Distributed prime sieve in heterogeneous computer clusters

Carlos Costa

Faculdade de Engenharia da Universidade do Porto
Rua Dr. Roberto Frias, s/n 4200-465 Porto, Portugal

carlos.costa@fe.up.pt / carloscosta.cmcc@gmail.com

Abstract. Prime numbers play a pivotal role in current encryption algorithms and given the rise of cloud computing, the need for larger primes never been so high. This increase in available computation power can be used to either try to break the encryption or to strength it by finding larger primes. With this in mind, this paper provides an analysis of different sieves implementations that can be used to generate primes up to 2^64. It starts by analyzing cache friendly sequential sieves with wheel factorization, then expands to multicore architectures and ends with a cache friendly segmented hybrid implementation of a distributed prime sieve, designed to efficiently use all the available computation resources of heterogeneous computer clusters with variable workload and to scale very well to any cluster size.

Keywords: distributed prime sieve, sieve of Eratosthenes, wheel factorization, distributed and parallel computing, OpenMP, OpenMPI, C++

1 Introduction

Prime numbers have been a topic of wide research given their important role in securing online transactions using asymmetric public key algorithms such as RSA [2]. In this context prime numbers are used to create the public key that is used by the sender to encrypt the message contents. The reason to use a product of two prime numbers to create the public key is because in the current computer architecture, it is computation infeasible to factor large prime numbers, and thus find the original primes used to create the public key, and used then to decrypt the message (with the progress made in quantum computers, this may not hold for very long). Besides encryption, prime numbers can also be used in hash functions to reduce the risk of collision and in pseudorandom number generators.

Although nowadays is more common to use primality test algorithms [3] to find large primes, sieves have been a known method to generate primes up to a given number. One of the most efficient prime number sieve was discovered by Eratosthenes [4] in the ancient Greece and can generate prime numbers up to a given number n with O(n log log n) operations. Other prime sieves were discovered since then, such as the sieve of Atkin [5] or the sieve of Sundaram [6], but a modified sieve of Eratosthenes is still considered to be the most efficient algorithm to use in the current computer architecture, and thus it was the one chosen to be used. This algorithm was implemented with 20 different variations, ranging from sequential form, passing to parallel on multicore architectures and ending in distributed computer clusters. Each of these 3 main implementations have several algorithm variations, to determine which strategy was more suitable for each usage. As such, it was developed algorithms variations that focused on using the minimum amount of memory, others focused on computing time, and others made a tradeoff between the two. Finally it was implemented segmented versions of both parallel and distributed algorithms to allow the computation of primes to the maximum number represented by the current computer architecture (2^64). Special attention was devoted to the implementation of the distributed algorithm version since is the one more suitable to be used to calculate primes up to 2^64 in reasonable time, because it was designed to scale very well to large clusters and was implemented to perform dynamic allocation of segments to nodes in order to efficiently use all the computation capacity of clusters with heterogeneous hardware and with variable workload.

On the next sections it will be presented each algorithm variation and their possible application, and it will be discussed the implementation results along with the speedup, efficiency and scalability analysis of the 3 main algorithms implementations.

2 Related work

In the course of the implementation of the several algorithm variations, it was performed a state of the art search to identify possible design improvements and to determined which implementation should be optimized in order to contributed to the already existing implementations publicly available.

From all the publicly implementations analyzed, special interest was devoted to the Prime Sieve [7] developed by Kim Walisch, since it is considered to be the fastest multicore implementation publicly available at the present date.

Other papers that influenced the strategies developed included [8] that details a very efficient use of the cache memory system for very large prime numbers, [9] [10] that explains how to use wheel factorization to considerably speed up the sieving process and [11] [12] that provide insights on how to implement the simple MPI version.

From this search it was determined that at the present date, a distributed implementation optimized for heterogeneous clusters could be of public interest and as such, was the implementation that was devoted most of work in development.

2.1 Sieve of Eratosthenes

The sieve of Eratosthenes was discovered in ancient Greece by Eratosthenes around the 3rd century B.C., and describes a method for calculating primes up to a given number n in O(n log log n) operations.

The algorithm can be described with the following pseudo code [4]:

2.1.1 Sieve of Eratosthenes algorithm

```
Input: an integer n > 1
Let P be an array of Boolean values, indexed by integers 2 to n, initially all set to
true.

 for i = 2, 3, 4, ..., not exceeding √n:
  if P[i] is true:
    for j = 2i, 3i, 4i, ..., not exceeding n :
      P[j] = false

Now all i such that P[i] is true are prime numbers
```

2.2 Wheel factorization

Wheel factorization [13] is a known optimization used to cross off multiples of several sieving primes. It can be used to considerable speed up the sieving process if used efficiently.

A modulo 210 wheel which has sieving primes 2, 3, 5, 7, can cross off as much as 77% of composites. Larger wheels may not have a reasonable improvement in % of composites crossed off, given the size of the table required [14].

The creation of the wheel sieve table can be described as follows:

2.2.1 Wheel factorization pseudo code for table creation

```
Input: a P list of known prime numbers
Let m be the product of the known prime numbers in the P list
Let C be an array of Boolean values with numbers in [1, m], initially all set to false.

foreach i in P
  for j = 2i, 3i, 4i, … not exceeding m
    C[j] = true

Now for all x numbers such as k=(x mod m) in [1, m], if C[k] is false, then x is a prob-
able prime number, and needs to be checked by other means to confirm if it is a prime
number or not. If C[k] is true, then x is a composite number of one of the primes in
list P
```

3 Implementation

In the following sections it will be presented the main concepts behind each prime sieve algorithm implemented. The order used was to denote the development of each variation in relation to the previous ones and to facilitate the identification of the optimizations that were introduced in each development iteration. In the simple algorithms it will be provided pseudo code to facilitate the explanation of the algorithm but for the more optimized versions, given the length of the algorithms it will only be explained the main concepts. As such, it is recommended to view the C++ implementation source code available in the git repository in [1].

3.1 Sequential prime sieve using trial division

This is the simplest algorithm of a prime number sieve and serves only to demonstrate that a naïve implementation, although correct, can be very inefficient.

The main idea is to cross off all odd j numbers that are composite of a previous prime number i. This can be achieved by checking if a j number is multiple of a previous i number by using the modulus operator. If the remainder of the integer division between j and i is 0, then j is multiple of i and is marked as composite.

One optimization applied that improves upon the original Eratosthenes algorithm is that the sieving can start at i^2 instead of 2i in order to avoid marking a composite number multiple times. For example 21 is multiple of 3 and 7, so there is no need to mark it as composite when sieving with prime 7 because it was already sieved with the prime 3. Also for the same reason, it is only necessary to check i until $\sqrt{n}$, because when $i > \sqrt{n}$, then j will exceed n every time and it will be useless to execute that portion of the code because it will never enter the if section.

One other optimization is to completely exclude even numbers, and this way reduce the computations and memory to half. As a result, the memory access has to be mapped from number to respective position and the increment to get the next number to check is 2 instead of 1.

3.1.1 Sequential prime sieve algorithm using trial division

```
Input: an integer n > 1
Let C be an array of Boolean values representing the odd numbers > 1, initially all set
to false
for i = 3, 5, 7, …, not exceeding √n
  if C[(i - 3) / 2] is false
    for j = i², i² + 2, i² + 4, i² + 6, …, not exceeding n
      if (j % i == 0) then C[(j - 3) / 2] = true
```

3.2 Sequential prime sieve using fast marking

Algorithm 3.1 is very inefficient since uses modulus operations to check for composites, and this is very computation intensive, especially if the maximum range to sieve is a large number, because it takes considerable more time to do a division in current hardware architectures than to do an addition. With this insight in mind, the previous algorithm was modified to use only additions.

The main idea now is to cross off all j numbers that we already know that are composites, because $i^2 + k*i$ is guaranteed to be composite. This way there is no need to use the modulus operator.

One other optimization in relation to the original sieve of Eratosthenes is that the increment to mark the next composite can be 2*i since adding i to an odd multiple of i will result in an even number, and even numbers are not prime, so they can be skipped.

3.1.1 Sequential prime sieve algorithm using fast marking

```
Input: an integer n > 1
Let C be an array of Boolean values representing the odd numbers > 1, initially all set
to false
for i = 3, 5, 7, …, not exceeding √n
  if C[(i - 3) / 2] is false
    for j = i², i² + 2i, i² + 4i, i² + 6i, …, not exceeding n
        C[(j - 3) / 2] = true
```

3.3 Sequential prime sieve using fast marking with block decomposition optimized for space

Although algorithm 3.2 is considerable faster than algorithm 3.1, it was suffering performance degradation by not using the cache memory system efficiently. This was happening because the same areas of array C were being loaded to cache memory several times, since the algorithm was sieving from i to the end of array C every time. This is extremely inefficiently, because the last elements of array C were being loaded to cache too many times, and since each cache miss forces the cpu to wait hundreds of clock cycles to load the values from main memory, this was affecting performance considerably.

In order to solve this problem, it was implemented a sieve with block decomposition, in order to load the values of the array C only one time to the cache, and then sieve all primes numbers up to √n in that block, before moving to the next.

3.1.1 Sequential prime sieve algorithm using fast marking and block decomposition optimized for space

```
Input: an integer n > 1
Let C be an array of Boolean values representing the odd numbers > 1, initially all set
to false
Let bs be the block size in number of elements
Let nb = n / bs be the number of blocks to use in sieving

calculatePrimesInBlock(3, 3 + bs)

for b = 1, 2, …, not exceeding nb
  a = b * bs
  b = a + bs, at most n + 1, because block is sieved in [a, b[
  removePrimesFromPreviousBlocks(a, b)
  calculatePrimesInBlock(a, b)

define calculatePrimesInBlock(a, b)
  for i = a, a + 2, a + 4, …, not exceeding √n
    if C[i] is false
      for j = $i^2$, $i^2$ + 2i, $i^2$ + 4i, $i^2$ + 6i, …, less than b
          C[j] = true

define removePrimesFromPreviousBlocks(a, b)
  for i = 0, 1, 2, …, not exceeding the position in C associated with √n
    if C[i] is false
      p = closest prime multiple of number associated with position i in relation to a
      for j = p, p + 2p, p + 4p, …, less than b
        C[j] = true
```

3.4 Sequential prime sieve using fast marking with block decomposition optimized for time

Algorithm 3.1 has a section of code that is constantly repeating computations. That section is the calculation of the closest prime number in relation to the beginning of the block. In order to prevent this repetition, this version keeps in memory the last prime multiple associated with each prime. And since it is only necessary to store the prime multiples associated to primes up to √n, this is a very reasonable tradeoff between space and computation time.

This was also the first variation in which was implemented a segmented sieve, in order to try to save some memory by keeping the bitset of bools only for the current block. But since all the primes found were kept in memory instead of being compacted in the bitset, it ended up consuming much more memory because each primes has 64 bits of storage.

3.5 Sequential prime sieve using fast marking with block decomposition optimized for space and time

This variation was implemented to reduce even more the repetition of computation that occur in the previous algorithm and to revert to the previous memory scheme.

The improvement is to avoid the repetition of the computation of the double of a prime to use as an increment in the removePrimesFromPreviousBlocks function. As such, once the prime double is calculated for the first time in calculatePrimesInBlock, it is associated to the prime multiple. Since there are very few primes to sieve in comparison to the maximum range of sieving, saving a pair of the current prime multiple and its double to avoid constant repetition of computations, is very reasonable.

3.6 Sequential prime sieve using fast marking with block decomposition optimized for space with modulo 30 wheel factorization

This was the first variation to introduce the use of the wheel factorization to speed up the sieving process.

Instead of using the wheel factorization to pre-sieve numbers and update the bitset of composite numbers, the current implementation uses the wheel sieve to determine the next possible prime given a number. This way the bitset is only updated in the positions that represent possible primes. This variation of use of the wheel sieve is more efficient for sieving, but also implies that to extract the primes from bitset only the positions associated with possible primes must be evaluated, since the other were not considered in the sieving process.

With this insight, the access to the bitset was changed in order to store only bits associated with possible primes, and reduced the memory consumption by 16%, since a wheel with modulo 30 removes 66% of composites, including the even numbers that were already left out in previous algorithms.

The problem with this method is that it introduces a lot of overhead to determined which position of the bitset is associated to a given prime (to extract the primes from the bitset), and in which position should be store the sieving analysis associated to a given number. Because of this, the computation time increased considerably, since memory access is a bottleneck of the algorithm. As a result, this version should be use when there is very little memory available. And even in these conditions the use of algorithm 14 is better, since is a truly segmented version and is completely configurable on how much memory it can use. On other hand, since algorithm 14 is implemented with OpenMP, it takes full advantage of today's computer architectures.

3.7 Sequential prime sieve using fast marking with block decomposition optimized for space with modulo 210 wheel factorization

This is the same version as algorithm 3.6, but uses a modulo 210 wheel to sieve 77% of the composites and as such, reduce memory consumption by 27% in relation to previous algorithms that do not use a wheel.

3.8 Sequential prime sieve using fast marking with block decomposition optimized for space and time and with modulo 30 wheel factorization

Since the use of the wheel to store only bits of possible primes introduced a lot of overhead, this version reverts to the previous scheme of storing all odd numbers, and uses the wheel to skip the sieving of the composites of the prime numbers 2, 3 and 5.
This way it was obtained a significant performance boost of almost 50% less computation time in comparison with the algorithms that did not use wheels.

3.9 Sequential prime sieve using fast marking with block decomposition optimized for space and time and with modulo 210 wheel factorization

This is the same algorithm as 3.8, but with a modulo 210 wheel, and as such, it skips also the composites of the prime number 7.

3.10 Sequential prime sieve using fast marking with block decomposition optimized for time and with modulo 30 wheel factorization

Since memory access is a bottleneck in all sieving algorithms, this version try to determine if using direct memory access without any need to offset calculations would improve performance.

The only difference between this variation and the previous is that it was reserved memory for all numbers, so that the number itself was the position within the bitset where the sieve analysis was going to be stored.

The results show that in fact performance did improve, but only for small ranges (less than 2^25).

The possible reason may be related to the fact that with this strategy, only half of the numbers can be stored in cache (comparing to only storing even numbers). And as such, although the overhead to access memory is reduced, the net loss in cache misses starts to outweigh the improved memory access when the range is increased.

3.11 Sequential prime sieve using fast marking with block decomposition optimized for time and with modulo 210 wheel factorization

This is the same version as the previous one but with a modulo 210 wheel, in order to skip the composites of the prime number 7.

3.12 Parallel prime sieve using fast marking with block decomposition optimized for space and time and with modulo 210 wheel factorization

With current computer architectures any computer intensive algorithm should be designed to take full advantage of the parallel computation resources available in multicore systems. Given that the previous algorithms were already implemented using block decomposition, and that the multicore systems use shared memory, the port of the best sequential algorithm (algorithm 9), to a parallel architecture using OpenMP, only required minor changes to the source code, in order to assign groups of blocks to different threads, and optimize the allocation of these blocks in order to avoid load unbalancing between the different threads.

3.13 Parallel prime sieve using fast marking with block decomposition optimized for time and with modulo 210 wheel factorization

This is the same algorithm as above, but trying again to see if using direct memory access would improve performance. And in fact it did, but not much. This variant is only faster than the previous one in ranges up to 2^26.

3.14 Segmented parallel prime sieve using fast marking with block decomposition optimized for space and time and with modulo 210 wheel factorization

Since the parallel algorithms were very fast to sieve primes up to 2^32 (about a second), it was developed a segmented variant to allow the computation of primes up to 2^64 using only the amount of memory that is specified in the command line arguments. This way, the algorithm can adapt to different hardware and still compute all primes up to 2^64 with very little overhead (about 3%) compared with the fastest OpenMP version (algorithm 12). This overhead is associated with the management of the segments (groups of blocks), and various checks to verify if export to file or prime count was specified by the user.

Given the results, this is the recommend algorithm to use in multicore systems when it is required to calculate very large primes (bigger than 2^35), using efficiently the memory available.

3.15 Distributed prime sieve using fast marking with block decomposition optimized for space and time and with modulo 210 wheel factorization

This is the first implementation of a distributed prime sieve algorithm prepared to run in homogeneous computer clusters.

It uses the best sequential algorithm (9), and splits evenly the workload among the processes.

In order to keep communication between processes to a minimum, each process computes the primes up to √n and then uses then to sieve their share of the workload.

The share of the workload [startNumber, endNumber[is computed as follows:
startNumber = processID * maxRange / numberProcessesInWorkgroup
endNumber = (processID + 1) * maxRange / numberProcessesInWorkgroup
processID ranges from [0, numberProcessesInWorkgroup – 1]

3.16 Distributed prime sieve using fast marking with block decomposition optimized for time and with modulo 210 wheel factorization

This is the same algorithm as above but using one less operation (shift left 1) per memory access. Is similar to previous algorithms optimized for time, but since the algorithm is now segmented, direct memory access can't be done. Like previous algorithms, this variant only performs better until 2^25.

3.17 Hybrid distributed prime sieve using fast marking with block decomposition optimized for space and time and with modulo 210 wheel factorization

The previous implementation completely disregards the fact that each node in the workgroup may be a multicore system, and as such, a hybrid version would take better advantage of the shared memory architecture to avoid the replication of the computation of the sieving primes up to √n in the same node.

This way, OpenMPI can be used to coordinate the distribution of the workload between different nodes in the cluster and then inside each node can be used an OpenMP variant optimized to take full advantage of the shared memory architecture.

3.18 Hybrid distributed prime sieve using fast marking with block decomposition optimized for time and with modulo 210 wheel factorization

This is similar to algorithm 16, but now applied to algorithm 17. Like other algorithms in the same category, it only performed better in relation to the previous algorithm when the max range was less than 2^25.

3.19 Hybrid distributed prime sieve using fast marking with block decomposition optimized for space and time, with modulo 210 wheel factorization and with dynamic scheduling

The previous distributed hybrid algorithms didn't take into account that the cluster in which they may be used can have heterogeneous hardware nodes with different processing capabilities, and even if the cluster is homogeneous, it may very well have variable workload. These external factors may have a damaging effect on the performance of the previous algorithms because some nodes may finish much sooner than others and then became idle, while others are still processing at full load. In order to avoid this, it was implemented a dynamic scheduling algorithm based in the variant 18, but with a control process that creates a given number of segments (specified by the user), and then distributes these segments dynamically according to the requests of the nodes that finish their segments. This way, if there is node that finishes sooner than others, it can continue contributing to the computation by requesting a new segment from the control process. With this strategy, the algorithm is ready to adapt to heterogeneous clusters with variable workloads and can be fine-tuned to a given network topology by specifying how many segments should be created, how many processes should be started and in which nodes, and how many threads should each process use.

3.20 Hybrid distributed prime sieve using fast marking with block decomposition optimized for time, with modulo 210 wheel factorization and with dynamic scheduling

Like in the previous algorithms it was implemented a variant optimized for time, but in this case it performed better than their equivalent (19) up to 2^30.

4 Methods used to calculate the results

The results were collected using Ubuntu 13.04 64 bits, and the source code [1] was compiled with –O3 –s flags using mpic++ with g++ 4.7.3, OpenMPI 1.4.5 and Boost 1.49.

To facilitate the compilation process, it is provided the Eclipse PTP project files, so it is only necessary to import the project into eclipse, fix any non-standard path locations, and build the project.

The sequential and parallel results were obtained in the following hardware configuration:

Clevo P370EM
CPU: i7-3630QM
Clock rate: 2400 MHz
L1 cache: 256 KB
L2 cache: 1024 KB
L3 cache: 6144 KB
Memory: 16 GB RAM DDR3 1600 MHz

For the distributed version was used the Clevo P370EM in conjunction with a Asus G51J, (which has roughly 50% less processing capability and 50% slower memory). To connect the two laptops was used a 100 Mbps Ethernet connection with a Technicolor TG582n router.

Asus G51J
CPU: i7-720QM
Clock rate: 1600 MHz
L1 cache: 256 KB
L2 cache: 1024 KB
L3 cache: 6144 KB
Memory: 4 GB RAM DDR3 1066 MHz

The program parameters used to obtain each result are detailed in [15].

The implementation is verified with the unit testing framework from the boost library, and also provides a manual verification option, by specifying a file name with the expected results.

It was optimized to run on a computer cluster and output the results to partial files using a network file system (each partial file has the specified file name appended with the range of prime numbers that it contains).

It also allows the sending of the results to a collector process, in order to merge then into a single file. This alternative is only recommended for testing purposes or when the maximum range is small. A better way of testing large ranges it to count the number of primes found by specifying the parameter -- countPrimesInNode

Since the purpose of this implementation is to compute large prime numbers, it also provides the option to only output the primes from the last segment (in case a segmented algorithm is used).

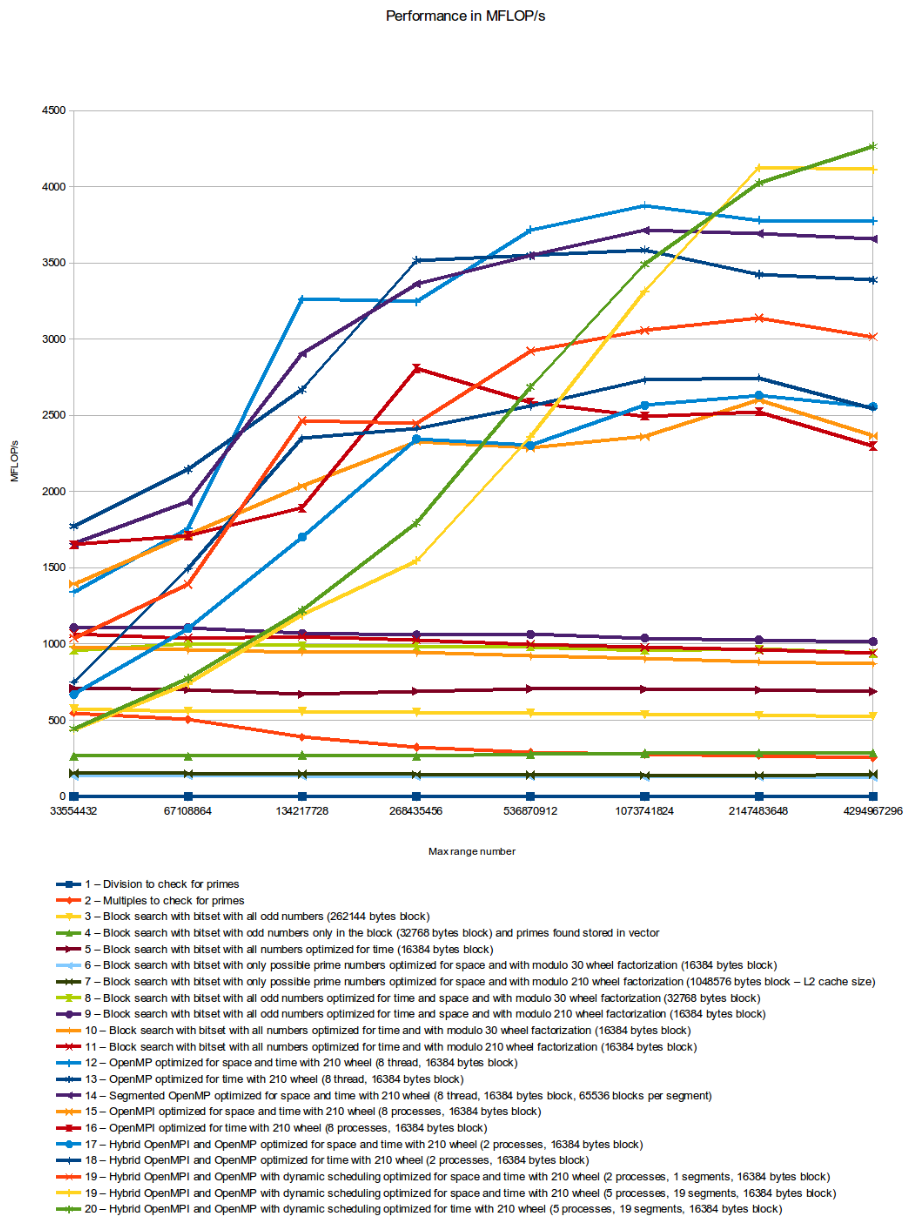
For a more detailed view of the parameter options allowed, the implementation provides a --help option, to explain the usage of each parameter. It also provides an interactive CLI in case no parameters are provided and the distributed algorithms are not used. To use the distributed algorithms is necessary to run the program with mpirun or other equivalent.

A more detailed usage is provided in [16].

5 Results and performance analysis

In this section will be presented the obtained results and will be given a brief analysis in each performance metric. Detail explanation of the reasons behind the different performances in each algorithm is presented in section 3.

5.1 Global performance comparison

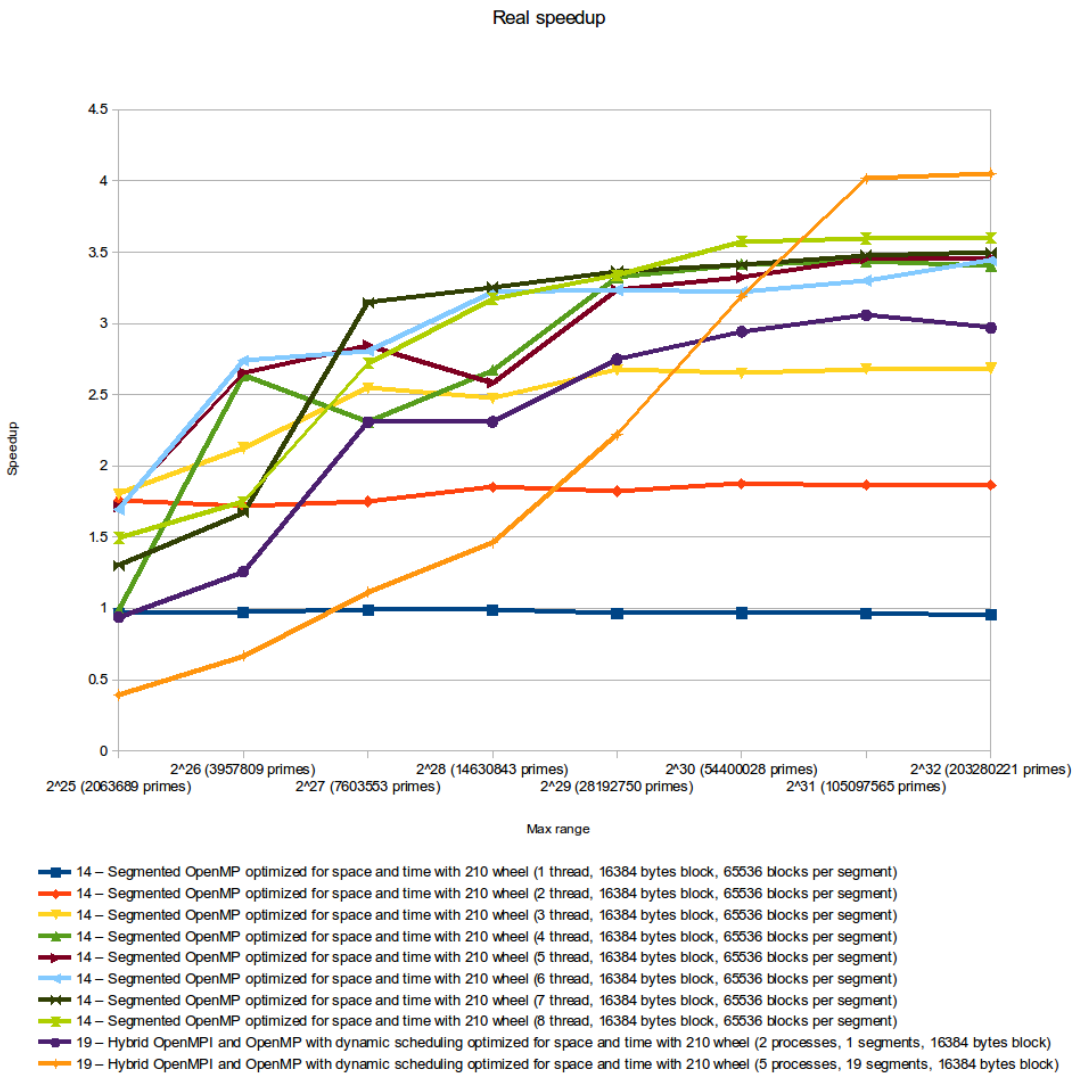


The previous chart gives a global overview of the performance of all implementations in ranges from 2^25 to 2^32.
This chart shows that most algorithms need a range of about 2^28 to reach their maximum performance. The only exceptions are the distributed algorithms, because the computation capacity was increased by 50% by adding a second laptop to the workgroup. As such, the performance will reach maximum in ranges larger than 2^32.

The reason for this is related to network latency and initializations overhead of processes and threads taking a considerable percentage of total computation time when the useful computations are relatively small (less than a second).

5.2 Performance analysis

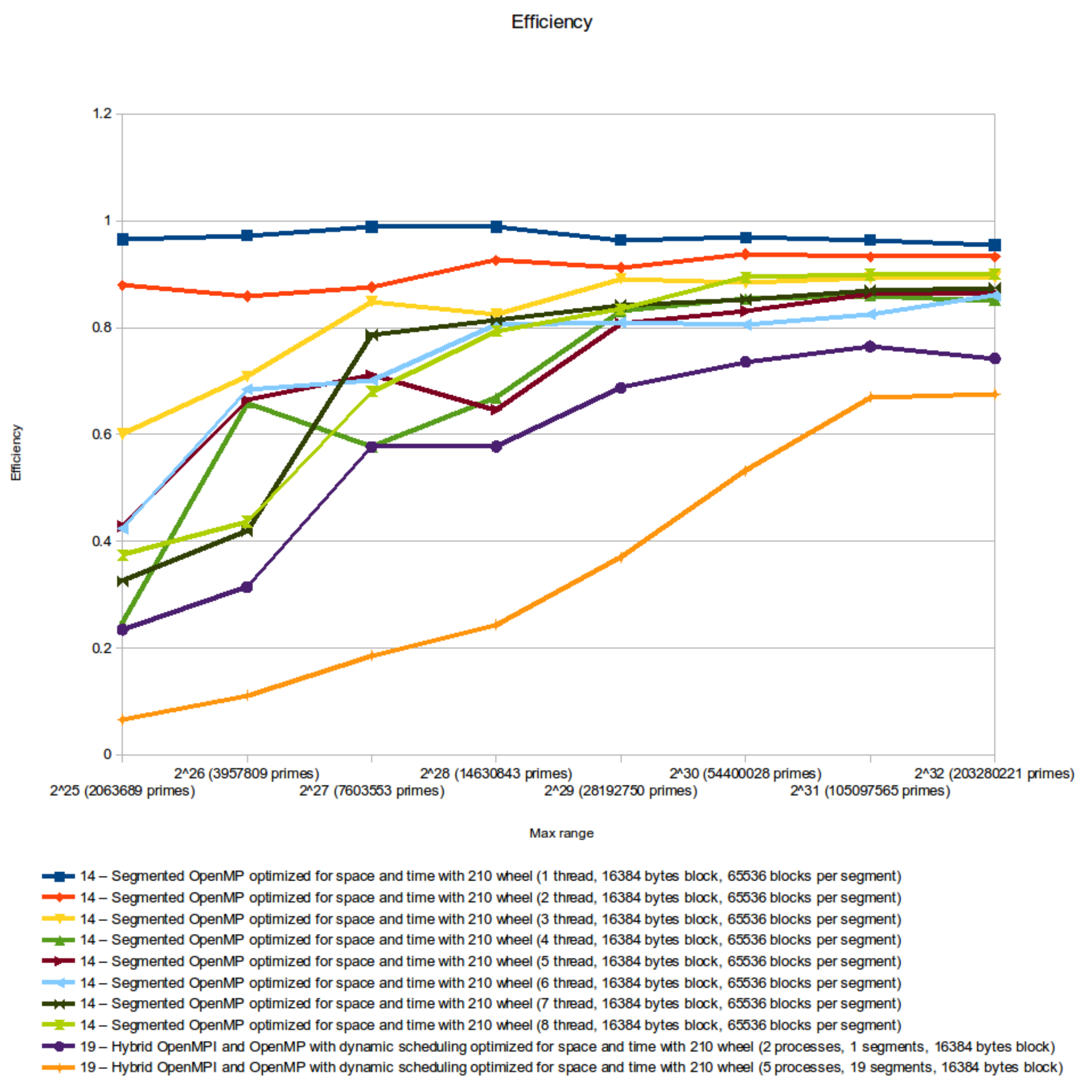
Real speedup. The following chart shows the obtained speedup when comparing execution times of the recommended parallel algorithm (12) and distributed algorithm (19) with the best sequential algorithm (9).



The results shows that the parallel version (14) runs best when using 8 threads on a quad core system with hyper-threading. But as expected, the result is not proportional to the number of threads but proportional to the number of real available cores. The speedup is not 4 because there is some sequential sections in the algorithm that cannot be parallelized.

It is also clear that the distributed algorithm should only be used for large ranges, since the overhead and latency of MPI calls can damage performance if the total computations are small. The speedup of the distributed version isn't higher because the second node has 50% less processing capability and uses a 50% slower memory when compared with the first node. This was done on purpose to determine the adaptability of the algorithm to heterogeneous nodes.

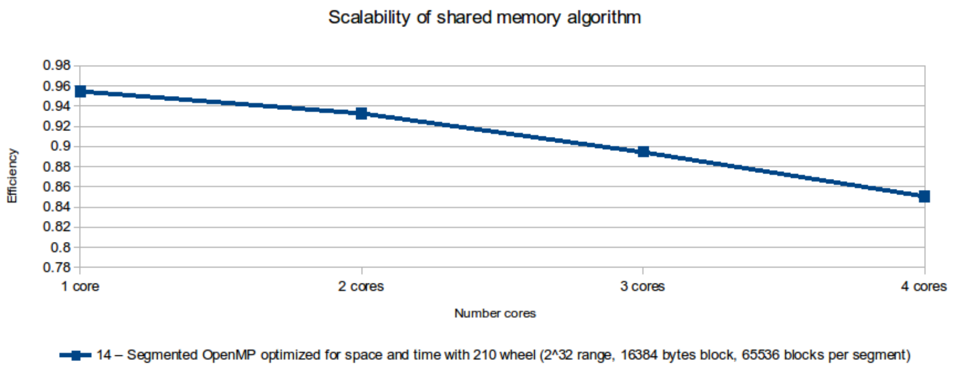
Efficiency. The following chart shows the obtained efficiency when comparing execution times of the recommended parallel algorithm (12) and distributed algorithm (19) with the best sequential algorithm (9). It was calculated based on the real speedup presented earlier.

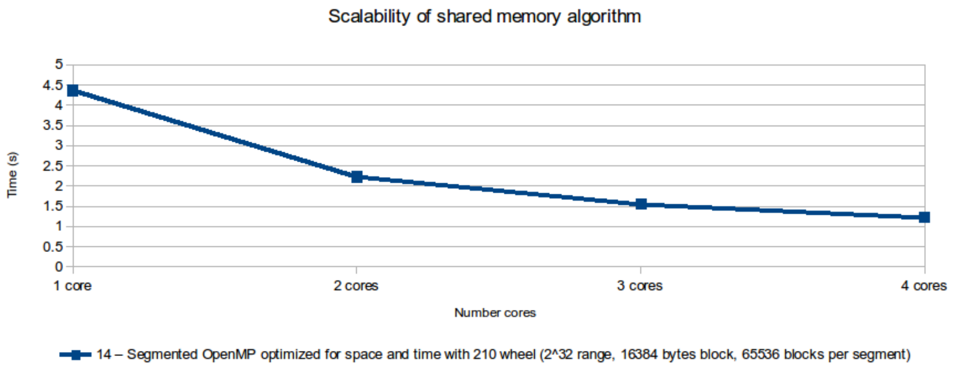


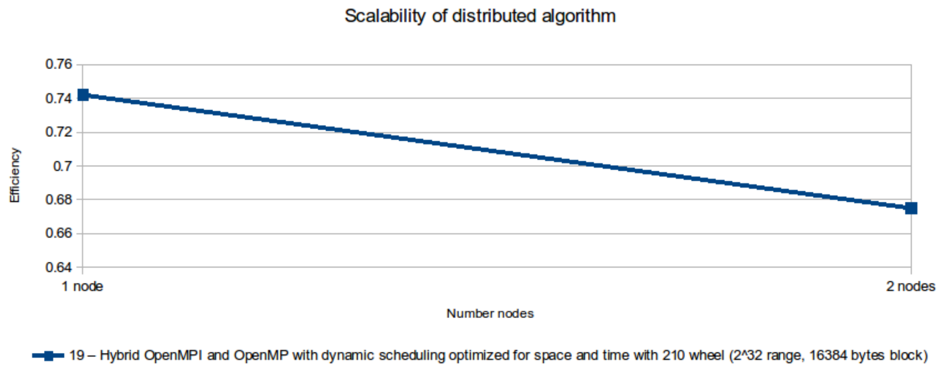
As expected, the best efficiency was obtained with the shared memory algorithm, since it keeps computation repetitions to a minimum and doesn't suffer from overhead of MPI calls and latency associated with network communications.

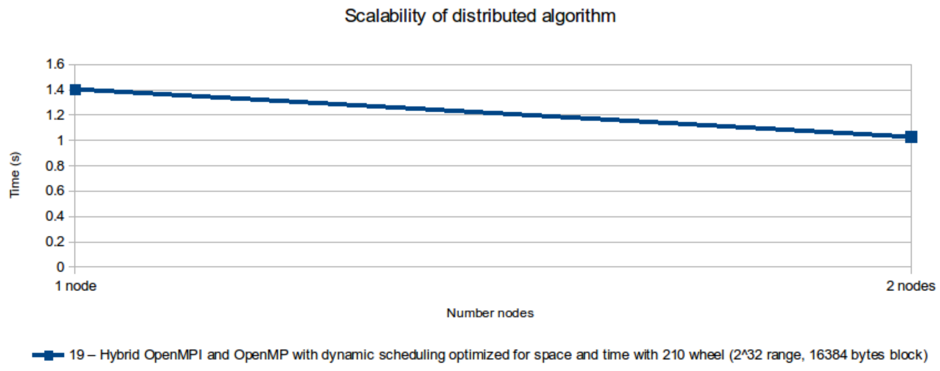
The reason for having 5 processes in algorithm 19 when running in 2 nodes is related to the fact that 1 process is only to control the dynamic allocation of segments to slave processes, and because on the current testing system, using only one process per node was resulting in lower performance, due to latency in the MPI calls when requesting a new segment from the control process. As such, the performance improved with 2 processes per node, because this way, when 1 process is waiting for a new segment allocation, the other one can still be using the node computation resources. With this strategy, idle periods are reduced when network latency is high.

Scalability. The following charts shows the obtained scalability when comparing execution times of the recommended parallel algorithm (12) and distributed algorithm (19) with the best sequential algorithm (9).









From the previous charts we can see that both algorithms scale relatively well, since their efficiency isn't significantly reduced when more computing resources are added, and the computation time is reduced consistently.

6 Conclusions

In this paper was presented several algorithms and optimizations that can be used to efficiently compute primes up to a specified number. It was discussed several variants of algorithms to maximize the efficiency in different computer architectures and cluster topologies.

The final implementation shows very reasonable efficiency and scalability and it can be used to compute primes numbers up to 2^64 in any computer architecture. For maximum efficiency was developed an OpenMP version to be used in traditional multicore computers, and a hybrid OpenMP and MPI with dynamic scheduling to be used in heterogeneous computer cluster, that may have computation nodes with different hardware capabilities and that most probably will have variable workload.

To improve the current implementation, an OpenACC variant is being implemented to take full advantage of the massive parallelism that current GPUs can provide, and a bucket sort algorithm [8] may be used to increase the cache hit rate for very large ranges.

References

1. Costa, C.: Distributed prime sieve C++ implementation (git repository), https://www.assembla.com/code/cpar-12-13/git/nodes/master/DistributedPrimeSieve
2. RSA Laboratories: PKCS #1 v2.2 - RSA Cryptography Standard (2012)
3. Wikipedia: Primality test, http://en.wikipedia.org/wiki/Primality_test, [Online, accessed 24-07-2013]
4. Wikipedia: Sieve of Eratosthenes, http://en.wikipedia.org/wiki/Sieve_of_Eratosthenes, [Online, accessed 24-07-2013]
5. Wikipedia: Sieve of Atkin, http://en.wikipedia.org/wiki/Sieve_of_Atkin, [Online, accessed 24-07-2013]
6. Wikipedia: Sieve of Sundaram, http://en.wikipedia.org/wiki/Sieve_of_Sundaram, [Online, accessed 24-07-2013]
7. Walisch, K.: Prime Sieve, https://code.google.com/p/primesieve/, [Online, accessed 24-07-2013]
8. Járai, A., Vatai, E.: Cache optimized linear sieve (2011)
9. Paillard,, G.: A fully distributed prime numbers generation using wheel sieve (2005)
10. Sorenson, J.: An Analysis of Two Prime Number Sieves (1991)
11. Wirian, D.: Parallel Prime Sieve – Finding prime numbers
12. Cordeiro, M.: Parallelization of the Sieve of Eratosthenes
13. Wikipeda: Wheel factorization, http://en.wikipedia.org/wiki/Wheel_factorization, [Online, accessed 25-07-2013]
14. Caldwell, C.: Wheel factorization, http://primes.utm.edu/glossary/page.php?sort=WheelFactorization, [Online, accessed 25-07-2013]
15. Costa, C.: Results of different implementations of prime sieves, https://www.assembla.com/code/cpar-12-13/git/nodes/master/DistributedPrimeSieve/docs/PrimesComputionTimes.ods
16. Costa, C.: Usage of Distributed prime sieve implementation, https://www.assembla.com/code/cpar-12-13/git/nodes/master/DistributedPrimeSieve/docs/usage.txt