U.PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Distributed Prime Sieve in Heterogeneous Computer Clusters

Carlos M. Costa, Altino M. Sampaio, and Jorge G. Barbosa

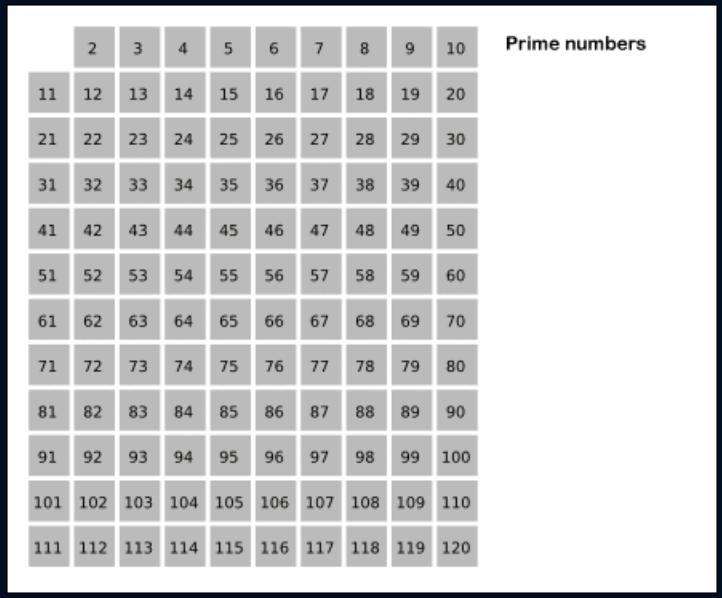
{carlos.costa,pro09002,jbarbosa}@fe.up.pt

Context

- ***Prime number***
  - Natural number greater than 1 that has no positive divisors other than 1 and itself
- ***Prime number sieve***
  - Algorithm able to find prime numbers up to a given limit
- ***Prime number applications***
  - Cryptography
    - Asymmetric public key algorithms (RSA)
    - Deffie-Hellman key exchange algorithm
  - Hash functions
  - Pseudo-random number generators

Prime number sieves

- ***Sieve of Atkin***
  - O(n / (log log n) operations
- ***Sieve of Sundaram***
  - O(n log n) operations
- ***Sieve of Eratosthenes***
  - O(n log log n) operations



Sieve of Eratosthenes animation

Algorithm 1 Sieve of Eratosthenes

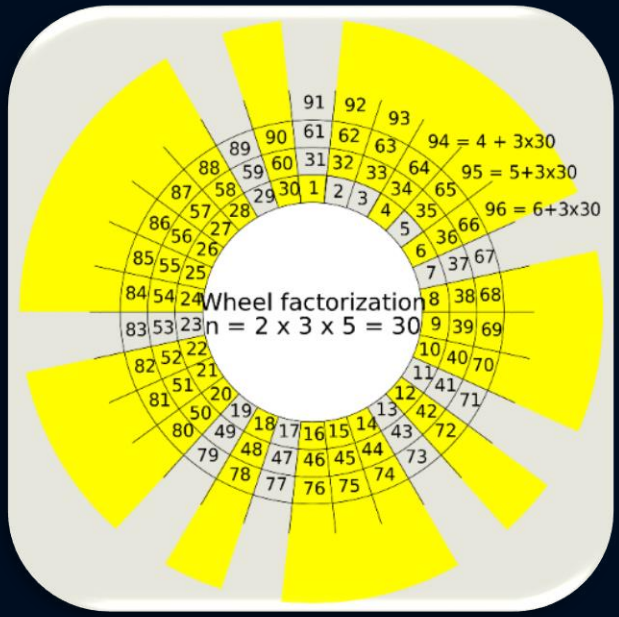
1: **Input:** an integer $n > 1$
2: ▷ Let P be an array of Boolean values, indexed by integers 2 to n, initially all set to true

3: **for** $i \in \{2, 3, 4, ..., n\}$ **do**
4: **if** $P[i] == true$ **then**
5: **for** $j \in \{2i, 3i, 4i, ..., n\}$ **do**
6: $P[j] = false$

7: ▷ Now all i, such that P[i] is true, are prime numbers

Wheel factorization

- Sieving optimization that can cross off multiples of several sieving primes.
- A 210 (2*3*5*7) wheel can remove as much as 77% of composite numbers.



Algorithm 2 Wheel Factorization

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

for $i \in P$ **do**
    for $j \in \{2i, 3i, 4i, ..., m\}$ **do**
        $C[j] = true$

▷ Now for all x numbers such as k=(x modulus m) in [1, m], if C[k] is false, then x is a probable 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

Sequential sieve with fast marking

- *Improvements over original Sieve of Eratosthenes:*
  - Sieving can start at i^2 (instead of 2i) and stop at $\sqrt{n}$ (instead of n)
    - Avoids marking the same composite number multiple times
  - All even numbers are skipped from sieving
    - Reduces sieving computations and memory consumption by 50%
  - Uses additions instead of multiplications to reduce CPU cycles
    - According to the Intel optimization manual (table C-20a. General Purpose Instructions), an add instruction has a latency of 1 and a throughput of 0.33, while an imul instruction has a latency of 3 and a throughput of 1. Which means an add instruction can be much faster than an imul when using pipelining.

Algorithm 3 Sequential Prime Sieve Using Fast Marking

1: **Input:** an integer $n > 1$
2: ▷ Let C be an array of Boolean values representing the odd numbers > 1, initially all set to false

3: **for** i $\in \{3, 5, 7, ..., \sqrt{n}\}$ **do**
4: **if** $C[(i-3)/2] == false$ **then**
5: **for** $j \in \{i^2, i^2 + 2i, i^2 + 4i, ..., n\}$ **do**
6: $C[(j-3)/2] = true$

7: ▷ Now all the positions in array C still marked as false, represent prime numbers

Sequential sieve with block decomposition

Algorithm 4 Sequential Prime Sieve Using Fast Marking With Block Decomposition Optimized For Space

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

5: calculatePrimesInBlock(3, 3 + bs)                          ▷ First block
6: for b ∈ {1, 2, 3, ..., nb} do                              ▷ Remaining blocks
7:     a = b × bs
8:     b = a + bs  ▷ Last block should have an upper limit of n + 1, because they are
   sieved in range [a, b[
9:     removePrimesFromPreviousBlocks(a, b)
10:    calculatePrimesInBlock(a, b)
11: ▷ Now all the positions in array C still marked as false, represent prime numbers

12: function calculatePrimesInBlock(a, b)                     ▷ Sieves in range [a, b[
13:    for i ∈ {a, a + 2, a + 4, ..., √n} do
14:        if C[i] == false then
15:            for j ∈ {i², i² + 2i, i² + 4i, i² + 6i, ..., b}  do      ▷ Not including b
16:                C[j] = true

17: function removePrimesFromPreviousBlocks(a, b)             ▷ Sieves in range [a, b[
18:    for i ∈ {0, 1, 2, ..., k}  do  ▷ k not exceeding the position in C associated with
   √n
19:        if C[i] == false then
20:            p = closest prime multiple of number associated with position i in rela-
   tion to a
21:        for j ∈ {p, p + 2p, p + 4p, ..., b}  do                 ▷ Not including b
22:            C[j] = true
```

Sequential sieve with block decomposition

- *Improvements over previous algorithm:*
  - Block decomposition of the sieving range greatly improves cache usage, because instead of picking a prime and removing all it's multiples, it removes all multiples of the sieving primes block by block. This achieves better performance because the sieving primes can remain in cache memory longer, reducing cache misses.
  - The reduction of cache misses improves performance because cache memory access is much faster than main memory (RAM) access.

Sequential sieve with block decomposition

- *Other improvements*
  - Line 20 of algorithm 4 spends a considerable amount of time computing the closest prime multiple to the beginning of the block. To avoid this, each sieving prime is associated with its last multiple.
  - Moreover, in line 18, we can see that the bitset of numbers is searched again to select the sieving primes. To avoid this extensive loop, a data structure with the sieving primes doubles and their last used multiples is kept in memory.
  - In algorithm 4, all numbers are stored in memory instead of only the odd ones. Both memory storage versions were implemented to test the overhead of the memory offset computations. The results show that for small ranges, direct memory access has better performance, but for large ranges, storing only the odd numbers is better. This is due to the fact that cache usage for large ranges has more impact than offset computations.
  - A modified 210 wheel sieved was also added to reduce the number of composites to check. In this case the wheel is used to compute the next composite number that needs to be checked (instead of being used to pre-sieve the numbers bitset).

Parallel segmented sieve

- Given that the sequential sieve was already using block decomposition, the parallelization was achieved by assigning different groups of blocks to different threads.

- This lead to the implementation of a segmented sieve, in which only the segment (group of blocks) that is being sieved is stored in memory (no longer stores all odd numbers). This allows the computation of primes to near 2^{64} with minimal memory requirements.

Distributed sieve

- Computing prime numbers to near 2^{64} can take a very long time.
- To speedup the processing, a computer cluster can be used.
- But some clusters can have very heterogeneous hardware.
- To solve this problem, it was implemented a dynamic scheduling algorithm to distribute segments across cluster nodes.
- This is a hybrid implementation in which OpenMPI is employed to coordinate the workload and OpenMP is used inside each cluster node.

Distributed sieve

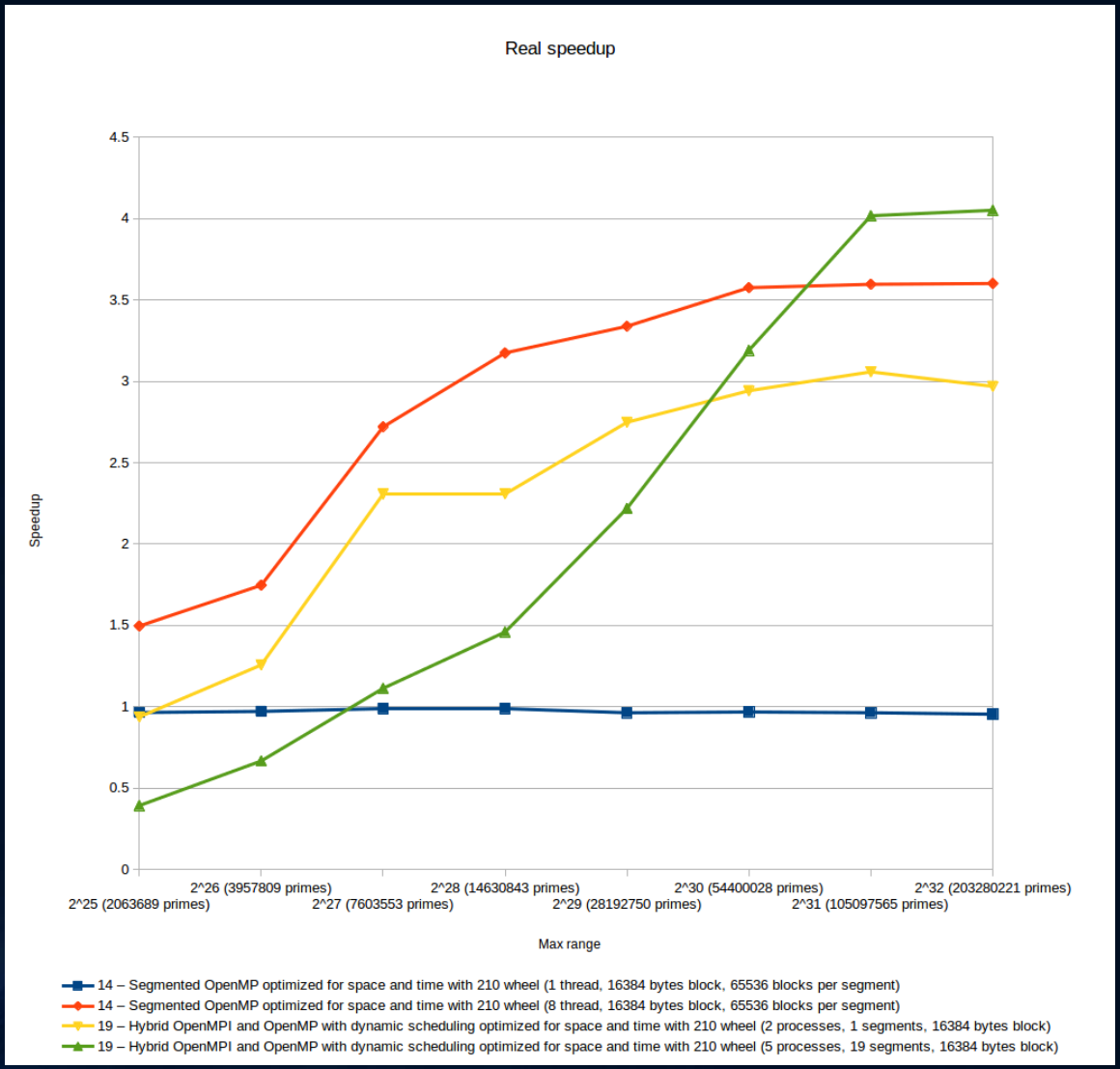
- ***Dynamic scheduling algorithm:***
  - Start a master process with a queue of segments for each cluster node.
  - Spawn slave processes (that will process the segments)
  - Each slave process asks the master for a new segment until all of then are processed
  - Every time the master process receives a request for a new segment, it replies with:
    - Last segment from the queue associated to the slave process
    - If queue is empty, selects segment from the largest queue (associated to other slave process)
    - Removes selected segment from the queue
  - After all segments are processed and results are collected, master replies to slaves that all work is done and they can terminate execution

- ***System performance***
  - By employing the dynamic scheduling algorithm, cluster nodes with high processing capability can keep contributing to the workload even after they have finished the processing of the segments that were initially allocated to then.
  - This way the system can adapt to different cluster topologies and avoids clusters node bottlenecks (in a static scheduling system, the processing time would be equivalent to the slowest cluster node).

Testing platforms

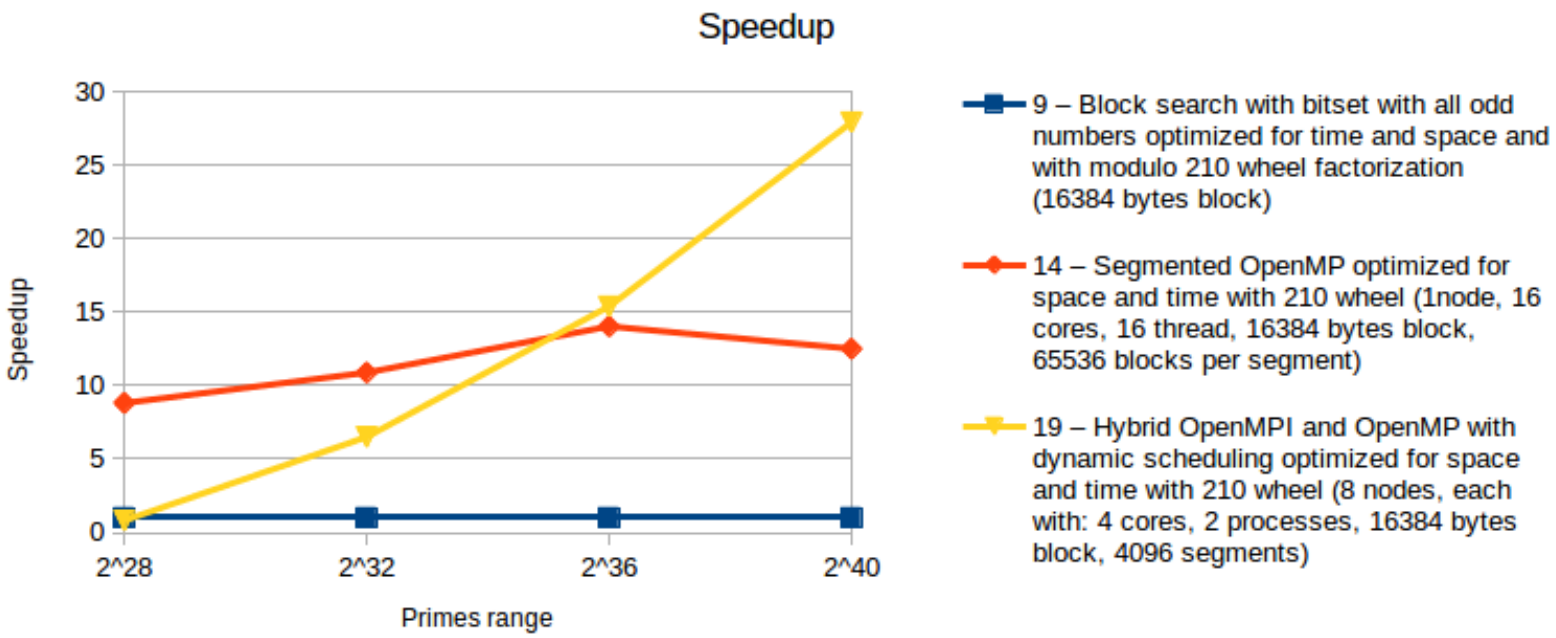
- The sieving implementations were tested in two different platforms:
  - ***Two laptops:***
    - Clevo P370EM
      - Intel i7-3630QM (quad core processor with Hyper-Threading – 2400 -> 3400 MHz clock rate)
      - 16 GB of RAM DDR3 1600 MHz
    - Asus G51J
      - Intel i7-720QM (quad core processor with Hyper-Threading – 1600 -> 2800 MHz clock rate)
      - 4 GB of RAM DDR3 1066 MHz
  - ***Computer cluster (Avalanche - Grid FEUP):***
    - Each cluster node with:
      - 16 cores -> two Intel Xeon E5-2450 (2100 -> 2900 MHz clock rate)
      - 64 GB RAM DDR3 1600 MHz
- The source code is available at:
  - https://github.com/carlosmccosta/Distributed-Prime-Sieve

Results - speedup



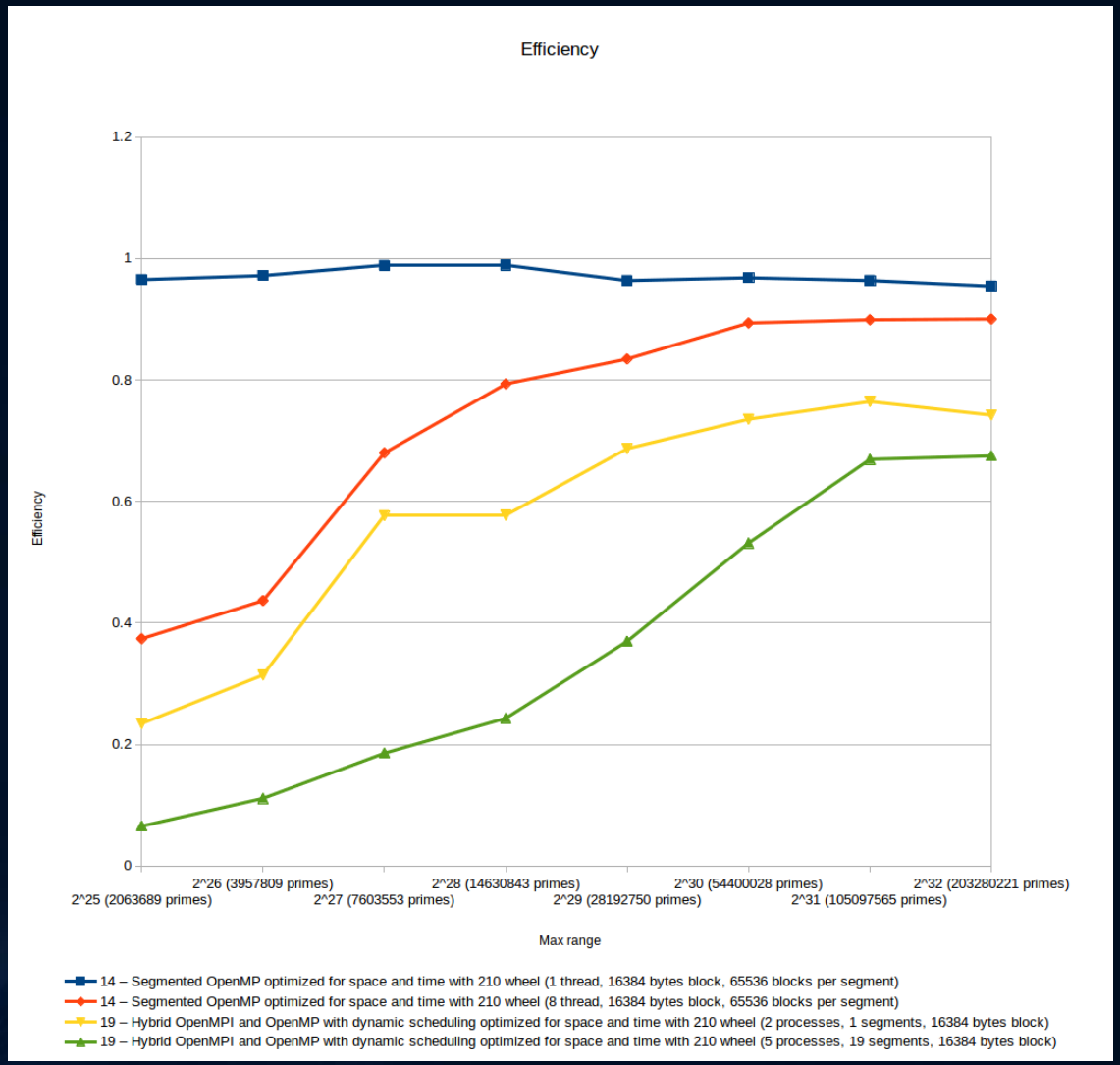
Speedup achieved with 2 laptops

Results – cluster speedup



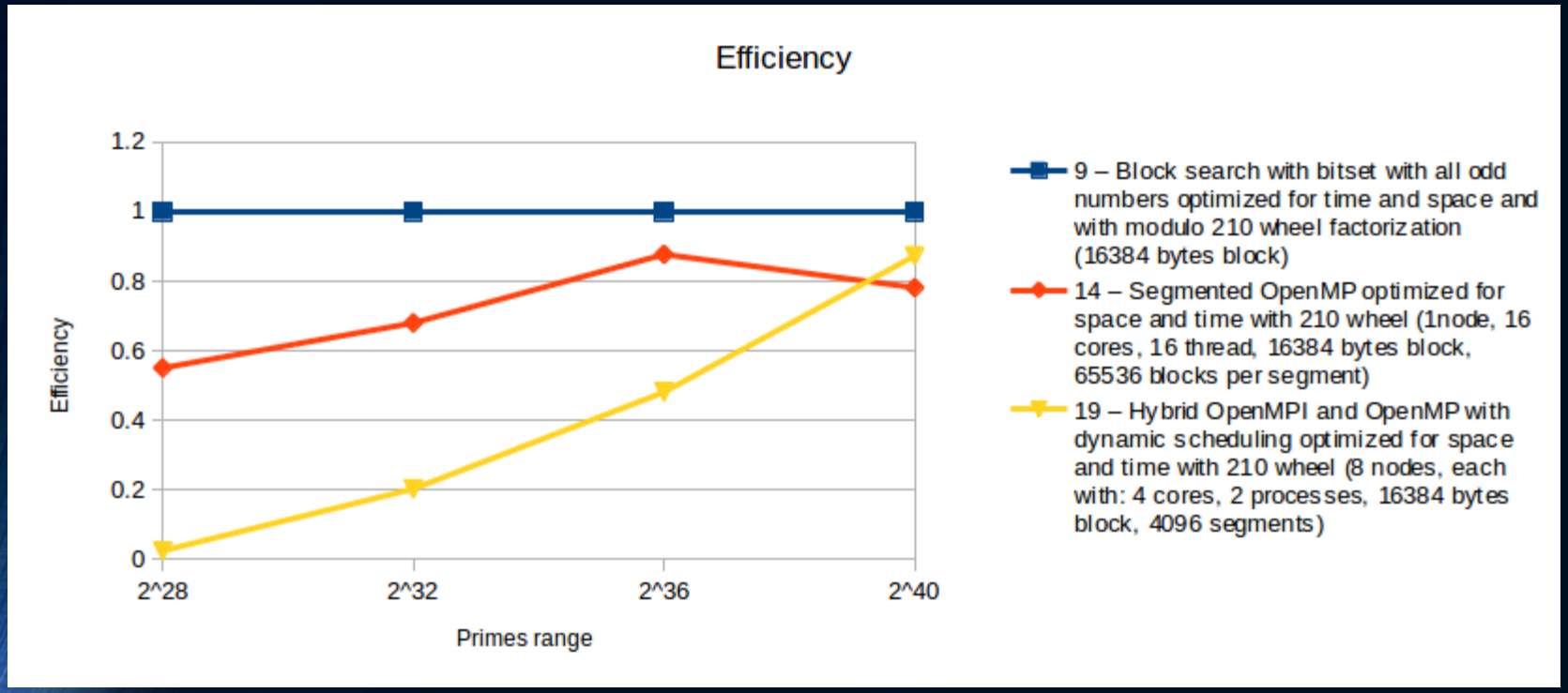
Speedup achieved with a computer cluster

Results - efficiency



Efficiency achieved with 2 laptops

Results – cluster efficiency



Efficiency achieved with a computer cluster

Thank you!

Questions?