement. The results can be seen in the tables below and the graphs overhead.

(n,p)	1	2	4	8	16
1000	0.018	0.040	0.0	0.065	0.132
10000	0.109	0.071	0.068	0.104	0.198
100000	1.192	0.623	0.371	0.312	0.425
1000000	13.017	6.565	3.414	1.987	1.596

Table 7: Estimated execution time for 1000 iterations of the parallel sieve.

(n,p)	1	2	4	8	16
1000	0.034	0.027	0.03	0.048	0.105
10000	0.323	0.171	0.103	0.084	0.229
100000	3.294	1.671	0.849	0.459	0.524
1000000	34.15	16.884	8.431	4.228	2.213

Table 8: Measured execution time for 1000 iterations of the parallel sieve.

As one can see, the estimations is not that accurate. This is to be expected: our estimation is based on our theoretical model using the theoretical BSP model, while our actual implementation has a lot of little details which are not directly included in our estimation. However, when looking at orders of magnitude our estimation is fairly accurate, as can be seen from the graphs overhead.

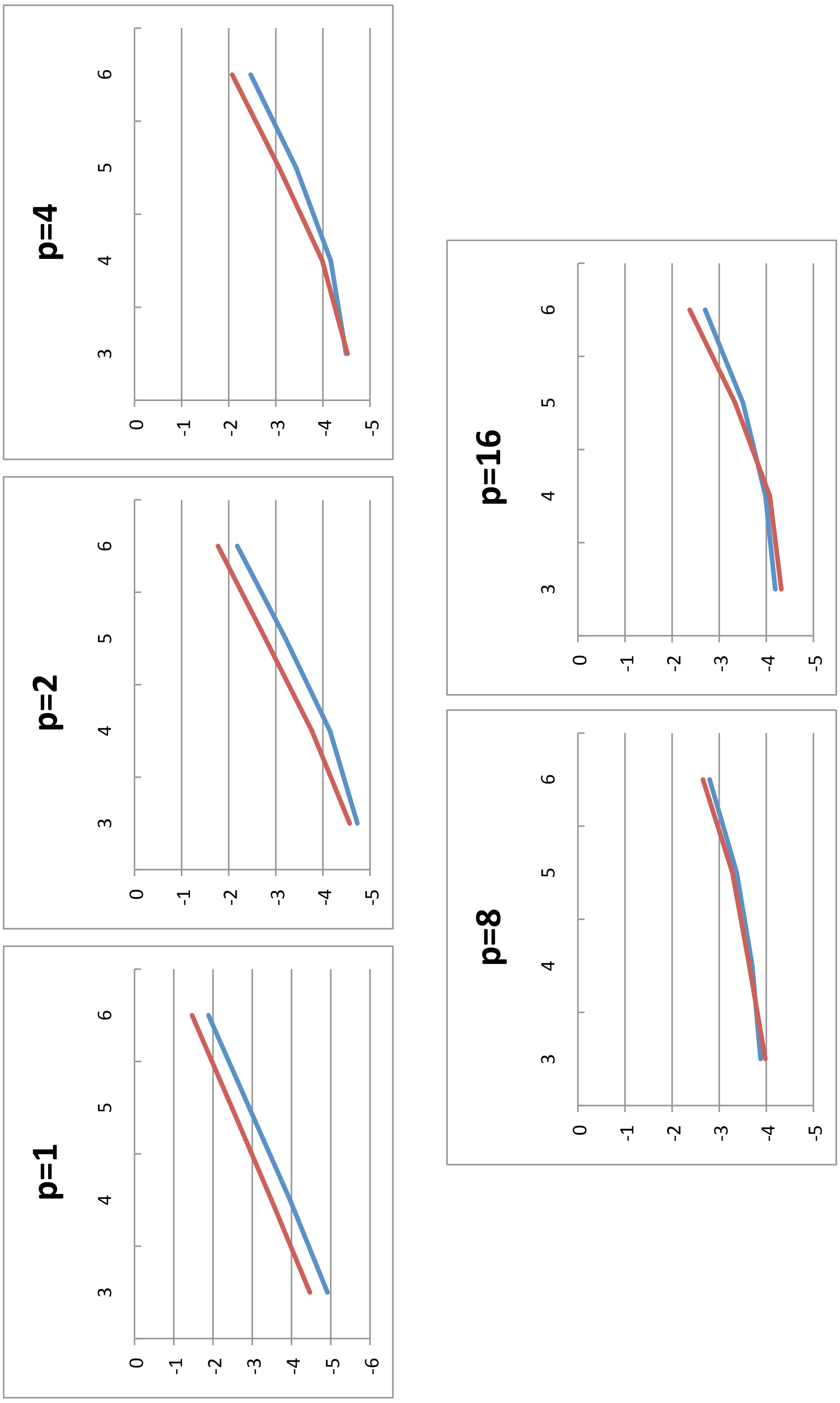

Figure 2: Graphs of the actual and approximated execution time for the parallel sieve; both axes are Log$_1$0. The approximation is blue, the measurement red.

2.3 Twin primes in parallel

We modified our parallel program to generate twin primes, that is, pairs of primes that differ by two. We determine all twin primes of the processor that has the largest of the two primes in its block. In order to do this we need to know if one of the last two numbers on the previous block is a prime number; we therefore communicate this information to the next processor.

The modifications are described in appendix D. As an illustration, we determined the twin pairs up to 100, giving the following result:

$$(3,5),(5,7),(11,13),(17,19),(29,31),(41,43),(59,61),(71,73)$$

2.4 Goldbach conjecture in parallel

Finally, we extended our program to test the Goldbach conjecture. We test if every even $2 < k \leq n$ is the sum of two primes. We will do this by computing all sums of prime numbers up to n. We will require p to be a square; that way we can split the prime numbers into $\sqrt{p}$ parts and distribute two-tuples of these $\sqrt{p}$ parts over our p processors using a Cartesian distribution.

Since we would need a lot of extra communication to determine how many primes each processor has, we keep it simple and redistribute the numbers up to n in $\sqrt{p}$ parts; we will use this as distribution for the rows and the columns. The easiest way to do this is to combine for each new part $\sqrt{p}$ parts of the original block distribution used in the parallel sieve. Since the number of prime numbers will decrease the further the block is, we do this in a cyclic way instead of through blocks.

Thus, if P_i is the set of prime numbers found on processor i, we get that processor s determines:

$$\bigcup\{P_i \mid 0 \leq i < p \mid \lfloor\frac{i}{\sqrt{p}}\rfloor = \lfloor\frac{s}{\sqrt{p}}\rfloor\} + \bigcup\{P_i \mid 0 \leq i < p \mid \lfloor\frac{i}{\sqrt{p}}\rfloor = s \mod \sqrt{p}\}$$

The sums up to n that arise in this way are sent back to the processor that originally owned this number in the block distribution used for the parallel sieve (taking care to avoid sending duplicates from the same processor). Then each processor checks the Goldbach conjecture for its own part. If it finds any counterexample, this is reported back to processor 0 which can thus determine if the Goldbach conjecture is true up to n or not.

We can make an easy optimization to this. Since + is commutative, we unnecessarily compute each sum twice. An easy way to save on this is to only do half the work on each processor by only using half of one of the prime arrays; that is, we let processor s determine:

$$\bigcup\{P_i \mid 0 \leq i < \lfloor\frac{p}{2}\rfloor \mid \lfloor\frac{i}{\sqrt{p}}\rfloor = \lfloor\frac{s}{\sqrt{p}}\rfloor\} + \bigcup\{P_i \mid 0 \leq i < p \mid \lfloor\frac{i}{\sqrt{p}}\rfloor = s \mod \sqrt{p}\}$$
$$\text{if } \lfloor\frac{s}{\sqrt{p}}\rfloor < s \mod \sqrt{p}$$
$$\bigcup\{P_i \mid 0 \leq i < p \mid \lfloor\frac{i}{\sqrt{p}}\rfloor = \lfloor\frac{s}{\sqrt{p}}\rfloor\} + \bigcup\{P_i \mid \lfloor\frac{p}{2}\rfloor \leq i < p \mid \lfloor\frac{i}{\sqrt{p}}\rfloor = s \mod \sqrt{p}\}$$
$$\text{if } \lfloor\frac{s}{\sqrt{p}}\rfloor > s \mod \sqrt{p}$$
$$\{x + y \mid x, y \in \bigcup\{P_i \mid 0 \leq i < p\rfloor \mid \lfloor\frac{i}{\sqrt{p}}\rfloor = \lfloor\frac{s}{\sqrt{p}}\rfloor\} \mid y \geq x\}$$
$$\text{if } \lfloor\frac{s}{\sqrt{p}}\rfloor = s \mod \sqrt{p}$$

The modifications to `bspsieve()` are described in appendix E. The program was used to test the Goldbach conjecture up to 1000000, for which it found no counterexample.

A Benchmark results

A.1 Huygens using puts

```
p= 1, r= 195.693 Mflop/s, g= 53.6, l= 544.7
p= 1, r= 195.804 Mflop/s, g= 53.3, l= 562.8
p= 1, r= 195.604 Mflop/s, g= 53.0, l= 571.5
p= 1, r= 194.673 Mflop/s, g= 52.6, l= 587.0
p= 1, r= 195.492 Mflop/s, g= 53.7, l= 507.0

p= 2, r= 146.605 Mflop/s, g= 36.0, l= 2881.5
p= 2, r= 193.738 Mflop/s, g= 56.1, l= 2181.9
p= 2, r= 194.908 Mflop/s, g= 56.6, l= 2035.6
p= 2, r= 194.361 Mflop/s, g= 56.7, l= 1987.1
p= 2, r= 193.503 Mflop/s, g= 56.0, l= 2153.3

p= 4, r= 195.432 Mflop/s, g= 57.1, l= 3739.1
p= 4, r= 195.504 Mflop/s, g= 56.5, l= 4201.0
p= 4, r= 195.415 Mflop/s, g= 58.0, l= 3885.4
p= 4, r= 194.918 Mflop/s, g= 57.5, l= 4247.2
p= 4, r= 195.654 Mflop/s, g= 58.9, l= 4002.2

p= 8, r= 194.922 Mflop/s, g= 57.5, l= 7939.4
p= 8, r= 195.341 Mflop/s, g= 58.2, l= 8215.0
p= 8, r= 193.878 Mflop/s, g= 60.9, l= 8451.7
p= 8, r= 195.593 Mflop/s, g= 57.3, l= 8019.9
p= 8, r= 194.938 Mflop/s, g= 56.2, l= 8697.9

p= 16, r= 194.172 Mflop/s, g= 63.8, l= 17999.0
p= 16, r= 195.010 Mflop/s, g= 58.3, l= 16667.1
p= 16, r= 195.595 Mflop/s, g= 58.4, l= 16632.1
p= 16, r= 195.252 Mflop/s, g= 56.9, l= 15870.1
p= 16, r= 194.128 Mflop/s, g= 60.6, l= 16410.7
```

A.2 MacBook using puts

```
p= 1, r= 284.247 Mflop/s, g= 24.2, l= 253.9
p= 1, r= 281.914 Mflop/s, g= 24.3, l= 220.0
p= 1, r= 114.350 Mflop/s, g= 9.6, l= 125.6
p= 1, r= 284.247 Mflop/s, g= 24.6, l= 202.1
p= 1, r= 281.499 Mflop/s, g= 24.4, l= 210.2

p= 2, r= 290.496 Mflop/s, g= 39.2, l= 1828.9
p= 2, r= 347.244 Mflop/s, g= 48.4, l= 1835.5
p= 2, r= 345.230 Mflop/s, g= 53.1, l= 1763.0
p= 2, r= 346.928 Mflop/s, g= 46.1, l= 2117.6
p= 2, r= 306.225 Mflop/s, g= 42.4, l= 1595.7

p= 4, r= 342.537 Mflop/s, g= 78.2, l= 17369.7
p= 4, r= 346.232 Mflop/s, g= 133.9, l= 19406.1
p= 4, r= 335.339 Mflop/s, g= 84.8, l= 16765.0
p= 4, r= 258.828 Mflop/s, g= 72.3, l= 12474.4
p= 4, r= 332.216 Mflop/s, g= 81.6, l= 16886.2
```

```
p= 8, r= 318.536 Mflop/s, g= 127.9, l= 56139.7
p= 8, r= 316.994 Mflop/s, g= 141.3, l= 54325.4
p= 8, r= 343.427 Mflop/s, g= 150.4, l= 60741.8
p= 8, r= 347.454 Mflop/s, g= 125.1, l= 64220.5
p= 8, r= 337.045 Mflop/s, g= 138.5, l= 61536.9

p= 16, r= 325.197 Mflop/s, g= 702.9, l= 438862.9
p= 16, r= 322.442 Mflop/s, g= 497.9, l= 358315.3
p= 16, r= 322.442 Mflop/s, g= 649.8, l= 327113.9
p= 16, r= 298.776 Mflop/s, g= 657.7, l= 296126.1
p= 16, r= 323.057 Mflop/s, g= 638.8, l= 334078.0
```

A.3 Huygens using gets

```
p= 1, r= 195.715 Mflop/s, g= 85.7, l= 783.5
p= 1, r= 195.604 Mflop/s, g= 85.8, l= 772.9
p= 1, r= 195.693 Mflop/s, g= 85.7, l= 780.0
p= 1, r= 195.693 Mflop/s, g= 85.9, l= 773.1
p= 1, r= 195.715 Mflop/s, g= 85.7, l= 786.5

p= 2, r= 195.648 Mflop/s, g= 92.2, l= 2880.9
p= 2, r= 194.919 Mflop/s, g= 92.9, l= 2796.6
p= 2, r= 194.685 Mflop/s, g= 89.5, l= 2281.6
p= 2, r= 195.604 Mflop/s, g= 92.7, l= 2857.2
p= 2, r= 195.559 Mflop/s, g= 92.3, l= 2848.9

p= 4, r= 195.151 Mflop/s, g= 91.7, l= 6775.2
p= 4, r= 195.626 Mflop/s, g= 92.7, l= 5871.0
p= 4, r= 195.626 Mflop/s, g= 93.3, l= 6537.0
p= 4, r= 189.658 Mflop/s, g= 81.3, l= 7678.6
p= 4, r= 195.237 Mflop/s, g= 93.4, l= 6499.5

p= 8, r= 188.356 Mflop/s, g= 82.4, l= 12610.0
p= 8, r= 192.754 Mflop/s, g= 80.4, l= 16044.4
p= 8, r= 186.076 Mflop/s, g= 91.3, l= 12334.3
p= 8, r= 195.648 Mflop/s, g= 92.7, l= 13224.4
p= 8, r= 195.346 Mflop/s, g= 92.9, l= 12666.9

p= 16, r= 194.219 Mflop/s, g= 82.4, l= 29550.8
p= 16, r= 195.339 Mflop/s, g= 97.2, l= 27329.8
p= 16, r= 194.127 Mflop/s, g= 96.4, l= 27003.6
p= 16, r= 194.221 Mflop/s, g= 83.6, l= 28618.0
p= 16, r= 195.293 Mflop/s, g= 96.1, l= 26935.7
```

A.4 MacBook using gets

```
p= 1, r= 306.838 Mflop/s, g= 36.8, l= 301.4
p= 1, r= 284.671 Mflop/s, g= 34.2, l= 281.6
p= 1, r= 284.623 Mflop/s, g= 33.3, l= 349.1
p= 1, r= 284.623 Mflop/s, g= 34.1, l= 316.5
p= 1, r= 284.623 Mflop/s, g= 33.9, l= 292.6

p= 2, r= 252.453 Mflop/s, g= 67.4, l= 892.1
p= 2, r= 348.017 Mflop/s, g= 83.2, l= 2210.2
```

```
p= 2, r= 293.668 Mflop/s, g= 74.1, l= 1373.6
p= 2, r= 347.982 Mflop/s, g= 83.8, l= 2398.1
p= 2, r= 348.299 Mflop/s, g= 91.6, l= 1684.9

p= 4, r= 305.629 Mflop/s, g= 97.4, l= 24551.1
p= 4, r= 321.556 Mflop/s, g= 98.1, l= 26505.6
p= 4, r= 347.719 Mflop/s, g= 122.8, l= 26421.8
p= 4, r= 336.988 Mflop/s, g= 107.6, l= 26400.7
p= 4, r= 322.223 Mflop/s, g= 106.9, l= 25578.7

p= 8, r= 315.736 Mflop/s, g= 240.1, l= 76402.3
p= 8, r= 341.649 Mflop/s, g= 244.9, l= 91816.9
p= 8, r= 321.588 Mflop/s, g= 236.6, l= 79591.4
p= 8, r= 334.666 Mflop/s, g= 247.8, l= 82430.0
p= 8, r= 344.629 Mflop/s, g= 170.3, l= 86192.9

p= 16, r= 326.547 Mflop/s, g= 1186.1, l= 439352.2
p= 16, r= 319.634 Mflop/s, g= 1066.1, l= 450887.0
p= 16, r= 341.282 Mflop/s, g= 1196.7, l= 469632.4
p= 16, r= 325.507 Mflop/s, g= 1182.3, l= 441010.8
p= 16, r= 323.032 Mflop/s, g= 1140.5, l= 446187.3
```

B Sequential sieve in C

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

//Perform the sieve of Eratosthenes (sequentially).
//Input:  integer n, integer array numlist of length n+1 initialized to
//        all ones.
//Output: numlist is modified such that numlist[i]==1 if i is prime
//        and numlist[i]==0 if i is not prime.
void seqsieve (int n, int *numlist) {
    int *numlist,sqrt_n,i,j,ops;

    //Perform the sieve, also counting the number of operations/cross-outs.
    sqrt_n = (int) sqrt(n);
    ops = 0;
    for (i=0;i<=sqrt_n;i++)
        if (numlist[i])
            for (j=i*i;j <= n; j += i)
            {
                numlist[j] = 0;
                ops++;
            }

    printf("Operations: %d\n",ops);
}

int main (int argc, const char * argv[]) {
    int *numlist,n;
    printf("What is n?\n"); fflush(stdout);
    scanf("%d", &n);

    if (n < 1) {
        printf("Error: n needs to be a non-negative integer.\n");
        return 1;
    }

    //Initialize the integer array.
    numlist = malloc((n+1)*sizeof(int));
    for (i=0;i<=n;i++)
        numlist[i]=1;
    numlist[0]=0;
    numlist[1]=0;

    seqsieve(n);
    return 0;
}
```

C Parallel sieve in C

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

int P;

//Block distribution: determine the size of the block on processor s.
//for an array of length n.
//Input:  integer b for the block size, integer s for the processor number,
//        integer n for the array length.
//Output: the size of the block on processor s.
int block_size (int b, int s, int n) {
    if ((s+1)*b <= n)
        return b;
    else if (n > s*b)
        return n - s*b;
    else
        return 0;
}

//Block distribution: determine the global index of the local index 0.
//Input:  integer b for the block size, integer s for the processor number,
//        integer n for the array length.
//Output: the global index of the local index 0; we return -1 if the block has
//        length 0.
int block_init (int b, int s, int n) {
    if (s*b < n)
        return s*b;
    else
        return -1;
}
```

```
//Perform the sieve of Eratosthenes in parallel, using a block distribution.
//At the end of the function, the array numlist is distributed over the
//processors in a block distribution. We have, for global indices i, that at
//the end numlist[i]==1 if i is prime and numlist[i]==0 if i is not prime.
//We first run a sequential sieve on processor 0 up to sqrt(n), broadcast the
//results and then perform the crossouts in parallel.
//Input: none
//Output: none
void bspsieve()
{
    int *numlist,n,i,j,p,s,b,n_loc,n_init,offset,sqrt_n;
    int *prime_list,prime_count,cur_prime_sq;

    bsp_begin(P);
    p = bsp_nprocs();
    s = bsp_pid();

    if (s == 0) {
        printf("What is n?\n");
        scanf("%d", &n);
        if (n < 1)
            bsp_abort("n needs to be a positive integer.\n");
        if (n < p*p)
            bsp_abort("n should be at least the square  of p.\n");
    }

    //Register and broadcast n
    bsp_push_reg(&n,sizeof(int));
    bsp_sync();
    bsp_get(0,&n,0,&n,sizeof(int));
    bsp_sync();

    //Determine the block size b
    if ((n+1) % p == 0)
        b = (n+1) / p;
    else
        b = (n+1) / p + 1;

    //Determine the local block size and the global index of the local index 0
    n_loc = block_size(b,s,n+1);
    n_init = block_init(b,s,n+1);

    //Prepare the list that will contain the prime numbers <= sqrt(n), i.e. the
    //result of the first sequential sieve. Observe that there are less than
    //sqrt(n) prime numbers <= sqrt(n), so we will use this as length of
    //the array.
    sqrt_n = (int) sqrt(n);
    prime_list = malloc(sqrt_n*sizeof(int));
    bsp_push_reg(&prime_count,sizeof(int));
    bsp_push_reg(prime_list,sqrt_n*sizeof(int));
    bsp_sync();

    //Initialize numlist.
    numlist = malloc(n_loc*sizeof(int));
    for (i=0;i<n_loc;i++)
        numlist[i]=1;

    //Perform the first sequential sieve up to sqrt(n) and broadcast the
    //results.
    if (s == 0) {
        seqsieve((int) sqrt(n),numlist);

        //Put the prime numbers up to sqrt(n) in a list
        prime_count = 0;
        for (i=0;i <= sqrt_n; i++)
            if (numlist[i]) {
                prime_list[prime_count] = i;
                prime_count++;
            }

        //Put the results in all processors.
        for (i=0; i < p; i++) {
            bsp_put(i,&prime_count,&prime_count,0,sizeof(int));
            bsp_put(i,prime_list,prime_list,0,prime_count*sizeof(int));
        }
    }
    bsp_sync();

    //Perform the parallel sieve using the result from the first sequential
    //sieve.
    for (i=0; i < prime_count; i++) {
        //We start crossing out at the square of the prime number.
        cur_prime_sq = prime_list[i]*prime_list[i];
        //The square of the current prime is before our initial index.
        if (cur_prime_sq < n_init) {
            //Calculate the local index of the first multiple of the current
            //prime in our block
            offset = n_init % prime_list[i];
            if (offset == 0)
                j = 0;
            else
                j = prime_list[i] - offset;
        }
        //The square of the current prime is in or beyond our block.
        else
            j = cur_prime_sq - n_init;

        //Perform the cross-outs
        for (;j < n_loc; j += prime_list[i])
            numlist[j] = 0;
    }

    bsp_end();
}

int main (int argc, char * argv[]) {
    bsp_init(bspsieve, argc, argv);
    /* Sequential part */
    printf("How many processors?\n");
    scanf("%d",&P);
    if (P > bsp_nprocs()) {
        printf("Sorry, not enough available.\n");
        exit(1);
    }

    /* Parallel part */
    bspsieve();

    /* Sequential part */
    exit(0);
}
```

D Twin primes in C

```
//Modifications to bspsieve() to determine twin primes.
//We determine the twin pairs on the processor containing the largest of
//the two primes. At the end the array twin_primes will contain the smallest
//of each twin prime pair.

//Add at the start of bspsieve():
    int last_prime,*twin_primes,twin_primes_count;

//Add before bsp_end():
    //Determine if one of our two last indices is a prime number and broadcast
    //this.
    //last_prime=-1 if the last local index represents a prime
    //last_prime=-2 if the second to last local index represents a prime
    //last_prime= 0 otherwise
    if (s != p-1) {
        if (numlist[n_loc-1])
            last_prime=-1;
        else if (numlist[n_loc-2])
            last_prime=-2;
        else
            last_prime=0;

        bsp_put(s+1,&last_prime,&last_prime,0,sizeof(int));
    }
    bsp_sync();


    //An array of length n_loc will certainly be long enough to contain the
    //first element of the local twin pairs (where 'local' means the last of
    //the pair is local).
    twin_primes = malloc(n_loc*sizeof(int));
    twin_primes_count=0;

    //Determine all twin primes.
    //The border cases
    if (s == 0)
        last_prime = 0;

    if (last_prime == -2 && numlist[0]) {
        twin_primes[twin_primes_count]=n_init-2;
        twin_primes_count++;
    }
    else if (last_prime == -1 && numlist[1]) {
        twin_primes[twin_primes_count]=n_init-1;
        twin_primes_count++;
    }

    //Determine the twin primes for which both primes are local
    for (i=0;i<n_loc-2;i++)
        if (numlist[i] && numlist[i+2]) {
            twin_primes[twin_primes_count]=n_init+i;
            twin_primes_count++;
        }
```

E Goldbach conjecture in C

```
//Extension to bspsieve() to check the Goldbach conjecture up to n. That is,
//check that every even number 2 < k <= n is the sum of two primes.

//Add at the start of bspsieve()
    int *full_numlist,sqrt_p,*row_primes,*column_primes,prow,pcolumn;
    int b2,sum,counterex,endpoint;

//Replace line 76-89 with:
    //Prepare the list that will contain the prime numbers on our own processor.
    //Observe that there are less than b local prime numbers, so we will use
    //this as length of the array.
    sqrt_n = (int) sqrt(n);
    prime_list = malloc(b*sizeof(int));
    bsp_push_reg(&prime_count,sizeof(int));
    bsp_push_reg(prime_list,b*sizeof(int));
    //Prepare the arrays that will contain the prime numbers in our row and
    //column to check the Goldbach conjecture.
    sqrt_p = (int) sqrt(p);
    row_primes = malloc(sqrt_p*b*sizeof(int));
    column_primes = malloc(sqrt_p*b*sizeof(int));
    for (i=0;i<sqrt_p*b;i++) {
        row_primes[i]=0;
        column_primes[i]=0;
    }
    bsp_push_reg(row_primes,sqrt_p*b*sizeof(int));
    bsp_push_reg(column_primes,sqrt_p*b*sizeof(int));

    //Initialize numlist.
    full_numlist = malloc((n+1)*sizeof(int));
    numlist = &full_numlist[n_init];
    bsp_push_reg(numlist,n_loc*sizeof(int));
    for (i=0;i<n_loc;i++)
    numlist[i]=1;
    bsp_sync();


//Add at the end of bspsieve():
    //We only need the prime number 2 to write 4 as the sum of two primes;
    //every other sum of 2 plus a different prime is odd. However, 2 plus a
    //different prime could again be a prime (which cannot happen for odd
    //primes, since their sum is even). To avoid improper behaviour we declare 2
    //to be non-prime for now.
    if (s == 0)
    numlist[2]=0;

    //Determine our processor row and column
    prow = s/sqrt_p;
    pcolumn = s%sqrt_p;

    //Determine the prime numbers on our processor
    prime_count=0;
    for (i=0;i<n_loc;i++)
    if (numlist[i]) {
        prime_list[prime_count]=n_init + i;
        prime_count++;
    }

    //Broadcast the prime numbers to the correct processors.
    //Broadcast to the correct row
    for (i=prow*sqrt_p;i < (prow+1)*sqrt_p; i++)
    bsp_put(i,prime_list,row_primes,pcolumn*b*sizeof(int),
        prime_count*sizeof(int));

    //Broadcast to the correct column
    //It might seem off that we start at s/sqrt_p; this is because we combine
    //sqrt_p parts of the original block distribution into one new part.
    for (i=s/sqrt_p;i < p;i += sqrt_p)
    bsp_put(i,prime_list,column_primes,pcolumn*b*sizeof(int),
        prime_count*sizeof(int));
    bsp_sync();

    //Add our primes
    //Since summation is commutative we only use half of one of the lists if we
    //are not on the diagonal (i.e. prow != pcolumn), and on the diagonal we
    //make sure not to add primes twice.
    b2 = b*sqrt_p;
    if (prow < pcolumn)
    endpoint = b2/2;
    else
    endpoint = b2;

    for (i=0;i<b2;i++)
    if (row_primes[i] != 0) {
        if (pcolumn < prow)
            j = b2/2;
            else if (pcolumn == prow)
                j = i;
                else
                    j=0;

                    for (j=0;j<b2;j++)
                        if (column_primes[j] != 0) {
                            sum = row_primes[i]+column_primes[j];
                            if (sum <= n)
                                full_numlist[sum]=2;
                        }
    }

    //Put the result back in the original block distribution used for the sieve.
    for (i=0;i<=n;i++)
    if (full_numlist[i] == 2)
    bsp_put(i/b,&full_numlist[i],numlist,(i%b)*sizeof(int),sizeof(int));

    bsp_push_reg(&counterex,sizeof(int));
    bsp_sync();

    //Determine the index to start checking from. Recall that we marked 2 as
    //non-prime, therefore 4=2+2 is an exception and we start at 6.
    if (n_init < 6)
    i = 6;
    else if (n_init % 2 == 0)
    i = 0;
    else
    i = 1;

    //Check if every even number >=6 was marked
    for (;i<n_loc && numlist[i]==2;i += 2) {}

    counterex = 0;
    if (i < n_loc) {
        counterex = n_init + i;
        bsp_put(0,&counterex,&counterex,0,sizeof(int));
    }
    bsp_sync();

    if (s == 0) {
        if (counterex)
            printf("Goldbach Conjecture is false, counterexample: %d\n",
                counterex);
            else
                printf("Goldbach Conjecture verified up to %d\n",n);
                }


//Add to main(), after bsp_init(..):
    int sqrt_P;

//Add to main, before bspsieve(); :
    sqrt_P = (int) sqrt(P);
    if (sqrt_P*sqrt_P != P) {
        printf("The number of processors should be a square.\n");
        exit(1);
    }
```