Project — Sieve of Eratosthenes parallelized with DPS

Course: Program parallelization on clusters of PCs
Author: Jeremiah Menétrey
Due date: Jan. 1, 2014

“Move 'ZIG'. For great justice.” — Captain

Description of the application

The sieve of Eratosthenes is an classic algorithm to find prime numbers up to a given integer. The algorithm works as follows:

Given a list of unmarked integers $[2..L]$:
1. Find n, the next unmarked number in the list
2. Mark all multiples of n, except for n itself
3. Repeat *1-2* until all numbers up to $\sqrt{L}$ have been processed
4. The numbers left unmarked in the list are prime numbers

Algorithmic complexity

The complexity of the algorithm depends on L and the number of primes in the list up to $\sqrt{L}$. It can be proved that the actual complexity is $O(n \log\log n)$, as the prime harmonic series asymptotically approaches $\log\log n$.[1]

Achievable speedup (Amdahl's Law)

The maximum achievable speedup for an application where a fraction f cannot be parallelized is

$$S_p = \frac{t_s}{t_p} \leq \frac{1}{f + \frac{1-f}{p}} = \frac{p}{1 + (p-1)f}$$

Serial part of the application:

- Initializing the list of approx. L numbers, i.e. looping once over the list;
- Retrieving the primes found by the sieve, which is also done with one loop over the list.

The actual sieving is considered parallelizable, and consists of $\sqrt{L}$ loops over the list of numbers. So we have $f = \frac{2}{\sqrt{L}}$ and therefore

$$S_p \leq \frac{1}{\frac{2}{\sqrt{L}} + \frac{1-\frac{2}{\sqrt{L}}}{p}} = \frac{p}{1 + (p-1)\frac{2}{\sqrt{L}}} = \frac{p\sqrt{L}}{2p + \sqrt{L} - 2}$$

The formula shows that the speedup grows as we add CPUs and increase the size of the list. See the annex for graphs that show the achievable speedup with varying number of processors p and limit of the list L. The dependency between those two factors and the limit of the achievable speedup are clearly visible.

Parallelization strategies

In this section, it is assumed that every process has a unique ID (called *pid*) and every process knowns its own ID as well as the IDs of all other processes (so they also know the number of processes).
The number of processes is denoted p, and processes *pid* is within the range $[1..p]$. There might also be a *master* process, which is not counted in p and has the *pid* 0.

All strategies below use the fact that chunks of the list of candidate primes need not to be sent over the network. It is sufficient to communicate the upper limit L of the original list, the size of the chunks S or both, to be able to build locally a representation of the chunk with the correct bounds, using the *pid* of the process. Except if stated otherwise, when the communication of a chunk is implied, L, the size of the chunks or both are actually sent over the network, depending on the situation.

First strategy: The easy one

The master process computes the primes up to $\sqrt{L}$. Every time it finds a prime, it broadcasts it to all the slaves. When done, it broadcasts a special FIN message indicating that it has finished.

The slaves are each responsible of a different chunk of the candidate primes. When a slave receives a prime from the master, it marks all the multiples of that prime in its chunk. When it receives the FIN message, the slave collects all unmarked numbers (i.e. all primes) in its chunk and sends them back to the master.

It must be noted that the master is actually doing a sieving with an upper limit of $\sqrt{L}$, which means that it can stop the sieving after reaching $\sqrt[4]{L}$ and still find all primes up to $\sqrt{L}$.

Discussing the strategy

This strategy is very simple and still might achieve good results by letting the master process handle a small portion of the list while allowing to parallelize the processing of the big portion.

This technique scales very well for the portion of the list greater than $\sqrt{L}$. In facts, for a given L, the size of the chunk processed by each slave is inversely proportional to the number of slaves. In the extreme case where there are $L - \lfloor\sqrt{L}\rfloor$ slaves (i.e. one slave for each number greater than $\sqrt{L}$), the limiting factor will be given by the rate at which the master will broadcast new primes, which is not parallelized with this strategy.

The downside of this strategy is the memory limitation: If there are not enough slaves to hold the entire second portion of the list, the algorithm won't work as-is. Caching the primes sent by the master might be a way to overcome the limitation, however the benefits of the on-the-fly sieving (sieving a chunk while the master continuously sends new primes) would not be applicable for all chunks after the first p ones.

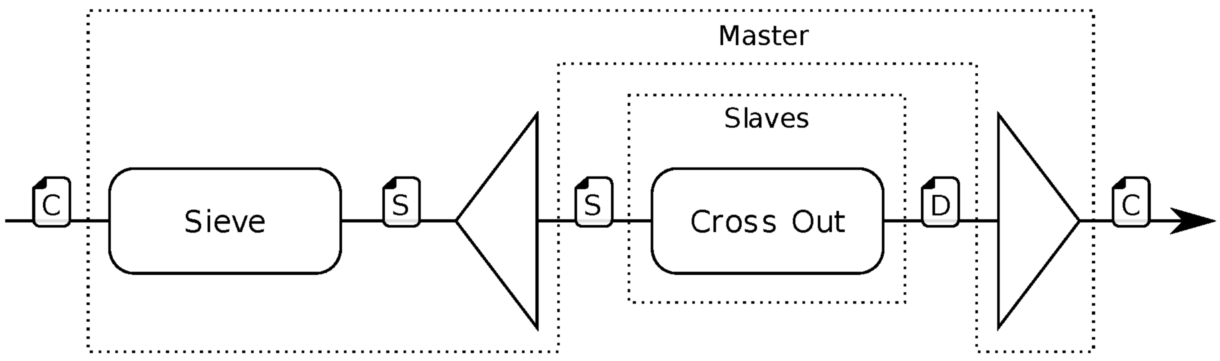
Computational complexity

Let k be the number of primes up to $\sqrt{L}$. The prime counting function[2] $\pi(n)$ gives the number of primes up to n, so we have the relation $k = \pi(L)$.

- Each slave will be handling a chunk of size $S = L/p$, and therefore will be performing k loops over S elements. So the parallel computation time is $k(L/p)$.
- The master will send kp messages (broadcasting the primes) and receive p messages (the primes found by the slaves), amounting to a total of $p(k+1)$ messages.
- Computation/Communication ratio: $\frac{kL}{(k+1)p^2} = O(L/p^2)$.

The ratio increases linearly with the number of candidate primes, but decreases quadratically with the number of CPUs. This means that for a fixed L, the number of CPUs has a highly negative impact, but for a fixed number of CPUs, the size of the list is neutral.

DPS flow graph



$\boxed{\text{C}}$ A dummy control message signaling the start or the end of the sieving.
$\boxed{\text{S}}$ Contains a list of primes and optionally the *FIN* signal.
$\boxed{\text{D}}$ Contains an *ACK* signal and optionally a list of primes.

The `Sieve` operation performed by the master is done in the same process as the `Split` and `Merge` operations, but in another thread. This allows DPS to optimize the communication between the `Sieve` and the `Split` operation, by not actually sending a message but rather sharing memory.

As DPS requires that each message sent through the `Split` generates a message received by the `Merge`, the slave processes must send an acknowledgment message to the merge, even if they are not able to send the list of primes in their chunk yet.

When the master sends the last prime to the slaves, it signals that there are no more primes coming by setting the *FIN* flag. When slaves have done processing their chunk with the last primes, they send the *ACK* message with a list of all the primes they found.

The master does the sieving up to $\sqrt{L}$ in the `Split` operation and collects the primes from the slaves in the `Merge`.

Second strategy: Incremental sieve

This strategy leads to a parallel incremental sieve, where each process is responsible for a chunk of the whole list (which might then theoretically be infinite).

The strategy involves a turn-based approach, where at step i the process responsible of the i-th chunk **finishes** that chunk, i.e. it finds all remaining primes in it.

At step i process with *pid* i performs the finishing sieve on its chunk then broadcasts the newly found primes to all other processes, which perform the sieve on their own chunk using the broadcast primes. Then at step $i+1$, process $i+1$ finishes its own chunk. Again, it broadcasts the newly found primes and all other processes perform the sieve on their own chunk with the primes found by process $i+1$. And so on...

Additionally, each process keeps the whole list of primes found so far, including those broadcast by other processes (which actually amounts to a kind of *shared* list of primes). This allows each process to start processing a new chunk as soon as it finishes its current chunk. A new chunk is first sieved using the shared list of primes, then the normal processing can resume. When process p finishes his chunk, process 1 can continue (as if it were the process $p+1$).

If the list of candidate primes is bounded (i.e. there is an upper limit L), the algorithm stops after the chunk l that contains $\sqrt{L}$ is finished. When process *pid* l finishes, it sends the new primes to other processes, which then perform the sieve using the new primes found by process l on all their remaining chunks (i.e. without communication). When done, all processes send the primes they found in that last big round (over several chunks at once) to the master.

Finally, the master process with *pid* 0 is only responsible for initiating the sieve, by broadcasting the initial conditions to all other processes, such as the limit L, an initial list of primes and the lower bound for the sieve (which is actually the upper bound L of the initial list of primes), the size of the chunks, the number of processes, and other parameters if needed. If the list of candidate primes is unbounded, process 0 can also be responsible of occasionally collecting the shared list of primes to save it or display it to the client.

Aside from communications from and to process 0, which are negligible, this strategy requires only the communication of newly found primes to all processes (the broadcast that updates the shared list of primes), which happens once for every chunk.

If we imagine the setting as a pipeline, it would be a five-stage pipeline with the following stages:

Stage	**Description**
Update	Receive new primes and update list
Sieve I	Sieve chunk using updated primes
Sieve II	Sieve chunk for new primes
Broadcast	Broadcast newly found primes
Sieve III	Sieve new chunk using list

Stages *Sieve II*, *Broadcast* and *Sieve III* are only performed by a process when it is its turn to finish a chunk. Non-finishing processes only perform the *Update* and *Sieve I* stages before exiting the pipeline.
Additionally, it must be noted that the *Update* and *Broadcast* stages must happen at the same time, i.e. all non-finishing processes must be in the *Update* stage when the finishing one is in the *Broadcast* stage.
Finally, it can be noted that this design allows for the *Sieve III* stage to take up to the time needed

for *Sieve I* and *Sieve II* combined, which might be useful as *Sieve III* actually does the same operation as *Sieve I* except that it uses all primes found so far instead of only the primes of the last update.
However, since the pipeline is only conceptual, it is also possible to relax the *Sieve III* stage by merging it with followings *Update* and *Sieve I* while buffering incoming broadcasts.

Discussing the strategy

This strategy is more involved than the first one and requires that processes are synchronized in order to implement the pipelined setting as described above. Hopefully, the synchronization is easily done using the communication stages *Update* and *Broadcast*.

The strategy distributes the serial part of the algorithm over several processes, which might hinder performances as the serial part is interleaving with communications in order to change round (in terms of the first strategy, it is like switching the role of *master* to another process).

The clear advantage of this strategy over the first one is that it allows to continue sieving as long as the available memory and the implementation allow to represent numbers the application has to deal with. Another advantage in the bounded version is that if the number of processes is limited and L is very big, this strategy will gracefully handle all the numbers while the first strategy will hit memory limitations (as each slave has to handle a chunk of size $(L - \sqrt{L})/p$) as it has to finishes in one "round". In this strategy, the chunks size is limited, e.g. to the maximum memory available, and the number of rounds is flexible.

On the communications side, their number is comparable to that of the first strategy in terms of number of sent primes, and even less if we count the number of messages: unlike the first strategy, where the master sends a message for at least each prime up to $\sqrt[4]{L}$, in this strategy a message is sent only once for each finished chunk.

Computational complexity

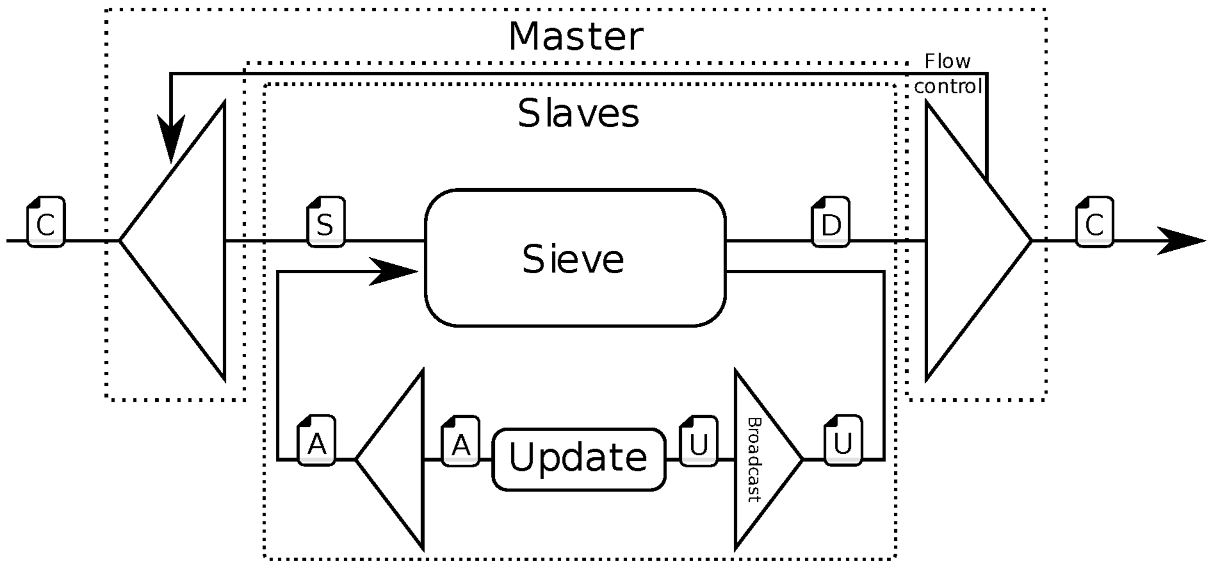
The bounded version of the strategy is used to formulate its computational complexity.

Again, let $k = \pi(\sqrt{L})$ the number of primes up to the square root of the bound L. The number of rounds is $r = \sqrt{L}/S + 1$. The size of the chunks S is arbitrary and the number of chunks $s = L/S$. l is the chunk that contains $\sqrt{L}$, i.e. $l = \sqrt{L}/S$, and by extension the number of the round in which chunk l is finished.

- Each process will have to loop at most k times over each of their chunk. The parallel computation time is therefore $ksS = kL$. Note that $k = O(\sqrt{L})$.
- Each process will send one message per chunk, up to chunk l. Then each process will send one message to the master. The total number of messages is therefore $l + p = p + \sqrt{L}/S$.
- The Computation/Communication ratio is therefore: $\frac{kL}{p+\sqrt{L}/S} = O(L^{1.5}/p)$.

This strategy is much more efficient than the first one in terms of computational complexity, as the ratio decreases only linearly with the number of CPUs but grows better than linearly with the number of candidate primes. The communications have much less impact with this strategy.

DPS flow graph



$\boxed{C}$ A dummy control message signaling the start or the end of the sieving.
$\boxed{S}$ Signals to the slaves that they can start the sieving. May contain an initial list of primes and other parameters.
$\boxed{D}$ Signals to the master that a slave finished the sieving. Contains a list of primes.
$\boxed{U}$ New primes broadcast to all slaves by the one that finished its chunk.
$\boxed{A}$ Dummy acknowledgment message used to terminate the broadcast.

Pairs of *Sieve* and *Update* operations have to be implemented within a same operating system process as they both have to access the local instance of the shared list of primes (*Update* writes to that list, while *Sieve* reads from it).

The *broadcast* stage is implemented with a `Split` operation, which is done only by the process that finishes its chunk. A $\boxed{U}$ message is sent to all processes, which update their local instance of the shared list of primes (*update* stage) and send back an acknowledgment message in order to terminate the broadcast with the `Merge` operation (a DPS requirement). When the broadcast is done, the broadcasting process can continue the sieving as described in the strategy above.

In the unbounded version of the strategy, it is desired that the master occasionally collects primes found so far. In order to achieve this, flow control is used between the master `Split` and `Merge` operations, which allows to send several $\boxed{S}$ messages to the slaves, which in turn can send several $\boxed{D}$ messages to the master with the primes found so far (or actually the primes found since the last $\boxed{D}$ message was sent). In the bounded version, the master sends only one $\boxed{S}$ to each slave, and the slaves send only one $\boxed{D}$ message when the whole sieving is done (in which case flow control is not needed).

Third strategy: Almost no communication

Each process is assigned a chunk of the list after $\sqrt{L}$. This can be done deterministically without communications.

Then each process performs the sieving up to $\sqrt{L}$, while also crossing-out the numbers in their respective chunk.

Finally, a *master* process gather all primes found by the other processes. The master can be one of the processes that performed the sieving.

Discussing the strategy

This strategy is a variation of the first one which minimizes the amount of communications. If the density of primes among candidate primes is such that the communication time exceeds the computation time to find a given amount of primes, e.g. when starting the sieving from 2 or 3, reducing the amount of communications is more beneficial than trying to parallelize every bit of the program.

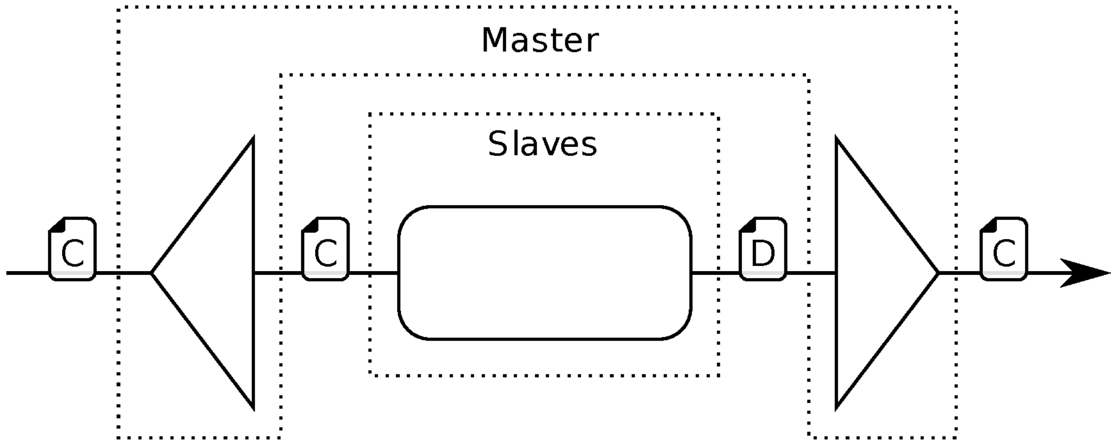
Part of the processing in this strategy is done by every node, which wastes resources, but this allows to reduce the amount of communications. It could also be possible to use one process to sieve the numbers up to $\sqrt{L}$, which would allow other processes to idle and therefore be available for unrelated processing, or they could also idle and consume less power, but then more communications would be required, although a more manageable amount than for the first strategy.

Computational complexity

Let $k = \pi(\sqrt{L})$ as defined previously.

- Each process will be handling a list of number composed by the numbers up to $\sqrt{L}$ and their respective chunk, which is of size $\frac{L-\sqrt{L}}{p}$ and they will therefore perform k loops over $\frac{L-\sqrt{L}}{p} + \sqrt{L}$ numbers.
- The only communications are those required to send the list of found primes to the master. Using a scenario where the master is not one of the processing node, only p messages are sent.
- Computation/Communication ratio: $O(\frac{k(L-\sqrt{L}(p-1))}{p^2})$. If we assume that $p << \sqrt{L}$, then the ratio becomes $O(\frac{kL}{p^2}) = O(\frac{L^{1.5}}{p^2})$.

DPS flow graph



C A dummy control message signaling the start or the end of the sieving.
D Contains a list of found primes.

The *master* sends a dummy message to all processes, making them start the sieving. When a process finishes its sieving, it sends the list of found primes back to the *master*. The master merges primes found by all the processes and terminates the program by sending a C message.

Detailed theoretical analysis

In order to analyze the strategies, computations and communications times were measured. Computations time is measured by running an implementation of the serial algorithm several times with different upper limit values, and to measure communications time a simple parallel DPS program was created, which measures the RTT of messages sent with different payloads from a master thread to slave threads (`Split`-`Process`-`Merge` flow, with the `Process` operation handled by distinct threads and actually just sending back the input message to the master).

For computations time, it was found that the processing time is linearly dependent on the number of primes or the number of candidate primes (the number of primes being almost linearly dependent on the number of candidate primes).

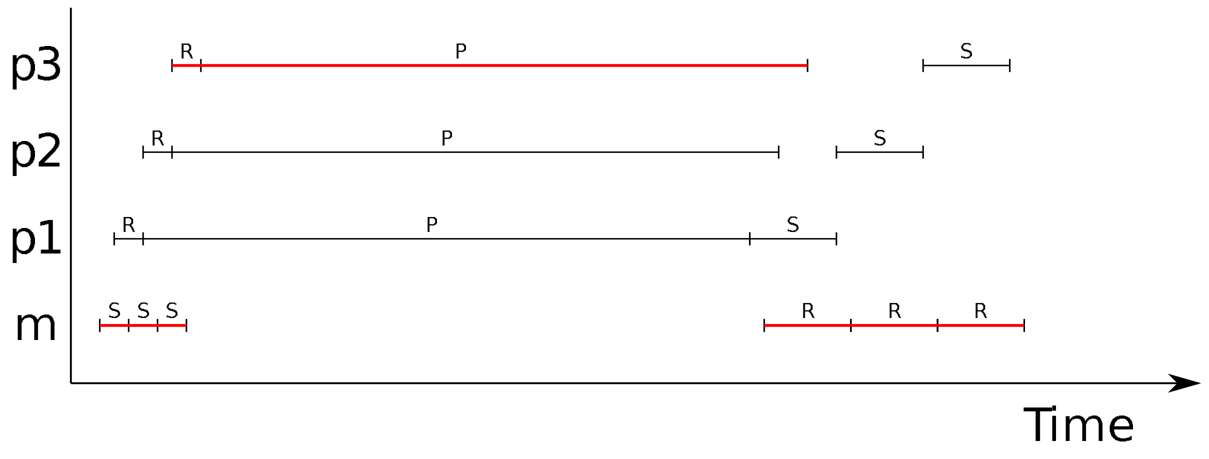
For communications time, it was also found that after a threshold on the payload size, the communication time over the size of the payload was approximatively constant.

The theoretical analysis below is made assuming those two observations hold in the general case. The values found empirically are approximatively:

- Processing time: 6.4e-6 ms for each candidate prime;
- Communication time: 5.0e-4 ms per prime in the payload of a message from one node to another (half of the RTT).
- Communication latency: Approx. 15 ms.
- Number of actual primes among candidate primes: 5.6% (mean for L in the range 1.25e7 to 3.2e9). The number of primes is actually decreasing as the limit L increases, but in order to simplify the analysis, a fixed value is used. This leads to an overestimation of the impact of communications. In fact, it is expected that $\lim\limits_{x \to \infty} \frac{\pi(n)}{n} = 0$.

Using those values, there is approximatively one prime found each 1.1e-4 ms, which makes the communication time five time larger than the computation time. In the light of this result, the third strategy is chosen for analysis and potential implementation, as the first strategy would require approximatively one message for each prime found up to $\sqrt{L}$, and the third one would require far less messages.

Timing diagram



Critical path (highlighted in red in the diagram):

- Send $\boxed{C}$ messages to processes 1, 2 and 3
- Sieving by process 3
- Receiving $\boxed{D}$ messages from processes 1, 2 and 3

From the above diagram, we can derive a speedup formula.

- t_p: total processing time (serial solution)
- t_{ps}: processing time for the sieving up to $\sqrt{L}$
- t_{pc}: processing time for the crossing-out after $\sqrt{L}$ (serial solution)
- t_l: communication latency
- t_r: time to send the whole list of primes

$t_{serial} = t_{ps} + t_{pc} = t_p$
$t_{par} = n \cdot t_l + t_{ps} + \frac{t_{pc}}{n} + t_l + n\,\frac{t_r}{n} = (n+1)t_l + t_r + t_{ps} + \frac{t_{pc}}{n}$

Note that $t_{ps} + \frac{t_{pc}}{n} = \sqrt{t_p} + \frac{t_p - \sqrt{t_p}}{n} = \frac{(n-1)\sqrt{t_p} + t_p}{n}$.
Then $t_{par} = (n+1)t_l + t_r + \frac{n-1}{n}\sqrt{t_p} + \frac{t_p}{n}$.

Finally, the speedup is

$$S_p = \frac{t_{serial}}{t_{par}} = \frac{t_p}{t_c + \frac{n-1}{n}\sqrt{t_p} + \frac{t_p}{n}}$$

where $t_c = (n+1)t_l + t_r$, the total communication time including latency.

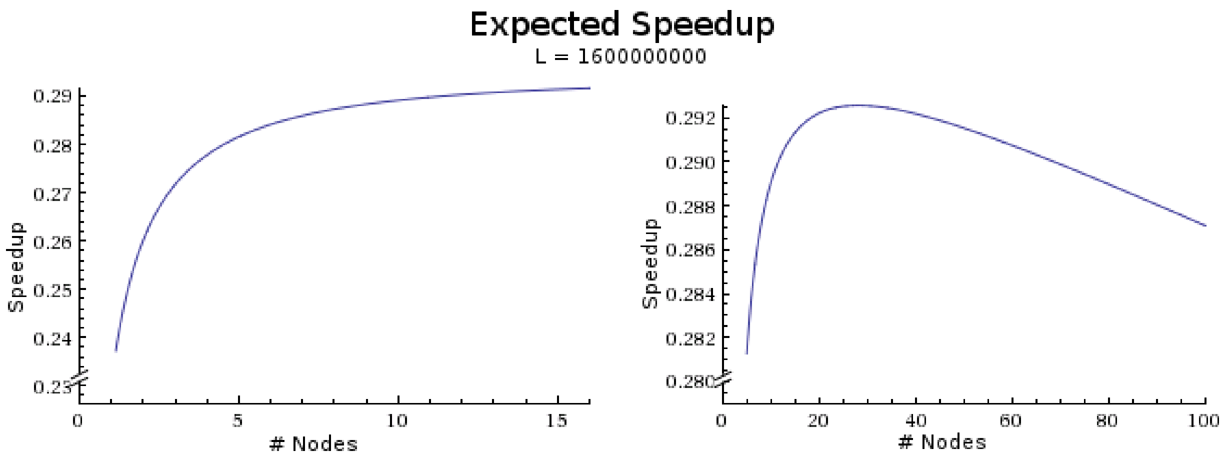
The following table shows the expected speedup for numbers up to 1600000000 (1.6×10^9) in which there are 79451833 primes.

n	S_p
2	0.260
4	0.278
6	0.284
8	0.287

n	S_p
10	0.289
12	0.290

This shows that even with almost no communication, the strategy is worse than the serial implementation. This is probably due to the final sending of all the primes to a single node, which handles one communication at a time. Switching to a distributed file system might improve the speedup by allowing to take the final sending out of the formula, as distributed file systems are inherently parallel. We can also note that the speedup slightly improves with the number of nodes.

If we take a step back, we can see that the speedup actually is maximal at 0.293 with 28.0427 CPUs. So using 29 nodes or more will make the speedup decrease as the gain in the computation factors won't compensate for the increasing communications.



Optimizations

The parallel algorithm is pretty much optimal in terms of computations. A small optimization can still be applied: Since the first part of the list of number (up to $\sqrt{L}$) is sieved with the standard, sequential algorithm to find all primes in it, it also can be optimized by sieving only to the square root of its biggest number, i.e. the numbers to process can be reduced to $\sqrt{\sqrt{L}} = \sqrt[4]{L}$.

With respect to the implementation, several optimizations can be though of.
If we relax the assumption that any node runs only one thread, we can use several threads in a single process and share the primes found in the first part of the list, which will reduce the overall amount of computations. Memory interactions between two threads should however be studied as concurrent memory accesses for several chunks might limit the benefits of that optimization when caches are involved.
On the communications side, the problem with the chosen implementation is that there is a huge bottleneck on the master, when it has to receive all primes at the end (which is done sequentially). One optimization could be to have a multi-threaded master, with one thread per slave so it can receive and process several messages at once. But ideally, what one would want is a truly parallel way of storing or processing found primes, which can be enabled with a distributed file system. It would effectively reduce the critical path to the process that has the longest processing and sending time, instead of having to add the time needed by each process to send their primes.

Finally, in order to address, or at least mitigate memory limitations, a better approach to memory usage should have been considered (see below the comments on the implementation).

Comments on the implementation

Once the parallelization strategy was decided, parallelizing the algorithm presented no real challenge in terms of coding.

Problems occured however because of memory limitations. The parallel algorithm implementation uses two `bitset` structures (`vector` -like structures optimized for booleans) which unlike vectors must be provided a size at compile time. For simplicity, the same size is used for both the first part of the list and the chunk, which is not optimal (we need only $\sqrt{L}$ numbers for one of them and the size of the other depends on the number of slaves). Additionally, when sending found primes back to the master, a vector is used to store and send the primes. Using a custom data structure, the memory used for the bitsets could be reused for building the list of primes before sending them, reducing the actually needed memory to only the equivalent of the two bitsets.

It is also possible to use vectors of booleans instead of bitset, which would solve the problem of oversized bitsets. However tests made using vectors showed that the processing time notably increased, so this solution was not retained as computation time was a bigger concern than the achievable size of the list.

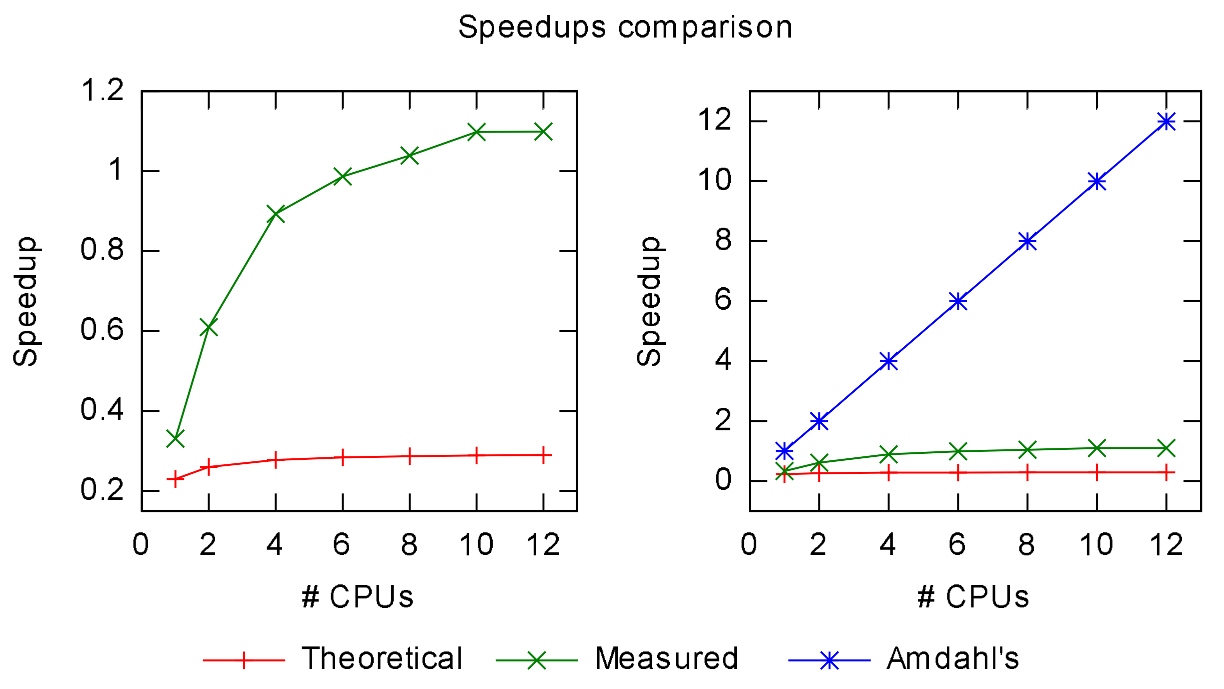
Performances evaluation

The table below shows the mean measured execution time for series of 5 executions on varying number of nodes. The effective speedup with respect to the serial implementation is also shown, as well as the predicted and Amdahl's speedups for reference.

n	**Execution time [ms]**	**Measured S_p**	**Predicted S_p**	**Amdahl's law S_p**
1 (Serial)	11905.4	-	-	-
1 (Parallel)	35996.38	0.331	0.230	1.000
2	19515.78	0.610	0.260	2.000
4	13327.40	0.893	0.278	3.999
6	12057.18	0.987	0.284	5.999
8	11454.46	1.039	0.287	7.997
10	10845.70	1.098	0.289	9.996
12	10828.62	1.099	0.290	11.993

We see that the parallel implementation beats the serial version in terms of execution time only when using more than 6 nodes. We also see a discrepancy with the predicted speedup when we compare them side-by-side, but also with respect to the predicted maximal speedup (which is 0.293 using 28 nodes).

Amdahl's law being an upper bound and not taking communications into account at all, the speedup obtained from that law (almost linear for small numbers of nodes) is not possibly achievable with the implemented algorithm.



Corrected model

The problems with the model used to predict the speedup are mainly due to network issues: It assumes a too simplistic view of the actual network.

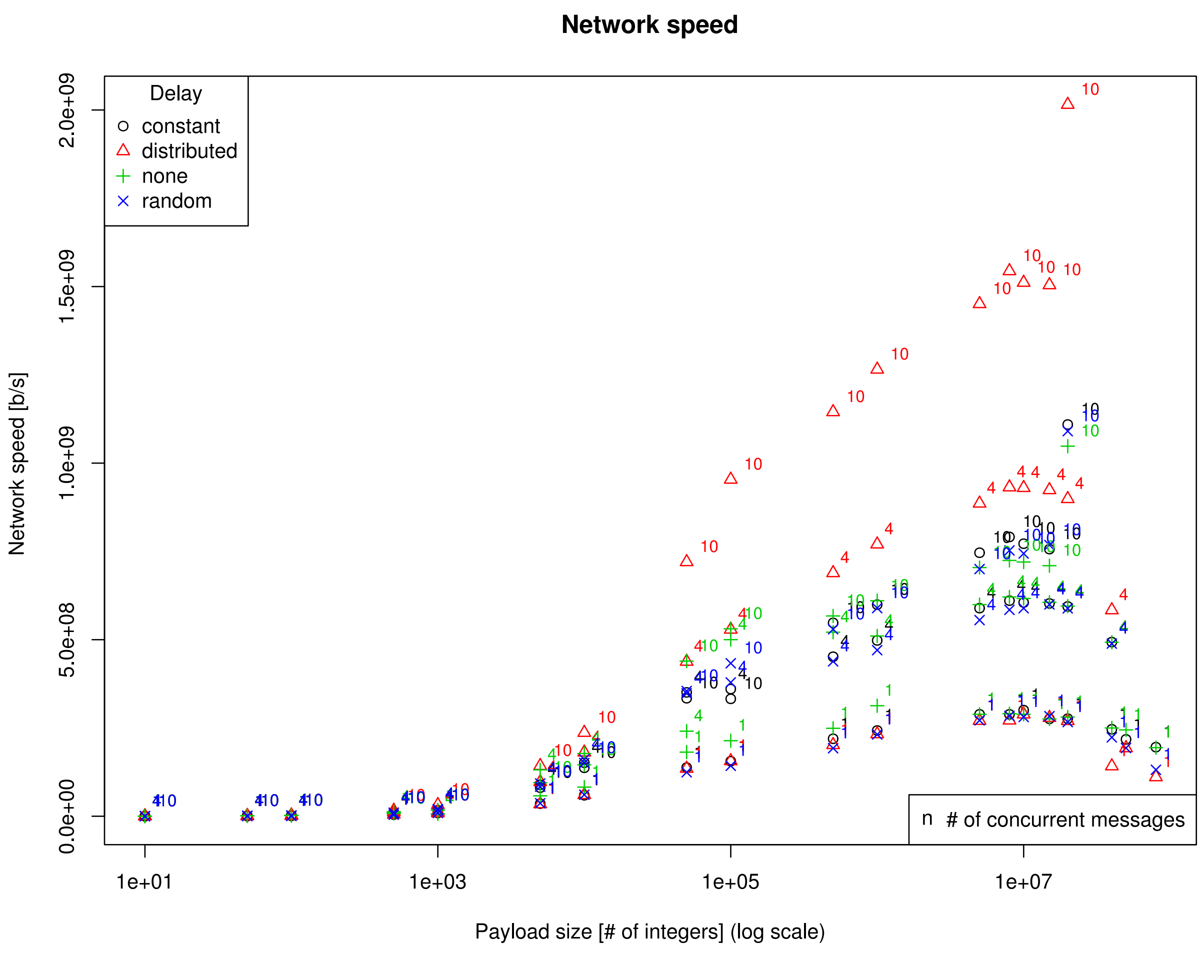
The main problem is that the values measured in order to get some constants for the theoretical analysis are not accurate enough, or even cannot be used as they have been to fit in the model. For instance, the measured network speed is heavily dependent on the number, the size and the temporal distribution of the messages sent to the master.

The graph below, computed from updated and more rigorous measurements, shows the network speed computed from the time it took to send messages around the network with varying parameters. The setup is the following: A master send a message to all slaves and the latter send it back to the master.

During the measurements, messages were sent with varying payload sizes and number of concurrent messages. For each series of tests, the network load put on the master was modified by delaying the sending of messages from the slaves. The following load management delays were used:

- None: No load control, all slaves send their message back as soon as they receive it
- Constant: Each slave delay the reply by a given (identical) amount of time
- Random: Each slave delay the reply by a random amount of time
- Distributed: Each slave delay the reply by a controlled amount, which should simulate an uncongested network

Note that large payloads made the application crash, so only measurements with a reduced number of concurrent messages were done for those payloads.



(The same graph with log scale on the Y-axis can be found in the annex.)

Analyzing the graph, several observations can be made about the network.

1. There are many factors that influence the network speed, which are directly related to t_r in the formula for speedup prediction. In facts, all tested factor influenced the speed at some point: payload size, number of concurrent messages and network load. The network load is directly dependent on the number of concurrent messages, but can be influenced by delaying some of them, so we have an independent measure of that factor by means of the "delay" factor, as explained above. Another factor, which was not directly influenced, is the total amount of data transiting through the network, i.e. the sum of the payloads of concurrent messages.
2. It is interesting to note that for large payload sizes, if the payload is divided into several messages sent by different nodes, the network transmit them faster than if a single node sends the whole payload.
3. DPS does not seem to manage its network load, which is clear by looking at the values for different delays for the same payload size. It is not apparent for small payloads though. Since network load is not managed, buffers might come into play. It follows that creating a model of the network might be hard. Fortunately, in the present case it is almost certain that the sieving algorithm won't behave as if the "distributed" delay was used, but lies in a model somewhere between the "constant" and "random" ones, i.e. all slaves performing the sieving finishes and send their primes at almost the same time.

Furthermore, the actual network model we need to build for the parallel sieve cannot be build by following one of the clear tendencies in the graph: Since we are interested in a varying number of nodes doing the sieve, we need to look at values for 1, 2, 4, ... concurrent messages and at the same time picking payload sizes of resp. 80000000, 40000000, 20000000, and so on. It could be possible to create a model by finding a function that uses all those parameters, but the model will become complicated.

Another point not covered by the speedup prediction model is the fact that messages have to be prepared before being able to be sent over the network: Prime numbers are encoded in a bitset during the sieve, which is sub-optimal in terms of size. So they must be *packed*, i.e. stored in a structure more fit for networked communications (in the current implementation: a list of integers). This packing takes some time, which should also appear in the model. The packing time t_{pac} has been measured to be 40 ns / prime, which amounts to approx. 3178 ms for $\pi(1.6e9)$ primes.

Furthermore, the assumption that the time needed to sieve the first part of the list up to $\sqrt{L}$ is $\sqrt{t_p}$ was wrong. The amount of time hitting numbers in the first part of the list has been measured to be less than 0.0001% of the total sieving time, which means that it is negligible.

Finally, the round trip time used to predict the speedup might not be accurate. RTT values between 5 and 80 ms have been observed. The new value retained for the latency t_l is 9 ms, which is the median value over a series of tests.

So the corrected model would be:

$$S_p = \frac{t_{serial}}{t_{par}} = \frac{t_p}{(n+1)t_l + N_t(n) + \frac{t_p + t_{pac}}{n}}$$

where $N_t(n)$ is a model of the network that predicts the transmission time for n messages of size $\frac{\pi(L)}{n}$.

As it is pretty hard to come up with a complete model for the network based on the network speed as a function of payload size and number of concurrent messages, a simpler function will be used, which is specific for $L = 1600000000$ and $n = 1..12$, and gives pretty accurate results. The chosen model is based on observations of actual transfer times and is given by:

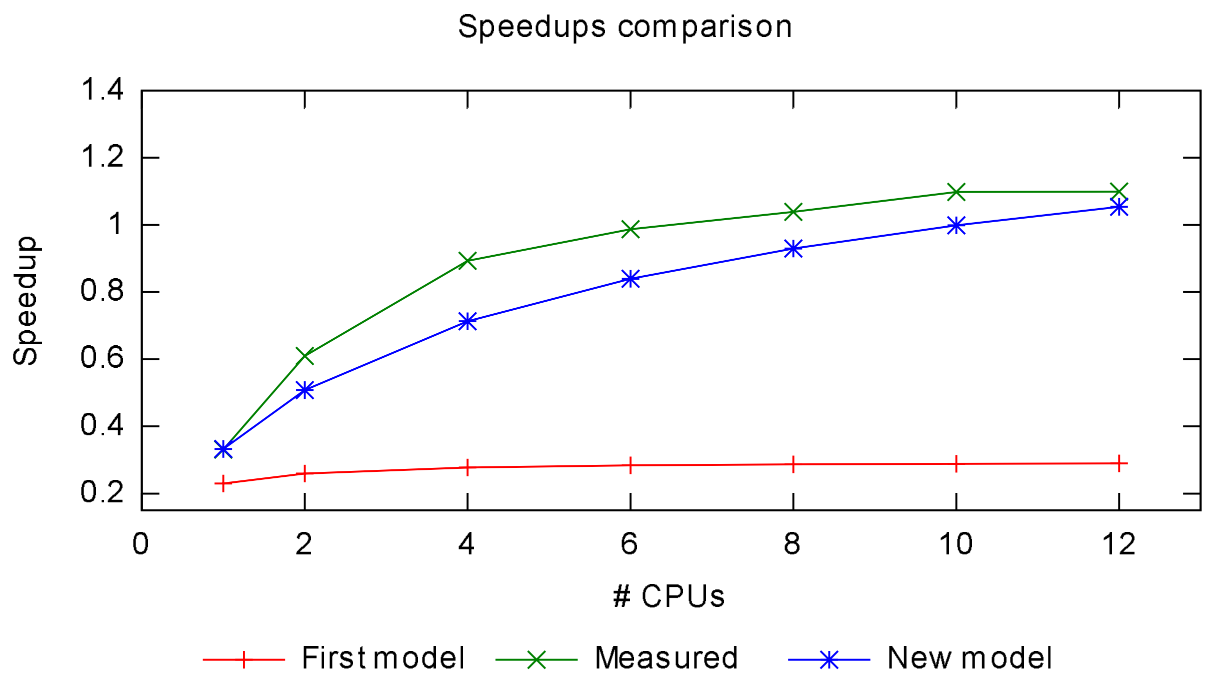
$$N_t(n) = t_r / \log_{10}(10n)$$

where $t_r = 20622 \text{ ms}$ is the time needed to transfer the whole list of primes in a single message, taken from new and more accurate measures. Note that due to testing conditions, an error of 2 seconds is likely, and this error alters enough the predicted speedup, which can be either above or under the measured speedup depending on the direction of the error.

The speedup values with the new formula are:

n	Execution time [ms]	Measured S_p	First model S_p	New model S_p
1	35996.38	0.331	0.230	0.333
2	19515.78	0.610	0.260	0.508
4	13327.40	0.893	0.278	0.713
6	12057.18	0.987	0.284	0.840

n	Execution time [ms]	Measured S_p	First model S_p	New model S_p
8	11454.46	1.039	0.287	0.930
10	10845.70	1.098	0.289	0.999
12	10828.62	1.099	0.290	1.054



The new speedup model is much more accurate than the first one as it takes into account a more complex network model. The difference that we can still see between the predicted speedup and the measured one comes from the fact that the network model is still not very accurate. However, creating an accurate network model would require lab conditions, with a dedicated network where most of its elements are tested and controlled. The actual testing conditions were pretty far from lab conditions in that a public computer room and the public network were used.

The new model is nevertheless accurate enough to show a clear similarity between the curves of its predicted speedup and the measured one, not only in shape but also in values.

Conclusion

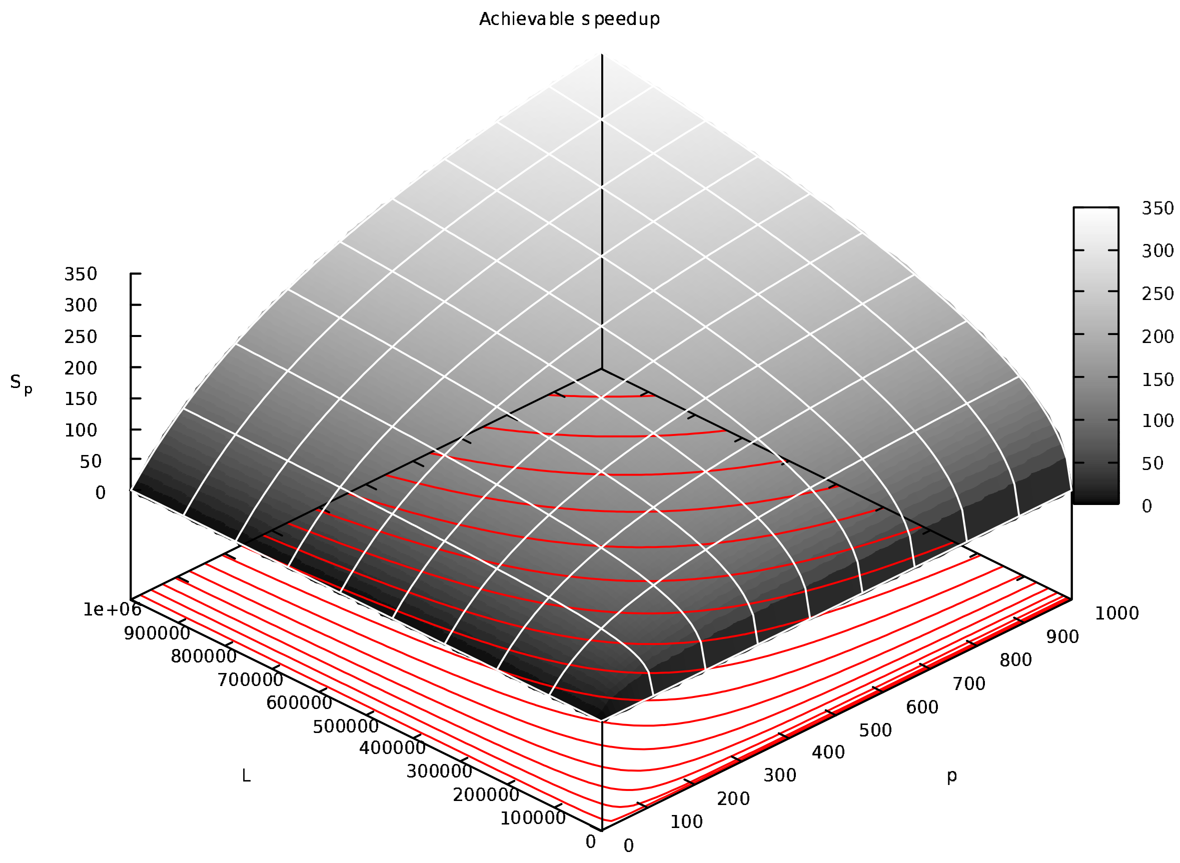
DPS is a neat attempt to get rid of errors in parallelized code by providing stronger guarantees than MPI does. However, it also introduces some rigidity that might hinder performances as it would require some tricks to implement communication channels that might be simpler to implement on MPI. One example is that the split operation requires the corresponding merge operation to receive exactly the same number of messages that the split operation sent. This prevents the easy implementation of simple streaming protocols, and might load the network more than required. Since the network is generally the bottleneck of a parallelized application (it was in this case), using DPS might not be always the best choice over MPI for some kinds of problems.

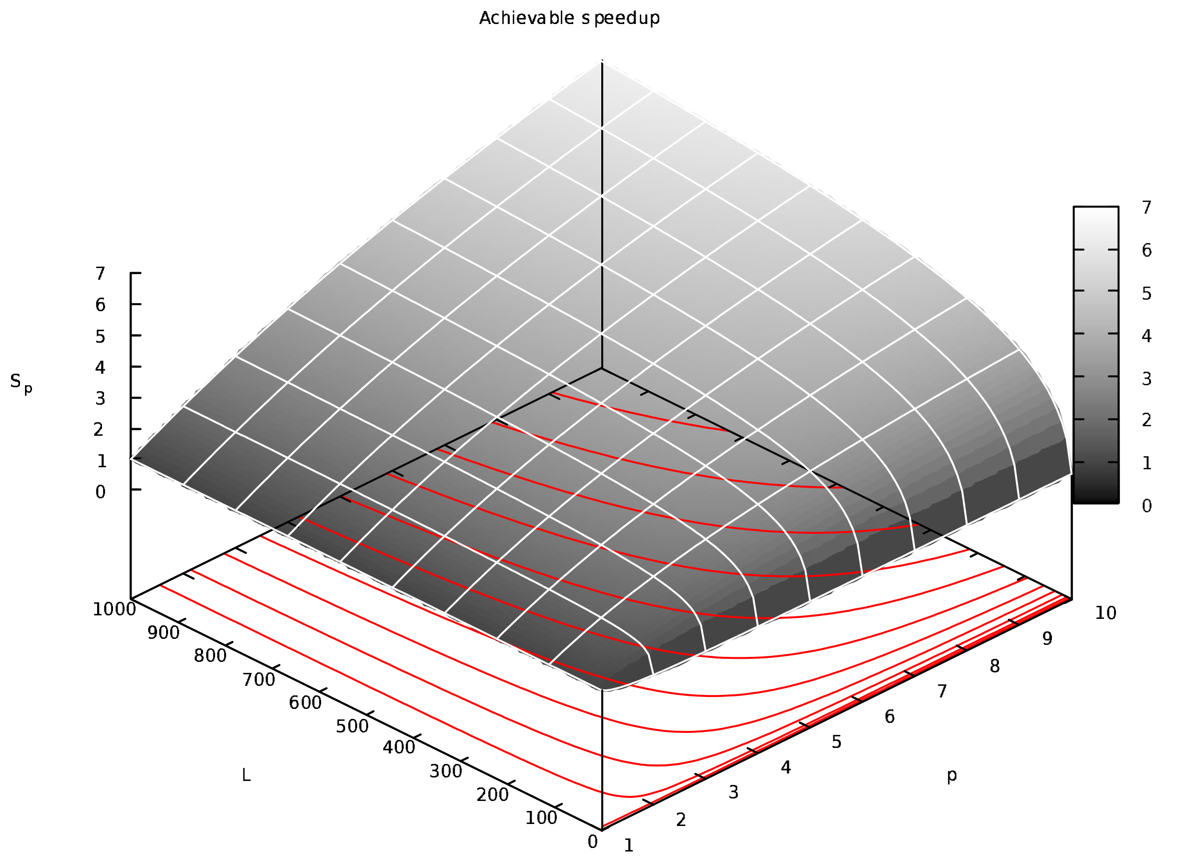
The parallelization strategy tried to cope with the network bottleneck by reducing communications to the strict minimum. However, as extended investigations of the network showed afterwards, having more balanced communication might have been a better choice even at the cost of a little more communications.

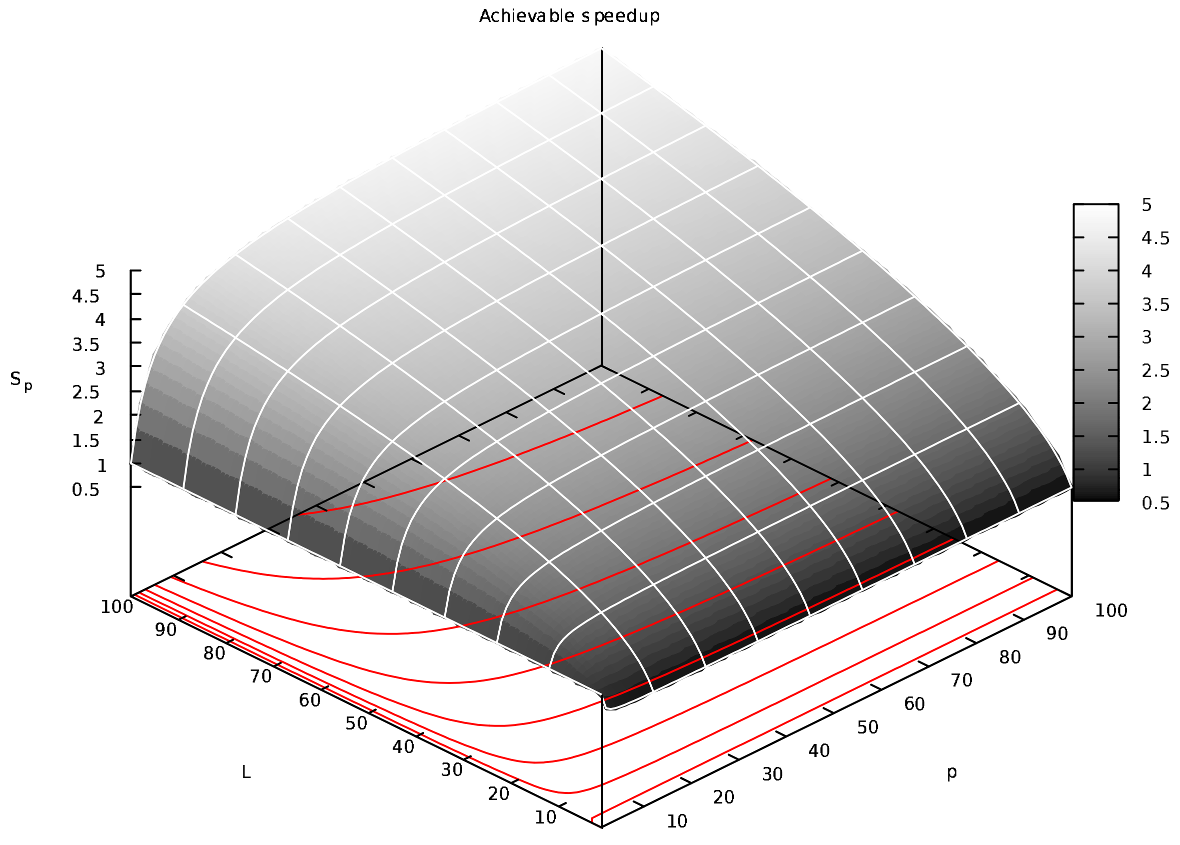
Annex

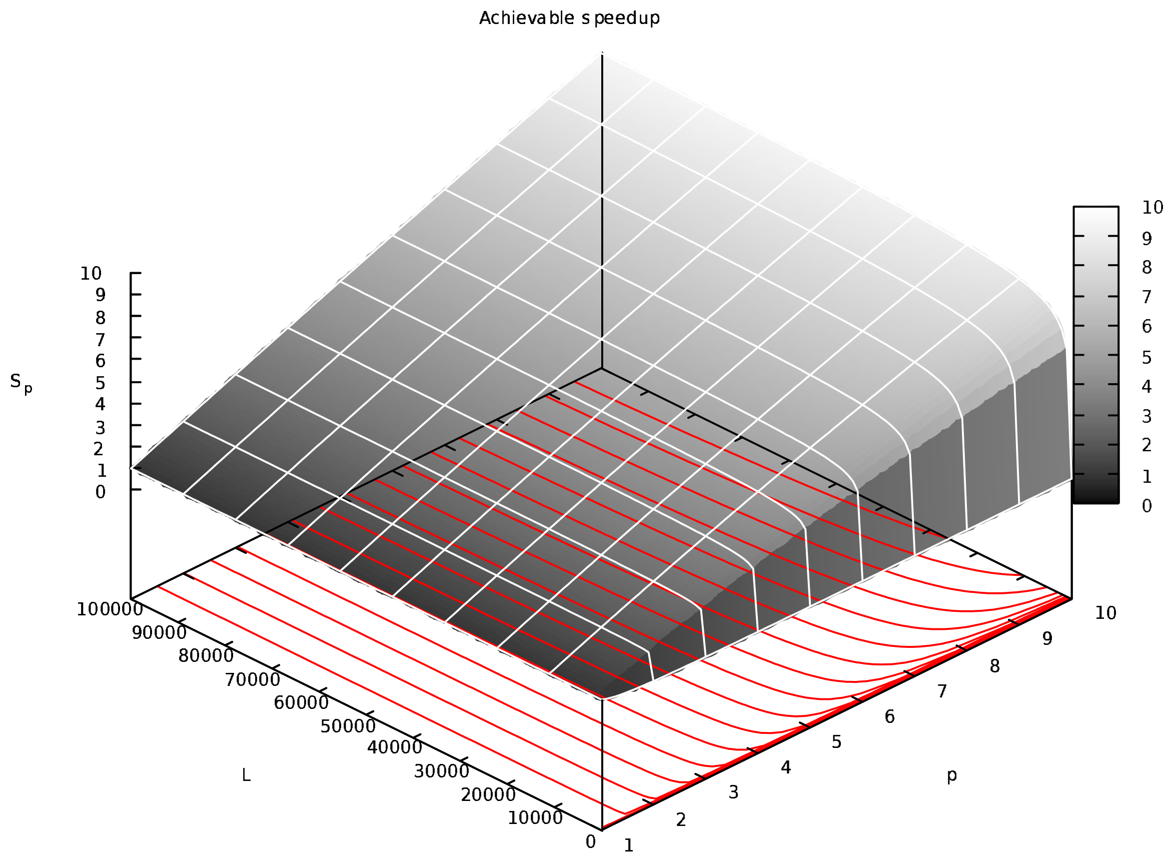
“ much report so amaze WOW! ”
— Doge

Achievable speedup: Graphs

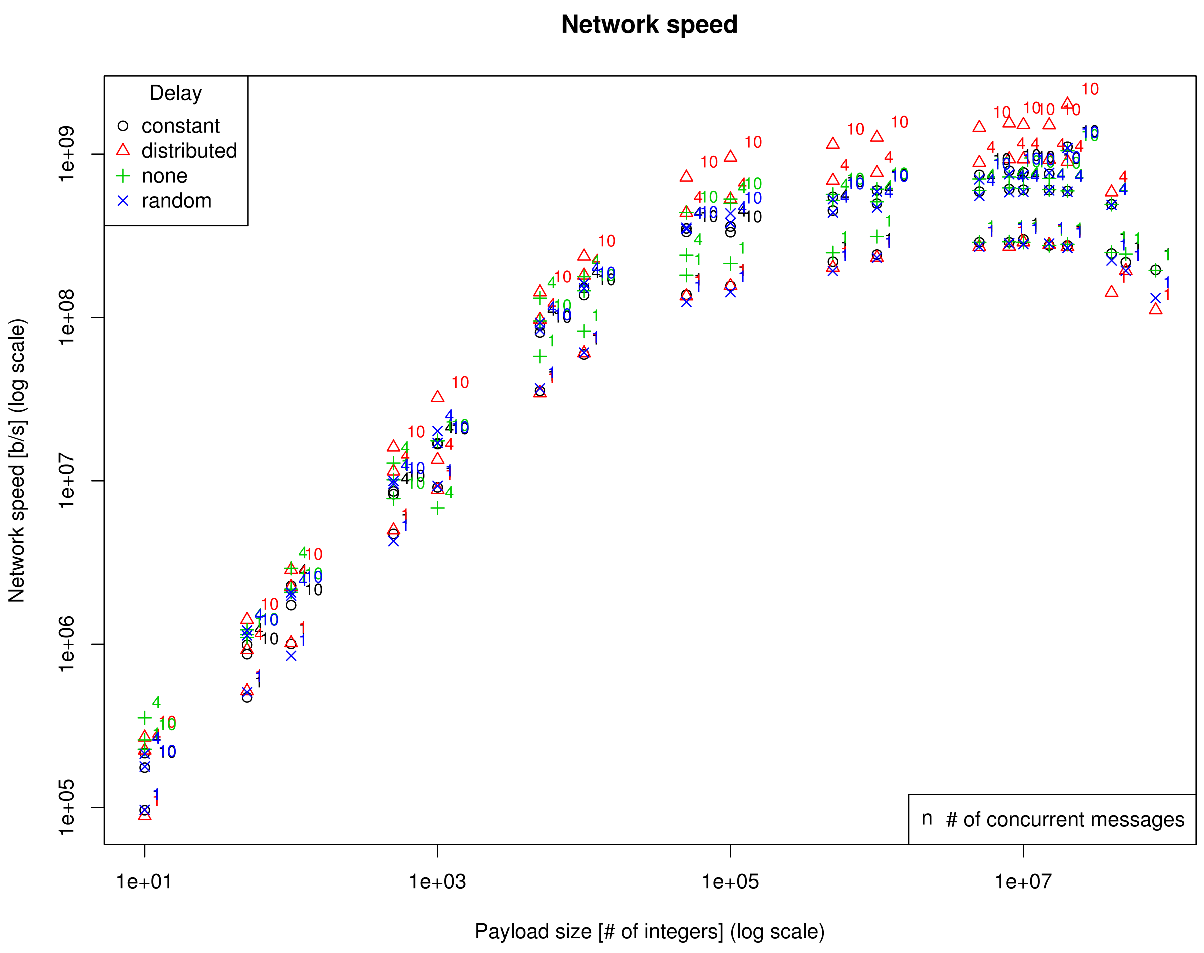



Achievable speedup
S_p
7
6
5
4
3
2
1
0
L
1000
900
800
700
600
500
400
300
200
100
0
p
1
2
3
4
5
6
7
8
9
10
7
6
5
4
3
2
1
0


Achievable speedup
S_p
5
4.5
4
3.5
3
2.5
2
1.5
1
0.5
L
100
90
80
70
60
50
40
30
20
10
p
10
20
30
40
50
60
70
80
90
100
5
4.5
4
3.5
3
2.5
2
1.5
1
0.5



Corrected model: Additional graphs



1. Sieve of Eratosthenes on Wikipedia (http://en.wikipedia.org/wiki/Sieve_of_Eratosthenes#Algorithm_complexity)
2. Prime number on Wikipedia (http://en.wikipedia.org/wiki/Prime_number#Number_of_prime_numbers_below_a_given_number)