Parallel Scalar Multiplication of Elliptic Curve Points

CSE 633
George Gunner
March 28, 2017
Professor: Dr. Russ Miller

Motivation

- Elliptic curves are commonly used in public-key cryptography
  - Digital Signatures
  - Symmetric Key Exchange
- Scalar multiplication of points on a curve is the most costly operation performed

Background – Finite Fields

- A **finite field** on p^n is the set of integers in $\{0, p^n\}$, where p is a prime and n is some positive integer
- Two types of finite fields are of interest
  - ►**Prime fields**, where n=1
    - ►Uses regular arithmetic, modulo a prime p
  - ►**Binary fields**, where p=2
    - ►Uses polynomial arithmetic, modulo an irreducible polynomial p

Background – Polynomial Arithmetic on a Finite Field

- The binary number $b_{n-1}||b_{n-2}||...||b_0$ represents the polynomial $\sum_{i=0}^{n-1} b_i x^i$
- Arithmetic operations defined in terms of polynomials, with coefficients computed modulo 2
- Squaring is efficiently achieved on binary fields
  - ►Inserting a 0 between consecutive bits of a number yields its square
  - ►O(n) time compared to $O(n^2)$ time for multiplication

Background – Non-Adjacent Forms

- A **non-adjacent form** (NAF) is an alternate representation for an integer k such that $k = \sum_{i=0}^{l-1} k_i 2^i$ where $k_i \in \{0, \pm 1\}$ and no two consecutive digits are nonzero
- A **windowed NAF** (wNAF) for k is the representation $k = \sum_{i=0}^{l-1} k_i 2^i$ such that $|k_i| < 2^{w-1}$ for a window size w, k_i is 0 or odd, and for any w consecutive digits, at most one is nonzero

Elliptic Curves

- General elliptic curve equation

$$y^2 + axy + by = x^3 + cx^2 + dx + e$$

- Two general types of curves are of interest:
  - **Prime curves:** $y^2 = x^3 + ax + b$
  - **Binary curves:** $y^2 + xy = x^3 + ax^2 + b$
    - Binary curve with certain properties called **Koblitz curves** allows field squaring to replace less efficient point doubling in scalar multiplication, which will be particularly suitable for a parallel implementation

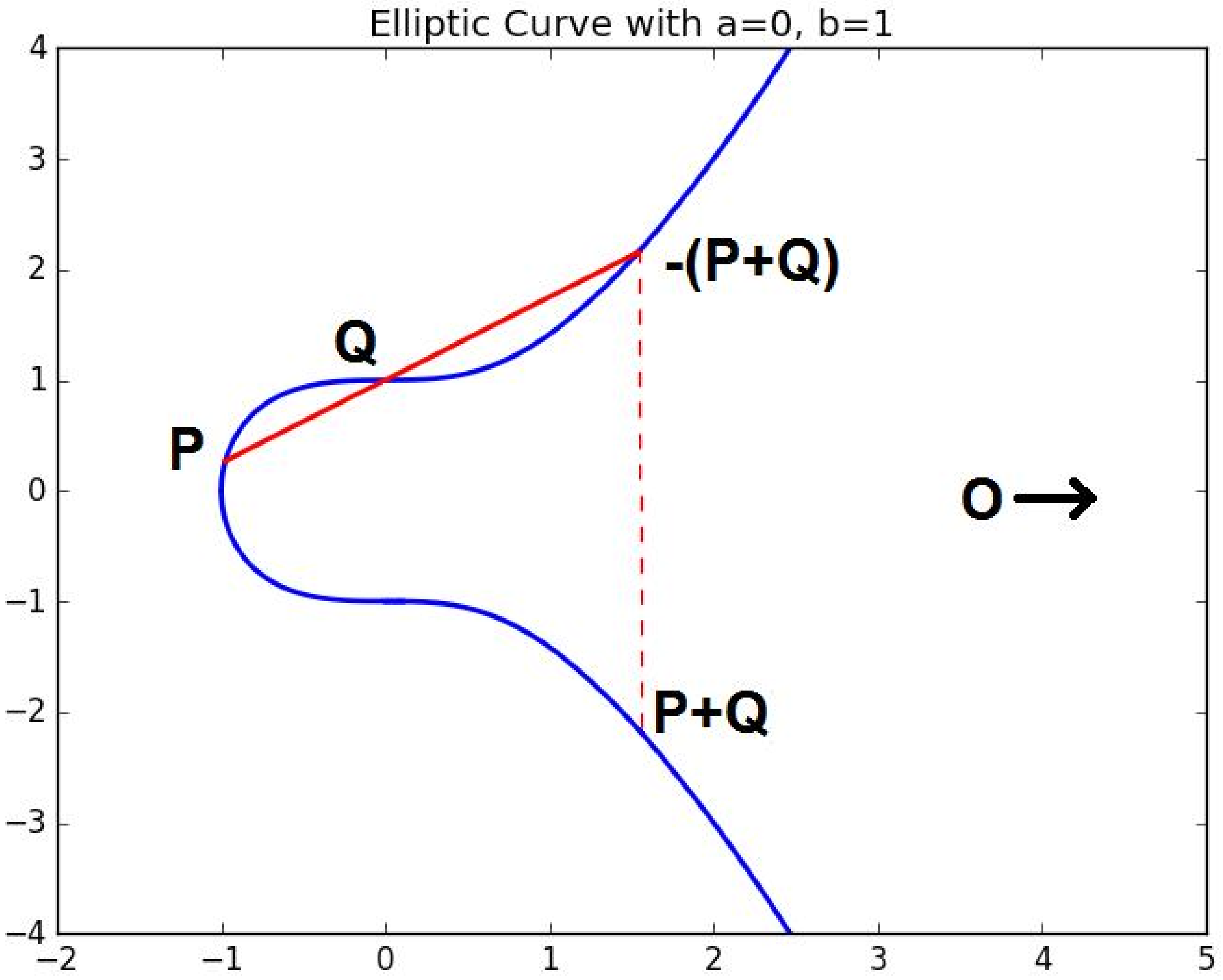
Elliptic Curve Coordinates

- Natural to think of curves and points in terms of **affine coordinates** (x, y) for geometric intuition and to describe algebraic properties
- Computation often more efficient when projecting on a higher dimensional space
  - ie. **Projective coordinates** (x, y, z) from the affine coordinates (x/z, y/z)
- **Compressed coordinates** can be used to transmit points with minimal size
  - The x affine coordinate and a bit signifying the corresponding y value to use

Prime Curves

- For a prime curve, if we have nonzero determinant $4a^3+27b^2 \neq 0 \pmod{p}$ we can define addition of points and form an abelian group:
  - Closure
  - Associativity
  - Commutativity
  - Identity Element (O, “point at infinity”)
  - Inverse Element (-P for a point P)
- Two basic point operations: point addition and point doubling

Prime Curves – Geometric Intuition



Prime Curves – Scalar Multiplication

- Basic approach is the "**double-and-add**" method to compute kP given $k=b_{n-1}||b_{n-2}||...||b_0$ the binary representation of k

```
Input: P, $k=b_{n-1}||b_{n-2}||...||b_0$
Output: Q = kP
    Q=0
    For i from 0 to n-1
        Q=2Q
        If $b_i=1$ then Q=Q+P
    Return Q
```

Prime Curves – Scalar Multiplication

- More efficient by a constant factor to use a **wNAF method**:

```
Input: P, k
Output: Q =kP
    Compute wNAF of $k=\sum_{i=0}^{l-1} k_i 2^i$
    Precompute jP for j={1, 3, ..., $2^{w-1}$-1}
    Q=O
    For i from l-1 to 0
        Q=2Q
        if $k_i$>0 then Q = Q+$k_i$P
        else if $k_i \neq 0$ then Q=Q-$k_i$P
    Return Q
```

Binary Curves

- Binary curves require b≠0 to define an abelian group
- General binary curves use same algorithms as prime curves to compute scalar multiplication
- Koblitz curves have a property which allows more efficient computation of scalar multiplication
  - Given a point (x, y) on the curve, (x^2, y^2) is also on the curve, and this can be used to replace point doubling by field squaring

Koblitz Curves – τ Operator

- Define the **τ operator** such that $\tau(x, y)=(x^2, y^2)$ and $\tau O=O$
  - Recall that squaring on a finite field over 2^m can be computed efficiently
- Given a point P, we have $(\tau^2+2)P=\mu\tau P$ where $\mu=(-1)^{1-a}$ where τ^j is the τ operator applied j times
- From the above result, we can consider τ as the complex number satisfying $\tau^2+2=\mu\tau$
  - $\tau=(\mu+\sqrt{-7})/2$
  - Allows a scalar to be expressed in terms of τ

Koblitz Curves – wτNAF

- A number $\kappa = r_0 + r_1\tau$ on the ring $\mathbb{Z}[\tau]$ has a wτNAF representation $\kappa = \sum_{i=0}^{l-1} u_i \tau^i$ where $u_i = \{0, \alpha_{\pm 1}, \alpha_{\pm 3}, \ldots, \alpha_{\pm(2^{w-1}-1)}\}$
  - ►The $\alpha_i = \beta_i + \gamma_i \tau$ for each window size are chosen so that each precomputed point requires at most a single point addition and a single application of τ during precomputation

Koblitz Curves – wτNAF

- Computing the wτNAF representation for a scalar results in a representation that is too long in general – ~2m digits for an m-bit scalar
- To get a suitable length representation, find a complex number ρ' such that ρ'≡k (mod δ) where $\delta=(\tau^m - 1)/(\tau - 1)$ using partial modulo reduction
  - ►The equivalence ensures that ρ'P≡kP, where ρ' has a sufficiently short representation bounded in length by m+a+3
  - ►High probability of finding ρ, the shortest representation based on a chosen parameter C

Koblitz Curves – wτNAF Multiplication

- The **wτNAF method** is as follows:

```
Input: P, $\rho'=\sum_{i=0}^{l-1} u_i \tau^i$
Output: $Q=\rho'P=kP$
    Precompute $P_u=\alpha_u P$ for $u\in\{\pm1, \pm3, \ldots, \pm(2^{w-1}-1)\}$
    $Q=O$
    For I from l-1 to 0
        $Q=\tau Q$
        If $u_i\neq0$ then
            Let u be such that $\alpha_u=u_i$ or $\alpha_{-u}=-u_i$
            If $u_i>0$ then $Q=Q+P_u$
            Else $Q=Q-P_u$
    Return Q
```

Securing Against Side Channel Attacks

- The computation methods considered so far depends on the input scalar
- Adversaries capable of side channel attacks, such as a timing attack, can exploit this to learn secret information
- Using a **Montgomery method** modifies multiplication algorithms in a simple way to take fixed time independent of the input scalar size
  - Performance decreased by a constant factor
  - Montgomery ladder used for prime curves
  - Dummy variable used for Koblitz curves

Parallel Scalar Multiplication

- Let k be an n-digit long scalar and suppose we have 2^m processors with $2^m \leq n$
  - In binary representation for prime curves
  - In wτNAF representation for Koblitz curves
- We can break k into 2^m parts:

$$k = k_{2^m}^m \| k_{2^m-1}^m \| \dots \| k_1^m$$

- Then compute the smaller products in parallel

$$k_{2^m}^m P, k_{2^m-1}^m P, \dots, k_1^m P \Rightarrow Q_{2^m}^m, Q_{2^m-1}^m, \dots, Q_1^m$$

Parallel Scalar Multiplication

- From these smaller products, we can then recursively recombine the Q values to obtain kP
  - For prime curves, we recombine via doubling

$$Q_{j/2}^{i}=2^{|k_{j-1}^{i+1}|}Q_{j}^{i+1}+Q_{j-1}^{i+1}$$

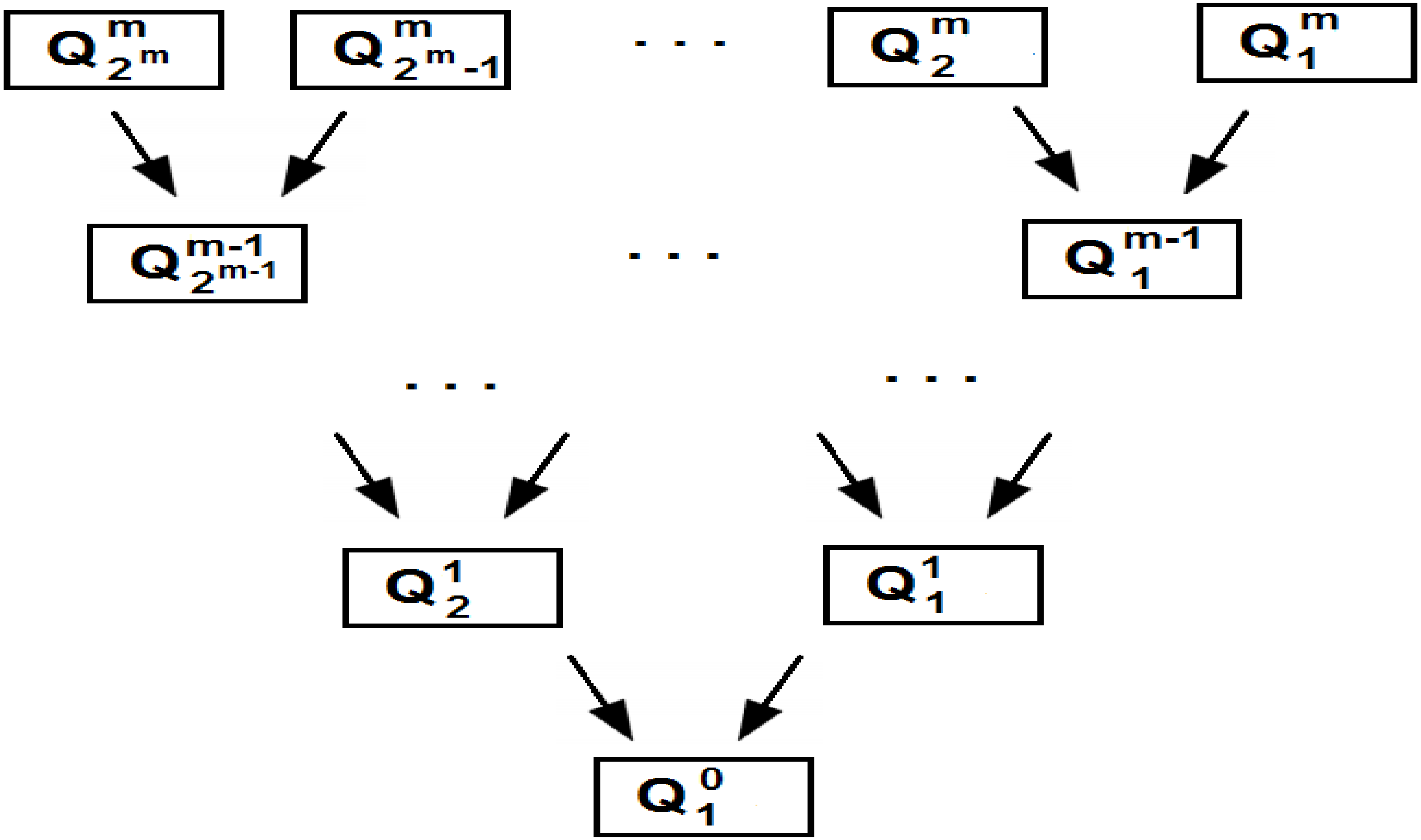
  - For Koblitz curves, we recombine via τ

$$Q_{j/2}^{i}=\tau^{|k_{j-1}^{i+1}|}Q_{j}^{i+1}+Q_{j-1}^{i+1}$$

  - We have Q_1^0=kP
  - In general denote the recombination function as $Q_{j/2}^{i}=f\left(Q_{j}^{i+1},Q_{j-1}^{i+1}\right)$

Parallel Scalar Multiplication

- The recombination steps can be represented as a tree:



$Q^{m}_{2^m}$ $Q^{m}_{2^m-1}$ . . . Q^{m}_{2} Q^{m}_{1}

$Q^{m-1}_{2^{m-1}}$ . . . Q^{m-1}_{1}

. . . . . .

Q^{1}_{2} Q^{1}_{1}

Q^{0}_{1}

Parallel Scalar Multiplication

- Putting this together, the algorithm for parallel scalar multiplication is:

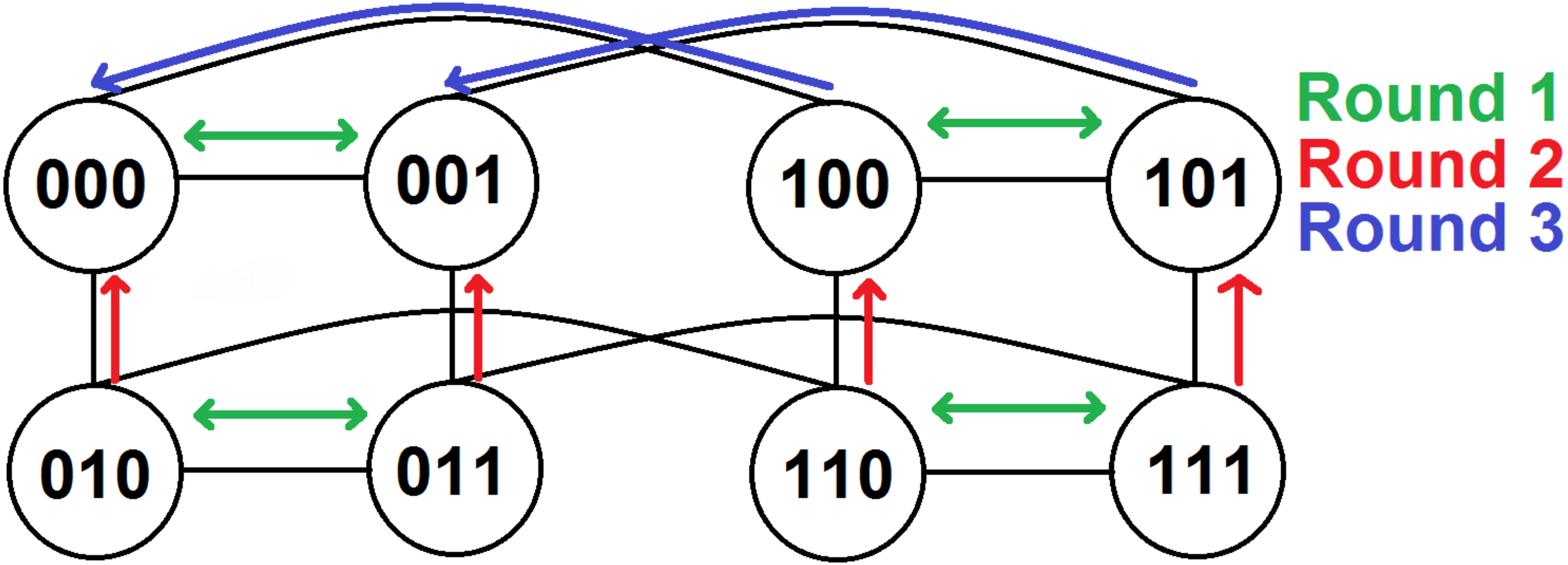
```
Input: $P, k = d^n_{2^n} || d^n_{2^n - 1} || \ldots || d^n_1$
Output: $Q = kP$
    $Q = O$
    for i=1 to $2^n$, in parallel
    $Q^n_i = d^j_n P$
    For i=n-1 to 0
        For j=i+1 to 1, in parallel
        $Q^i_{j/2} = f\left(Q^{i+1}_j, Q^{i+1}_{j-1}\right)$
    Return $Q^1_0$
```

Parallel Scalar Multiplication

- Hypercube and tree topologies naturally suited
    - Tree suitable for pipelining
    - Hypercube could interweave multiple multiplications together
- A linear structure can also be used, but has worse running time than a hypercube or tree
    - Better asymptotic throughput than a tree
- Higher throughput with no speedup can also be achieved by a simple division of processors, with results distributed across processors

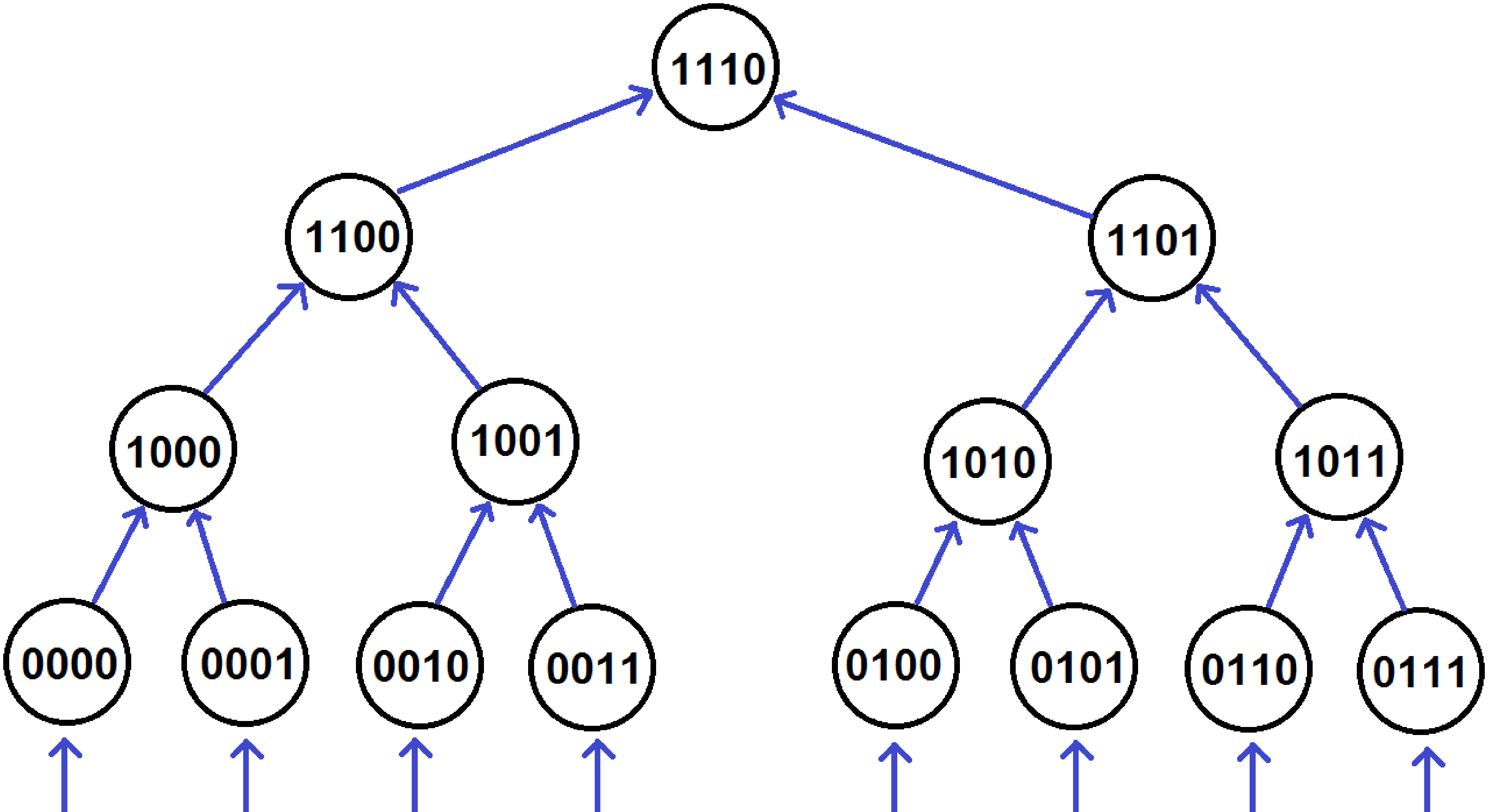
Parallel Scalar Multiplication

- Messages exchanged in a hypercube with 2 interweaved multiplications and 8 processors



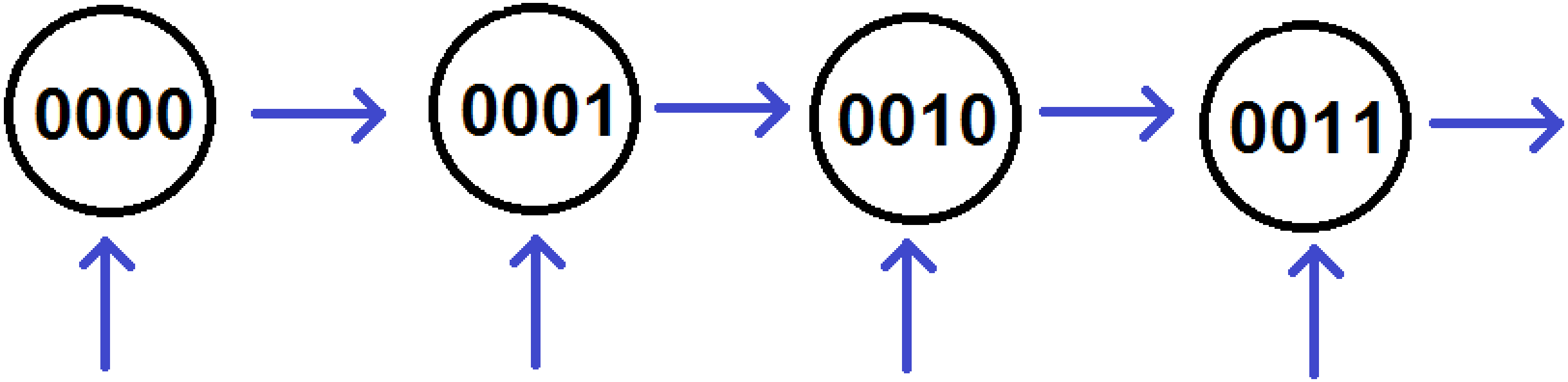
Parallel Scalar Multiplication

- Messages exchanged while pipelining multiplications in a tree



Parallel Scalar Multiplication

- Messages exchanged while pipelining multiplications in a linear array



Asymptotic Running Time - Sequential

- In terms of point additions (A), point doublings (D), field size (m), and processors (p)
    - The tau operator is asymptotically more efficient than other point operations
- For a prime curve, **m** point doublings and on average **m/(1 + w)** point additions are required for a window size of w with 2^{w-2} precomputation work
- Asymptotic running time is thus:
    - General: O(mD+mA)
    - Koblitz: O(mA)

Asymptotic Running Time - Hypercube & Tree

- First round computes multiplication of size m/p sequentially, requiring O(m/p D + m/p A) time
- The i-th (of log p total) recombination round requires 2^i m/p point doublings and one addition
- Theoretical optimal speedup using m/4 processors
- Asymptotic parallel running time is thus:
  - General: O(mD + (m/p + log p)A) when $2^n<m/4$
O(mD + (log m)A) when $2^n \geq m/4$
  - Koblitz: O((m/p + log p)A) when $2^n<m/4$
O((log m)A) when $2^n \geq m/4$

Asymptotic Running Time - Linear

- Each processor computes in parallel a sequential multiplication of size m/p, requiring O(m/p) time
- Recombination requires O(m/p) point doublings per processor, except the last one, and a single point addition
- Asymptotic parallel running time is thus:
    - ►General: O(mD + (m/p + p)A)
    - ►Koblitz: O((m/p + p)A)

Asymptotic Throughput

- Throughput in a tree is determined by the maximum of the root’s computation time and the leaves’ computation time:
  - ►General: O(1 / max(m/p (D + A), m D)
  - ►Koblitz: O(1 / (m/p A))
- Throughput in a linear array is determined by the computation time in a single node:
  - ►General: O(1 / (m/p D + m/p A))
  - ►Koblitz: O(1 / (m/p A))

Practical Running Time & Throughput

- Parallel overhead - O(log p) time for a tree or hypercube and O(p) time for a linear array
  - ►Network delays (MPI)
  - ►Packing/unpacking overhead (MPI)
  - ►Synchronization delays (OpenMP)
- Constant factors impact running time
  - ►Window sizes vary based on subscalar size, limiting speedup for regular multiplication

Practical Running Time & Throughput

- Sequential portion of multiplication – point doubling or tau operator and scalar conversion
  - Large sequential portion due to point doubling cost for general curves limits speedup
  - More efficient tau operator reduces sequential portion, but sequential portion becomes more significant with many processors
  - Sequential portion more significant for regular multiplication, further limiting speedup

Experimental Parameters

- 10 standard NIST curves: P-192, P-224, P-256, P-384, P-521, K-163, K-233, K-283, K-409, K-571
- Number of cores varied from 1-128
- Input form of scalar – NAF or binary
- Number of simultaneous multiplications varied from 1-16 (hypercube)
- Multiplication type – Montgomery or regular
- Logical topologies – Hypercube, Tree, Linear
- OpenSSL used to handle basic point operations
- GMP/MPFR to handle large rationals/floats

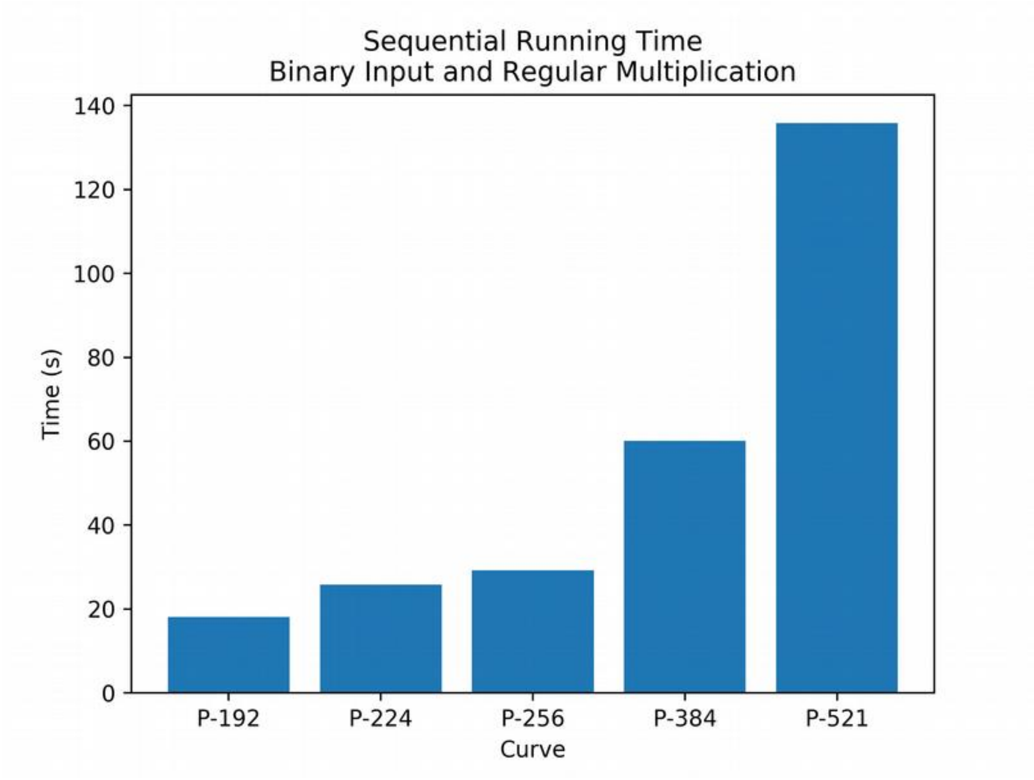
Experimental Setup

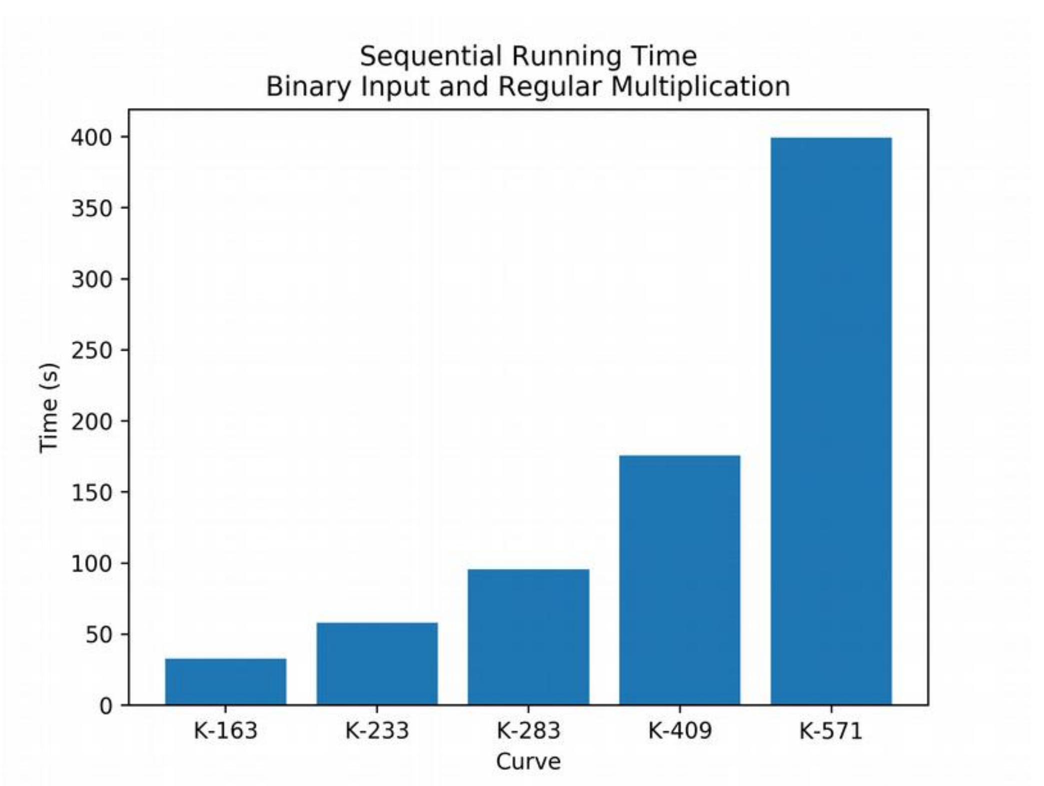
- 16 core machines utilized for all tests at UB CCR:
  - Intel E5-2660 Xeon (dual 8 core)
  - Infiniband Network (when using >16 cores)
- MPI Thread Safety for Hybrid Approach
  - Tree/hypercube: MPI_THREAD_SERIALIZED
  - Linear: MPI_THREAD_MULTIPLE
- Points and scalars generated at random
- 50,000 total multiplications performed for each experiment

Experimental Setup

- Linear and tree running time is not measured directly, but estimated
  - ►Tree running time estimated by estimated by summing average running time at each tree level excluding the time spent waiting for other processors
  - ►Linear running time estimated by summing the the time spent in each node sequentially plus the time spent in parallel

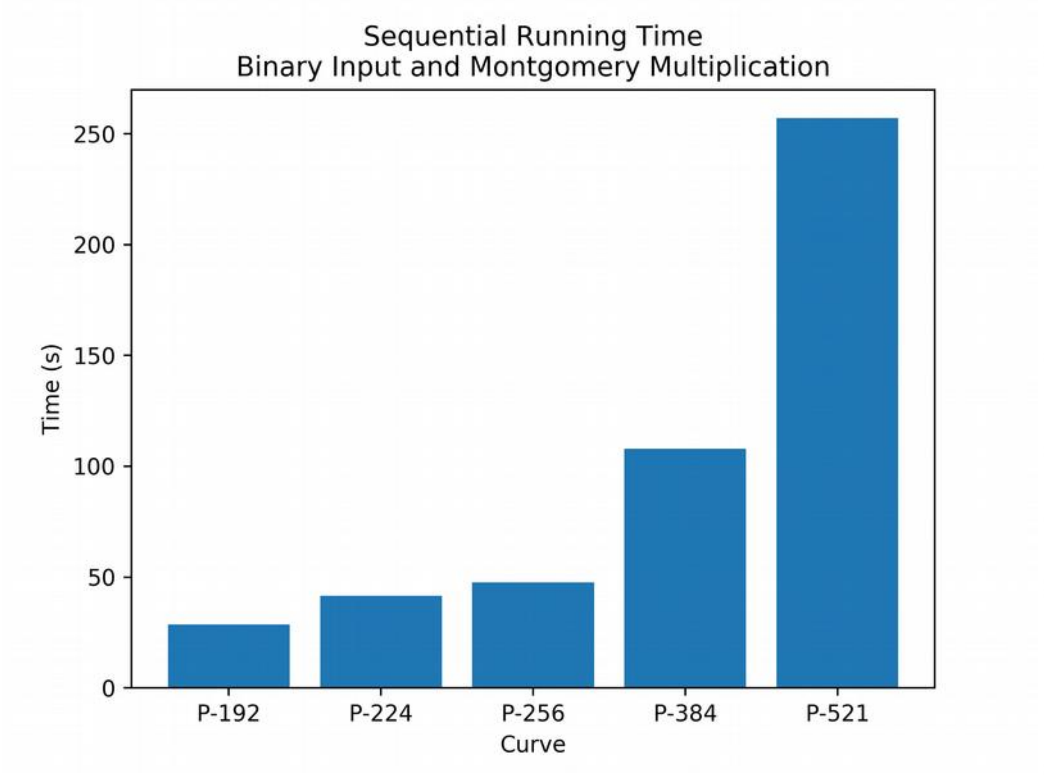
Sequential Running Time

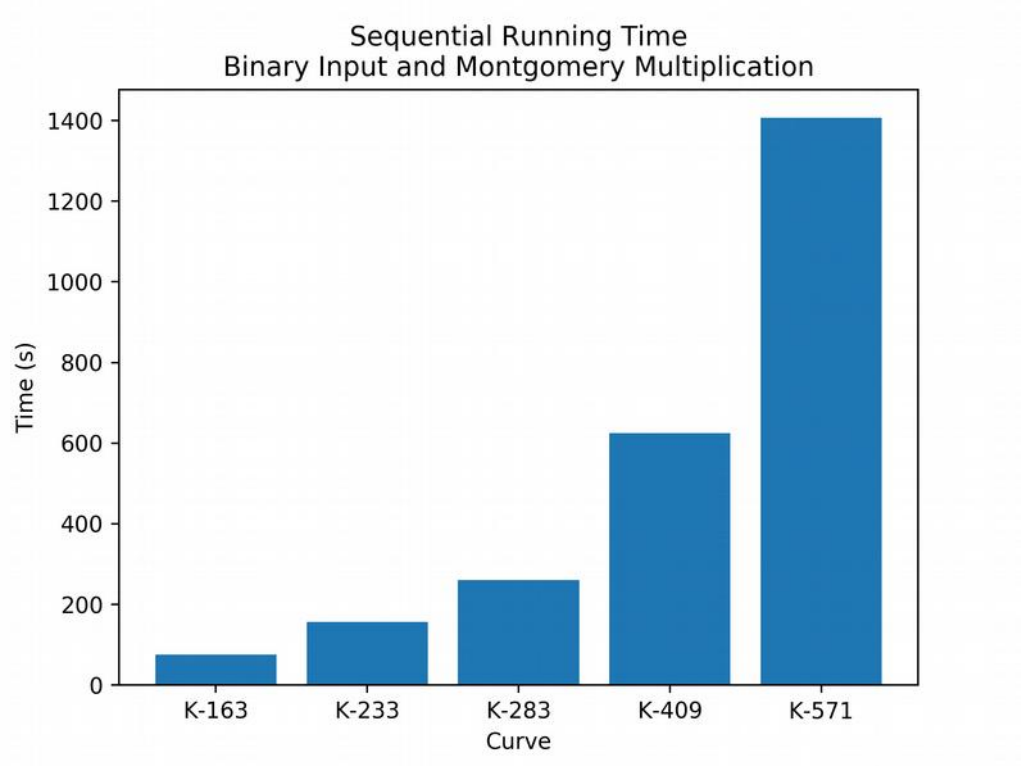




- Koblitz curves (right) exhibit slower running times due to less support in OpenSSL and binary curves in general being better suited for hardware implementations

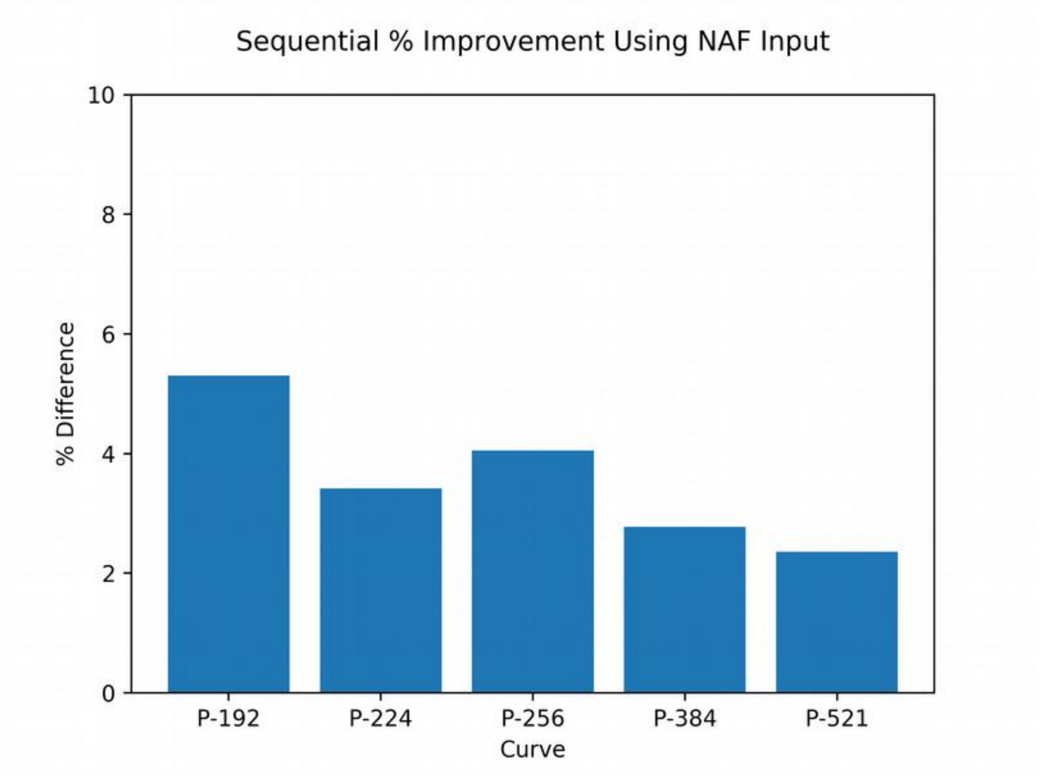
Sequential Running Time

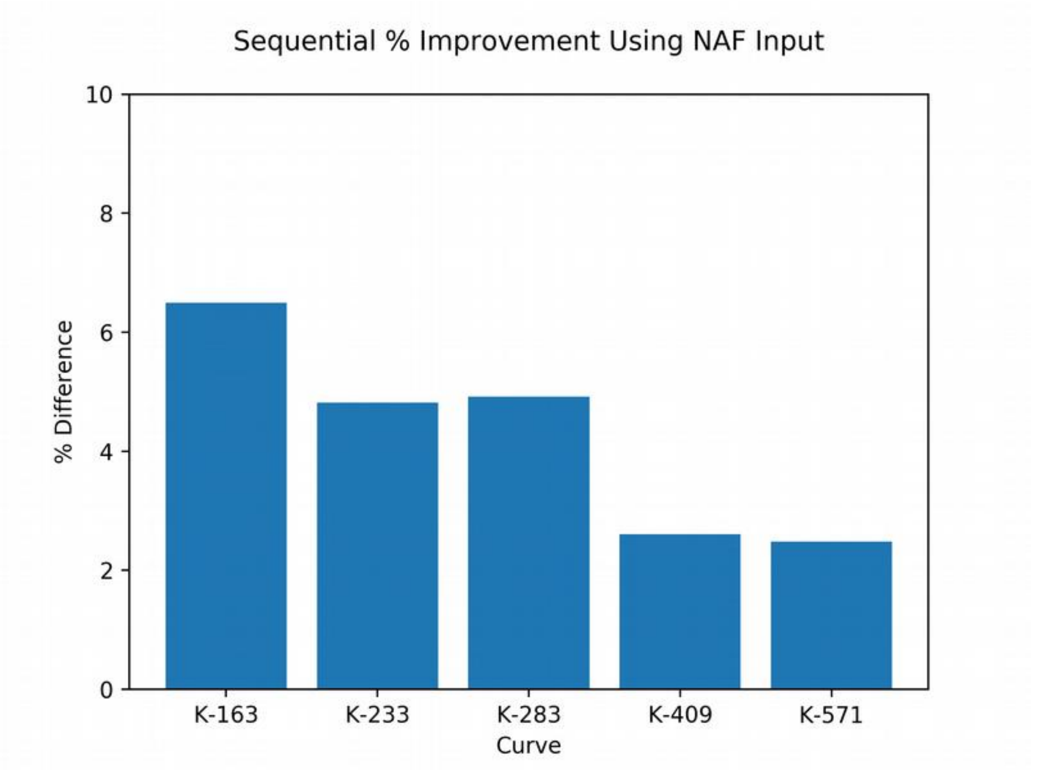

Sequential Running Time
Binary Input and Montgomery Multiplication


Sequential Running Time
Binary Input and Montgomery Multiplication

- Montgomery methods up to 3.5 slower than regular multiplications (previous)
- Performance hit worse for Koblitz curves

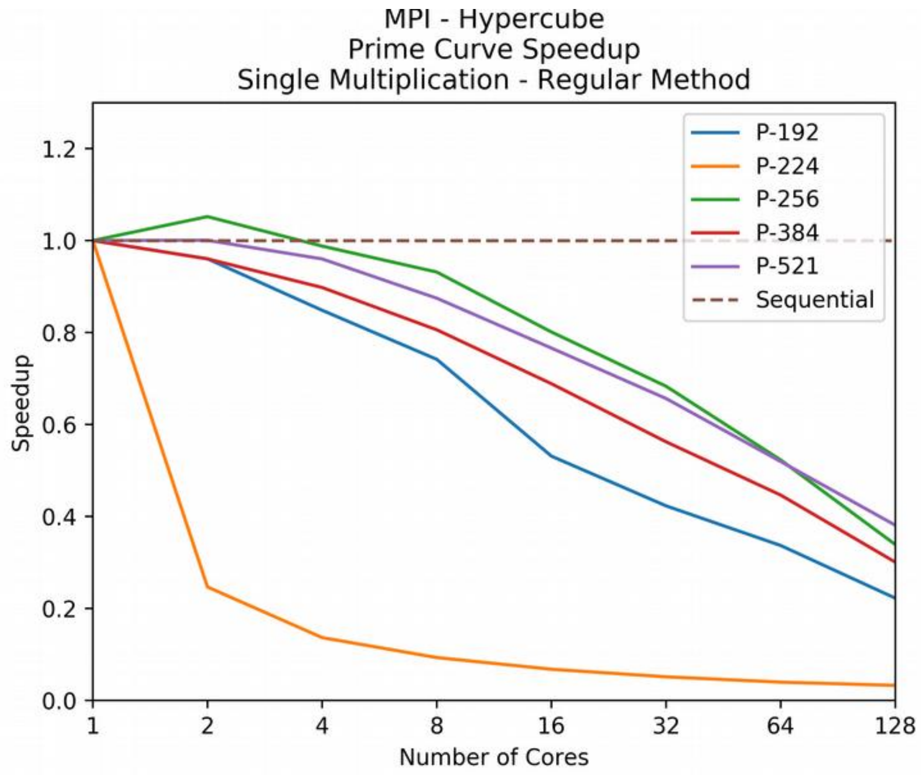
Sequential Running Time

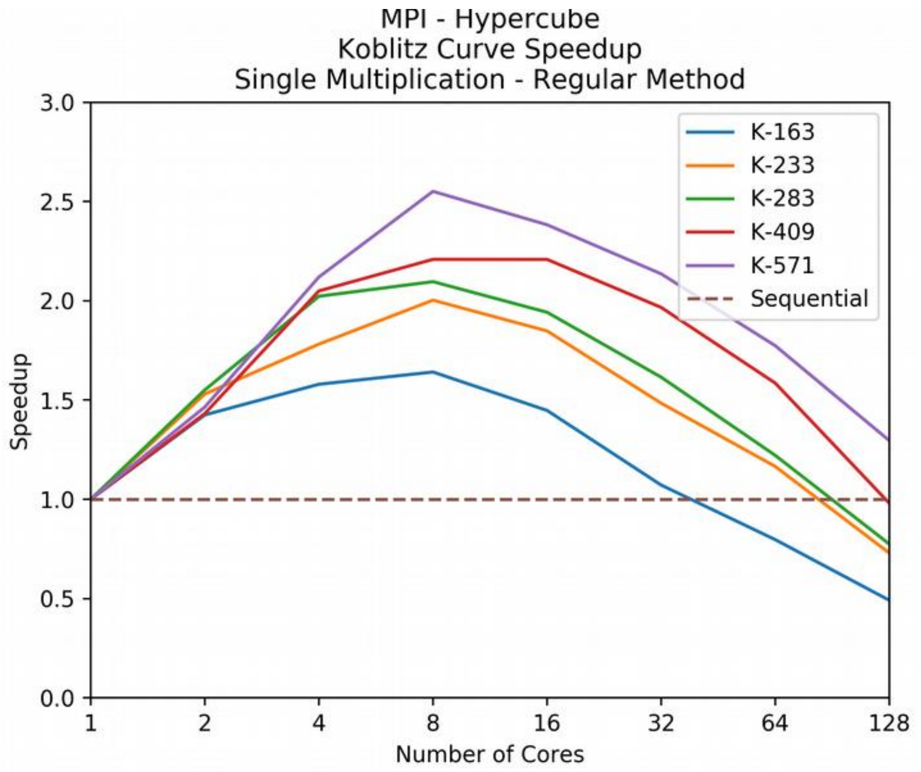




- Small improvement using NAF input
- Going forward, only binary input is presented
  - ►Results for NAF input show slight improvement

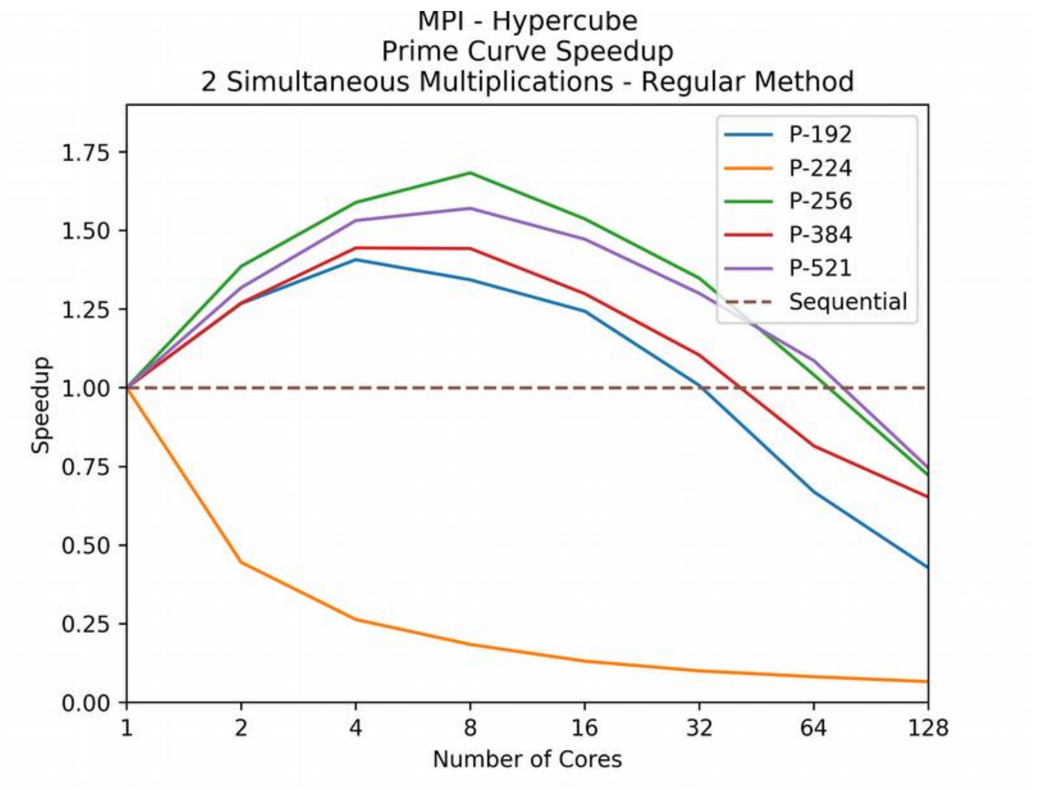
Hypercube Speedup

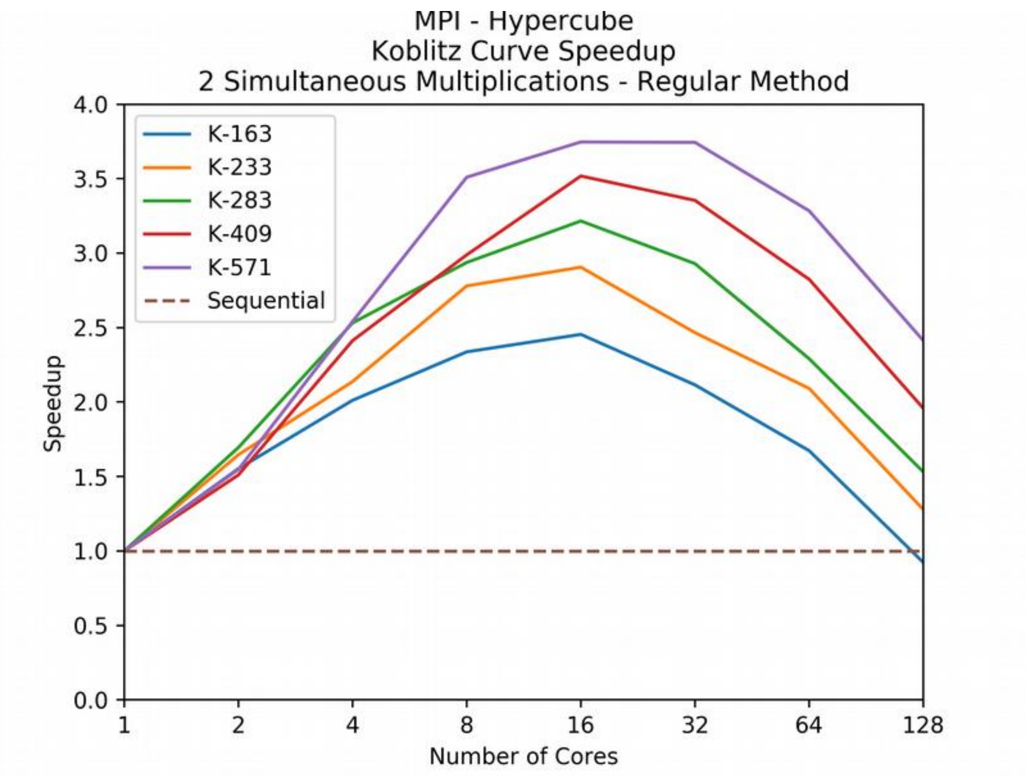

MPI - Hypercube
Prime Curve Speedup
Single Multiplication - Regular Method


MPI - Hypercube
Koblitz Curve Speedup
Single Multiplication - Regular Method

- Large parallel overhead limits speedup for prime curves in particular
  - ► Worse than sequential except P-256 using 2 cores

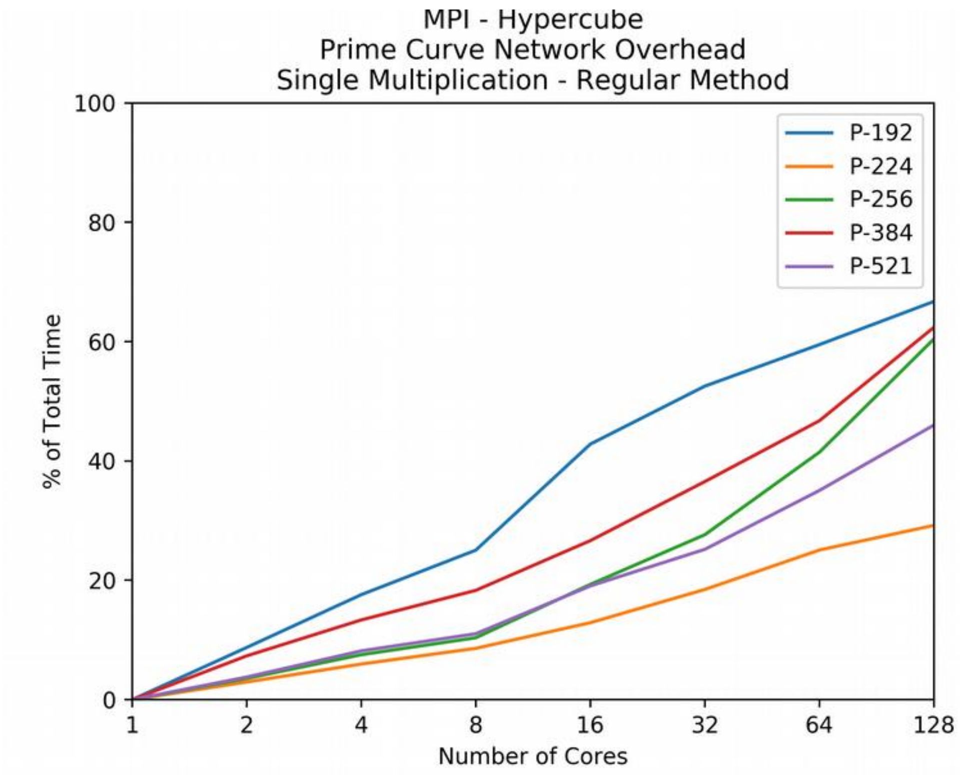
Hypercube Speedup



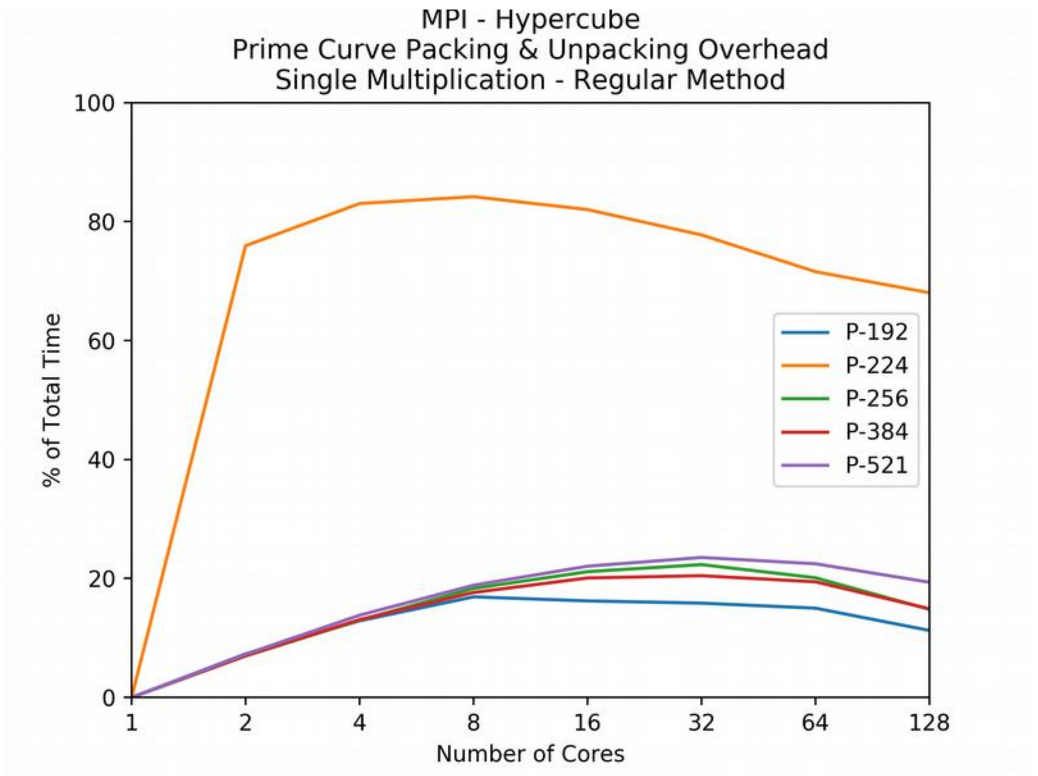


- Interweaving worse than dividing processors
  - ►Same holds for other configurations – further graphs on simultaneous multiplications omitted

Hypercube Overhead



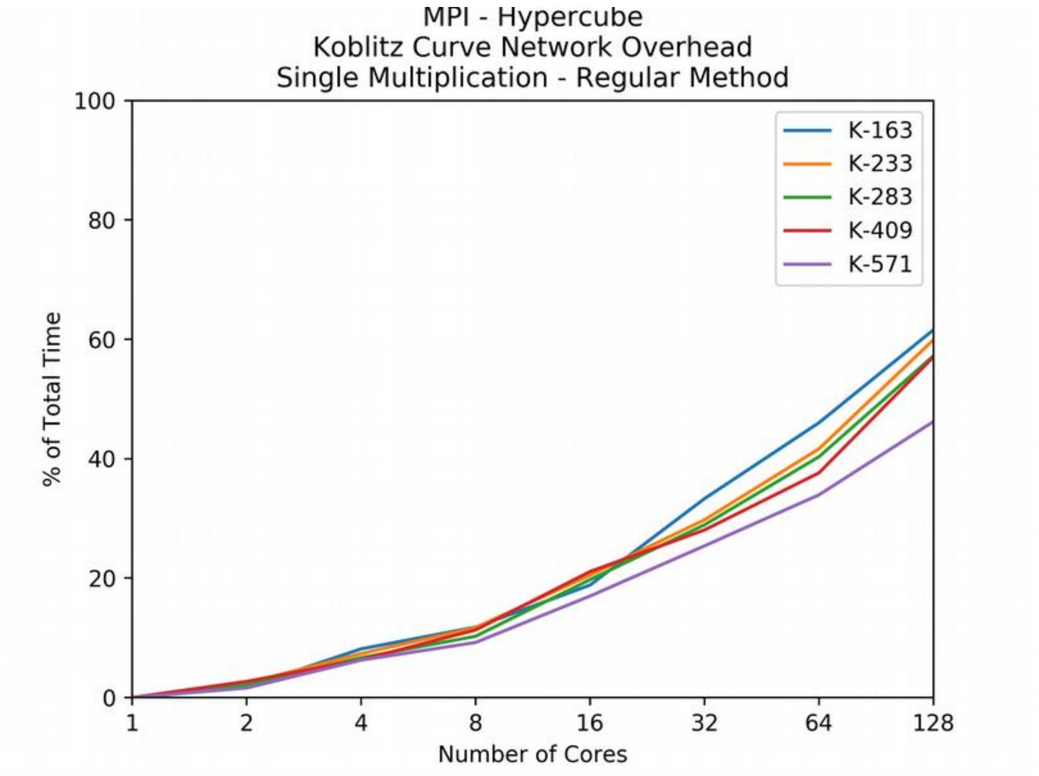
MPI - Hypercube
Prime Curve Network Overhead
Single Multiplication - Regular Method

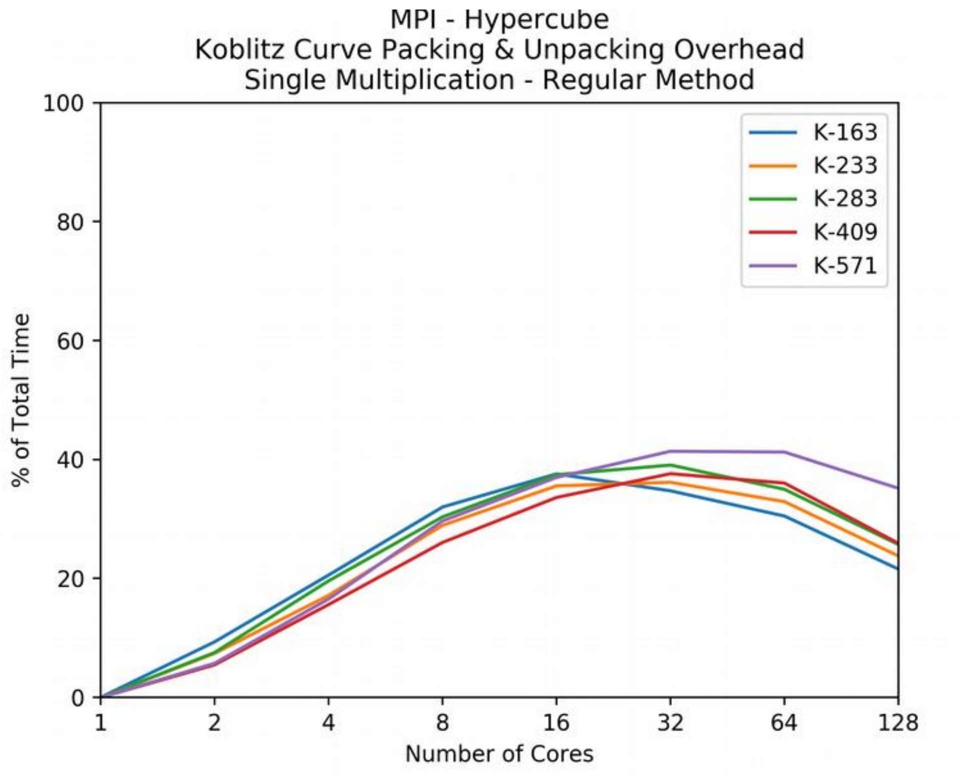


MPI - Hypercube
Prime Curve Packing & Unpacking Overhead
Single Multiplication - Regular Method

- Overhead grows with number of cores
- OpenSSL optimizations for P-224 at expense of packing/unpacking time explain its results

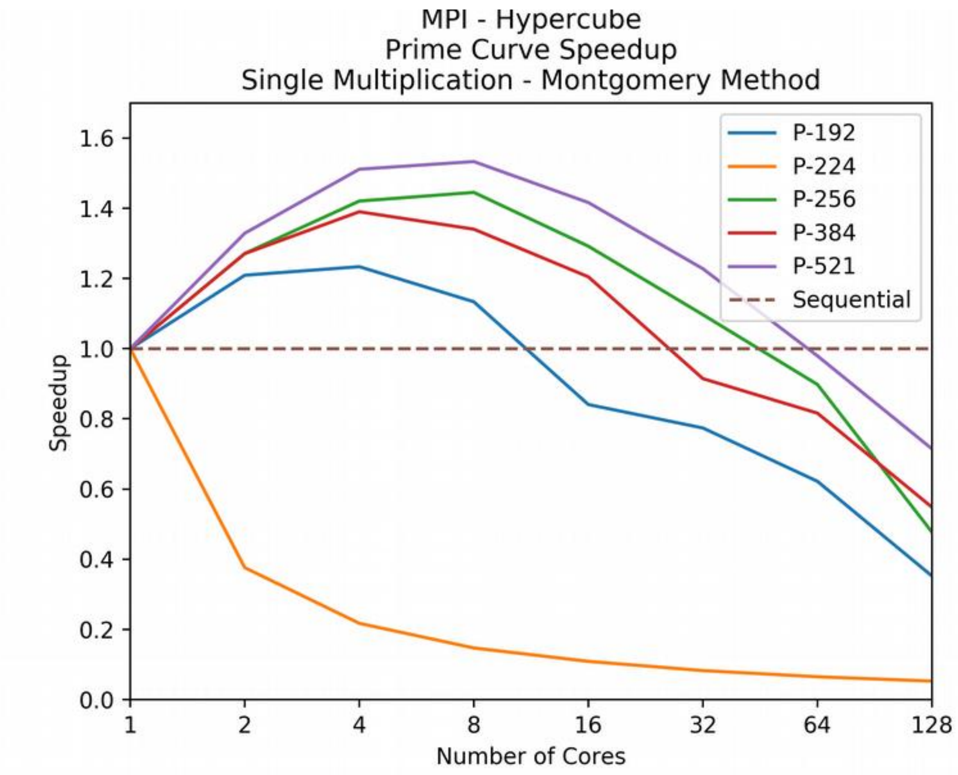
Hypercube Overhead

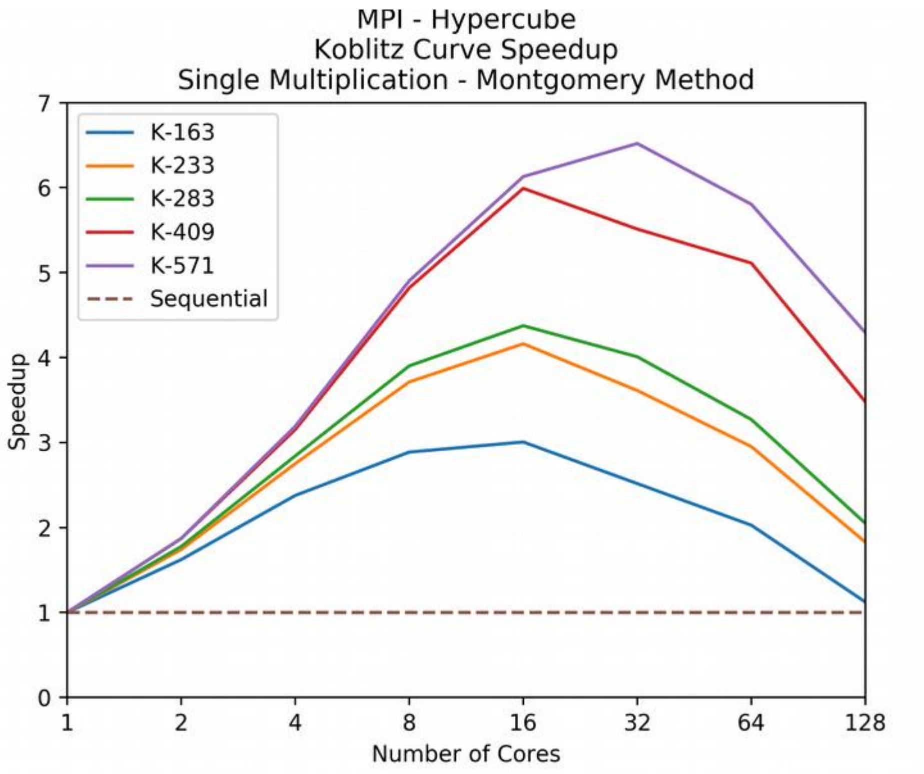




- More time spent on packing/unpacking overhead for Koblitz curves
- Generally less networking delays for Koblitz curves

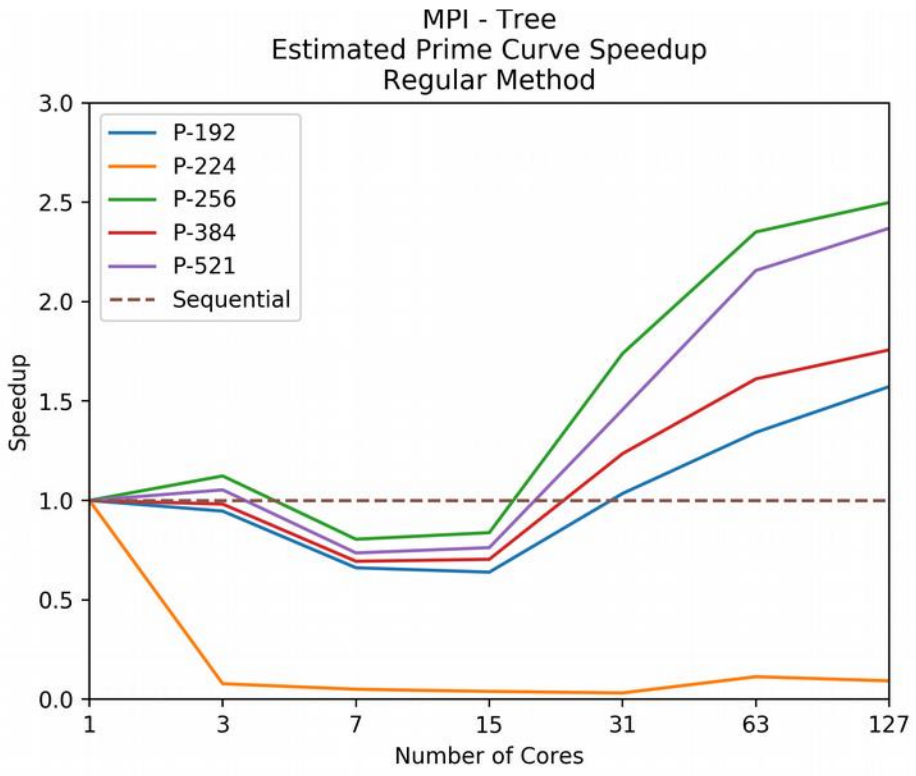
Hypercube Speedup

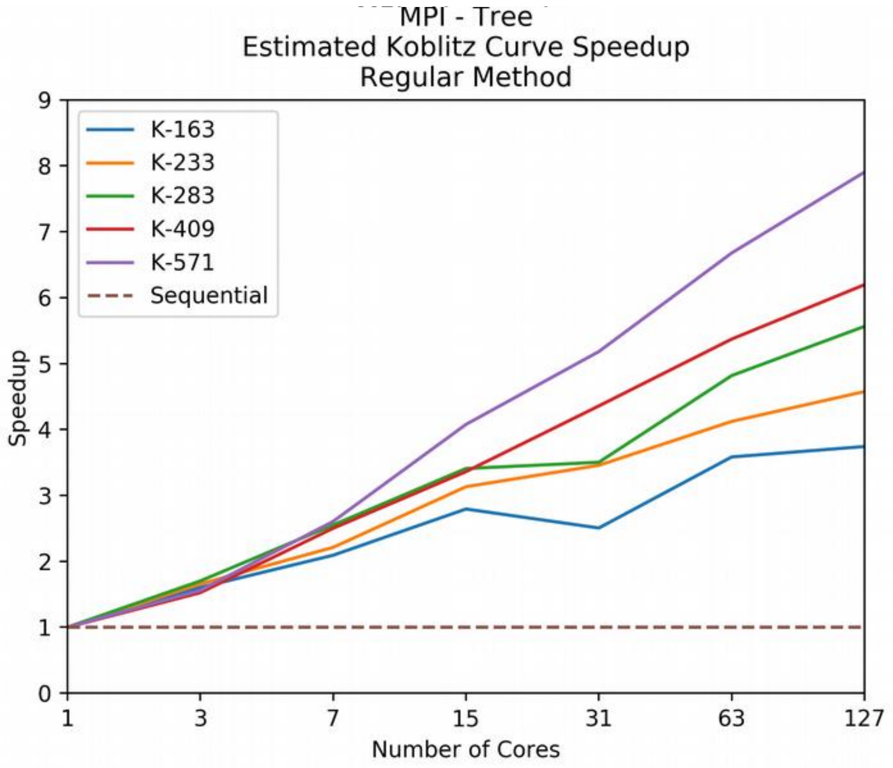




- Better speedup using a Montgomery method
- Prime curves show limited speedup due to larger sequential portion

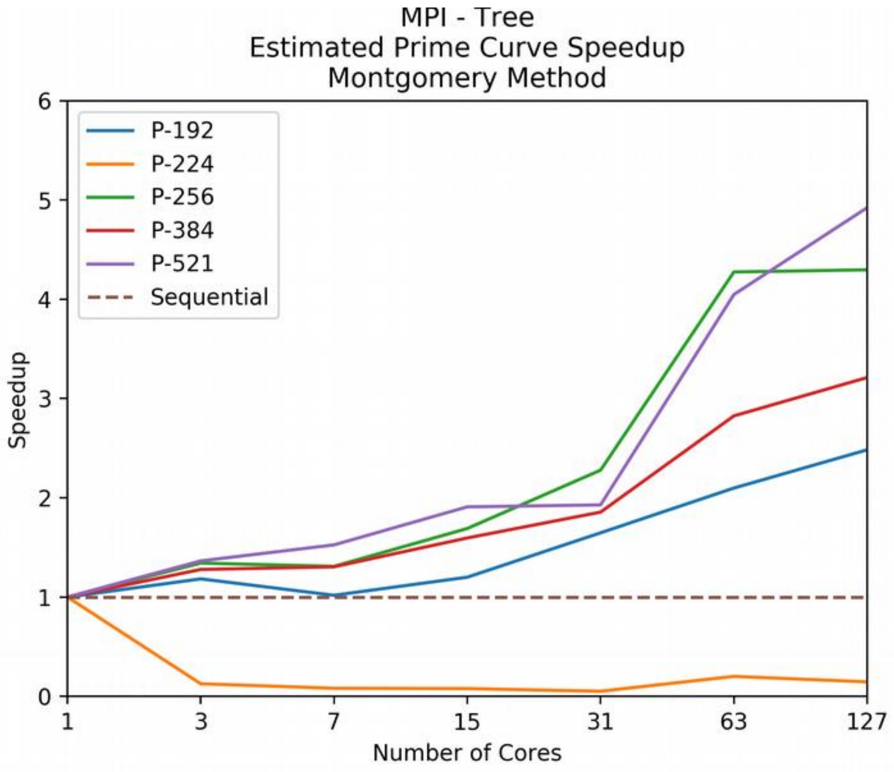
Tree Speedup

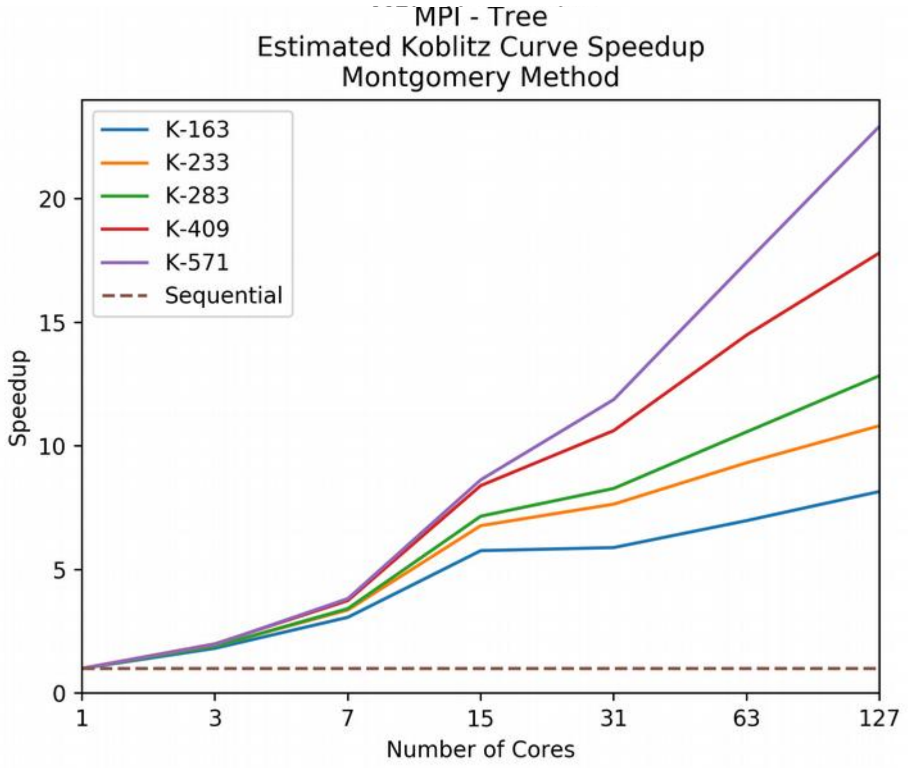

MPI - Tree
Estimated Prime Curve Speedup
Regular Method


MPI - Tree
Estimated Koblitz Curve Speedup
Regular Method

- Better speedup than equivalent hypercube as communications spread out over more time
- Overhead/constant factors outweigh parallel benefits for prime curves with <15 processors

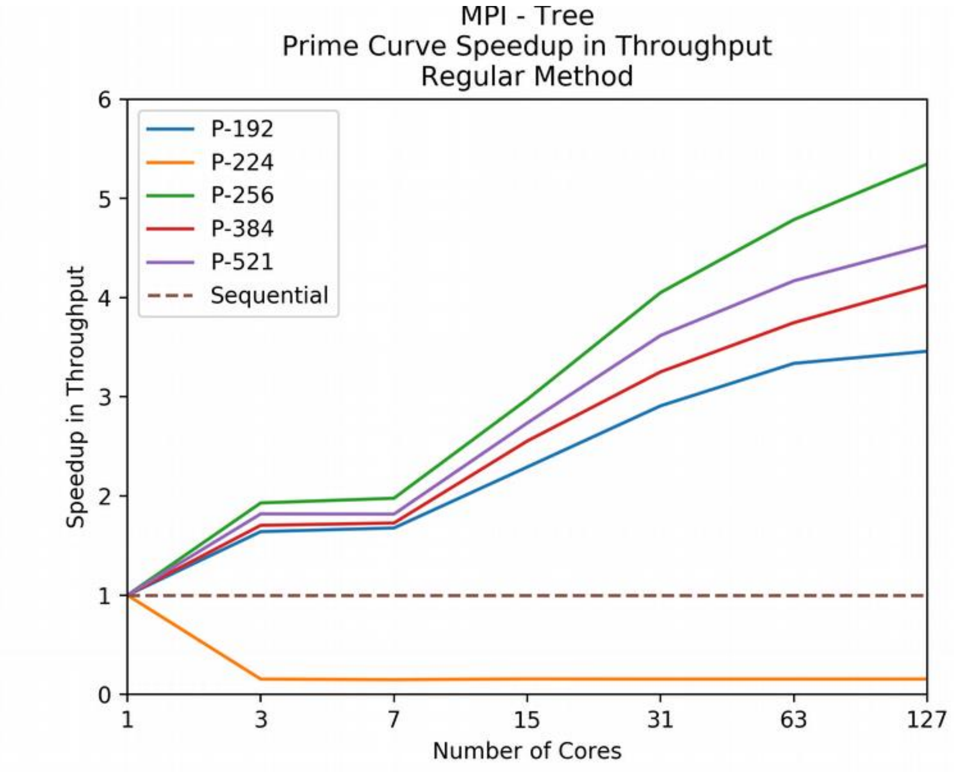
Tree Speedup

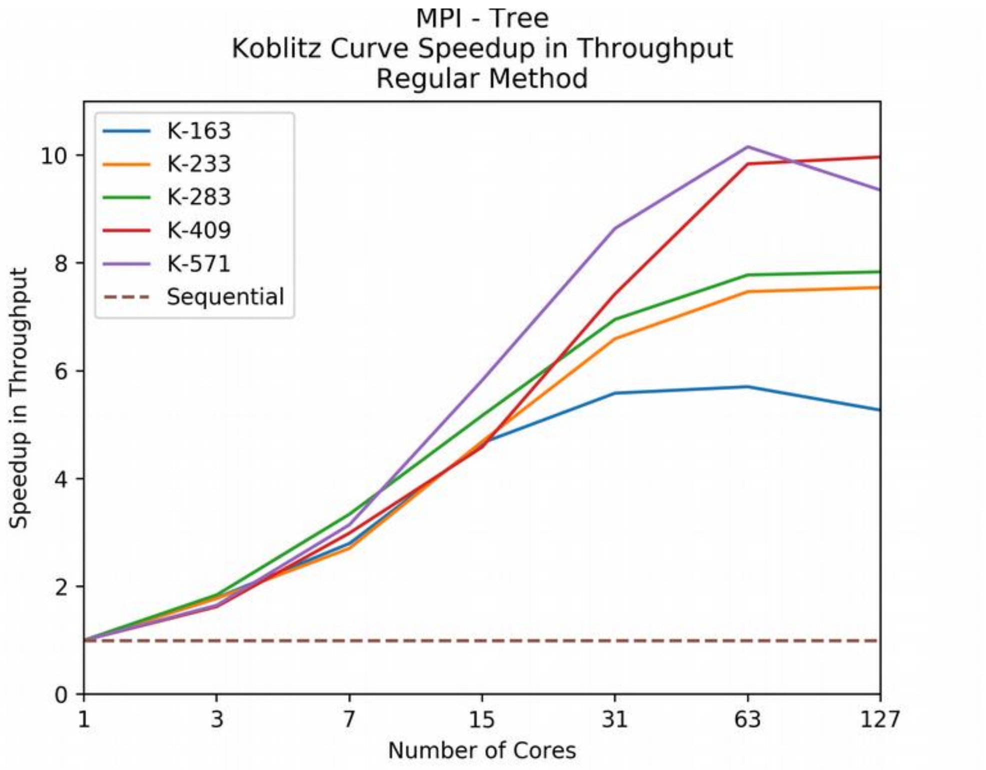

MPI - Tree
Estimated Prime Curve Speedup
Montgomery Method


MPI - Tree
Estimated Koblitz Curve Speedup
Montgomery Method

- Better speedup using Montgomery method

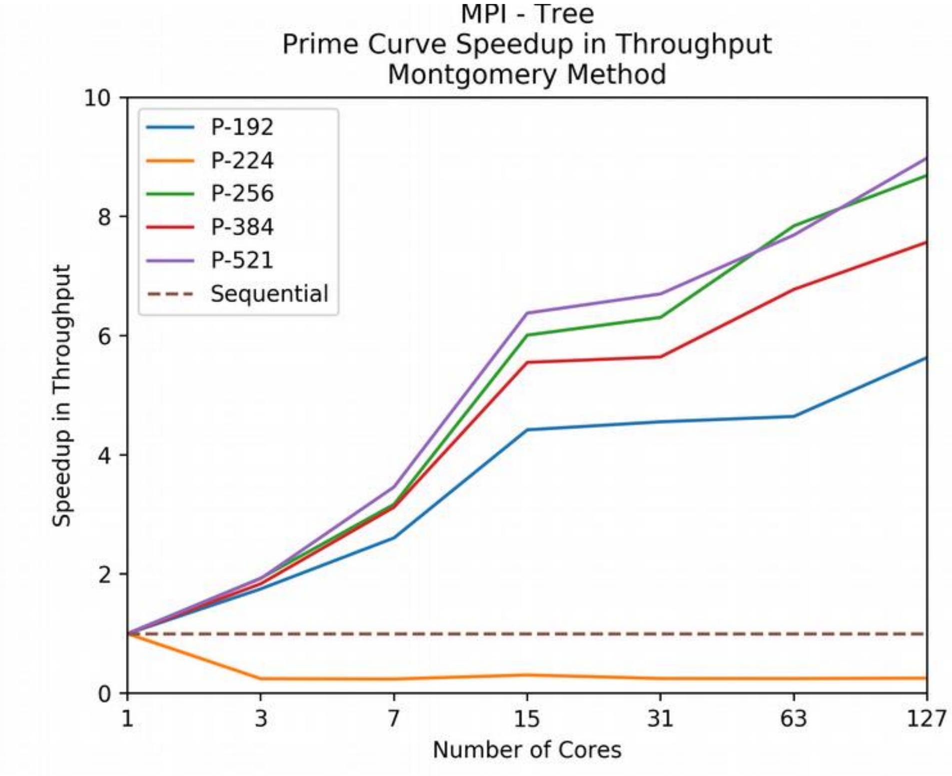
Tree Throughput



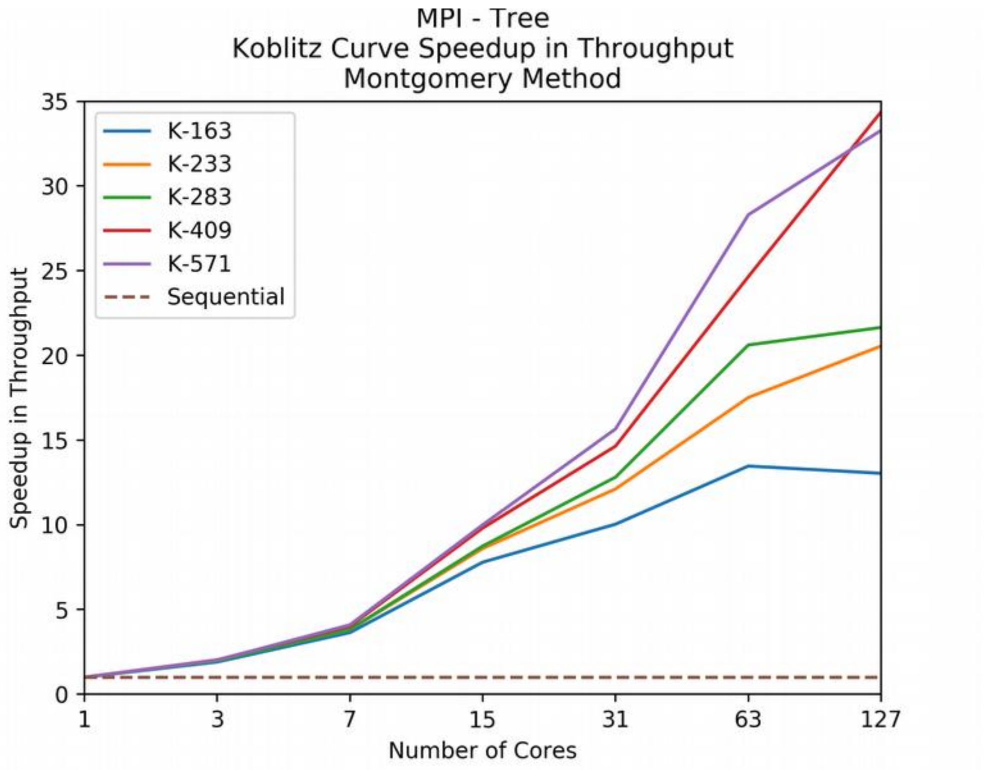


- Throughput continues to improve (except P-224) as number of cores increased
- Better throughput by using processors sequentially, but worse speedup in some cases

Tree Throughput



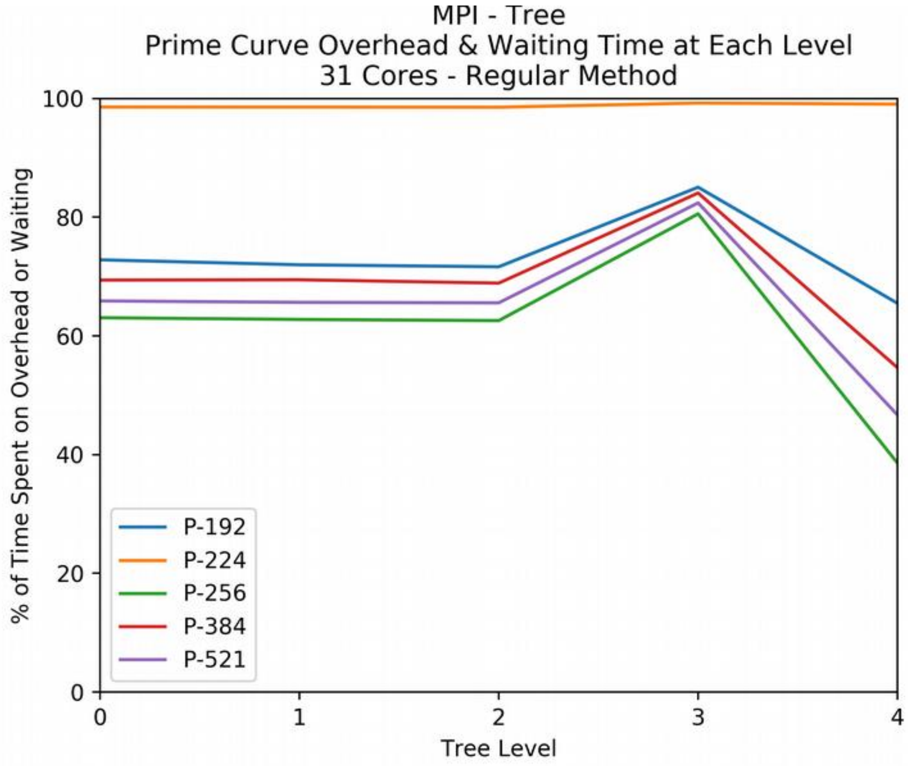
MPI - Tree
Prime Curve Speedup in Throughput
Montgomery Method

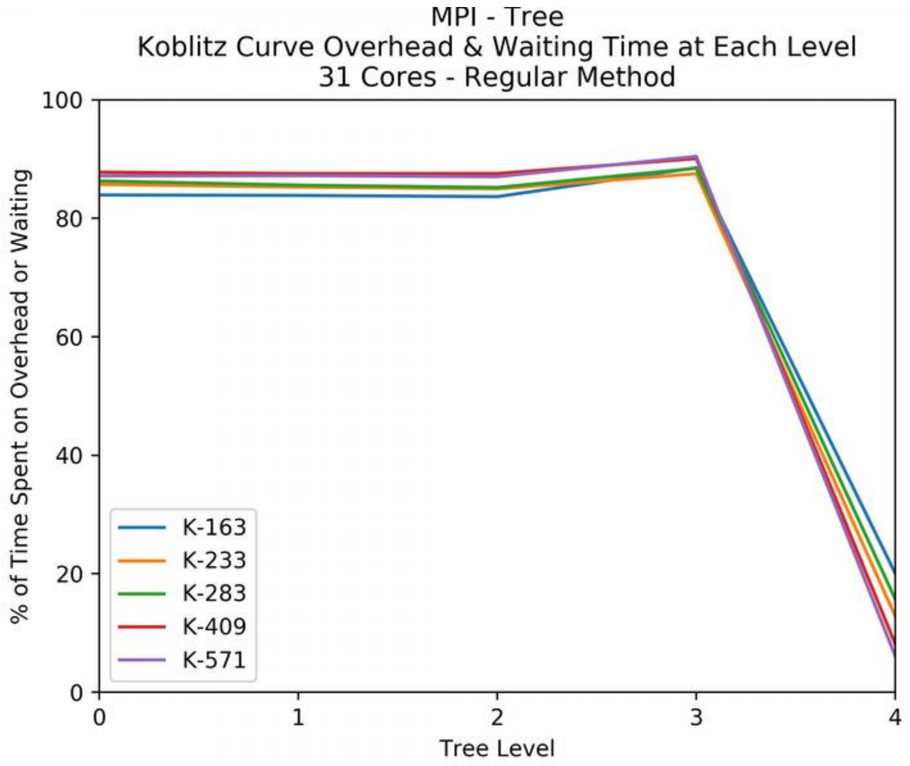


MPI - Tree
Koblitz Curve Speedup in Throughput
Montgomery Method

- Throughput continues to improve (except P-224) as number of cores increased
- Better throughput by using processors sequentially, but worse speedup in some cases

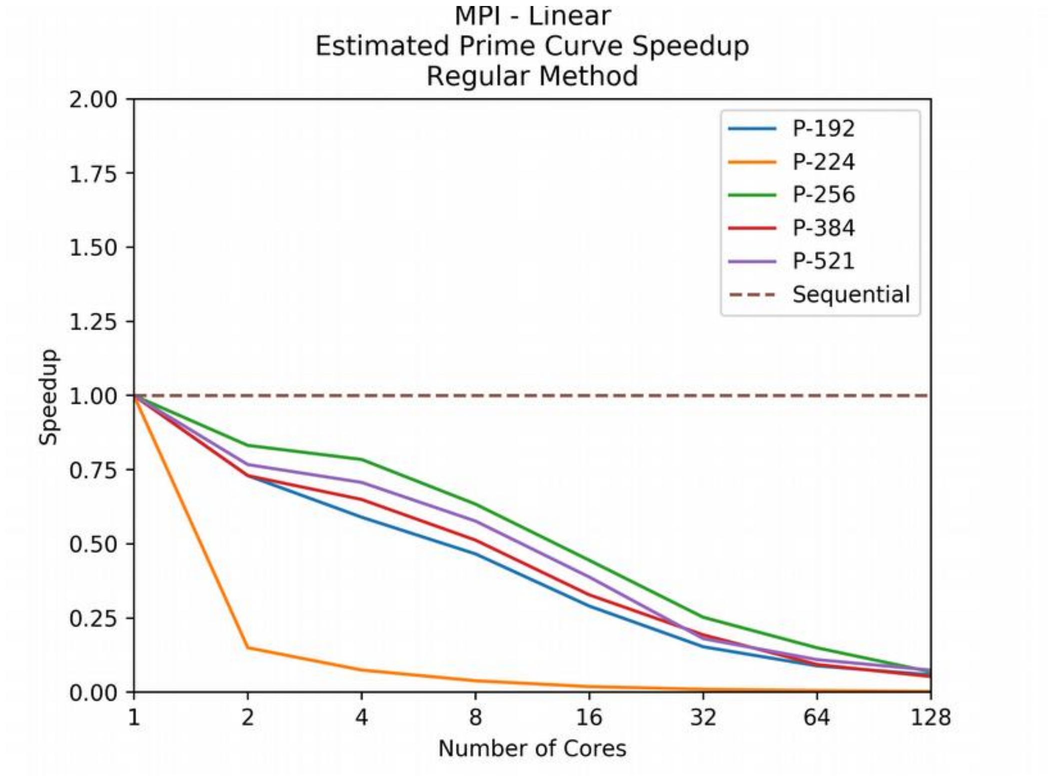
Time Spent Waiting or on Parallel Overhead in Tree

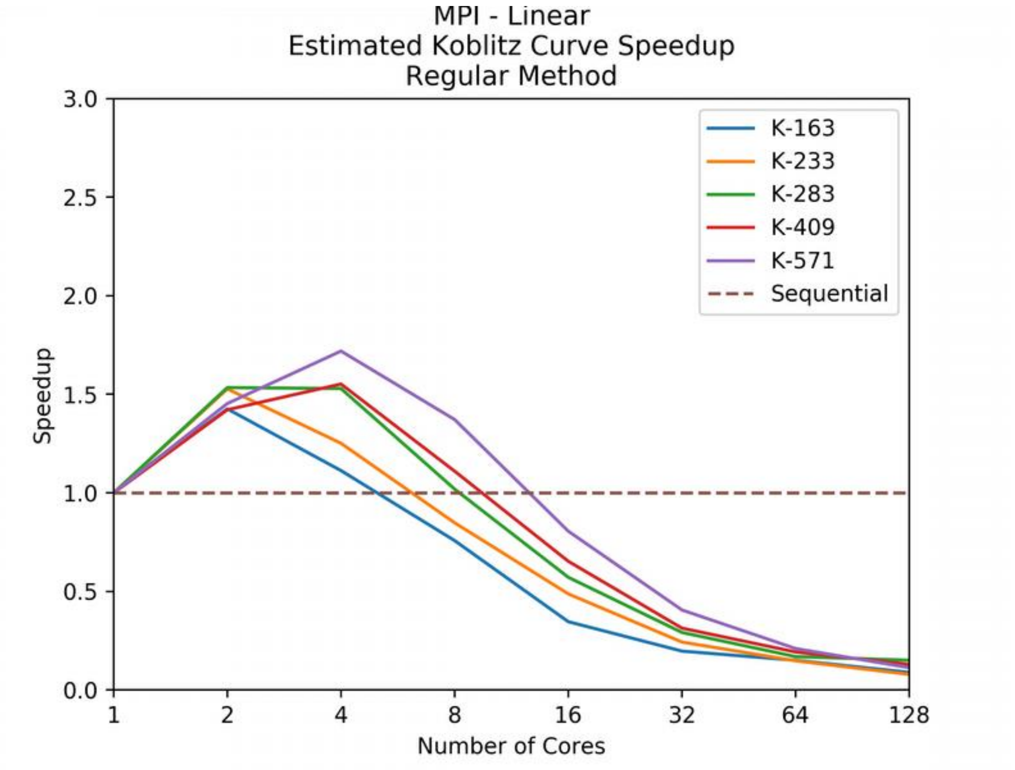




- Large amount of idle time, waiting for other processors at non-leave levels
- Similar results for other configurations

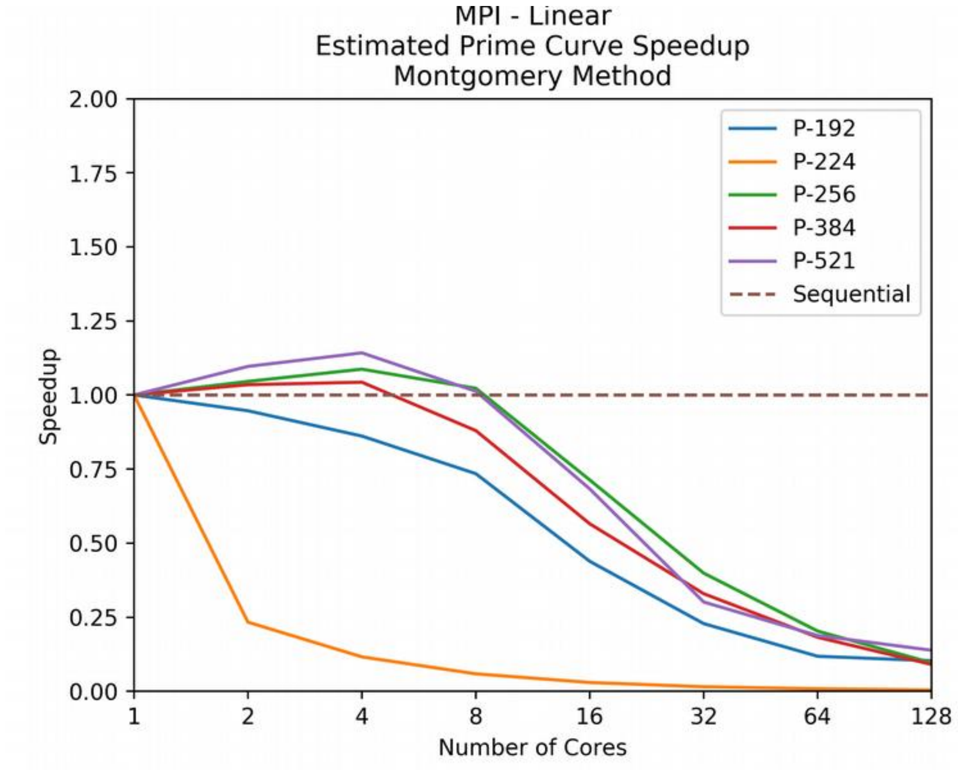
Linear Speedup

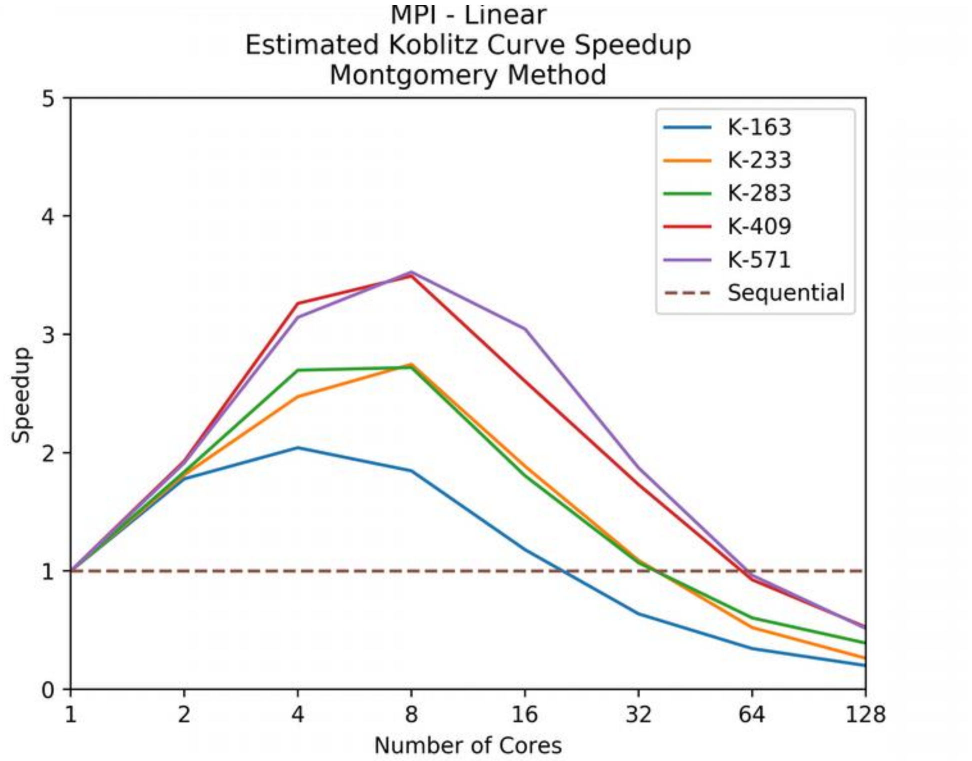




- Strictly worse than sequential for prime curves
- For Koblitz curves, 2 cores give speedup comparable to 2 core hypercube or 3 core tree and worse otherwise

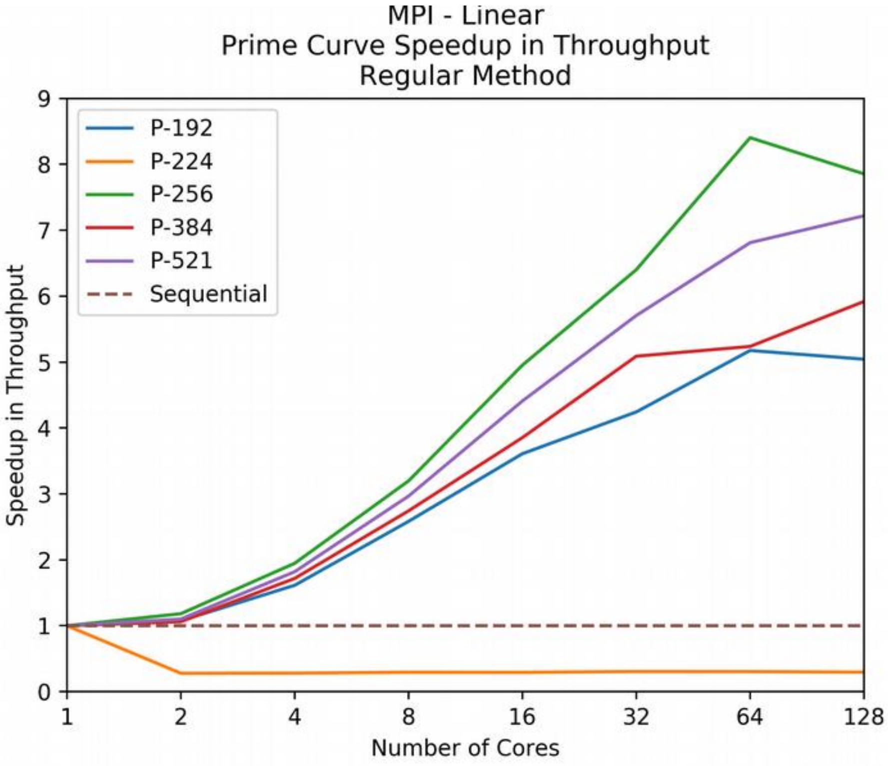
Linear Speedup

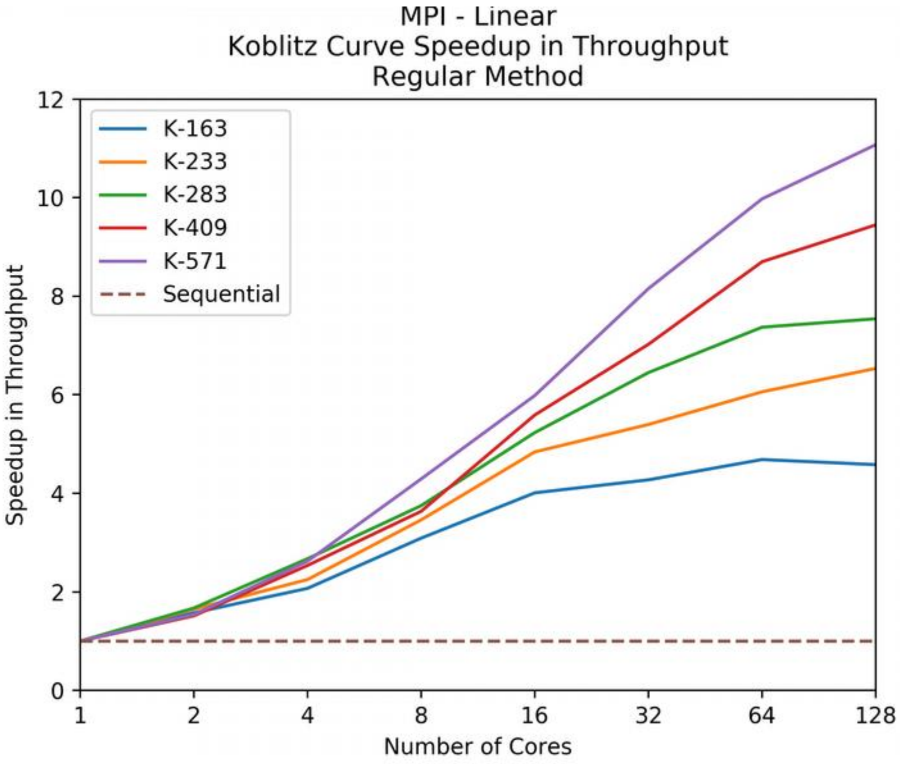




- Montgomery method shows marginal speedup for prime curves, worse than hypercube or tree
- Better speedup for some Koblitz curves for 2-4 cores compared to 2-4 core hypercube or 3-7 core tree

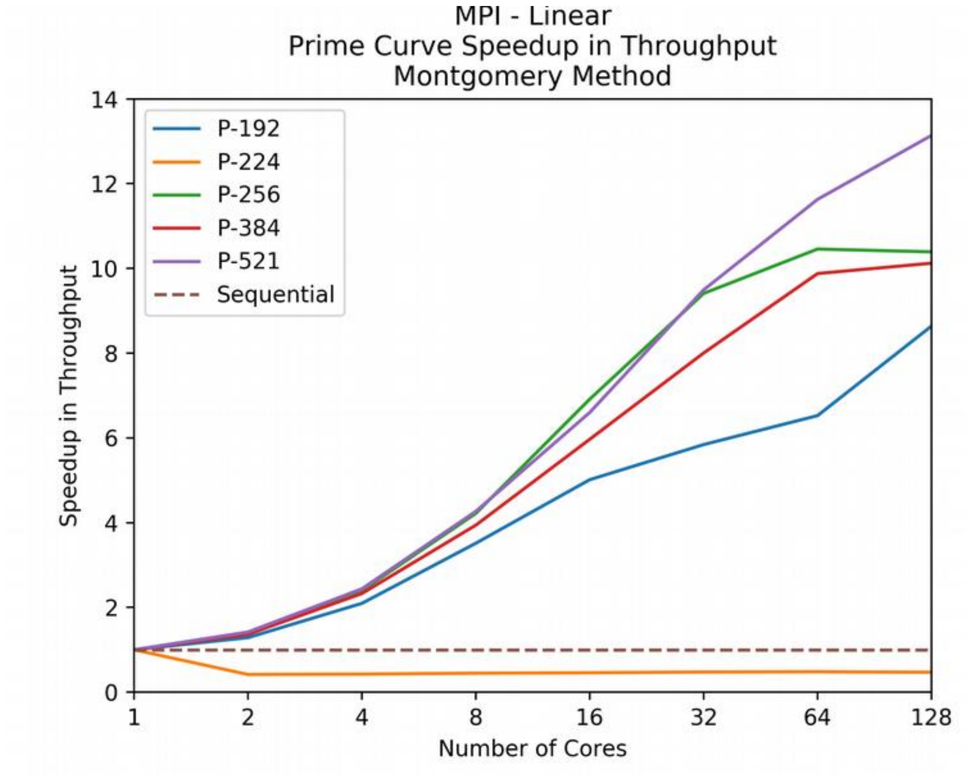
Linear Throughput

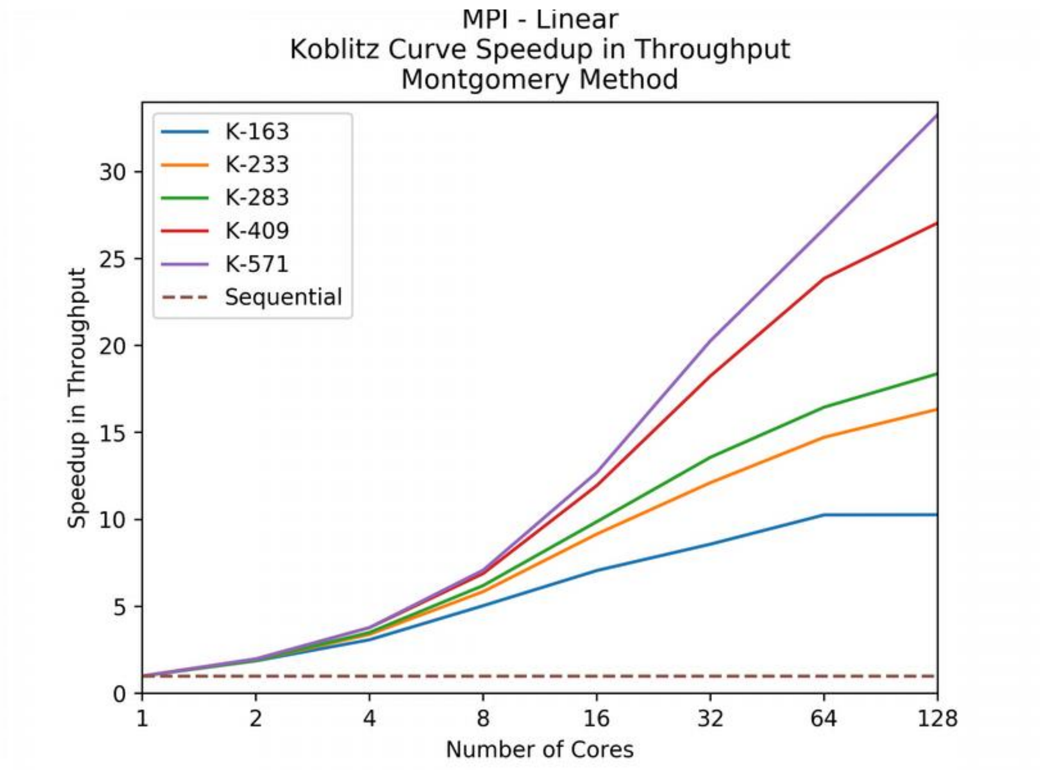

MPI - Linear
Prime Curve Speedup in Throughput
Regular Method


MPI - Linear
Koblitz Curve Speedup in Throughput
Regular Method

- Throughput is generally a bit better than a tree
  - ►Strictly better to distribute multiplications sequentially on prime curves using since no speedup advantages and worse throughput

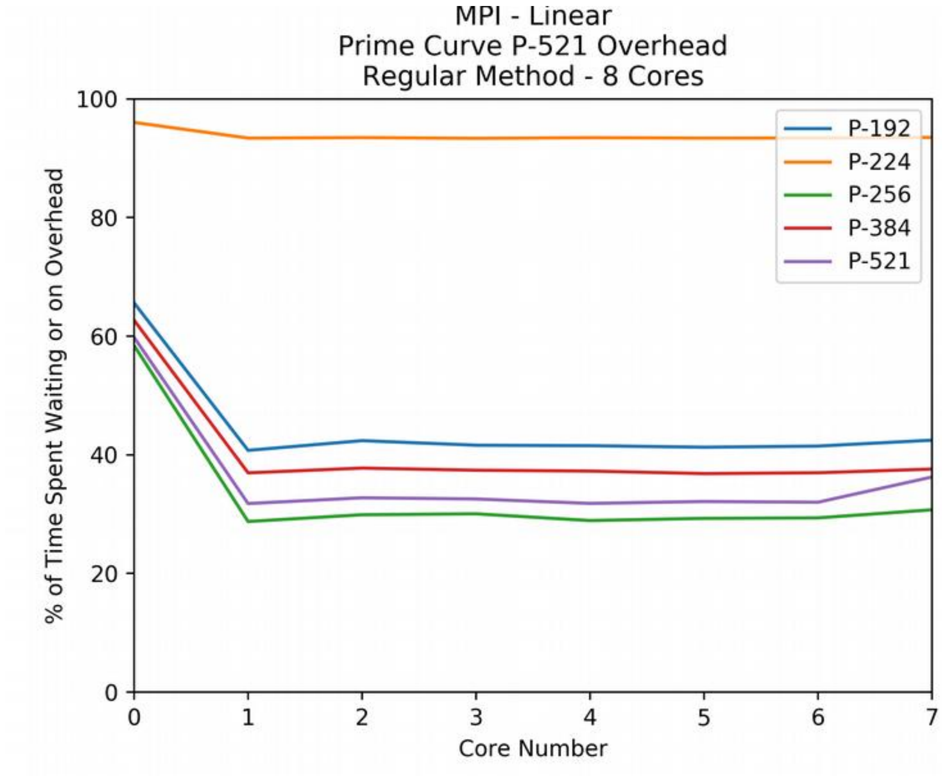
Linear Throughput

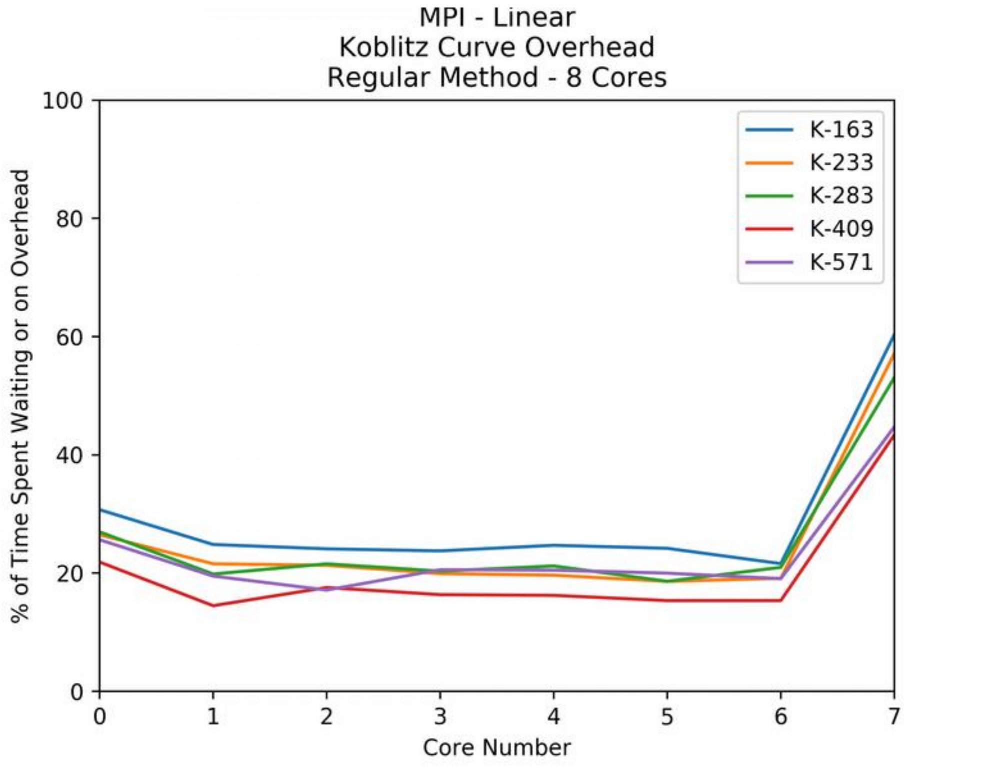




- Slightly better throughput than a tree when using few cores

Linear Overhead





- Generally linear overhead takes up less overall time
- Similar results for other configurations

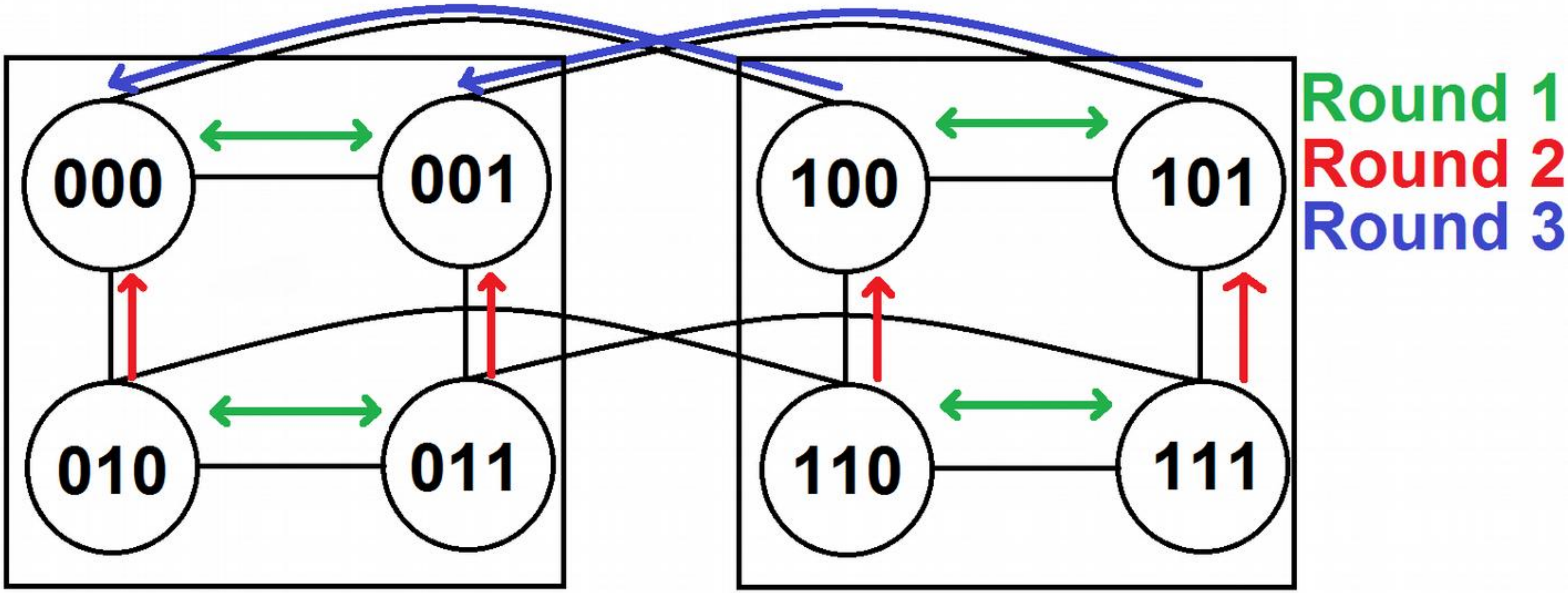
MPI Conclusions

- Packing/unpacking time for some curves and network delays limit achievable speedup and throughput
- Simultaneous communication can cause congestion limiting speedup, as seen with a tree achieving better speedup than an equivalent hypercube
- Trees generally offer good balance between speedup and throughput
- Linear array never good for prime curves, and better than a tree for Koblitz curves with a small number of cores available

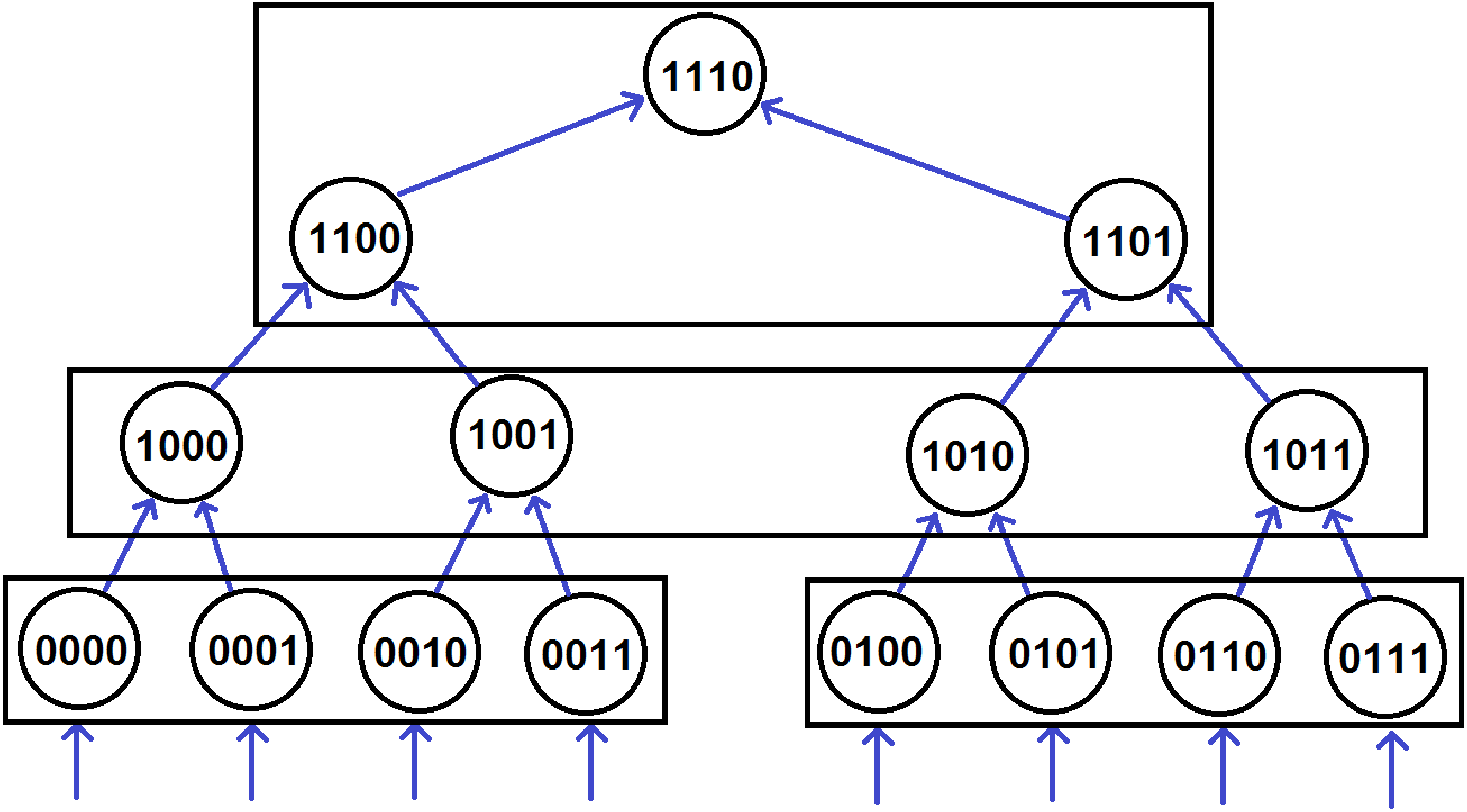
Challenges Moving to a Hybrid Approach

- Explicit synchronization required in OpenMP
- Results from MPI indicate limiting MPI calls could be beneficial
  - ►Where possible, MPI calls are merged, but this requires additional synchronization
- Where to use OpenMP vs MPI?
  - ►Based on rounds in hypercube topology
  - ►Based on level in tree topology
  - ►Based on neighbors in linear topology

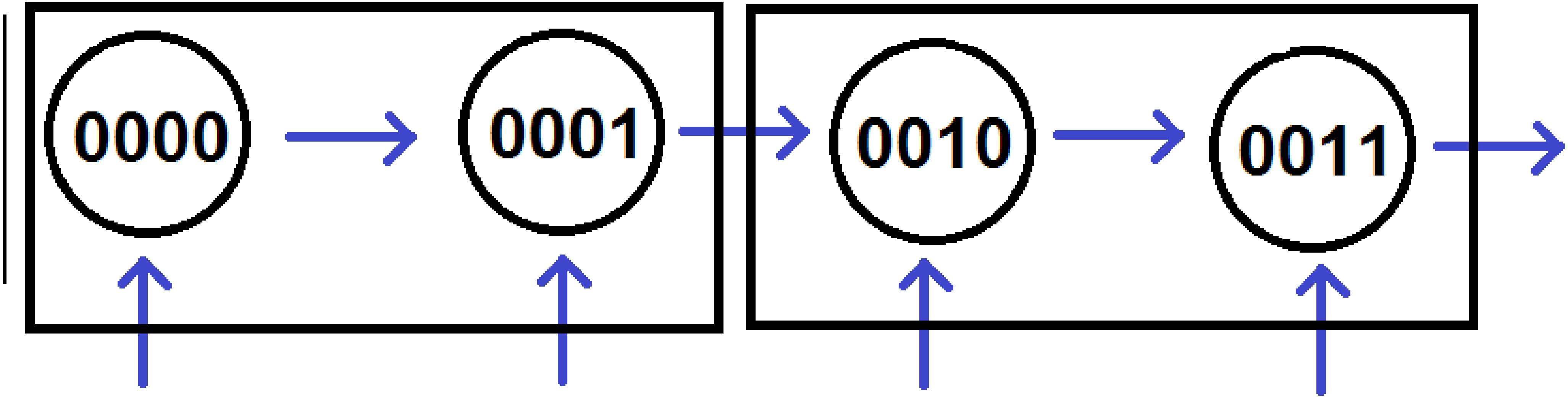
Hybrid Hypercube with 2 MPI nodes and 4 threads



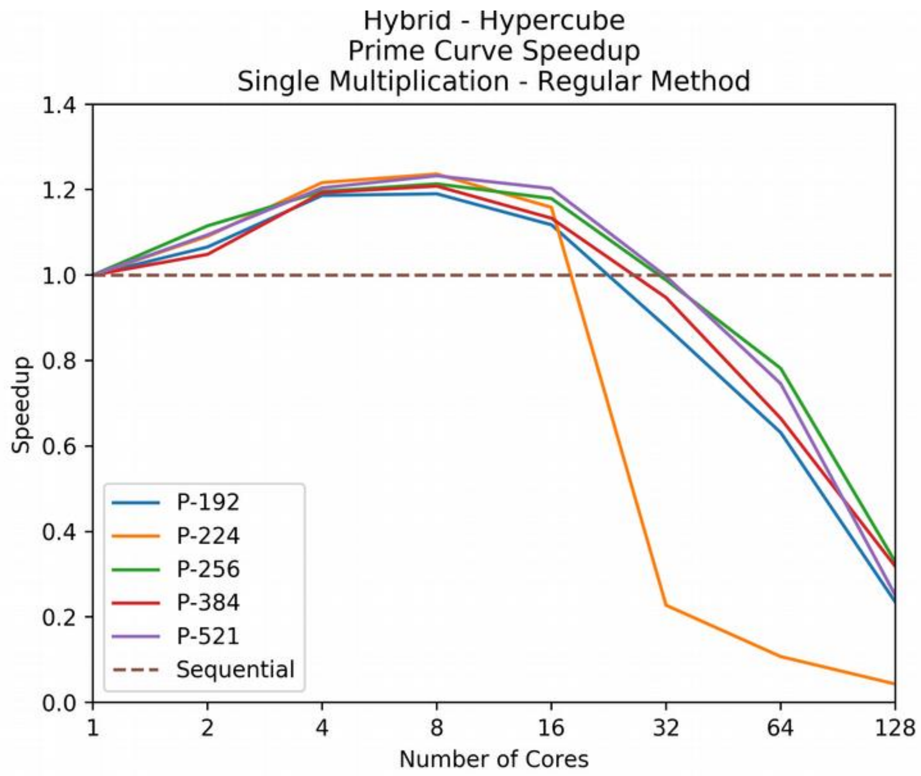
Hybrid Tree with 4 MPI nodes and 4 threads

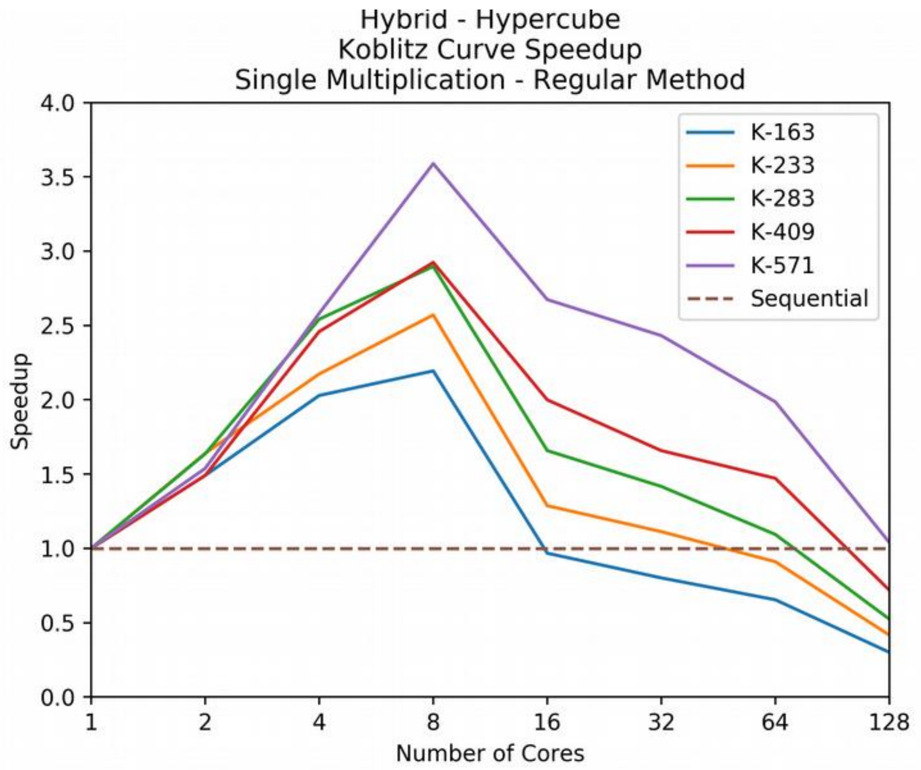


Hybrid Linear with 2 MPI nodes and 2 threads



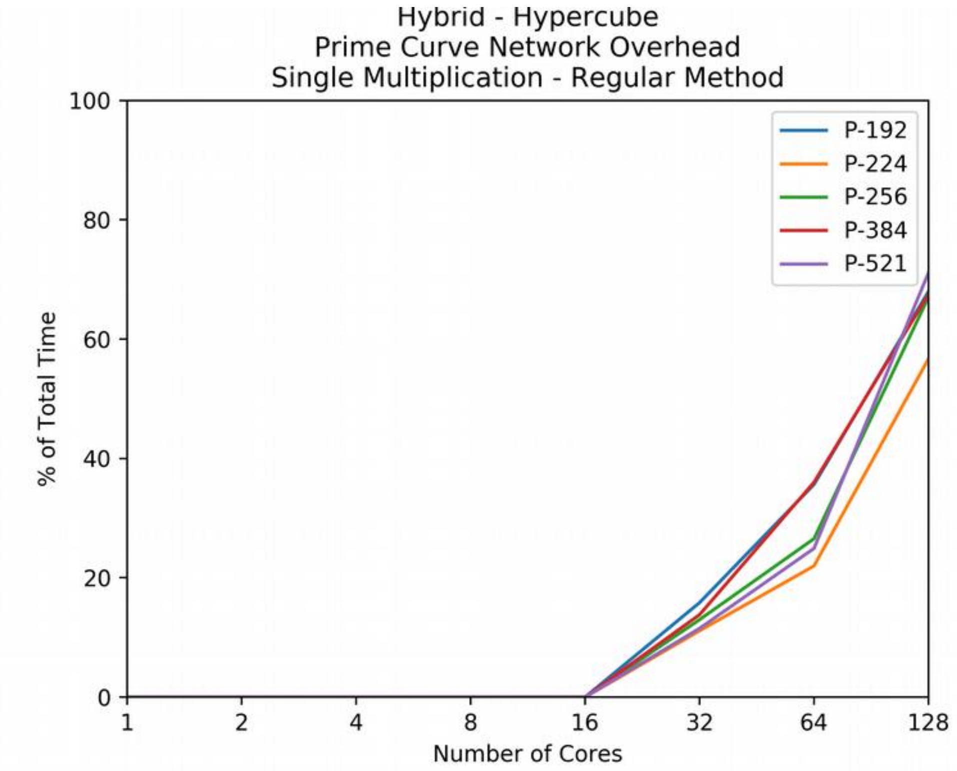
Hypercube Speedup

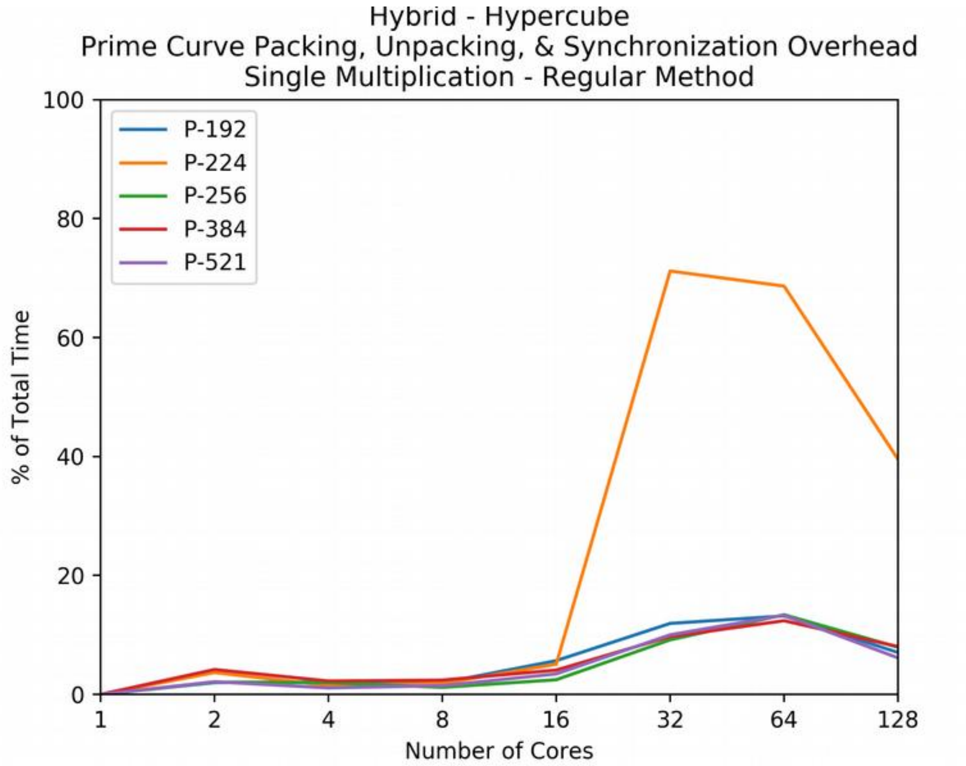

Hybrid - Hypercube
Prime Curve Speedup
Single Multiplication - Regular Method


Hybrid - Hypercube
Koblitz Curve Speedup
Single Multiplication - Regular Method

- Better speedup than MPI until 16 cores for prime curves and 8-16 cores for Koblitz curves
    - Performance impact for >8 cores may be due to frequent cache misses between processors

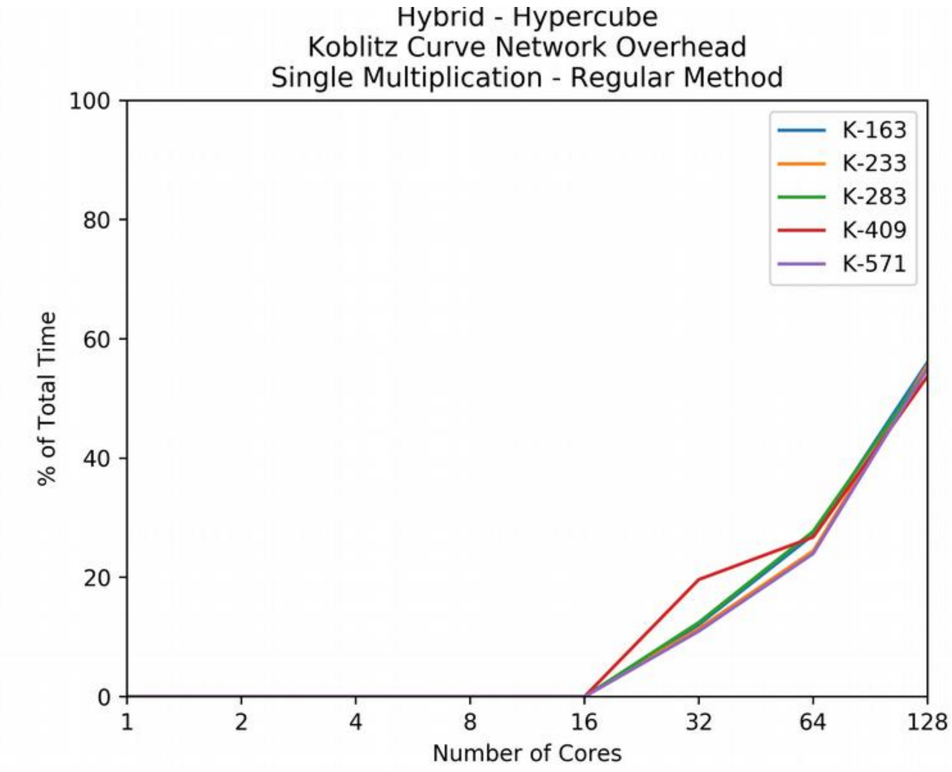
Hypercube Overhead

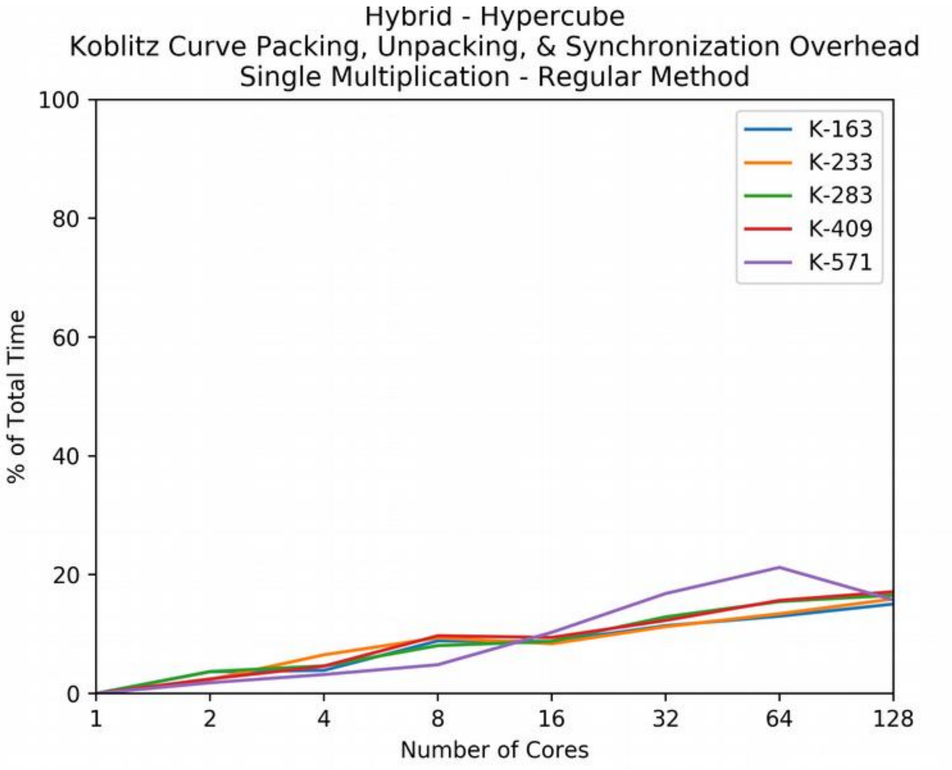

Hybrid - Hypercube
Prime Curve Network Overhead
Single Multiplication - Regular Method


Hybrid - Hypercube
Prime Curve Packing, Unpacking, & Synchronization Overhead
Single Multiplication - Regular Method

- OpenMP has less overhead compared to MPI
- Network delays with hybrid approach (>16 cores) quickly become significant

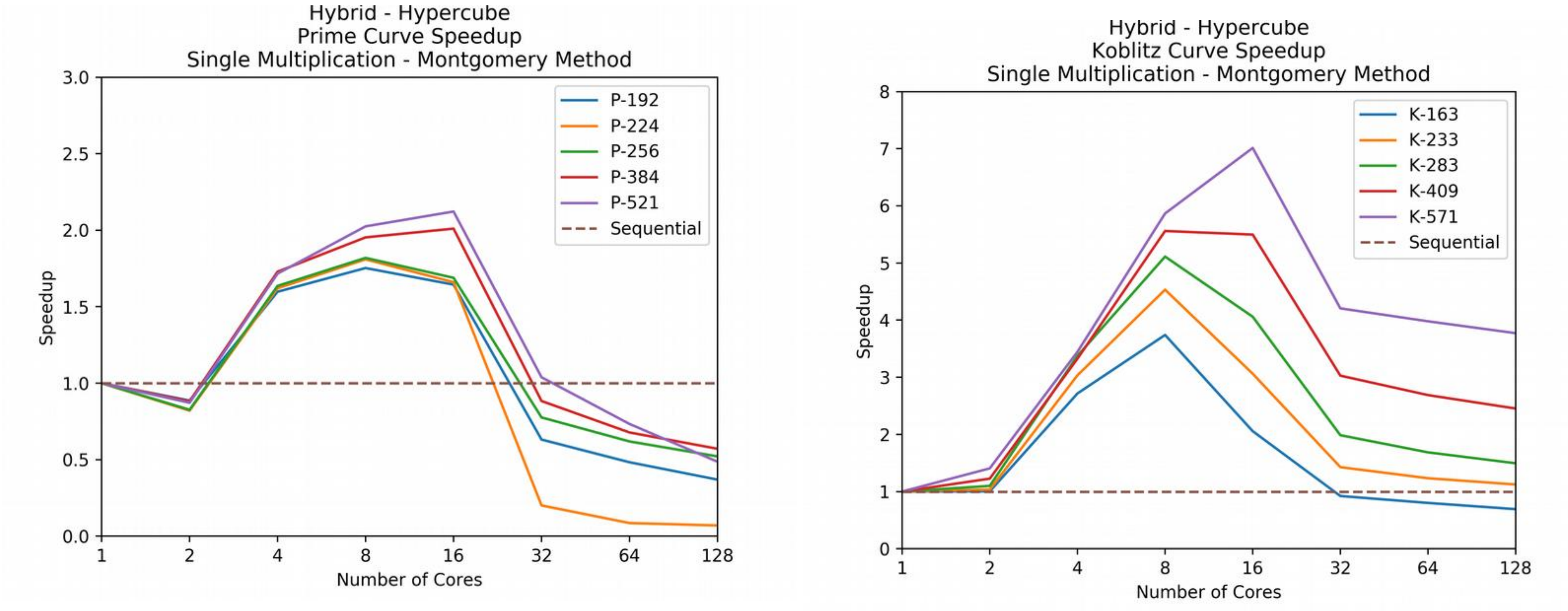
Hypercube Overhead





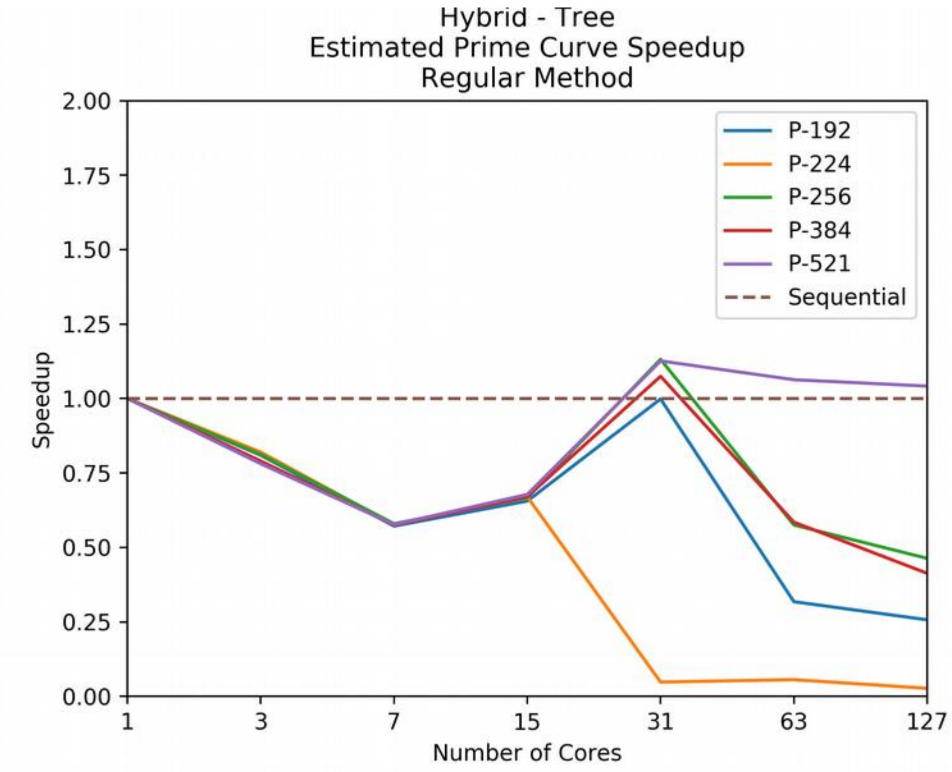
- Montgomery method shows less networking overhead, and more time spent on other overhead

Hypercube Speedup

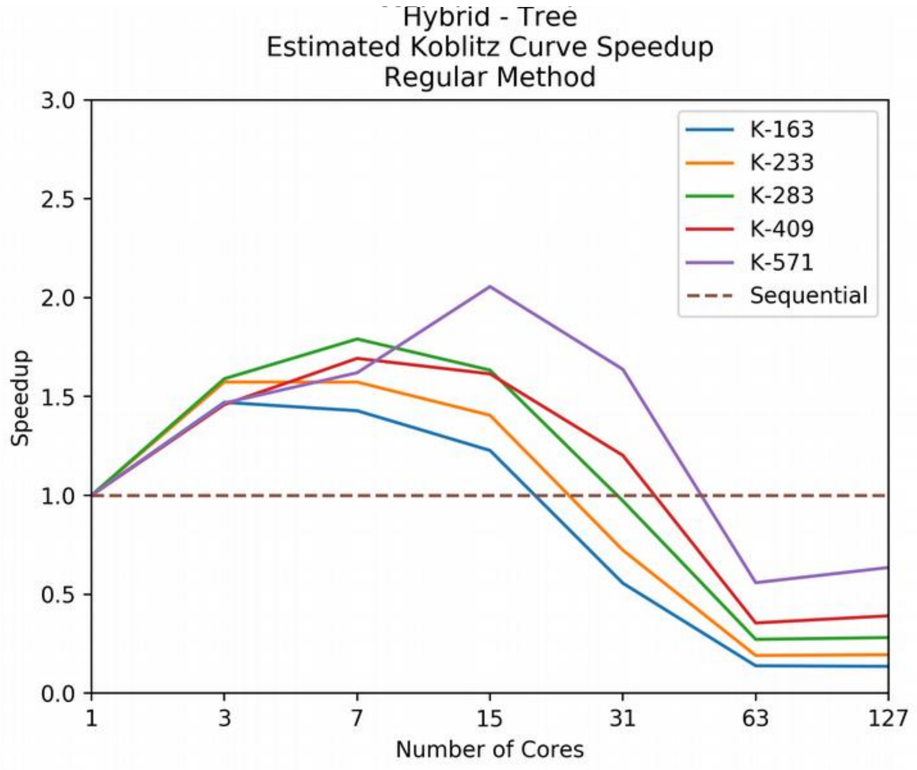


- Montgomery methods offer better speedup up to 8-16 cores with an initial performance hit at 2 cores compared to MPI

Tree Speedup



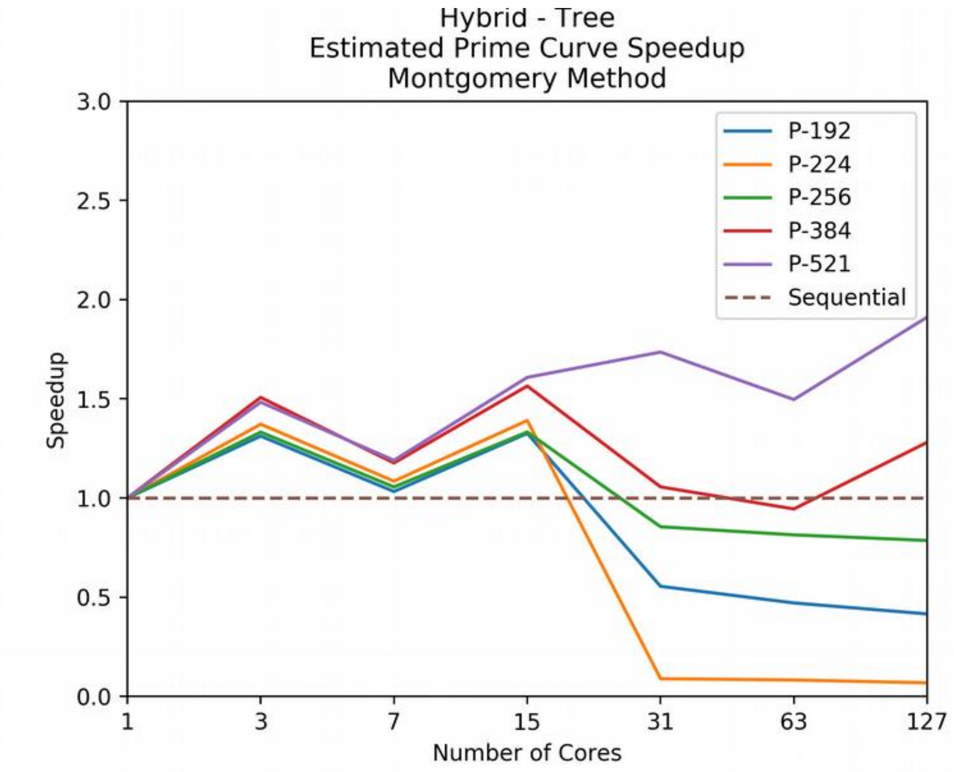
Hybrid - Tree
Estimated Prime Curve Speedup
Regular Method



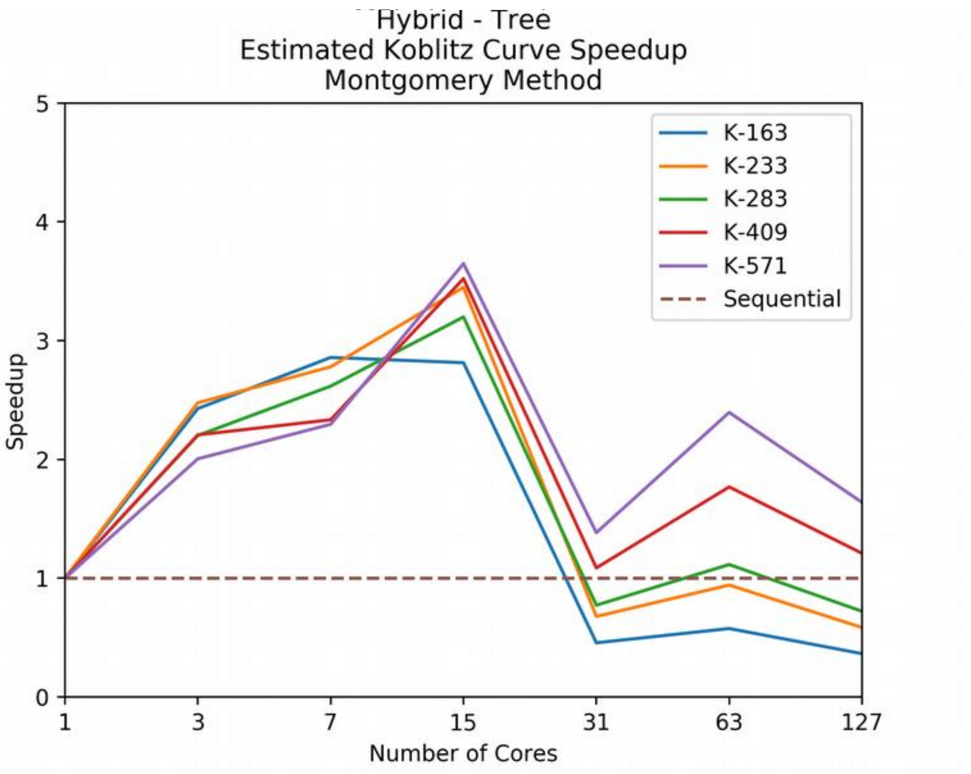
Hybrid - Tree
Estimated Koblitz Curve Speedup
Regular Method

- Tree performs worse than in MPI
  - ►Synchronization costs for a tree greater than speedup attainable from the parallel algorithm

Tree Speedup



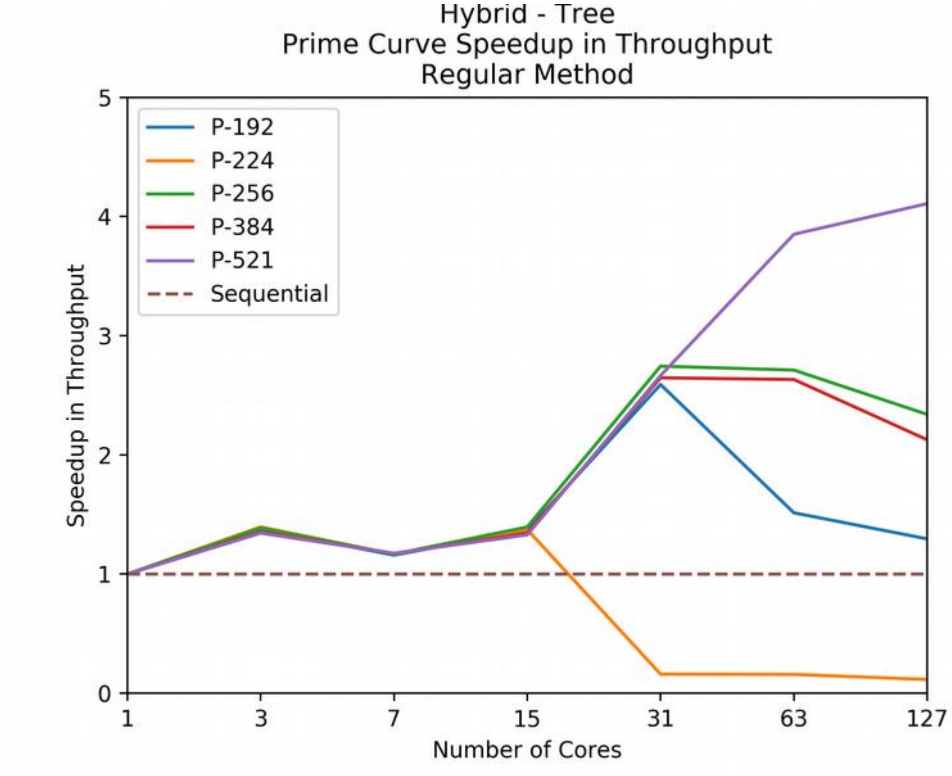
Hybrid - Tree
Estimated Prime Curve Speedup
Montgomery Method



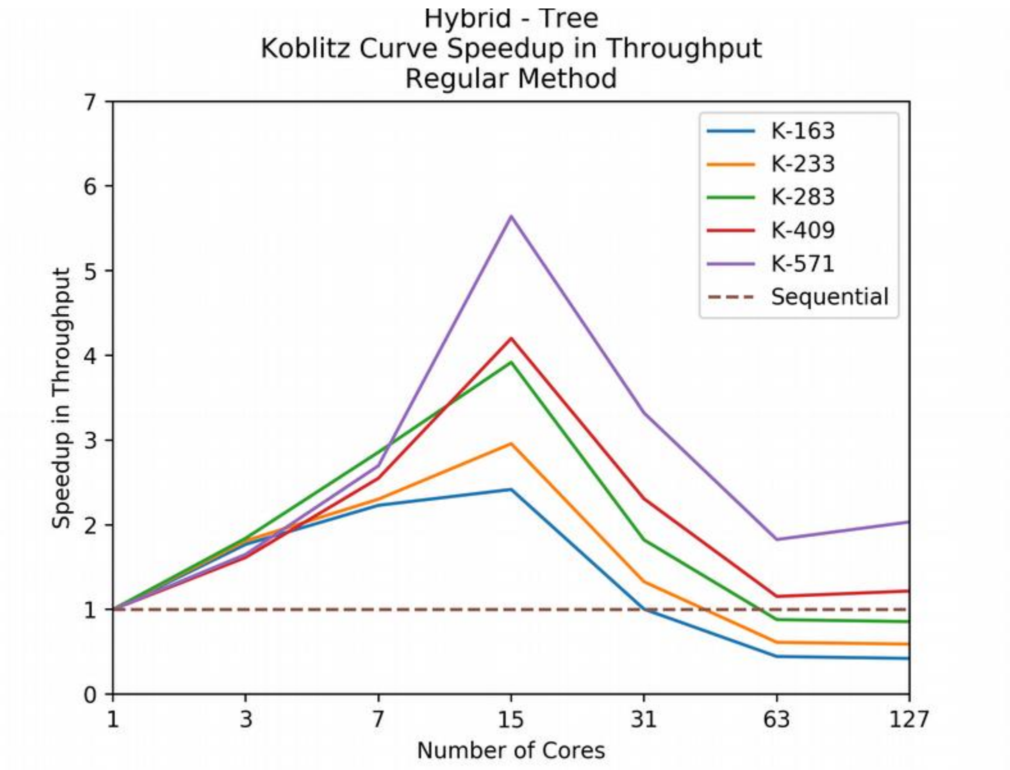
Hybrid - Tree
Estimated Koblitz Curve Speedup
Montgomery Method

- Tree performs worse than in MPI
  - ►Synchronization costs for a tree greater than speedup attainable from the parallel algorithm

Tree Throughput



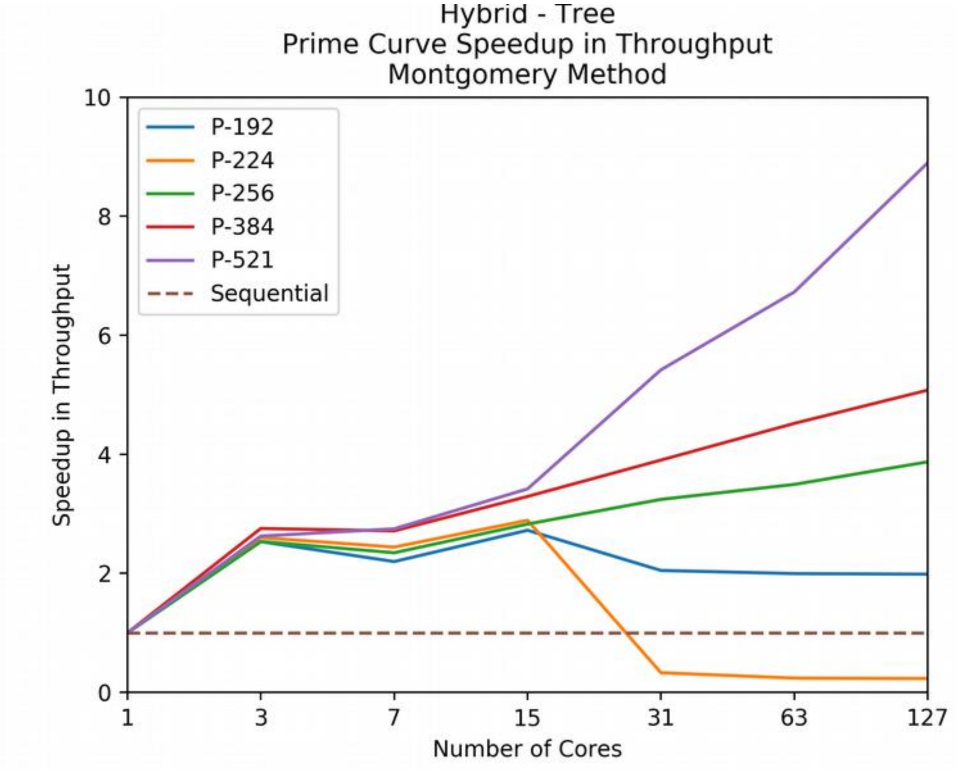
Hybrid - Tree
Prime Curve Speedup in Throughput
Regular Method

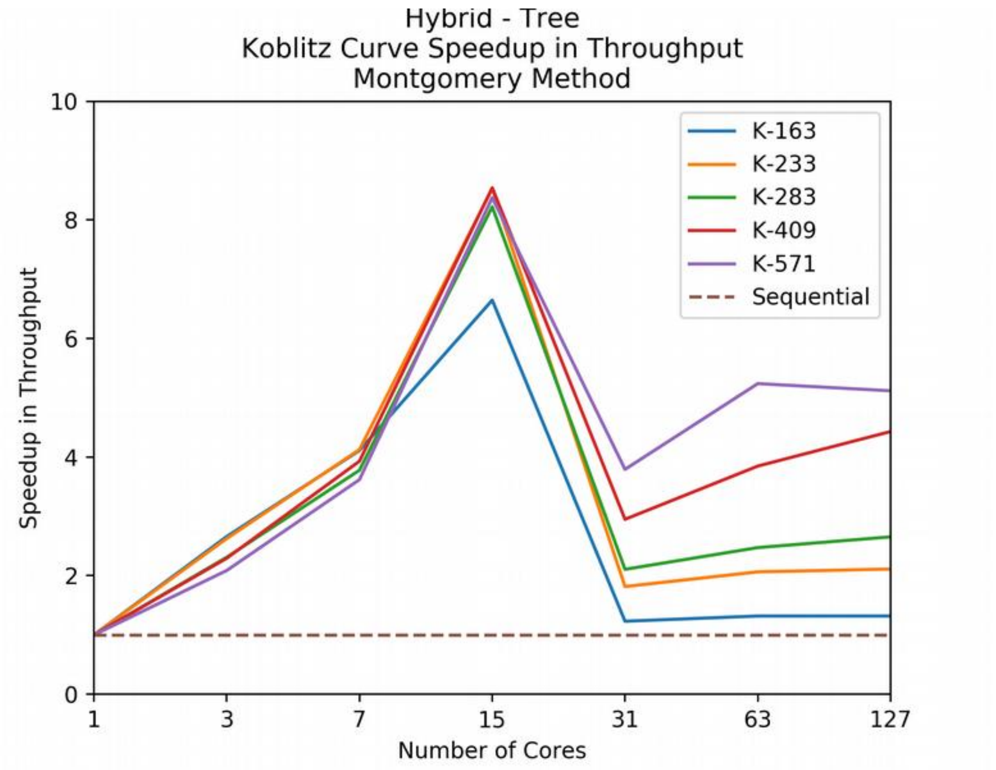


Hybrid - Tree
Koblitz Curve Speedup in Throughput
Regular Method

- Throughput for some curves comparable to throughput in MPI up to 15 cores
    - ►Synchronization delays with >15 cores limits throughput

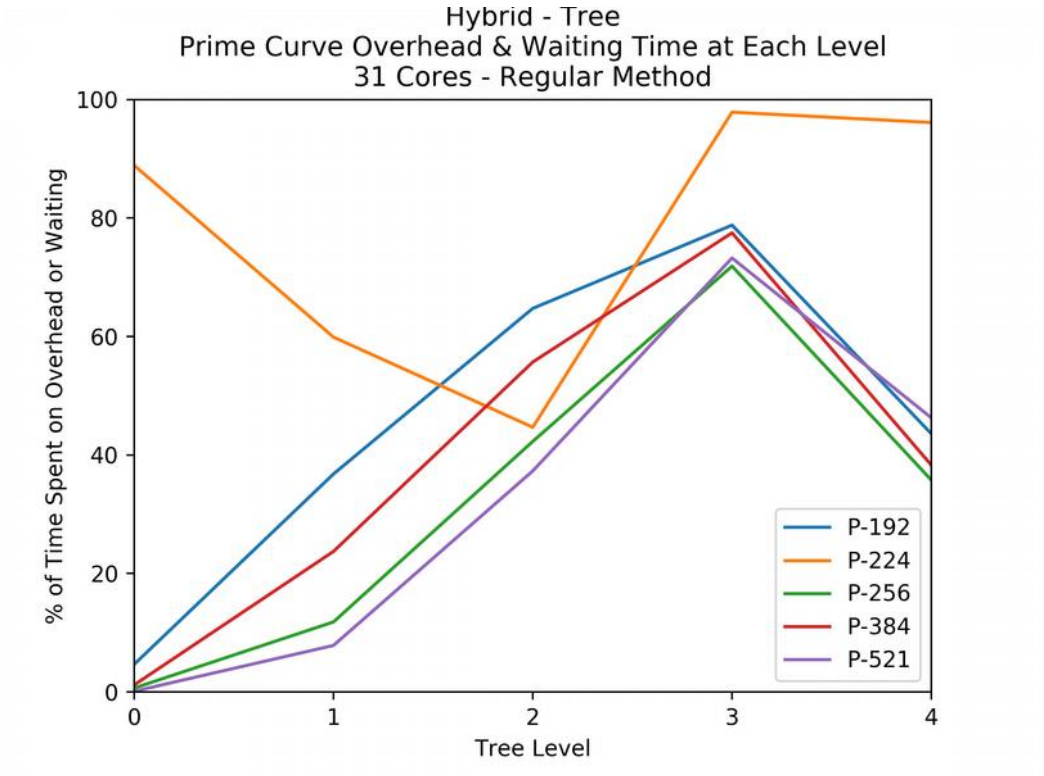
Tree Throughput

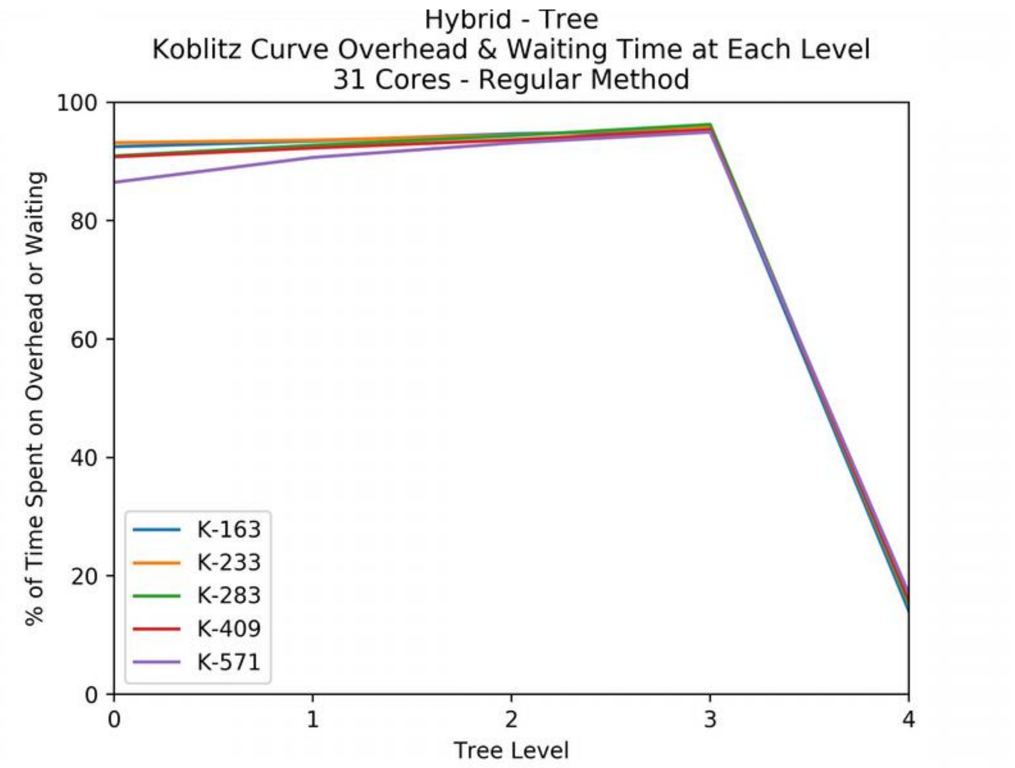




- Throughput for some Koblitz curves comparable to throughput in MPI up to 15 cores
  - ►Synchronization delays with >15 cores limits throughput

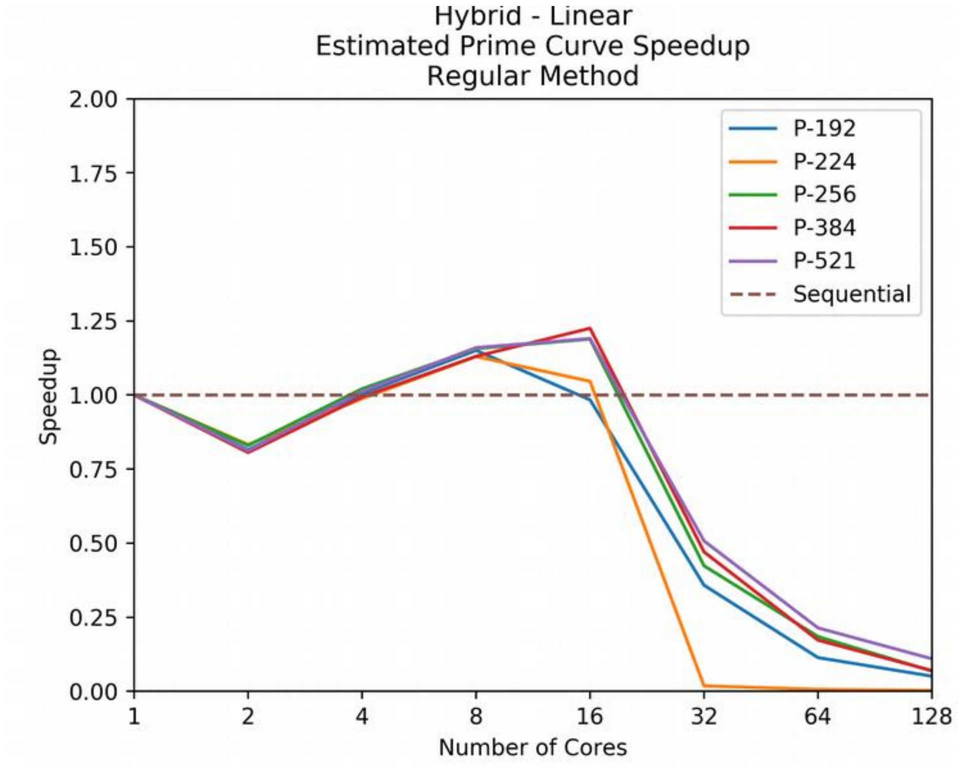
Time Spent Waiting or on Parallel Overhead in Tree

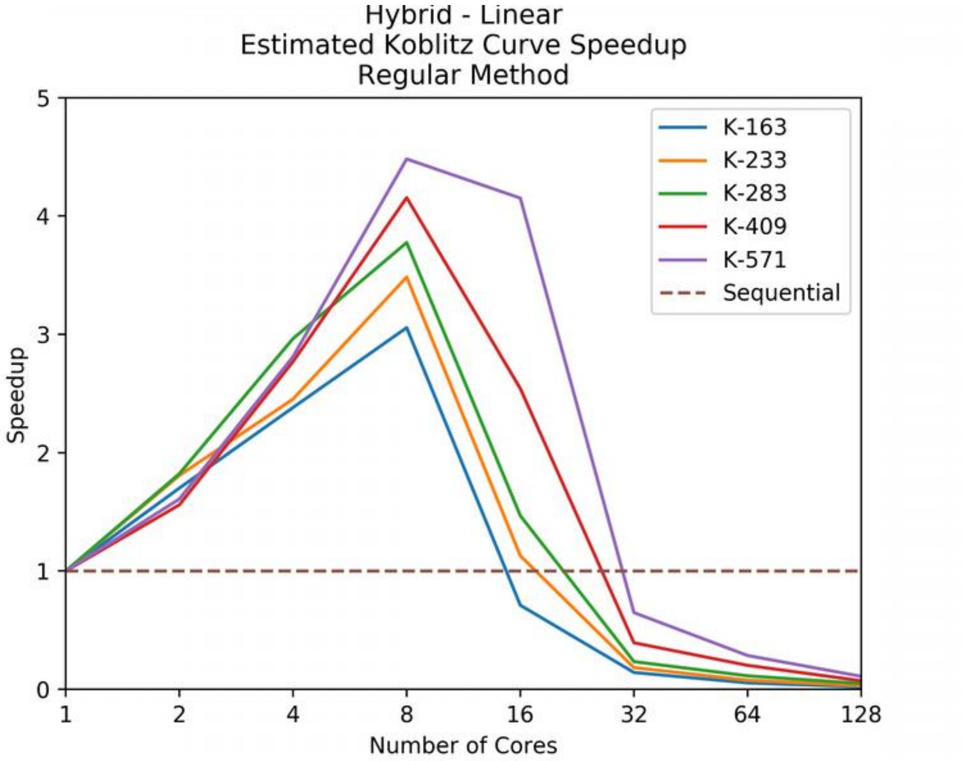

Hybrid - Tree
Prime Curve Overhead & Waiting Time at Each Level
31 Cores - Regular Method


Hybrid - Tree
Koblitz Curve Overhead & Waiting Time at Each Level
31 Cores - Regular Method

- Significant overhead costs and idle time (Koblitz curves)
  - ►Additional costs incurred from setting locks used for synchronization

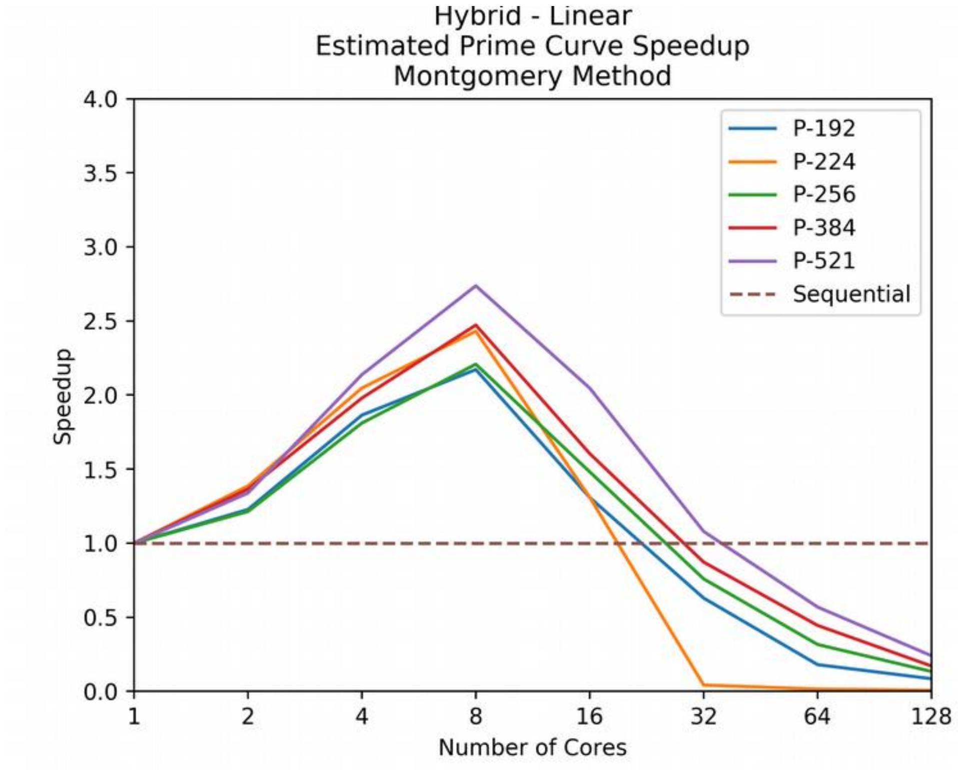
Linear Speedup



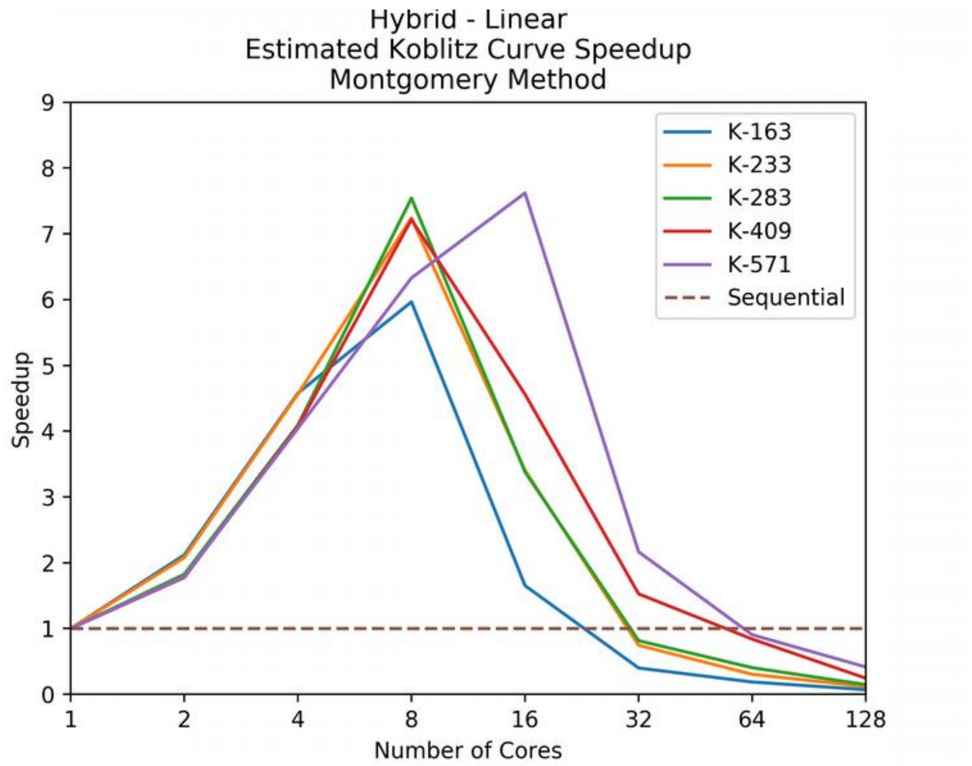


- Better speedup than in MPI with <16 cores for prime curves and 8 cores for Koblitz curves
  - ►For prime curves, parallel overhead overwhelms algorithm’s speedup when using 2-4 cores

Linear Speedup



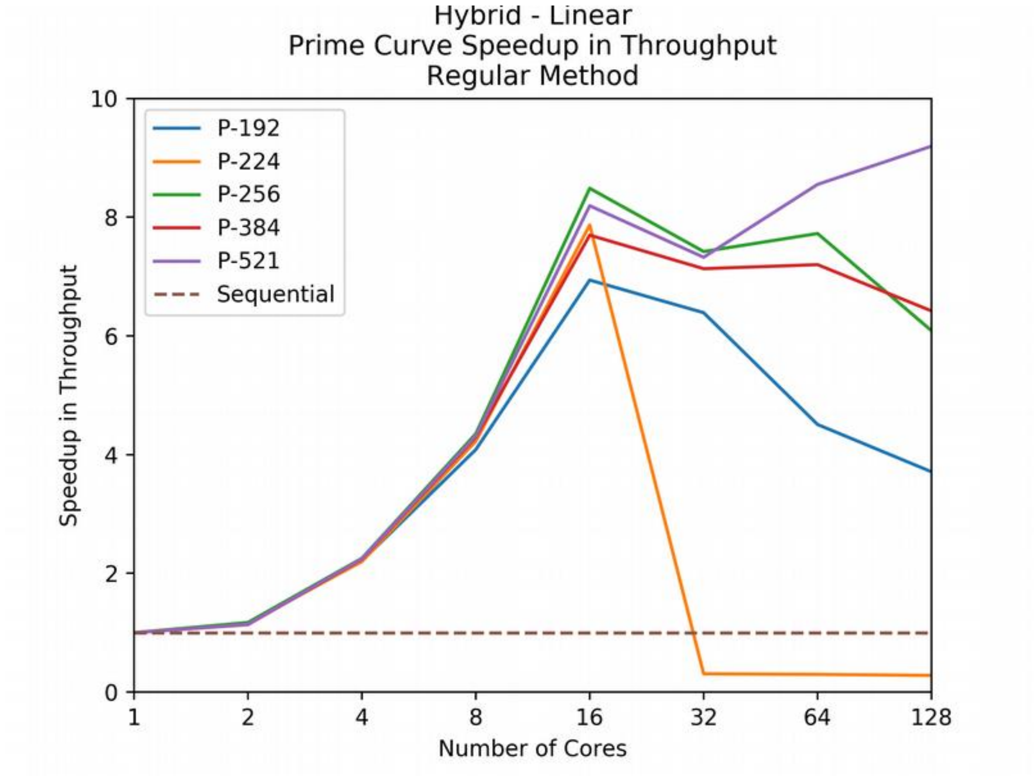
Hybrid - Linear
Estimated Prime Curve Speedup
Montgomery Method

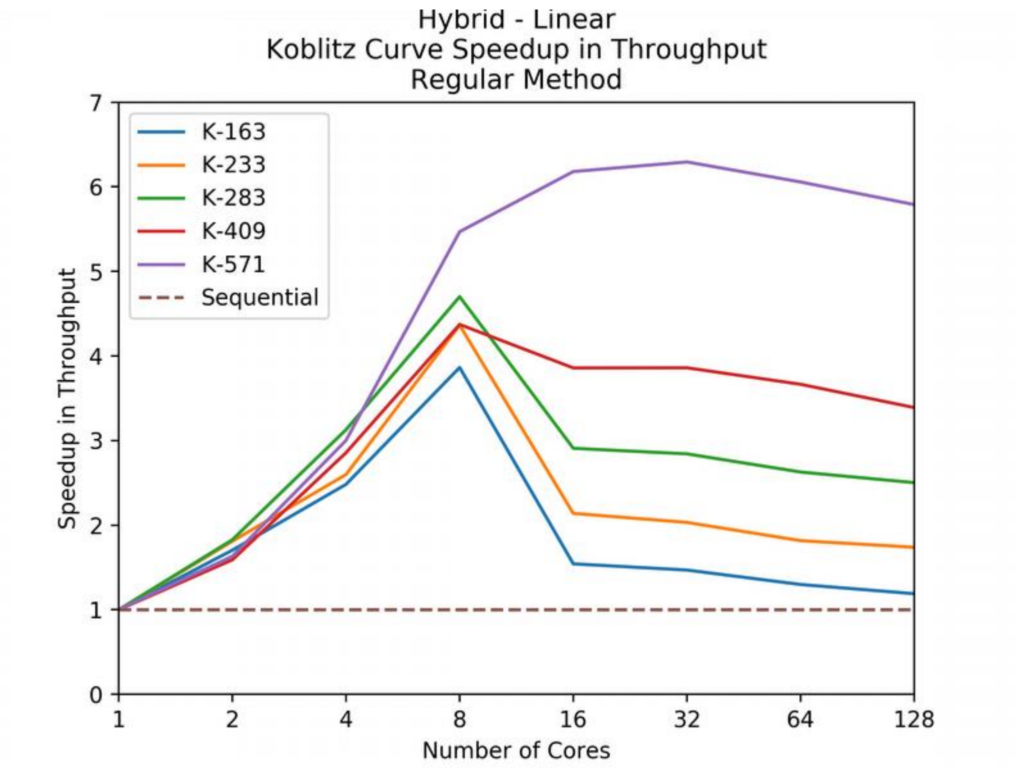


Hybrid - Linear
Estimated Koblitz Curve Speedup
Montgomery Method

- Surprisingly better speedup than a hypercube
  - Less synchronization costs
  - Performance hit at >8 cores

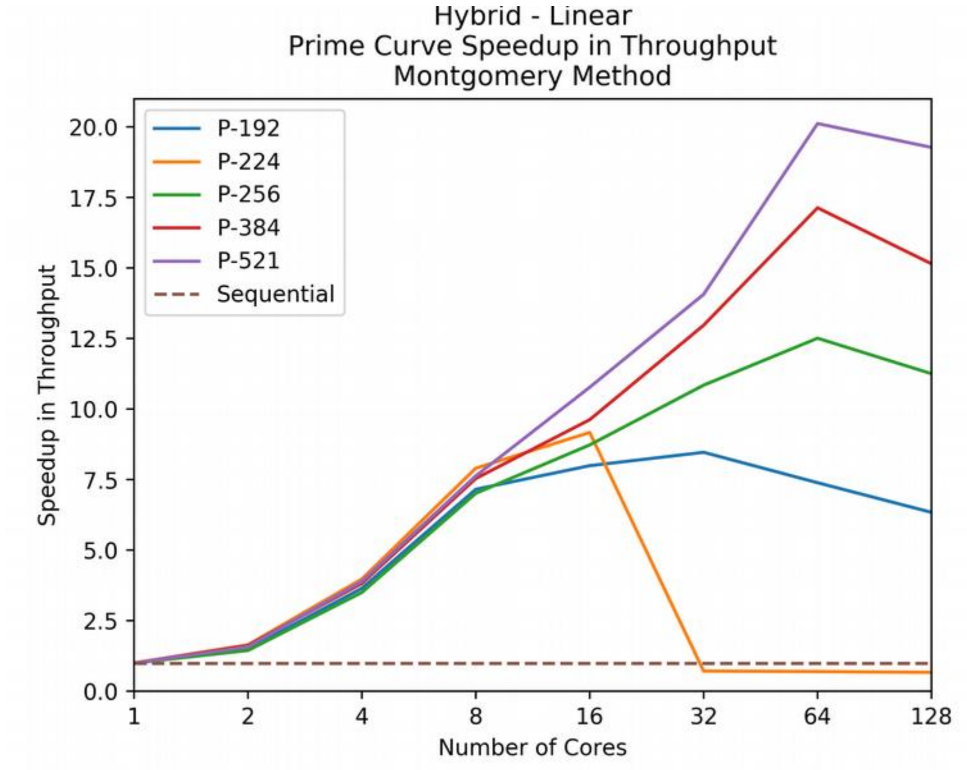
Linear Throughput

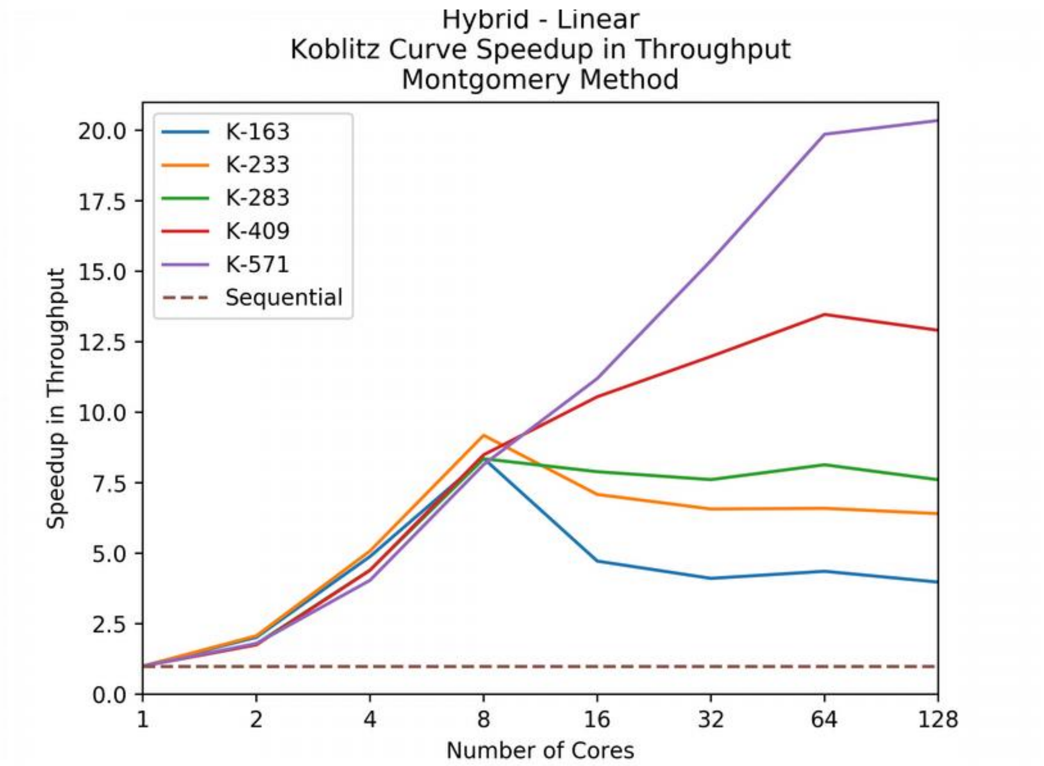




- Generally better throughput than when using MPI with linear array
  - ►Performance hit when hybrid approach is used and when two processors per compute node used

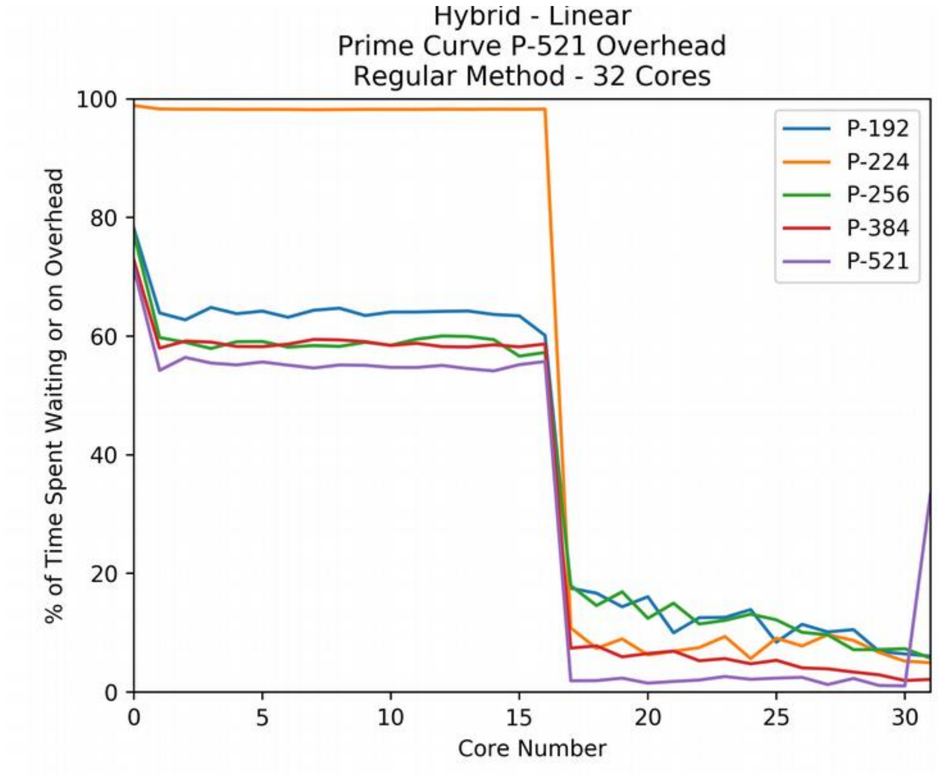
Linear Throughput

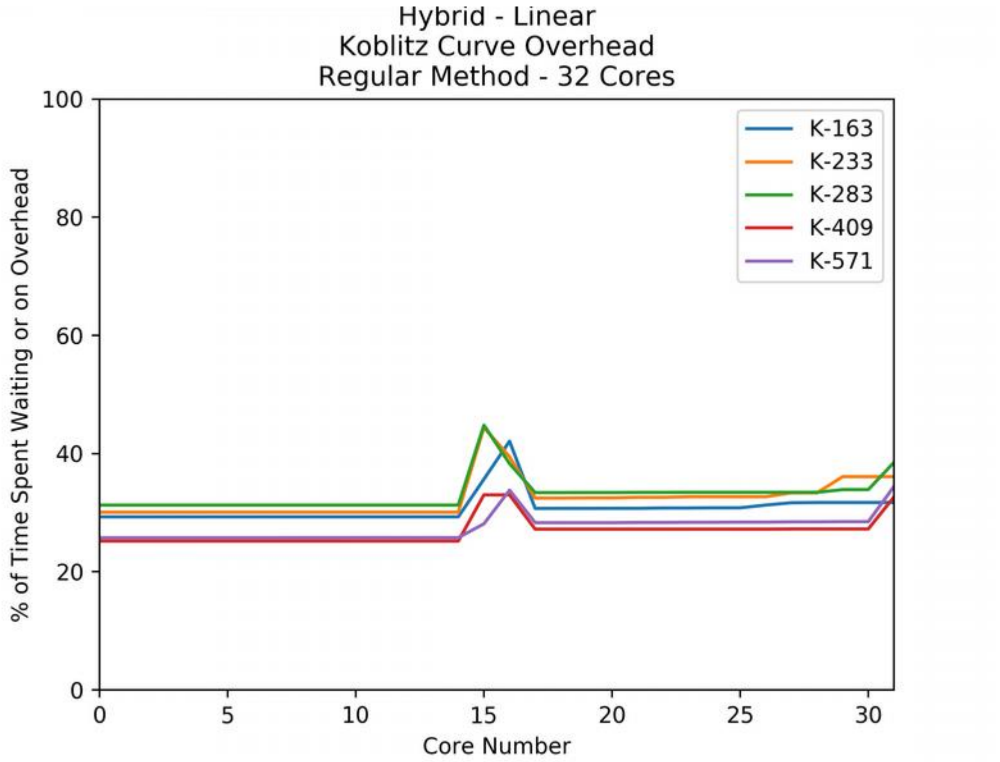

Hybrid - Linear
Prime Curve Speedup in Throughput
Montgomery Method


Hybrid - Linear
Koblitz Curve Speedup in Throughput
Montgomery Method

- Better throughput when using <8-16 cores than in MPI
  - ►Performance hit when hybrid approach is used and when two processors per compute node used

Linear Overhead





- Large overhead when utilizing multiple MPI nodes for prime curves corresponding to network delays
- Koblitz have nearly constant overhead for all cores with spikes near MPI node boundaries

Hybrid Conclusions

- Synchronization delays can be worse than networking delays in MPI in some cases
- Observed performance moving to 16 cores significantly impacted the hybrid approach
  - Frequent cache misses using multiple processors may be the cause for these results
- Linear array showed better speedup than other structures, but worse throughput than in MPI
  - Less overhead compared to other structures
- Merging MPI calls may not have been beneficial

Overall Conclusions

- Best logical structure depends on number of cores available, desired throughput, desired speedup, and curve type
  - ►Koblitz curves better suited for parallelization
  - ►Splitting cores sequentially best for maximizing throughput
  - ►MPI tree gives generally good balance between speedup and throughput, for many cores
  - ►OpenMP linear array gives generally good balance between speedup and throughput for few cores

Future & Related Work

- Large amount of time in a tree is spent waiting for other processors for non-leaves, and it may be possible to merge some non-leave nodes
- Combining topologies may yield better throughput results in some cases
- Parallelism at the point or field level is also possible using a fixed number of processors
- Multiple multiplications on the same point can use globally precomputed values for better performance
  - Key generation

Future & Related Work

- Better results can likely be achieved if suspected frequent cache misses due to dual-processor compute nodes are accounted for
    - ►One method to account for this is to use 2 MPI nodes per server (1 per processor), with 8 threads used per MPI node so MPI takes care of it
- Not merging MPI calls may be better suited for hypercubes and trees in the hybrid approach

References

- Keke Wu, Huiyun Li, Dingju Zhu: Fast and scalable parallel processing of scalar multiplication in elliptic curve cryptosystems. Security and Communication Networks 5(6): 648-657 (2012)
- Hankerson, Darrel R., Scott A. Vanstone, and A. J. Menezes. Guide to elliptic curve cryptography. New York: Springer, 2003. Print.
- Jerome A. Solinas: Efficient Arithmetic on Koblitz Curves. Des. Codes Cryptography 19(2/3): 195-249 (2000)
- Recommended Elliptic Curves For Federal Government Use. NIST Computer Security Resource Center. 1999.