Asynchronous Distributed Actor-based Approach to Jaccard Similarity for Genome Comparisons

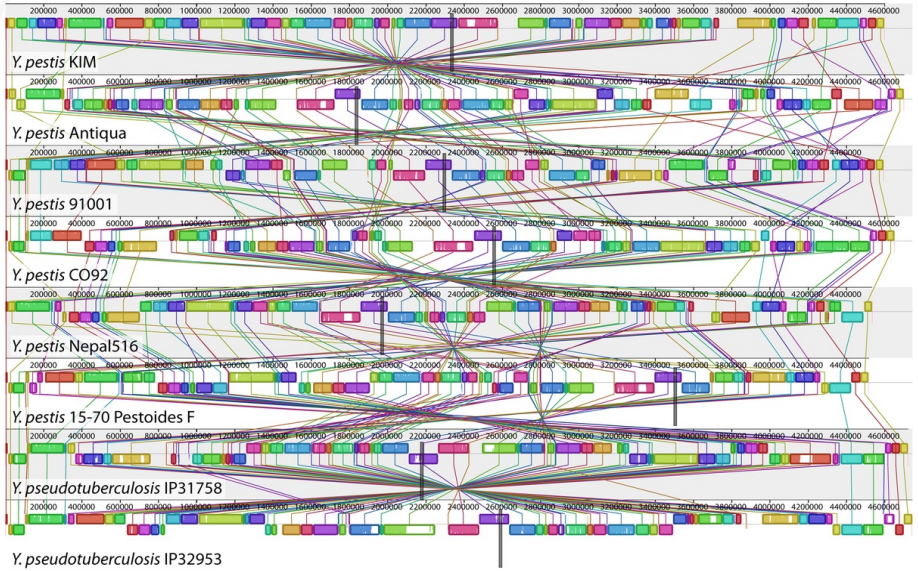
Youssef Elmougy, Akihiro Hayashi, Vivek Sarkar

Habanero Extreme Scale Software Research Lab
Georgia Institute of **Tech**nology

Corresponding Author: yelmougy3@gatech.edu

Importance of Genome Similarity and Genetic Distances

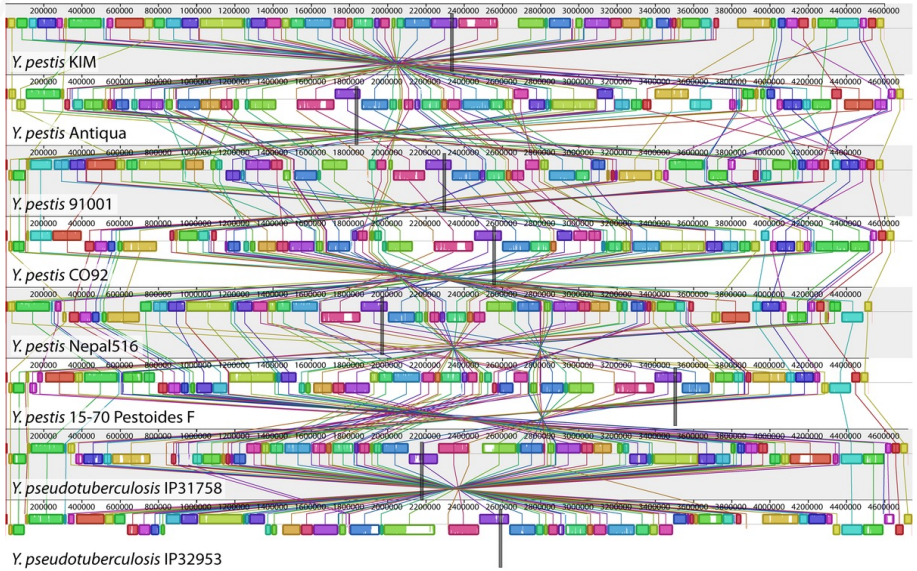
Computational Biology and Comparative Genomics



Assemblers
Metagenomic Profiling
Clustering
Retrieval of Sequencing Samples

Importance of Genome Similarity and Genetic Distances

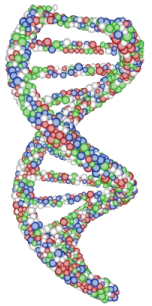
Computational Biology and Comparative Genomics



Assemblers
Metagenomic Profiling
Clustering
Retrieval of Sequencing Samples

Computing the genome similarity among two DNA sequencing data sets is assessed by computing their

Jaccard Similarity, $\mathcal{J}(X,Y) = \frac{|X \cap Y|}{|X \cup Y|}$

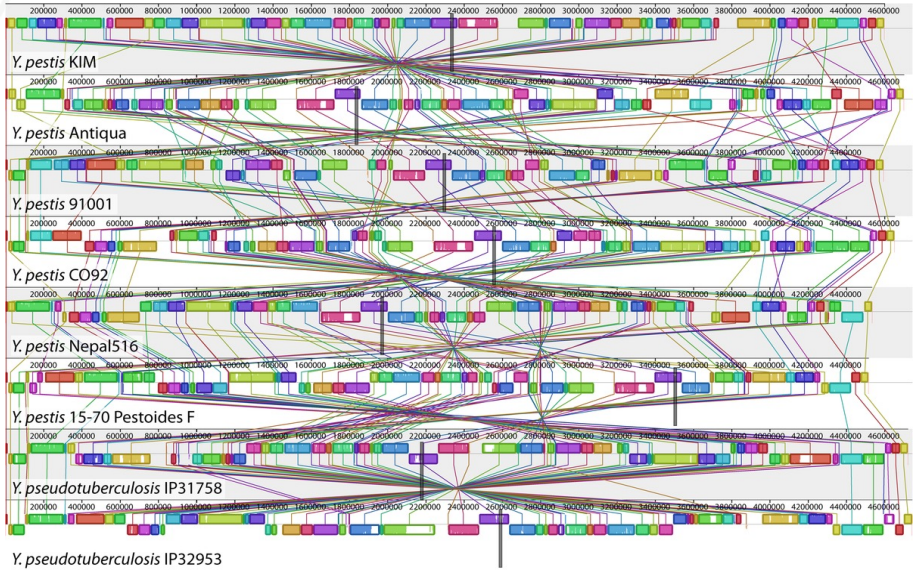


...TTTAGCCA...

...ACTAGCGA...

Importance of Genome Similarity and Genetic Distances

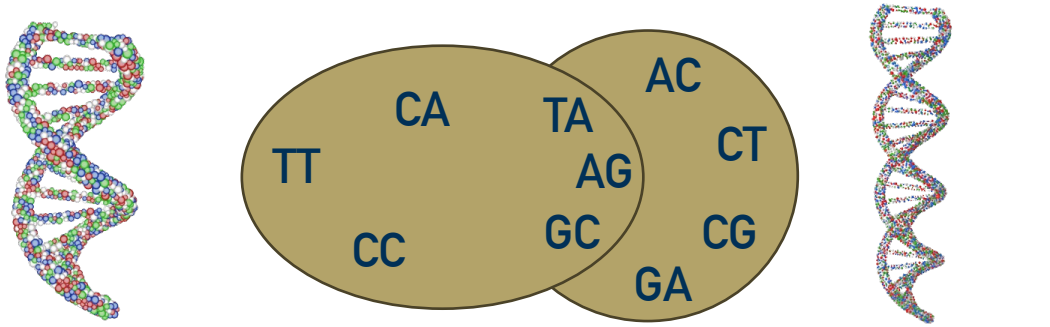
Computational Biology and Comparative Genomics



Assemblers
Metagenomic Profiling
Clustering
Retrieval of Sequencing Samples

Computing the genome similarity among two DNA sequencing data sets is assessed by computing their

Jaccard Similarity, $\mathcal{J}(X,Y) = \frac{|X \cap Y|}{|X \cup Y|}$



...TTTAGCCA...

...ACTAGCGA...

$$\mathcal{J}(\text{DNA}_1, \text{DNA}_2) = \frac{|\text{DNA}_1 \cap \text{DNA}_2|}{|\text{DNA}_1 \cup \text{DNA}_2|} = \frac{|\text{TA,AG,GC}|}{|\text{CA,TT,CC,TA,AG,GC,AC,CT,CG,GA}|} = 0.3$$

$$d_{\mathcal{J}} = 1 - \mathcal{J} = 1 - 0.3 = 0.7$$

Current Approaches are *NOT Efficient* for Scalable and Distributed Computation

Prior/Current Approaches:

Sequential or Single Server

All-Pairs[21]

Parallel Cell/B.E.[23]

Jaccard-PageRank[22]

MapReduce Implementation

Wikipedia Similarity Team[24]

3-Stage Approach for Set-Similarity Joins[25]

Implementations for Computational Biology and Comparative Genomics

GenomeAtScale[10]

BELLA[46]

Mash[5]

Dashing[11]

SOTA

Current Approaches are *NOT Efficient* for Scalable and Distributed Computation

Prior/Current Approaches:

Sequential or Single Server

All-Pairs[21] Parallel Cell/B.E.[23]

Jaccard-PageRank[22]

MapReduce Implementation

Wikipedia Similarity Team[24] 3-Stage Approach for Set-Similarity Joins[25]

Implementations for Computational Biology and Comparative Genomics

GenomeAtScale[10] BELLA[46]

Mash[5] Dashing[11]

SOTA

CHALLENGE

Continuous *growth of sequencing datasets* and *increase in number of genomes*

Inherently a computationally challenging problem

Current Approaches are *NOT Efficient* for Scalable and Distributed Computation

Prior/Current Approaches:

Sequential or Single Server

All-Pairs[21] Parallel Cell/B.E.[23]

Jaccard-PageRank[22]

MapReduce Implementation

Wikipedia Similarity Team[24] 3-Stage Approach for Set-Similarity Joins[25]

Implementations for Computational Biology and Comparative Genomics

GenomeAtScale[10] BELLA[46]

Mash[5] Dashing[11]

SOTA

CHALLENGE

Continuous *growth of sequencing datasets* and *increase in number of genomes*

Inherently a computationally challenging problem

This motivates the need for an ***efficient, fast, and scalable solution*** that can *process datasets of growing magnitudes* in *different domains*!

Our Main Contributions

Our Main Contributions

Distributed, Scalable, and Asynchronous Algorithm for Computing $\mathcal{J}$ and $d_{\mathcal{J}}$

based on the Actor-based Programming system

Our Main Contributions

Distributed, Scalable, and Asynchronous Algorithm for Computing $\mathcal{J}$ and $d_{\mathcal{J}}$

based on the Actor-based Programming system

Apply Algorithm to High-Scale Computations of Jaccard Similarity for Genome Comparisons and Genetic Distances

achieved 4.94× performance compared to SOTA

Our Main Contributions

Distributed, Scalable, and Asynchronous Algorithm for Computing $\mathcal{J}$ and $d_{\mathcal{J}}$

based on the Actor-based Programming system

Apply Algorithm to High-Scale Computations of Jaccard Similarity for Genome Comparisons and Genetic Distances

achieved 4.94× performance compared to SOTA

Perform a Deep Dive HWPC Study to Realize Performance Benefits

achieved 3.6× and 5.5× performance for execution time and HWPC results compared to SOTA

Jaccard Similarity Calculation

Jaccard Similarity Calculation

Jaccard Similarity, $\mathcal{J}(X,Y)$, is an operation that computes the overlap of two data sets X and Y by calculating the ratio between the *cardinality of their set intersection* and the *cardinality of their set union*.

$$\mathcal{J}(X,Y) = \frac{|X \cap Y|}{|X \cup Y|} = \frac{|X \cap Y|}{|X| + |Y| - |X \cap Y|}$$

Jaccard Similarity Calculation

Jaccard Similarity, $\mathcal{J}(X,Y)$, is an operation that computes the overlap of two data sets X and Y by calculating the ratio between the *cardinality of their set intersection* and the *cardinality of their set union*.

$$\mathcal{J}(X,Y) = \frac{|X \cap Y|}{|X \cup Y|} = \frac{|X \cap Y|}{|X| + |Y| - |X \cap Y|}$$

Jaccard Distance, $\mathrm{d}_{\mathcal{J}}$, calculates the dissimilarity between sets. $\mathrm{d}_{\mathcal{J}}(X,Y) = 1 - \mathcal{J}(X,Y)$

If both sets are empty or contain the exact same elements, then $\mathcal{J}(X,Y) = 1$ and $d_{\mathcal{J}} = 0$.

Jaccard Similarity Calculation

Jaccard Similarity, $\mathcal{J}(X,Y)$, is an operation that computes the overlap of two data sets X and Y by calculating the ratio between the *cardinality of their set intersection* and the *cardinality of their set union*.

$$\mathcal{J}(X,Y) = \frac{|X \cap Y|}{|X \cup Y|} = \frac{|X \cap Y|}{|X| + |Y| - |X \cap Y|}$$

Jaccard Distance, $\mathrm{d}_{\mathcal{J}}$, calculates the dissimilarity between sets. $\mathrm{d}_{\mathcal{J}}(X,Y) = 1 - \mathcal{J}(X,Y)$

If both sets are empty or contain the exact same elements, then $\mathcal{J}(X,Y) = 1$ and $d_{\mathcal{J}} = 0$.

Genome-based Similarity ← Scope of this work

Graph-based Similarity

Jaccard Similarity Calculation

Jaccard Similarity, $\mathcal{J}(X,Y)$, is an operation that computes the overlap of two data sets X and Y by calculating the ratio between the *cardinality of their set intersection* and the *cardinality of their set union*.

$$\mathcal{J}(X,Y) = \frac{|X \cap Y|}{|X \cup Y|} = \frac{|X \cap Y|}{|X| + |Y| - |X \cap Y|}$$

Jaccard Distance, $\mathrm{d}_{\mathcal{J}}$, calculates the dissimilarity between sets. $\mathrm{d}_{\mathcal{J}}(X,Y) = 1 - \mathcal{J}(X,Y)$

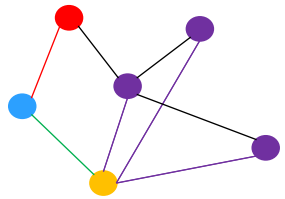
If both sets are empty or contain the exact same elements, then $\mathcal{J}(X,Y) = 1$ and $d_{\mathcal{J}} = 0$.

Genome-based Similarity ← Scope of this work

Graph-based Similarity

Rep. similarity of the neighborhoods of two vertices connected by an edge

$$\mathcal{J}_{u,v} = \frac{|U \cap V|}{|U \cup V|} = \frac{\gamma_{u,v}}{d_u + d_v - \gamma_{u,v}}, \forall (u,v) \in G$$



Jaccard Similarity Calculation

Jaccard Similarity, $\mathcal{J}(X,Y)$, is an operation that computes the overlap of two data sets X and Y by calculating the ratio between the *cardinality of their set intersection* and the *cardinality of their set union*.

$$\mathcal{J}(X,Y) = \frac{|X \cap Y|}{|X \cup Y|} = \frac{|X \cap Y|}{|X| + |Y| - |X \cap Y|}$$

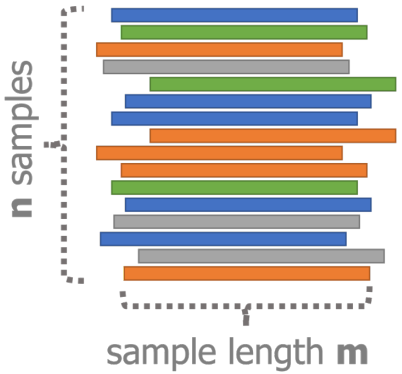
Jaccard Distance, $\mathrm{d}_{\mathcal{J}}$, calculates the dissimilarity between sets. $\mathrm{d}_{\mathcal{J}}(X,Y) = 1 - \mathcal{J}(X,Y)$

If both sets are empty or contain the exact same elements, then $\mathcal{J}(X,Y) = 1$ and $d_{\mathcal{J}} = 0$.

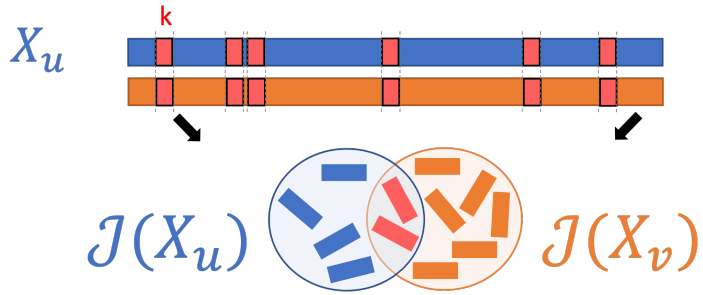
`Genome-based Similarity` ← Scope of this work

Rep. similarity between all pairs of multisets with respect to the fraction of k-mers shared between them

Data samples: $X = \{X_1, \ldots, X_n\}$ where $|X_i| = m$



$$\mathcal{J}(X_u, X_v) = \frac{|X_u \cap X_v|}{|X_u \cup X_u|} = \frac{\text{common k-mers}}{\text{total present k-mers}}, \qquad u, v \in \{1, \ldots n\}$$



State-of-the-Art Similarity Algorithm: GenomeAtScale [10]

The SOTA algorithm leverages the distributed memory numerical library *Cyclops Tensor Framework* (CTF) which provides routines for SpMM and SpMV (the main approach of the SOTA).

Data Samples: $\boldsymbol{X} = \{X_1, \ldots, X_n\}$

Indicator Matrix: $\boldsymbol{A} \in \mathbb{B}^{m \times n}$ $\quad a_{ij} = \begin{cases} 1 : i \in X_j \\ 0 : otherwise \end{cases}$

Similarity Matrix: $\boldsymbol{S} \in \mathbb{R}^{n \times n}$ $\quad s_{ij} = \mathcal{J}(X_i, X_j) = \frac{b_{ij}}{c_{ij}}$

where $\boldsymbol{B}, \boldsymbol{C} \in \mathbb{N}^{n \times n}$

$$|X_i \cap X_j| \Longrightarrow b_{ij} = \sum_k a_{ki} a_{kj} \Longrightarrow \boldsymbol{B} = \boldsymbol{A}^T \boldsymbol{A}$$

$$|X_i \cup X_j| \Longrightarrow c_{ij} = \sum_k a_{ki} + \sum_k a_{kj} - b_{ij}$$

[10] M. Besta, R. Kanakagiri, H. Mustafa, M. Karasikov, G. Rätsch, T. Hoefler, and E. Solomonik, "**Communication-Efficient Jaccard Similarity for High-Performance Distributed Genome Comparisons**," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2020, pp. 1122–1132.

State-of-the-Art Similarity Algorithm: GenomeAtScale [10]

The SOTA algorithm leverages the distributed memory numerical library *Cyclops Tensor Framework* (CTF) which provides routines for SpMM and SpMV (the main approach of the SOTA).

Data Samples: $\boldsymbol{X} = \{X_1, \ldots, X_n\}$

Indicator Matrix: $\boldsymbol{A} \in \mathbb{B}^{m \times n}$ $\quad a_{ij} = \begin{cases} 1 : i \in X_j \\ 0 : otherwise \end{cases}$

Similarity Matrix: $\boldsymbol{S} \in \mathbb{R}^{n \times n}$ $\quad s_{ij} = \mathcal{J}(X_i, X_j) = \frac{b_{ij}}{c_{ij}}$

where $\boldsymbol{B}, \boldsymbol{C} \in \mathbb{N}^{n \times n}$

$$|X_i \cap X_j| \Longrightarrow b_{ij} = \sum_k a_{ki} a_{kj} \Longrightarrow \boldsymbol{B} = \boldsymbol{A}^T \boldsymbol{A}$$

$$|X_i \cup X_j| \Longrightarrow c_{ij} = \sum_k a_{ki} + \sum_k a_{kj} - b_{ij}$$

GenomeAtScale algorithm steps to calculate $\boldsymbol{S}$:

[10] M. Besta, R. Kanakagiri, H. Mustafa, M. Karasikov, G. Rätsch, T. Hoefler, and E. Solomonik, "**Communication-Efficient Jaccard Similarity for High-Performance Distributed Genome Comparisons**," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2020, pp. 1122–1132.

State-of-the-Art Similarity Algorithm: GenomeAtScale [10]

The SOTA algorithm leverages the distributed memory numerical library *Cyclops Tensor Framework* (CTF) which provides routines for SpMM and SpMV (the main approach of the SOTA).

Data Samples: $\boldsymbol{X} = \{X_1, \ldots, X_n\}$

Indicator Matrix: $\boldsymbol{A} \in \mathbb{B}^{m \times n}$ $\quad a_{ij} = \begin{cases} 1 : i \in X_j \\ 0 : otherwise \end{cases}$

Similarity Matrix: $\boldsymbol{S} \in \mathbb{R}^{n \times n}$ $\quad s_{ij} = \mathcal{J}(X_i, X_j) = \frac{b_{ij}}{c_{ij}}$

where $\boldsymbol{B}, \boldsymbol{C} \in \mathbb{N}^{n \times n}$

$$|X_i \cap X_j| \Longrightarrow b_{ij} = \sum_k a_{ki} a_{kj} \Longrightarrow \boldsymbol{B} = \boldsymbol{A}^T \boldsymbol{A}$$

$$|X_i \cup X_j| \Longrightarrow c_{ij} = \sum_k a_{ki} + \sum_k a_{kj} - b_{ij}$$

GenomeAtScale algorithm steps to calculate $\boldsymbol{S}$:

(1) Divide $\boldsymbol{A}$'s rows into batches with $\hat{m}$ rows and loop through each batch;

[10] M. Besta, R. Kanakagiri, H. Mustafa, M. Karasikov, G. Rätsch, T. Hoefler, and E. Solomonik, "**Communication-Efficient Jaccard Similarity for High-Performance Distributed Genome Comparisons**," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2020, pp. 1122–1132.

State-of-the-Art Similarity Algorithm: GenomeAtScale [10]

The SOTA algorithm leverages the distributed memory numerical library *Cyclops Tensor Framework* (CTF) which provides routines for SpMM and SpMV (the main approach of the SOTA).

Data Samples: $\boldsymbol{X} = \{X_1, \ldots, X_n\}$

Indicator Matrix: $\boldsymbol{A} \in \mathbb{B}^{m \times n}$ $\quad a_{ij} = \begin{cases} 1 : i \in X_j \\ 0 : otherwise \end{cases}$

Similarity Matrix: $\boldsymbol{S} \in \mathbb{R}^{n \times n}$ $\quad s_{ij} = \mathcal{J}(X_i, X_j) = \frac{b_{ij}}{c_{ij}}$

where $\boldsymbol{B}, \boldsymbol{C} \in \mathbb{N}^{n \times n}$

$$|X_i \cap X_j| \Longrightarrow b_{ij} = \sum_k a_{ki} a_{kj} \Longrightarrow \boldsymbol{B} = \boldsymbol{A}^T \boldsymbol{A}$$

$$|X_i \cup X_j| \Longrightarrow c_{ij} = \sum_k a_{ki} + \sum_k a_{kj} - b_{ij}$$

GenomeAtScale algorithm steps to calculate $\boldsymbol{S}$:

(1) Divide $\boldsymbol{A}$'s rows into batches with $\hat{m}$ rows and loop through each batch;

(2) Remove zero rows within the batch using a distributed sparse vector;

[10] M. Besta, R. Kanakagiri, H. Mustafa, M. Karasikov, G. Rätsch, T. Hoefler, and E. Solomonik, "**Communication-Efficient Jaccard Similarity for High-Performance Distributed Genome Comparisons**," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2020, pp. 1122–1132.

State-of-the-Art Similarity Algorithm: GenomeAtScale [10]

The SOTA algorithm leverages the distributed memory numerical library *Cyclops Tensor Framework* (CTF) which provides routines for SpMM and SpMV (the main approach of the SOTA).

Data Samples: $\boldsymbol{X} = \{X_1, \ldots, X_n\}$

Indicator Matrix: $\boldsymbol{A} \in \mathbb{B}^{m \times n}$ $\quad a_{ij} = \begin{cases} 1 : i \in X_j \\ 0 : otherwise \end{cases}$

Similarity Matrix: $\boldsymbol{S} \in \mathbb{R}^{n \times n}$ $\quad s_{ij} = \mathcal{J}(X_i, X_j) = \frac{b_{ij}}{c_{ij}}$

where $\boldsymbol{B}, \boldsymbol{C} \in \mathbb{N}^{n \times n}$

$$|X_i \cap X_j| \Longrightarrow b_{ij} = \sum_k a_{ki} a_{kj} \Longrightarrow \boldsymbol{B} = \boldsymbol{A}^T \boldsymbol{A}$$

$$|X_i \cup X_j| \Longrightarrow c_{ij} = \sum_k a_{ki} + \sum_k a_{kj} - b_{ij}$$

GenomeAtScale algorithm steps to calculate $\boldsymbol{S}$:

(1) Divide $\boldsymbol{A}$'s rows into batches with $\hat{m}$ rows and loop through each batch;

(2) Remove zero rows within the batch using a distributed sparse vector;

(3) Compress row segments with bitmasking;

[10] M. Besta, R. Kanakagiri, H. Mustafa, M. Karasikov, G. Rätsch, T. Hoefler, and E. Solomonik, "**Communication-Efficient Jaccard Similarity for High-Performance Distributed Genome Comparisons**," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2020, pp. 1122–1132.

State-of-the-Art Similarity Algorithm: GenomeAtScale [10]

The SOTA algorithm leverages the distributed memory numerical library *Cyclops Tensor Framework* (CTF) which provides routines for SpMM and SpMV (the main approach of the SOTA).

Data Samples: $\boldsymbol{X} = \{X_1, \ldots, X_n\}$

Indicator Matrix: $\boldsymbol{A} \in \mathbb{B}^{m \times n}$ $\quad a_{ij} = \begin{cases} 1 : i \in X_j \\ 0 : otherwise \end{cases}$

Similarity Matrix: $\boldsymbol{S} \in \mathbb{R}^{n \times n}$ $\quad s_{ij} = \mathcal{J}(X_i, X_j) = \frac{b_{ij}}{c_{ij}}$

where $\boldsymbol{B}, \boldsymbol{C} \in \mathbb{N}^{n \times n}$

$$|X_i \cap X_j| \Longrightarrow b_{ij} = \sum_k a_{ki} a_{kj} \Longrightarrow \boldsymbol{B} = \boldsymbol{A}^T \boldsymbol{A}$$

$$|X_i \cup X_j| \Longrightarrow c_{ij} = \sum_k a_{ki} + \sum_k a_{kj} - b_{ij}$$

GenomeAtScale algorithm steps to calculate $\boldsymbol{S}$:

(1) Divide $\boldsymbol{A}$'s rows into batches with $\hat{m}$ rows and loop through each batch;

(2) Remove zero rows within the batch using a distributed sparse vector;

(3) Compress row segments with bitmasking;

(4) Compute and accumulate partial scores into $\boldsymbol{B}$ and $\boldsymbol{C}$ matrices;

[10] M. Besta, R. Kanakagiri, H. Mustafa, M. Karasikov, G. Rätsch, T. Hoefler, and E. Solomonik, "**Communication-Efficient Jaccard Similarity for High-Performance Distributed Genome Comparisons**," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2020, pp. 1122–1132.

State-of-the-Art Similarity Algorithm: GenomeAtScale [10]

The SOTA algorithm leverages the distributed memory numerical library *Cyclops Tensor Framework* (CTF) which provides routines for SpMM and SpMV (the main approach of the SOTA).

Data Samples: $\boldsymbol{X} = \{X_1, \ldots, X_n\}$

Indicator Matrix: $\boldsymbol{A} \in \mathbb{B}^{m \times n}$ $\quad a_{ij} = \begin{cases} 1 : i \in X_j \\ 0 : otherwise \end{cases}$

Similarity Matrix: $\boldsymbol{S} \in \mathbb{R}^{n \times n}$ $\quad s_{ij} = \mathcal{J}(X_i, X_j) = \frac{b_{ij}}{c_{ij}}$

where $\boldsymbol{B}, \boldsymbol{C} \in \mathbb{N}^{n \times n}$

$$|X_i \cap X_j| \Longrightarrow b_{ij} = \sum_k a_{ki} a_{kj} \Longrightarrow \boldsymbol{B} = \boldsymbol{A}^T \boldsymbol{A}$$

$$|X_i \cup X_j| \Longrightarrow c_{ij} = \sum_k a_{ki} + \sum_k a_{kj} - b_{ij}$$

GenomeAtScale algorithm steps to calculate $\boldsymbol{S}$:

(1) Divide $\boldsymbol{A}$'s rows into batches with $\hat{m}$ rows and loop through each batch;

(2) Remove zero rows within the batch using a distributed sparse vector;

(3) Compress row segments with bitmasking;

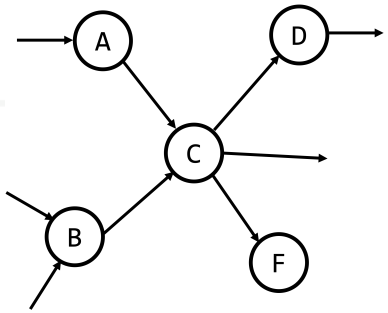
(4) Compute and accumulate partial scores into $\boldsymbol{B}$ and $\boldsymbol{C}$ matrices;

(5) Derive the final similarity scores $\boldsymbol{S}$ based on $\boldsymbol{B}$ and $\boldsymbol{C}$ matrices.

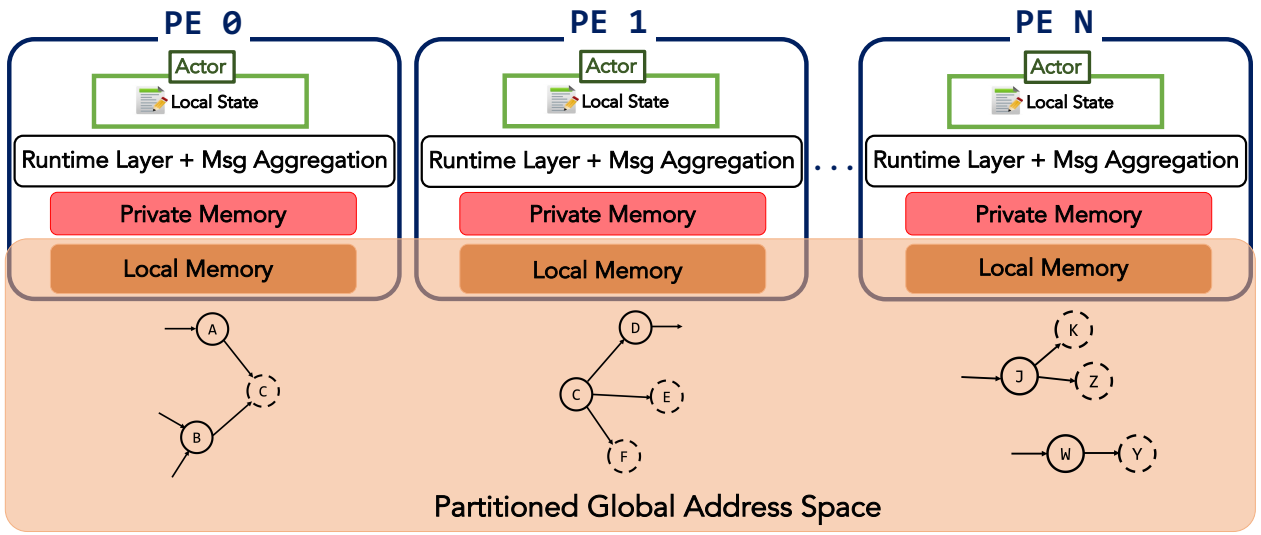
[10] M. Besta, R. Kanakagiri, H. Mustafa, M. Karasikov, G. Rätsch, T. Hoefler, and E. Solomonik, "**Communication-Efficient Jaccard Similarity for High-Performance Distributed Genome Comparisons**," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2020, pp. 1122–1132.

Actor-Based Programming System: Backend of Our Approach

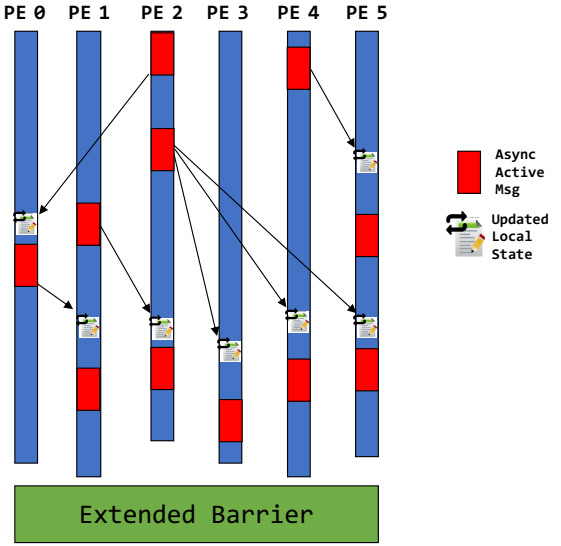
Sample Graph



Distributed Actor Runtime



FA-BSP Execution Model



Move compute to data via asynchronous active messages

- Presents a lightweight, asynchronous computation model
- Utilizes fine-grained asynchronous actor messages to express point-to-point remote operations
- Treats actors as primitives of computation, where actors are inherently isolated and share no mutable state
- Actors process messages sequentially within its mailbox, thereby avoiding data races and synchronization
- The unit of computation (PE or Actor) has a one-to-one relation to a physical CPU core

NOTE: “Actor” and “Selector” will be used interchangeably

[28] “Hclib-actor documentation,” https://hclib-actor.com, 2022.
[31] S. R. Paul, A. Hayashi, K. Chen, Y. Elmougy, and V. Sarkar, “**A Fine-grained Asynchronous Bulk Synchronous Parallelism Model for PGAS Applications**,” *Journal of Computational Science*, vol. 69, p. 102014, 2023.

Distributed, Asynchronous, and Scalable Actor-Based Jaccard Similarity for Genome Comparisons

Our approach is the first solution that employs a novel distributed and asynchronous algorithm resulting in just **two computation supersteps** compared to synchronous BSP-based approaches utilizing many barriers.

Distributed, Asynchronous, and Scalable Actor-Based Jaccard Similarity for Genome Comparisons

Our approach is the first solution that employs a novel distributed and asynchronous algorithm resulting in just **two computation supersteps** compared to synchronous BSP-based approaches utilizing many barriers.

```
/* Input: (shared) transposed (n x m) matrix A in CSR format with offsets stored
        in A->offset and nonzeros stored in A->nonzero
 * Output: (shared) (n x n) matrix J with jaccard similarities */

/* Utility functions
   * my_pe() : returns PE number of calling PE
   * n_pes() : returns number of PEs running in the application
   * get_remote_pe(x) : returns owner PE for element x
   * get_local_index(x) : returns local index in shared array for element x
   * get_global_id(x) : returns global ID for element x
   * binary_search(arr[:], x) : returns array index for element x if found
   * barrier() : blocks until async local/remote ops are completed on all PEs
   * d(x) : returns degree of element x (number of nonzeros)
*/
```

```
// initialization and start of the Selector instance
Selector<1> jac_selector;

int sender_PE = my_pe();
int n_local_rows = n / n_pes();
for (int v = 0; v < n_local_rows; v++) { // loop through locally stored rows
  int v_global = get_global_id(v); // get global ID for v (local operation)
  // get offset indices for the row
  for (int k = A->offset[v]; k < A->offset[v+1]; k++) {
    int u = A->nonzero[k]; // get nonzero using offset index (local operation)
    /* calculate intersection, the loop in L30 & L48 is application-dependant:
     * for general jaccard measures, the nonzeros of v are looped through
         for(int kk = A->offset[index]; kk < A->offset[index+1]; kk++){...}
     * for genome-based similarity, all pairs of samples are looped through
         for(int next_sample = v_id; next_sample < n; next_sample++){...} */
    for (int kk = A->offset[v]; kk < A->offset[v+1]; kk++) {
      int v_nonzero = A->nonzero[kk]; if (v_nonzero == u) continue;
      int remote_PE = get_remote_pe(u);
      // asynchronous msg sent to loop through shared array on remote_PE
      jac_selector.send(0, remote_PE, [=]() {
        if (binary_search(A->nonzero[A->offset[get_local_index(u)] :
                          A->offset[get_local_index(u)+1]], v_nonzero))
          // found common element, update local counter
          intersection[get_local_index(v_global), get_local_index(u)]++;
      });
} } }

// automatic termination of the Selector instance
jac_selector.done(0); // the mailbox will be terminated through the runtime's
      automatic termination protocol after guaranteeing all its messages have
      been received and executed
barrier(); // FA-BSP model, ensure all async messages have been executed before
       all PEs move forward
```

SUPERSTEP 1

```
// calculate jaccard similarity J
for (int v = 0; v < n_local_rows; v++) {
  // get offset indices for the row
  for (int k = A->offset[v]; k < A->offset[v+1]; k++) {
    int u = A->nonzero[k]; // get nonzero using offset index (local operation)
    // J = intersection / (d(x) + d(y) - intersection)
    J[v,u] = intersection[v,u] / (d(v) + d(u) - intersection[v,u]);
} }
```

SUPERSTEP 2

Distributed, Asynchronous, and Scalable Actor-Based Jaccard Similarity for Genome Comparisons

Our approach is the first solution that employs a novel distributed and asynchronous algorithm resulting in just **two computation supersteps** compared to synchronous BSP-based approaches utilizing many barriers.

Iterate through nonzero entries in locally stored rows

```
/* Input: (shared) transposed (n x m) matrix A in CSR format with offsets stored
        in A->offset and nonzeros stored in A->nonzero
 * Output: (shared) (n x n) matrix J with jaccard similarities */

/* Utility functions
    * my_pe() : returns PE number of calling PE
    * n_pes() : returns number of PEs running in the application
    * get_remote_pe(x) : returns owner PE for element x
    * get_local_index(x) : returns local index in shared array for element x
    * get_global_id(x) : returns global ID for element x
    * binary_search(arr[:], x) : returns array index for element x if found
    * barrier() : blocks until async local/remote ops are completed on all PEs
    * d(x) : returns degree of element x (number of nonzeros)
*/
```

```
// initialization and start of the Selector instance
Selector<1> jac_selector;

int sender_PE = my_pe();
int n_local_rows = n / n_pes();
for (int v = 0; v < n_local_rows; v++) { // loop through locally stored rows
  int v_global = get_global_id(v); // get global ID for v (local operation)
  // get offset indices for the row
  for (int k = A->offset[v]; k < A->offset[v+1]; k++) {
    int u = A->nonzero[k]; // get nonzero using offset index (local operation)
    /* calculate intersection, the loop in L30 & L48 is application-dependant:
     * for general jaccard measures, the nonzeros of v are looped through
          for(int kk = A->offset[index]; kk < A->offset[index+1]; kk++){...}
     * for genome-based similarity, all pairs of samples are looped through
          for(int next_sample = v_id; next_sample < n; next_sample++){...} */
    for (int kk = A->offset[v]; kk < A->offset[v+1]; kk++) {
      int v_nonzero = A->nonzero[kk]; if (v_nonzero == u) continue;
      int remote_PE = get_remote_pe(u);
      // asynchronous msg sent to loop through shared array on remote_PE
      jac_selector.send(0, remote_PE, [=]() {
        if (binary_search(A->nonzero[A->offset[get_local_index(u)] :
                          A->offset[get_local_index(u)+1]], v_nonzero))
          // found common element, update local counter
          intersection[get_local_index(v_global), get_local_index(u)]++;
      });
} } }

// automatic termination of the Selector instance
jac_selector.done(0); // the mailbox will be terminated through the runtime's
      automatic termination protocol after guaranteeing all its messages have
      been received and executed
barrier(); // FA-BSP model, ensure all async messages have been executed before
       all PEs move forward
```

SUPERSTEP 1

```
// calculate jaccard similarity J
for (int v = 0; v < n_local_rows; v++) {
  // get offset indices for the row
  for (int k = A->offset[v]; k < A->offset[v+1]; k++) {
    int u = A->nonzero[k]; // get nonzero using offset index (local operation)
    // J = intersection / (d(x) + d(y) - intersection)
    J[v,u] = intersection[v,u] / (d(v) + d(u) - intersection[v,u]);
} }
```

SUPERSTEP 2

Distributed, Asynchronous, and Scalable Actor-Based Jaccard Similarity for Genome Comparisons

Our approach is the first solution that employs a novel distributed and asynchronous algorithm resulting in just **two computation supersteps** compared to synchronous BSP-based approaches utilizing many barriers.

Iterate through nonzero entries in locally stored rows

```
/* Input: (shared) transposed (n x m) matrix A in CSR format with offsets stored
        in A->offset and nonzeros stored in A->nonzero
 * Output: (shared) (n x n) matrix J with jaccard similarities */

/* Utility functions
    * my_pe() : returns PE number of calling PE
    * n_pes() : returns number of PEs running in the application
    * get_remote_pe(x) : returns owner PE for element x
    * get_local_index(x) : returns local index in shared array for element x
    * get_global_id(x) : returns global ID for element x
    * binary_search(arr[:], x) : returns array index for element x if found
    * barrier() : blocks until async local/remote ops are completed on all PEs
    * d(x) : returns degree of element x (number of nonzeros)
*/
```

```
// initialization and start of the Selector instance
Selector<1> jac_selector;

int sender_PE = my_pe();
int n_local_rows = n / n_pes();
for (int v = 0; v < n_local_rows; v++) { // loop through locally stored rows
  int v_global = get_global_id(v); // get global ID for v (local operation)
  // get offset indices for the row
  for (int k = A->offset[v]; k < A->offset[v+1]; k++) {
    int u = A->nonzero[k]; // get nonzero using offset index (local operation)
    /* calculate intersection, the loop in L30 & L48 is application-dependant:
     * for general jaccard measures, the nonzeros of v are looped through
          for(int kk = A->offset[index]; kk < A->offset[index+1]; kk++){...}
     * for genome-based similarity, all pairs of samples are looped through
          for(int next_sample = v_id; next_sample < n; next_sample++){...} */
    for (int kk = A->offset[v]; kk < A->offset[v+1]; kk++) {
      int v_nonzero = A->nonzero[kk]; if (v_nonzero == u) continue;
      int remote_PE = get_remote_pe(u);
      // asynchronous msg sent to loop through shared array on remote_PE
      jac_selector.send(0, remote_PE, [=]() {
        if (binary_search(A->nonzero[A->offset[get_local_index(u)] :
                          A->offset[get_local_index(u)+1]], v_nonzero))
          // found common element, update local counter
          intersection[get_local_index(v_global), get_local_index(u)]++;
      });
} } }

// automatic termination of the Selector instance
jac_selector.done(0); // the mailbox will be terminated through the runtime's
      automatic termination protocol after guaranteeing all its messages have
      been received and executed
barrier(); // FA-BSP model, ensure all async messages have been executed before
       all PEs move forward
```

SUPERSTEP 1

```
// calculate jaccard similarity J
for (int v = 0; v < n_local_rows; v++) {
  // get offset indices for the row
  for (int k = A->offset[v]; k < A->offset[v+1]; k++) {
    int u = A->nonzero[k]; // get nonzero using offset index (local operation)
    // J = intersection / (d(x) + d(y) - intersection)
    J[v,u] = intersection[v,u] / (d(v) + d(u) - intersection[v,u]);
} }
```

SUPERSTEP 2

Calculate the intersection of local elements and each nonzero within their data samples

Distributed, Asynchronous, and Scalable Actor-Based Jaccard Similarity for Genome Comparisons

Our approach is the first solution that employs a novel distributed and asynchronous algorithm resulting in just **two computation supersteps** compared to synchronous BSP-based approaches utilizing many barriers.

Iterate through nonzero entries in locally stored rows

Async. message sent to the owner PE to access its data sample and check for element-wise similarity using binary search; if intersection is found, counter is incremented

Calculate the intersection of local elements and each nonzero within their data samples

```
1  /* Input: (shared) transposed (n x m) matrix A in CSR format with offsets stored
          in A->offset and nonzeros stored in A->nonzero
2     * Output: (shared) (n x n) matrix J with jaccard similarities */
```

```
15 // initialization and start of the Selector instance
16 Selector<1> jac_selector;
17
18 int sender_PE = my_pe();
19 int n_local_rows = n / n_pes();
20 for (int v = 0; v < n_local_rows; v++) { // loop through locally stored rows
21   int v_global = get_global_id(v); // get global ID for v (local operation)
22   // get offset indices for the row
23   for (int k = A->offset[v]; k < A->offset[v+1]; k++) {
24     int u = A->nonzero[k]; // get nonzero using offset index (local operation)
25     /* calculate intersection, the loop in L30 & L48 is application-dependant:
26        * for general jaccard measures, the nonzeros of v are looped through
27            for(int kk = A->offset[index]; kk < A->offset[index+1]; kk++){...}
28        * for genome-based similarity, all pairs of samples are looped through
29            for(int next_sample = v_id; next_sample < n; next_sample++){...} */
30     for (int kk = A->offset[v]; kk < A->offset[v+1]; kk++) {
31       int v_nonzero = A->nonzero[kk]; if (v_nonzero == u) continue;
32       int remote_PE = get_remote_pe(u);
33       // asynchronous msg sent to loop through shared array on remote PE
34       jac_selector.send(0, remote_PE, [=]() {
35         if (binary_search(A->nonzero[A->offset[get_local_index(u)] :
36                           A->offset[get_local_index(u)+1]], v_nonzero))
37           // found common element, update local counter
38           intersection[get_local_index(v_global), get_local_index(u)]++;
39       });
40 } } }
41
42 // automatic termination of the Selector instance
43 jac_selector.done(0); // the mailbox will be terminated through the runtime's
        automatic termination protocol after guaranteeing all its messages have
        been received and executed
44 barrier(); // FA-BSP model, ensure all async messages have been executed before
        all PEs move forward
45
```

SUPERSTEP 1

```
46 // calculate jaccard similarity J
47 for (int v = 0; v < n_local_rows; v++) {
48   // get offset indices for the row
49   for (int k = A->offset[v]; k < A->offset[v+1]; k++) {
50     int u = A->nonzero[k]; // get nonzero using offset index (local operation)
51     // J = intersection / (d(x) + d(y) - intersection)
52     J[v,u] = intersection[v,u] / (d(v) + d(u) - intersection[v,u]);
53 } }
```

SUPERSTEP 2

Distributed, Asynchronous, and Scalable Actor-Based Jaccard Similarity for Genome Comparisons

Our approach is the first solution that employs a novel distributed and asynchronous algorithm resulting in just **two computation supersteps** compared to synchronous BSP-based approaches utilizing many barriers.

Iterate through nonzero entries in locally stored rows

Async. message sent to the owner PE to access its data sample and check for element-wise similarity using binary search; if intersection is found, counter is incremented

Calculate the intersection of local elements and each nonzero within their data samples

```
1  /* Input: (shared) transposed (n x m) matrix A in CSR format with offsets stored
          in A->offset and nonzeros stored in A->nonzero
2     * Output: (shared) (n x n) matrix J with jaccard similarities */
```

```
15 // initialization and start of the Selector instance
16 Selector<1> jac_selector;
17
18 int sender_PE = my_pe();
19 int n_local_rows = n / n_pes();
20 for (int v = 0; v < n_local_rows; v++) { // loop through locally stored rows
21   int v_global = get_global_id(v); // get global ID for v (local operation)
22   // get offset indices for the row
23   for (int k = A->offset[v]; k < A->offset[v+1]; k++) {
24     int u = A->nonzero[k]; // get nonzero using offset index (local operation)
25     /* calculate intersection, the loop in L30 & L48 is application-dependant:
26        * for general jaccard measures, the nonzeros of v are looped through
27            for(int kk = A->offset[index]; kk < A->offset[index+1]; kk++){...}
28        * for genome-based similarity, all pairs of samples are looped through
29            for(int next_sample = v_id; next_sample < n; next_sample++){...} */
30     for (int kk = A->offset[v]; kk < A->offset[v+1]; kk++) {
31       int v_nonzero = A->nonzero[kk]; if (v_nonzero == u) continue;
32       int remote_PE = get_remote_pe(u);
33       // asynchronous msg sent to loop through shared array on remote PE
34       jac_selector.send(0, remote_PE, [=]() {
35         if (binary_search(A->nonzero[A->offset[get_local_index(u)] :
36                           A->offset[get_local_index(u)+1]], v_nonzero))
37           // found common element, update local counter
38           intersection[get_local_index(v_global), get_local_index(u)]++;
39       });
40 } } }
41
42 // automatic termination of the Selector instance
43 jac_selector.done(0); // the mailbox will be terminated through the runtime's
        automatic termination protocol after guaranteeing all its messages have
        been received and executed
44 barrier(); // FA-BSP model, ensure all async messages have been executed before
        all PEs move forward
45
```

SUPERSTEP 1

```
46 // calculate jaccard similarity J
47 for (int v = 0; v < n_local_rows; v++) {
48   // get offset indices for the row
49   for (int k = A->offset[v]; k < A->offset[v+1]; k++) {
50     int u = A->nonzero[k]; // get nonzero using offset index (local operation)
51     // J = intersection / (d(x) + d(y) - intersection)
52     J[v,u] = intersection[v,u] / (d(v) + d(u) - intersection[v,u]);
53 } }
```

SUPERSTEP 2

Barrier ensures all async. messages have been processed

Distributed, Asynchronous, and Scalable Actor-Based Jaccard Similarity for Genome Comparisons

Our approach is the first solution that employs a novel distributed and asynchronous algorithm resulting in just **two computation supersteps** compared to synchronous BSP-based approaches utilizing many barriers.

Iterate through nonzero entries in locally stored rows

Async. message sent to the owner PE to access its data sample and check for element-wise similarity using binary search; if intersection is found, counter is incremented

Calculate the intersection of local elements and each nonzero within their data samples

[fully local operation]
Calculate the final $\mathcal{J}$ values

```
1  /* Input: (shared) transposed (n x m) matrix A in CSR format with offsets stored
          in A->offset and nonzeros stored in A->nonzero
2     * Output: (shared) (n x n) matrix J with jaccard similarities */
15 // initialization and start of the Selector instance
16 Selector<1> jac_selector;
17
18 int sender_PE = my_pe();
19 int n_local_rows = n / n_pes();
20 for (int v = 0; v < n_local_rows; v++) { // loop through locally stored rows
21   int v_global = get_global_id(v); // get global ID for v (local operation)
22   // get offset indices for the row
23   for (int k = A->offset[v]; k < A->offset[v+1]; k++) {
24     int u = A->nonzero[k]; // get nonzero using offset index (local operation)
25     /* calculate intersection, the loop in L30 & L48 is application-dependant:
26        * for general jaccard measures, the nonzeros of v are looped through
27            for(int kk = A->offset[index]; kk < A->offset[index+1]; kk++){...}
28        * for genome-based similarity, all pairs of samples are looped through
29            for(int next_sample = v_id; next_sample < n; next_sample++){...} */
30     for (int kk = A->offset[v]; kk < A->offset[v+1]; kk++) {
31       int v_nonzero = A->nonzero[kk]; if (v_nonzero == u) continue;
32       int remote_PE = get_remote_pe(u);
33       // asynchronous msg sent to loop through shared array on remote PE
34       jac_selector.send(0, remote_PE, [=]() {
35         if (binary_search(A->nonzero[A->offset[get_local_index(u)] :
36                           A->offset[get_local_index(u)+1]], v_nonzero))
37           // found common element, update local counter
38           intersection[get_local_index(v_global), get_local_index(u)]++;
39       });
40 } } }
41
42 // automatic termination of the Selector instance
43 jac_selector.done(0); // the mailbox will be terminated through the runtime's
        automatic termination protocol after guaranteeing all its messages have
        been received and executed
44 barrier(); // FA-BSP model, ensure all async messages have been executed before
        all PEs move forward
45
```

```
46 // calculate jaccard similarity J
47 for (int v = 0; v < n_local_rows; v++) {
48   // get offset indices for the row
49   for (int k = A->offset[v]; k < A->offset[v+1]; k++) {
50     int u = A->nonzero[k]; // get nonzero using offset index (local operation)
51     // J = intersection / (d(x) + d(y) - intersection)
52     J[v,u] = intersection[v,u] / (d(v) + d(u) - intersection[v,u]);
53 } }
```

SUPERSTEP 1

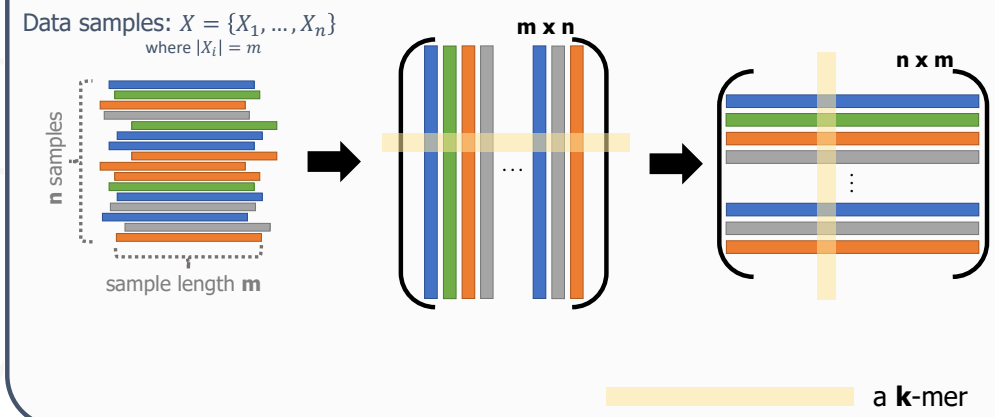
SUPERSTEP 2

Barrier ensures all async. messages have been processed

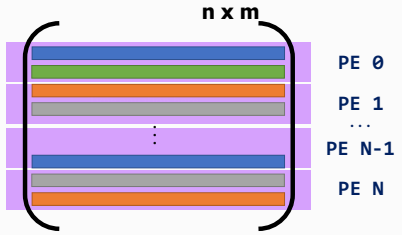
Distributed, Asynchronous, and Scalable Actor-Based Jaccard Similarity for Genome Comparisons

Distributed, Scalable, and Asynchronous Actors Approach to Jaccard Similarity for Genome Comparisons

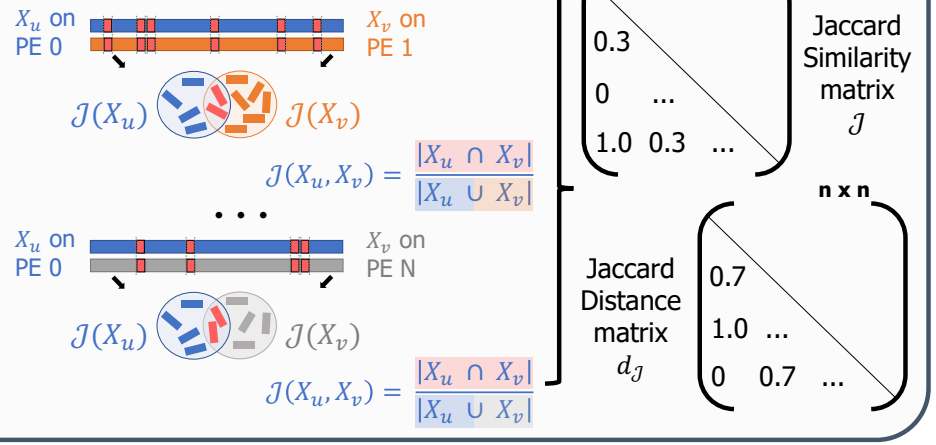
1 create indicator matrix from data samples and transpose matrix



2 create batch for each PE

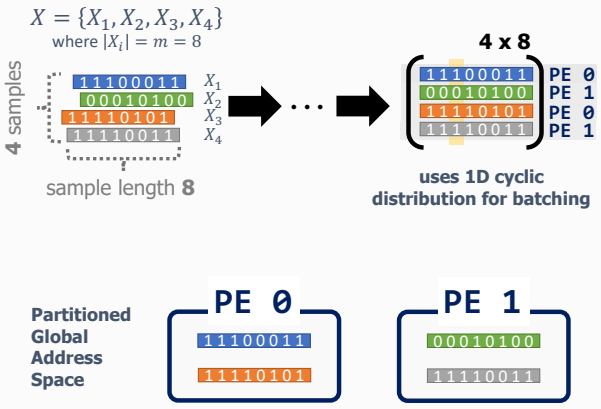


3 async. communication and computation to calculate Jaccard similarity

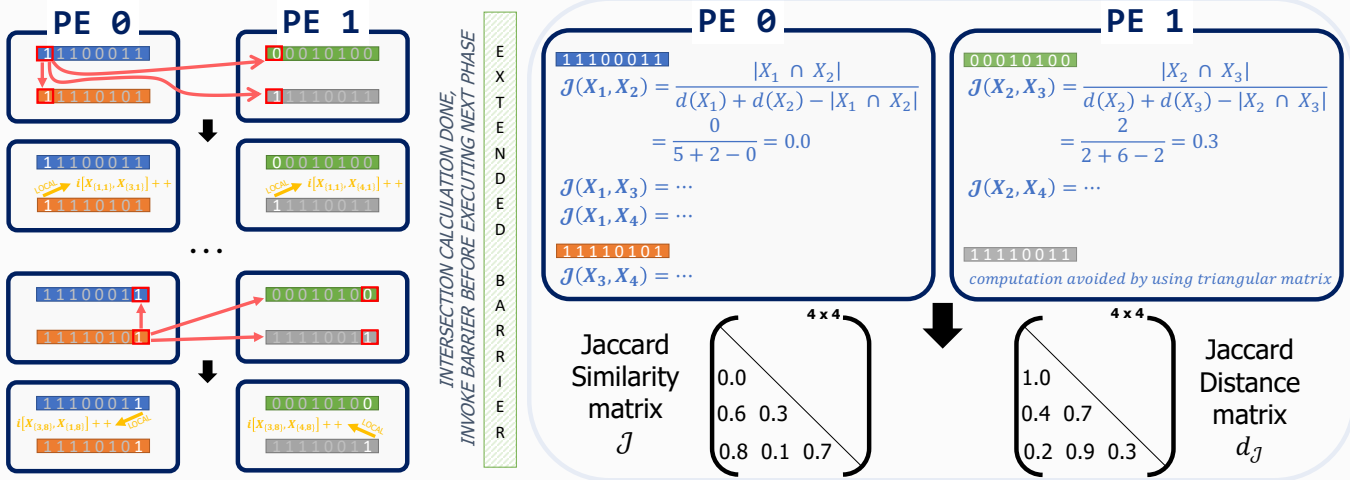


Example Execution Flow using ($m = 8, n = 4, \#PEs = 2$)

1, 2



3 execution from the perspective of **PE0** (assume SPMD, all PEs execute same steps)



DESCRIPTIONS:

- a **k**-mer
- Send an asynchronous message to other PEs to check if the same k-mer is present in any other data samples [CODE: **Listing 1, L34**]
- Highlights the k-mer of interest in the comparison for each data sample [CODE: **Listing 1, L35-36**]
- Update a local intersection counter since a common k-mer was found [CODE: **Listing 1, L38**]
- Extended barrier to wait for completion of sending and receiving async. messages before proceeding [CODE: **Listing 1, L44**]
- Calculate the final $\mathcal{J}$ values [CODE: **Listing 1, L47-53**]

Evaluation: Experimental Setup and Architecture

SCALE – number of data sample rows in the dataset as a power of two
[small-scale: 2^{10}, medium-scale: 2^{14}, large-scale: 2^{20}, extra-large-scale: 2^{25}]

Evaluation: Experimental Setup and Architecture

- Experiments conducted on the CPU nodes of the **Perlmutter supercomputer** at the National Energy Research Scientific Computing Center (NERSC)
  - 2x AMD EPYC 7763 (Milan) CPUs
  - 64 physical cores per CPU
  - 512 GB of DDR4 memory
  - 32KB L1 dcache, 32KB L1 icache, 512KB L2 cache, 2560 4K pages TLB per physical CPU core
  - 1x HPE Cray Slingshot Interconnect

SCALE – number of data sample rows in the dataset as a power of two
[small-scale: 2^{10}, medium-scale: 2^{14}, large-scale: 2^{20}, extra-large-scale: 2^{25}]

Evaluation: Experimental Setup and Architecture

- Experiments conducted on the CPU nodes of the **Perlmutter supercomputer** at the National Energy Research Scientific Computing Center (NERSC)
  - 2x AMD EPYC 7763 (Milan) CPUs
  - 64 physical cores per CPU
  - 512 GB of DDR4 memory
  - 32KB L1 dcache, 32KB L1 icache, 512KB L2 cache, 2560 4K pages TLB per physical CPU core
  - 1x HPE Cray Slingshot Interconnect
- Configuration: 32 PEs/node for our algorithm, 32 MPI processes/node for SOTA

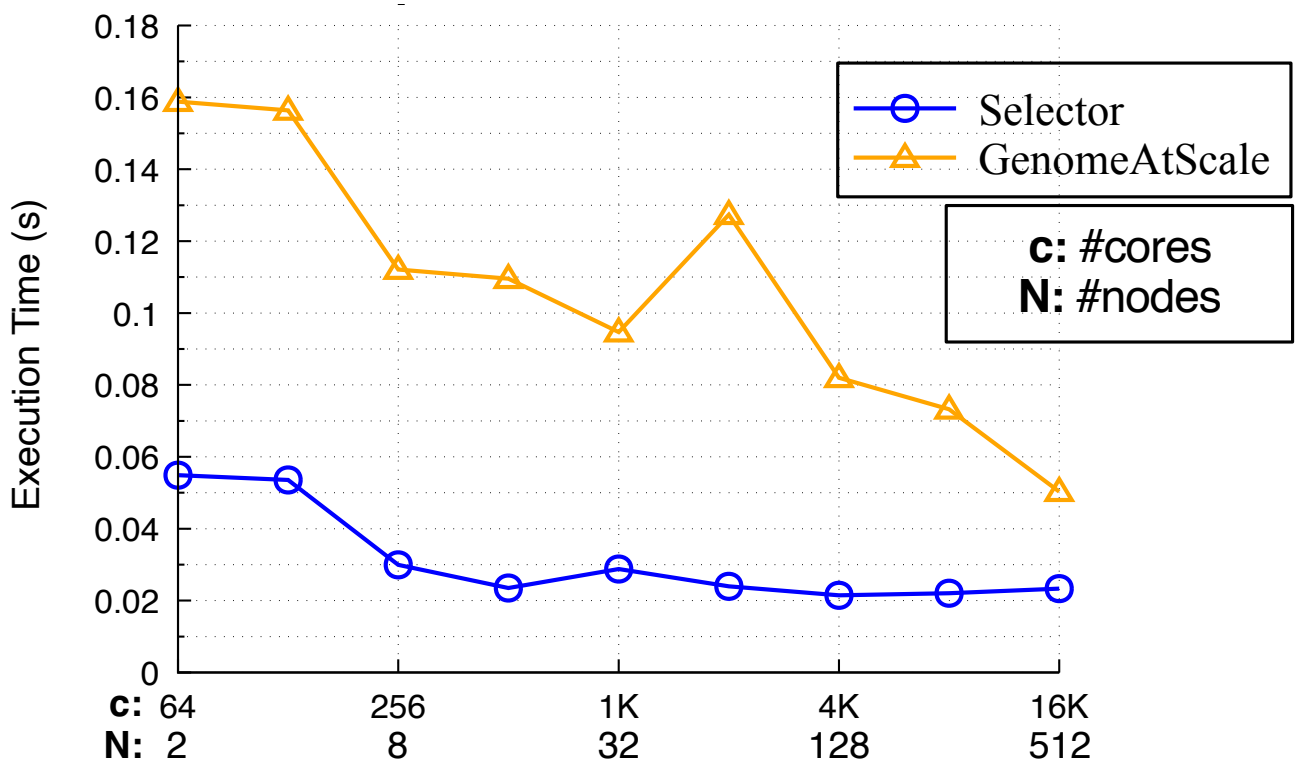
SCALE – number of data sample rows in the dataset as a power of two
[small-scale: 2^{10}, medium-scale: 2^{14}, large-scale: 2^{20}, extra-large-scale: 2^{25}]

Evaluation: Experimental Setup and Architecture

- Experiments conducted on the CPU nodes of the **Perlmutter supercomputer** at the National Energy Research Scientific Computing Center (NERSC)
  - 2x AMD EPYC 7763 (Milan) CPUs
  - 64 physical cores per CPU
  - 512 GB of DDR4 memory
  - 32KB L1 dcache, 32KB L1 icache, 512KB L2 cache, 2560 4K pages TLB per physical CPU core
  - 1x HPE Cray Slingshot Interconnect
- Configuration: 32 PEs/node for our algorithm, 32 MPI processes/node for SOTA
- Evaluations:
  - Strong scaling experiments using a synthetic dataset of medium scale (scale=14) and extra-large scale (scale=25, larger than SOTA)
  - Weak scaling experiments using a synthetic dataset of scale=12 to =20
  - Strong scaling experiments using two real world datasets (*E. coli genome* and *SARS CoV2 genome*)

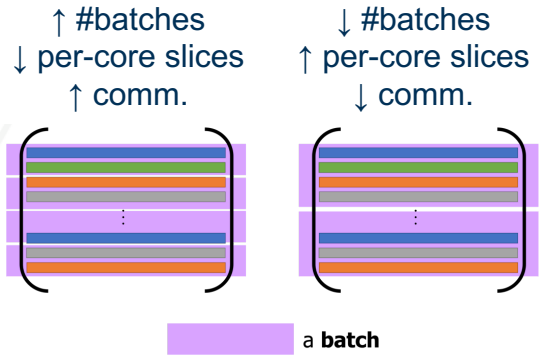
SCALE – number of data sample rows in the dataset as a power of two
[small-scale: 2^{10}, medium-scale: 2^{14}, large-scale: 2^{20}, extra-large-scale: 2^{25}]

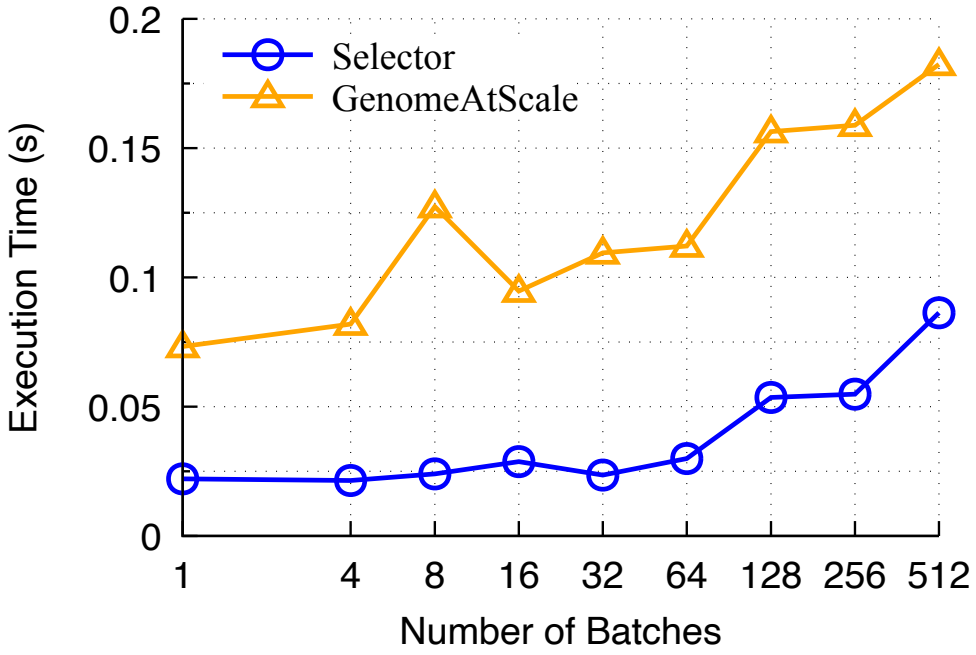
Performance Analysis: Strong Scaling using Synthetic Dataset (scale=14)



The increased performance can be attributed to the exploitation of asynchronous communication/computation and actor-level parallelism

Performance Analysis: Sensitivity to Batches





Our optimal number of batches is n/N

where

n is *#data samples*

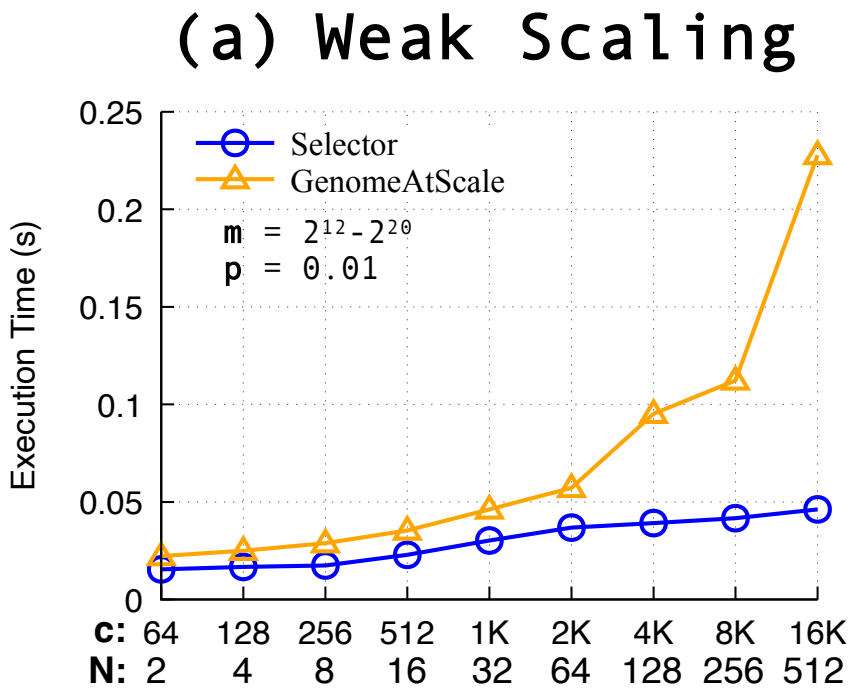
N is #PEs

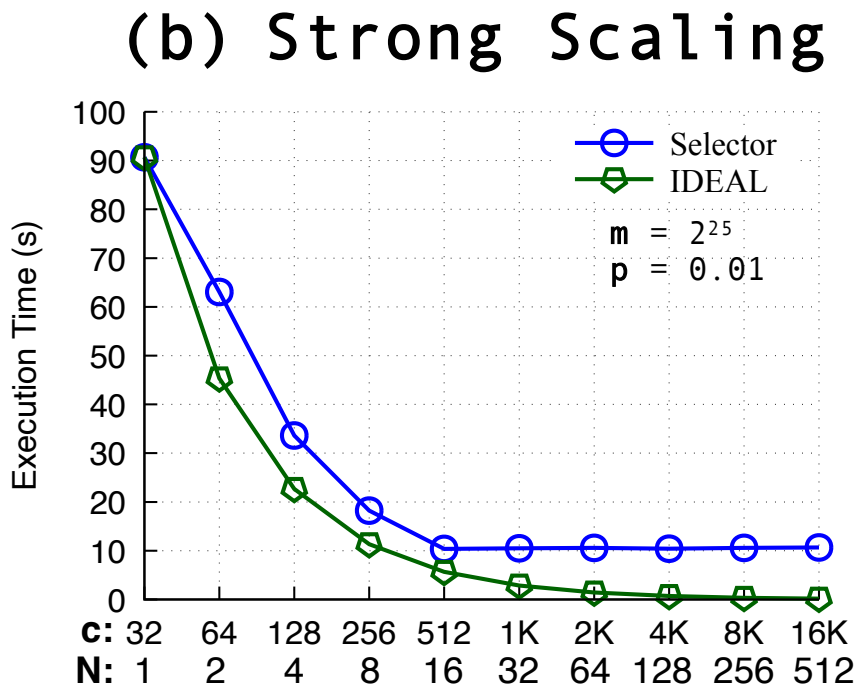
Batch size has a direct correlation with the amount of communication and hence execution time

Larger number of batches means that per-core slices of the computation matrix will be lower, increasing remote communication among cores of different locals

Our algorithm scales efficiently and performs better than the SOTA by 3.6x due to its async. execution and the underlying FA-BSP model

Performance Analysis: Scalability on Synthetic Data

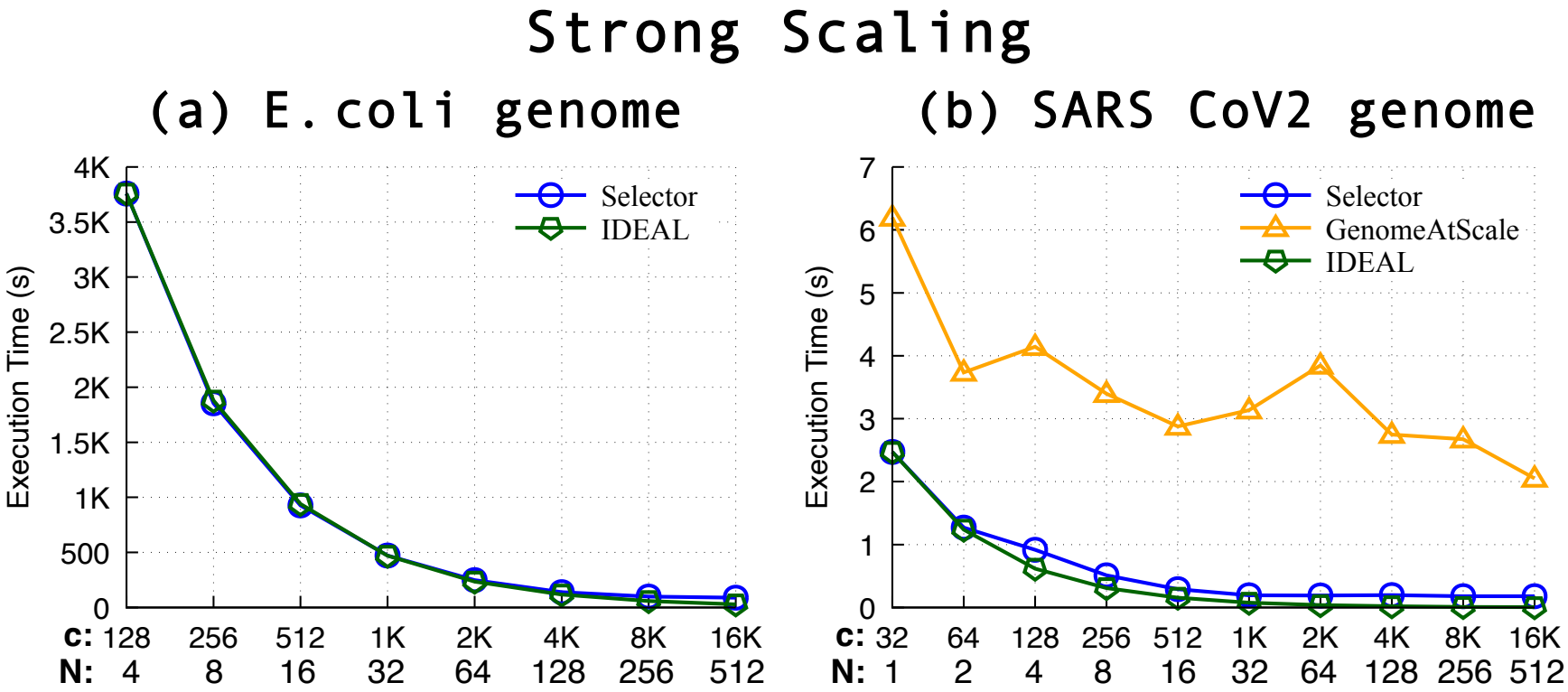




↑ 4.94x at the largest scale

The *SOTA approach is unable to process datasets of large scale without memory issues*, ***using a min. of 512 nodes (16K cores) to process the scale=20 dataset*** while **our approach can process and evaluate the scale=25 dataset using only a min. of 1 node (32 cores)**

Performance Analysis: Scalability on Real Data



3.7k secs on 4 nodes, scaling to <90 secs on 512 nodes

At the highest node count (512), we achieve a 42.8x and 13.8x speedup relative to the smallest node count (4) respectively for both genomes (11.4x better performance compared to SOTA for SARS CoV2 genome)

Why do we achieve this performance increase?

Hardware Metrics Evaluation

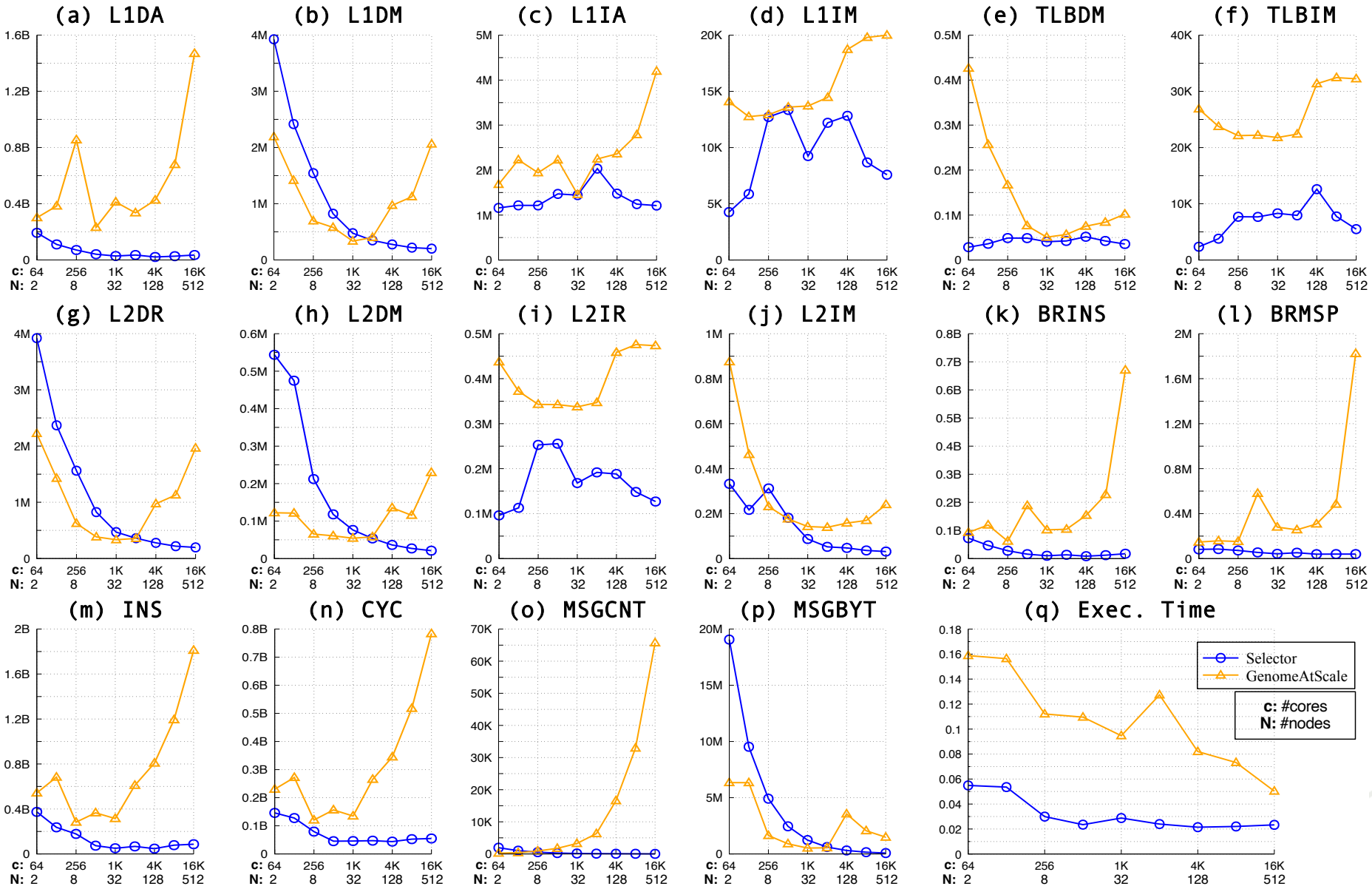
We use the `PAPI` library (v7.0.0.1) and the `perf` tool (v5.14.21) as part of the `CrayPat` performance analysis tool to access these performance counters.

	Abr.	Counter	Description
L1 data cache	L1DA	PAPI_L1_DCA	L1 data cache accesses
	L1DM	PAPI_L1_DCM	L1 data cache misses
L1 instruction cache	L1IA	perf::PERF_COUNT_HW_CACHE_L1I:ACCESS	L1 instruction cache accesses
	L1IM	perf::PERF_COUNT_HW_CACHE_L1I:MISS	L1 instruction cache misses
L2 data cache	L2DR	PAPI_L2_DCR	L2 data cache reads
	L2DM	PAPI_L2_DCM	L2 data cache misses
L2 instruction cache	L2IR	PAPI_L2_ICR	L2 instruction cache reads
	L2IM	PAPI_L2_ICM	L2 instruction cache misses
TLBs	TLBDM	PAPI_TLB_DM	Data translation lookaside buffer misses
	TLBIM	PAPI_TLB_IM	Instruction translation lookaside buffer misses
Branches	BRINS	PAPI_BR_INS	Branch instructions
	BRMSP	PAPI_BR_MSP	Conditional branch instructions mispredicted
Network Messages	MSGCNT	MSG_COUNT	Number of point-to-point communication across PEs
	MSGBYT	MSG_BYTES	Size (bytes) of point-to-point communication across PEs
Cycles	CYC	PAPI_TOT_CYC	Total cycles
Instructions	INS	PAPI_TOT_INS	Instructions completed

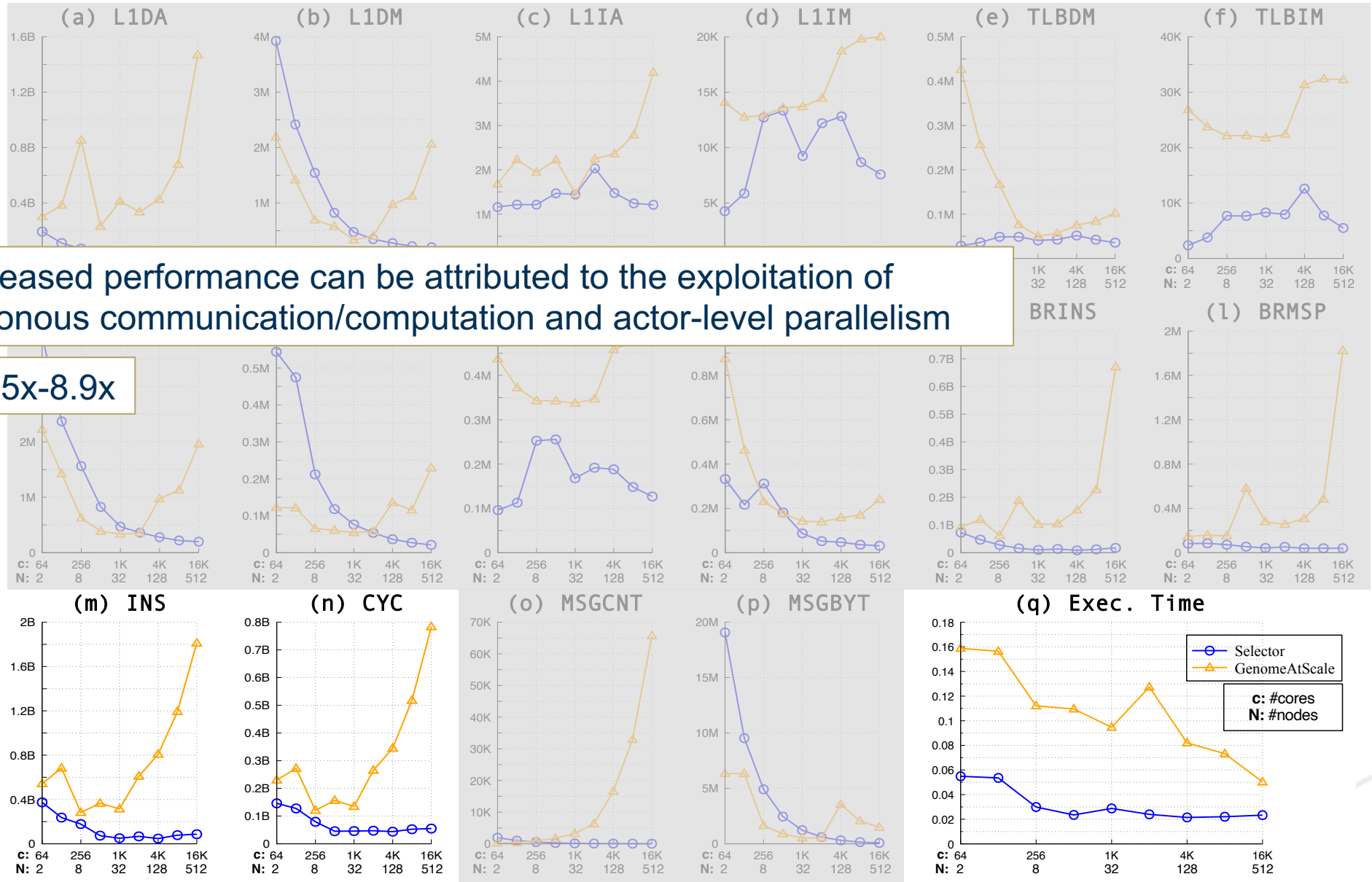
Experiments: Strong Scaling using Synthetic Dataset (scale=14)

Performance Analysis

All numbers are per-core averages.
Lower is better.



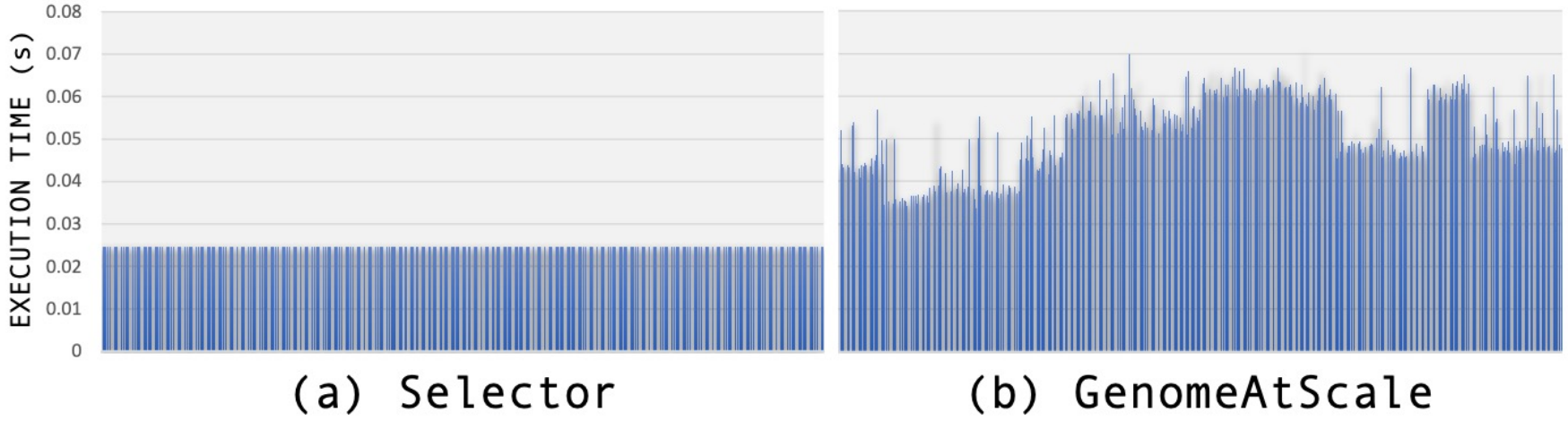
Performance Analysis: Execution Time, INS, CYC



The increased performance can be attributed to the exploitation of asynchronous communication/computation and actor-level parallelism

↓ 5.5x-8.9x

Performance Analysis: Load Imbalance



Workload per core for a 512 core experiment.

Relative Load Imbalance (RLI) measures the relative difference in workload among processing units

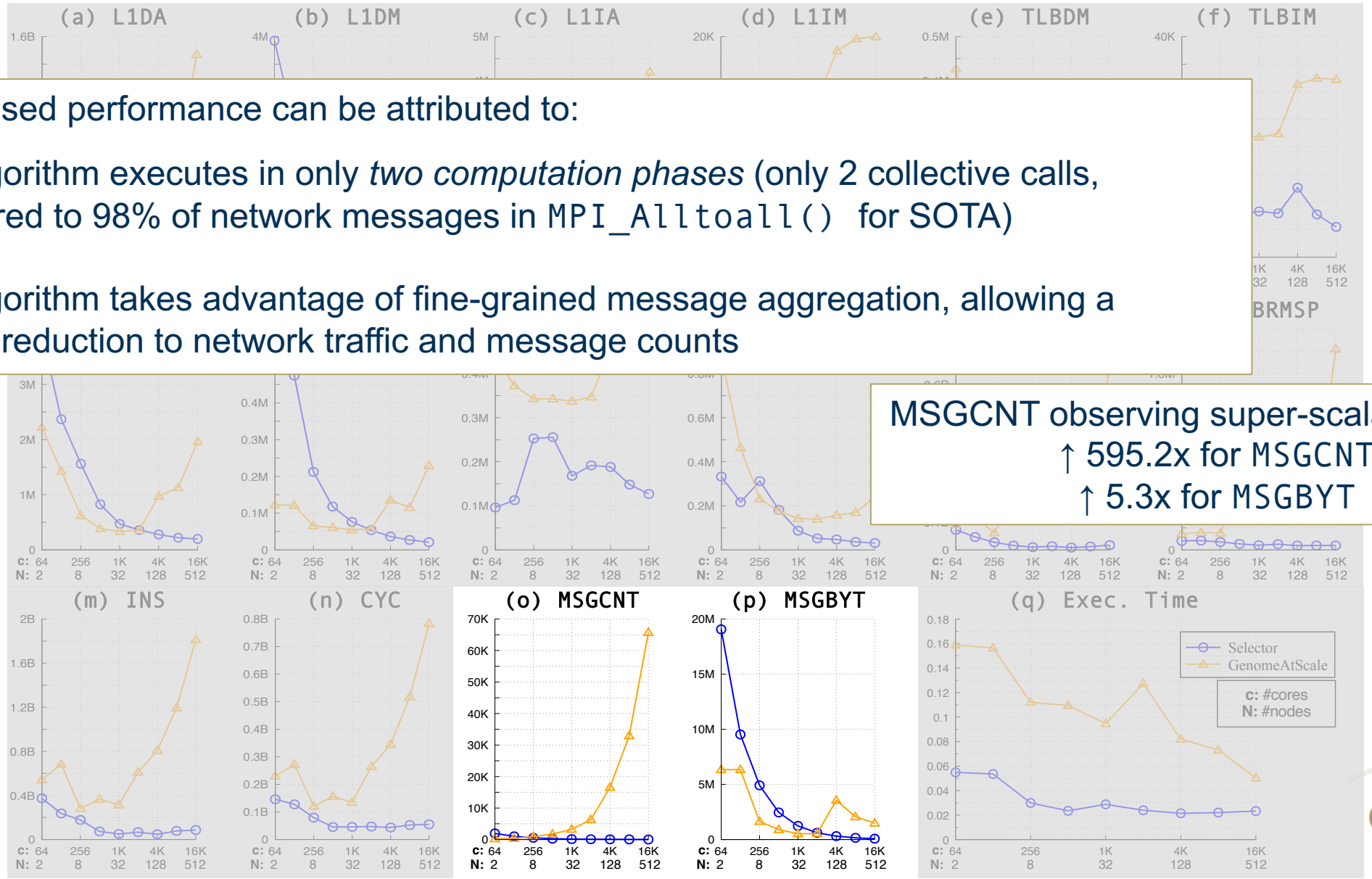
Our approach outperforms the SOTA by 592x in terms of RLI

Performance Analysis: Network Messages

The increased performance can be attributed to:

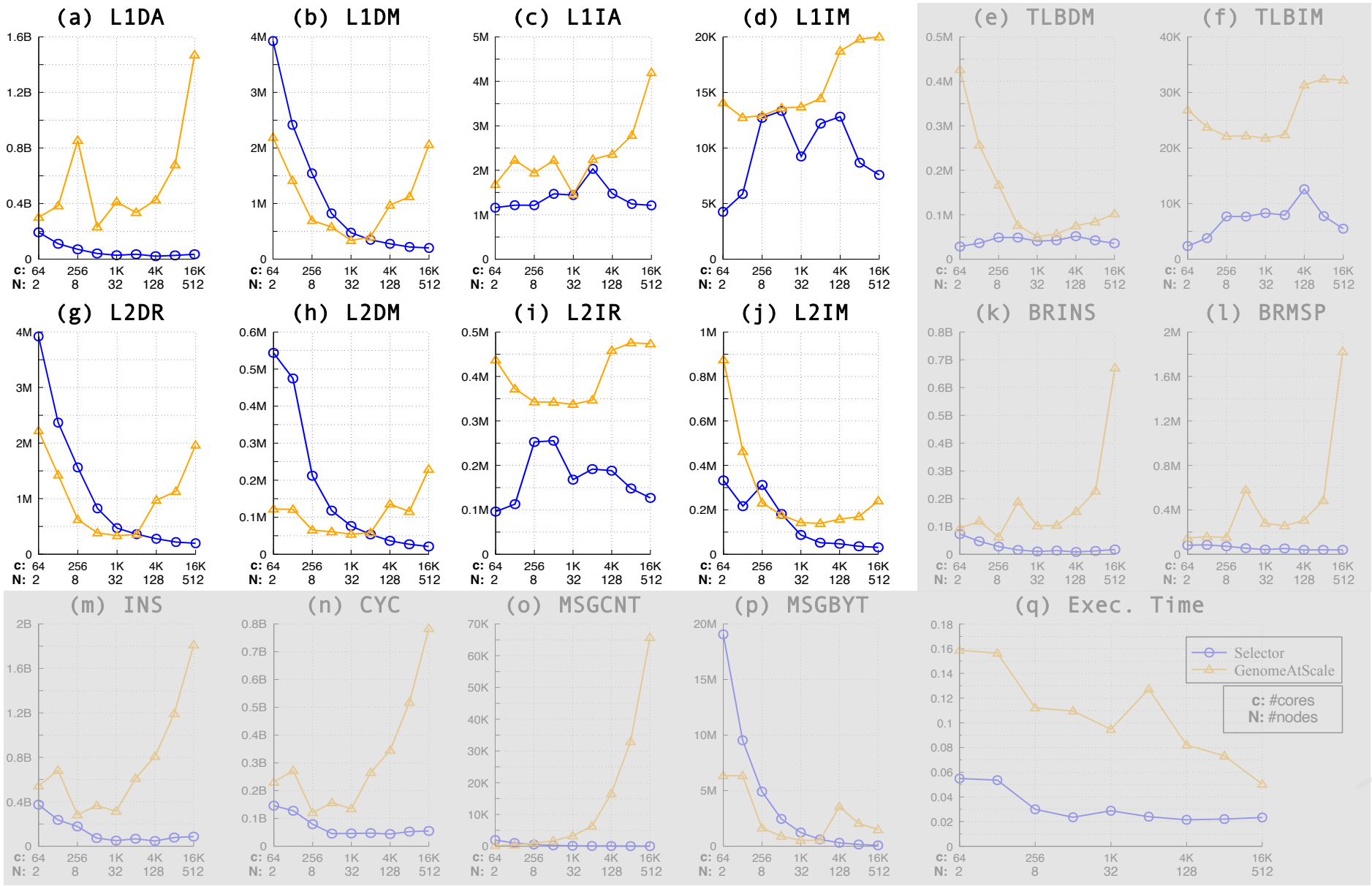
(1) Our algorithm executes in only *two computation phases* (only 2 collective calls, compared to 98% of network messages in `MPI_Alltoall()` for SOTA)

(2) Our algorithm takes advantage of fine-grained message aggregation, allowing a significant reduction to network traffic and message counts

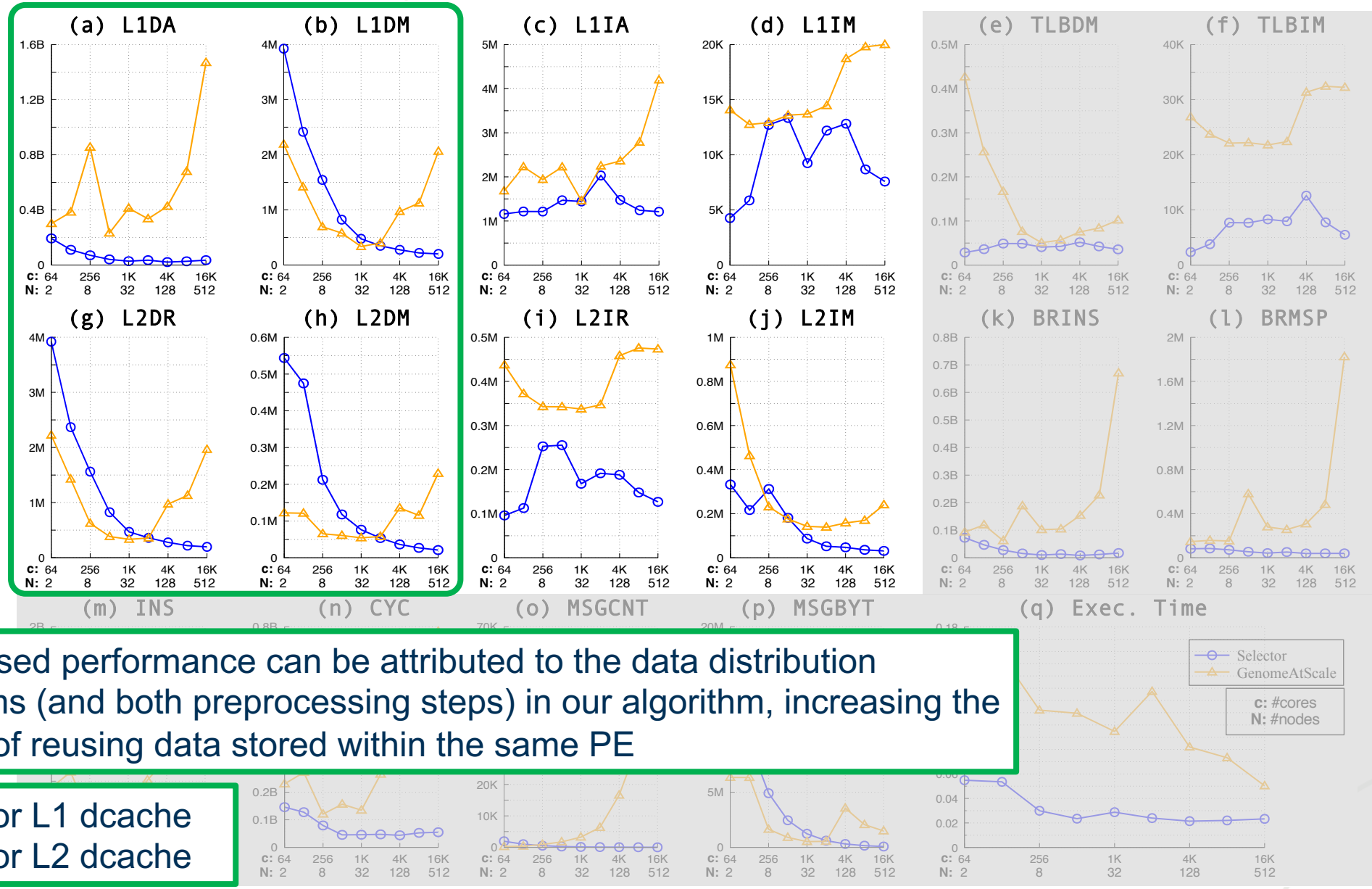


MSGCNT observing super-scalar speedup
↑ 595.2x for `MSGCNT`
↑ 5.3x for `MSGBYT`

Performance Analysis: L1,L2 Caches



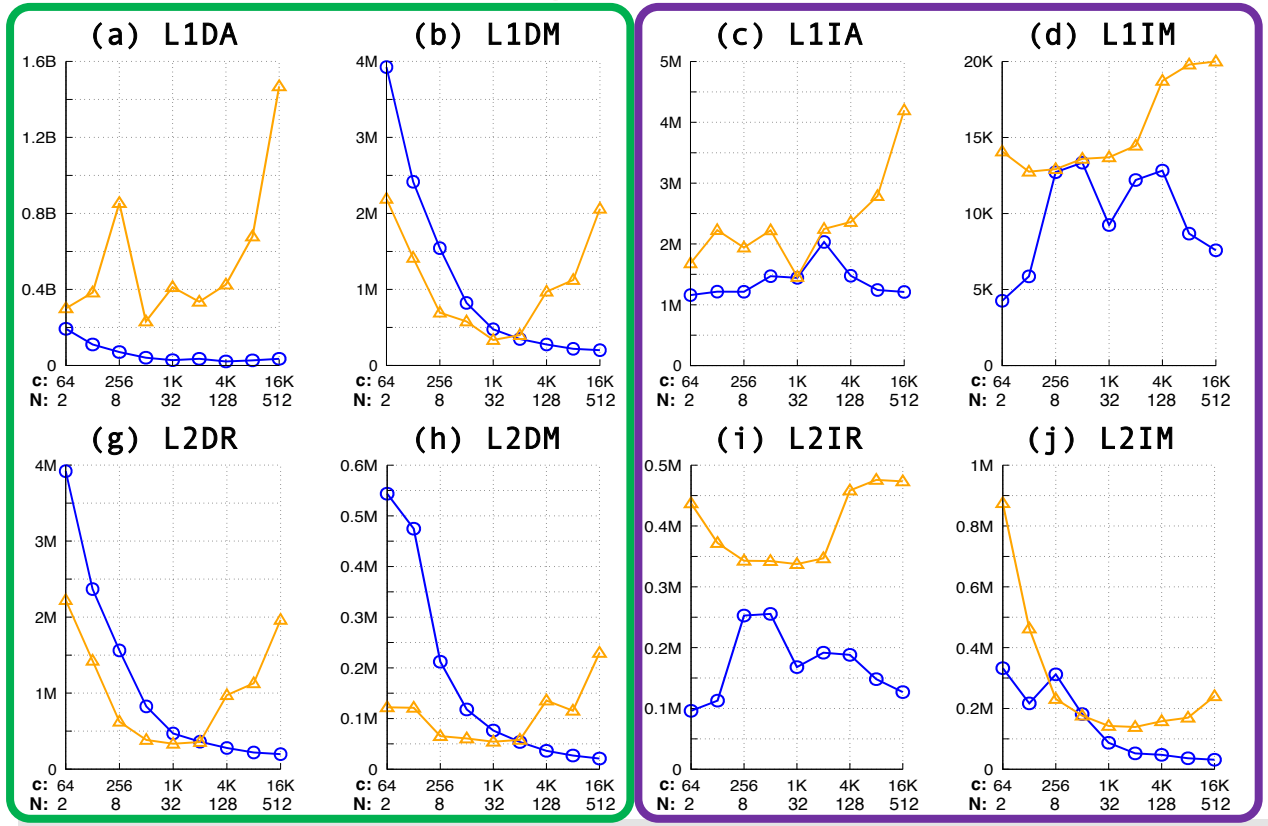
Performance Analysis: L1,L2 Caches



The increased performance can be attributed to the data distribution mechanisms (and both preprocessing steps) in our algorithm, increasing the possibility of reusing data stored within the same PE

↑ 8.7x for L1 dcache
↑ 2.5x for L2 dcache

Performance Analysis: L1,L2 Caches



These effects can be attributed to:

(1) The reusing of the same remote message handler code across all memory requests

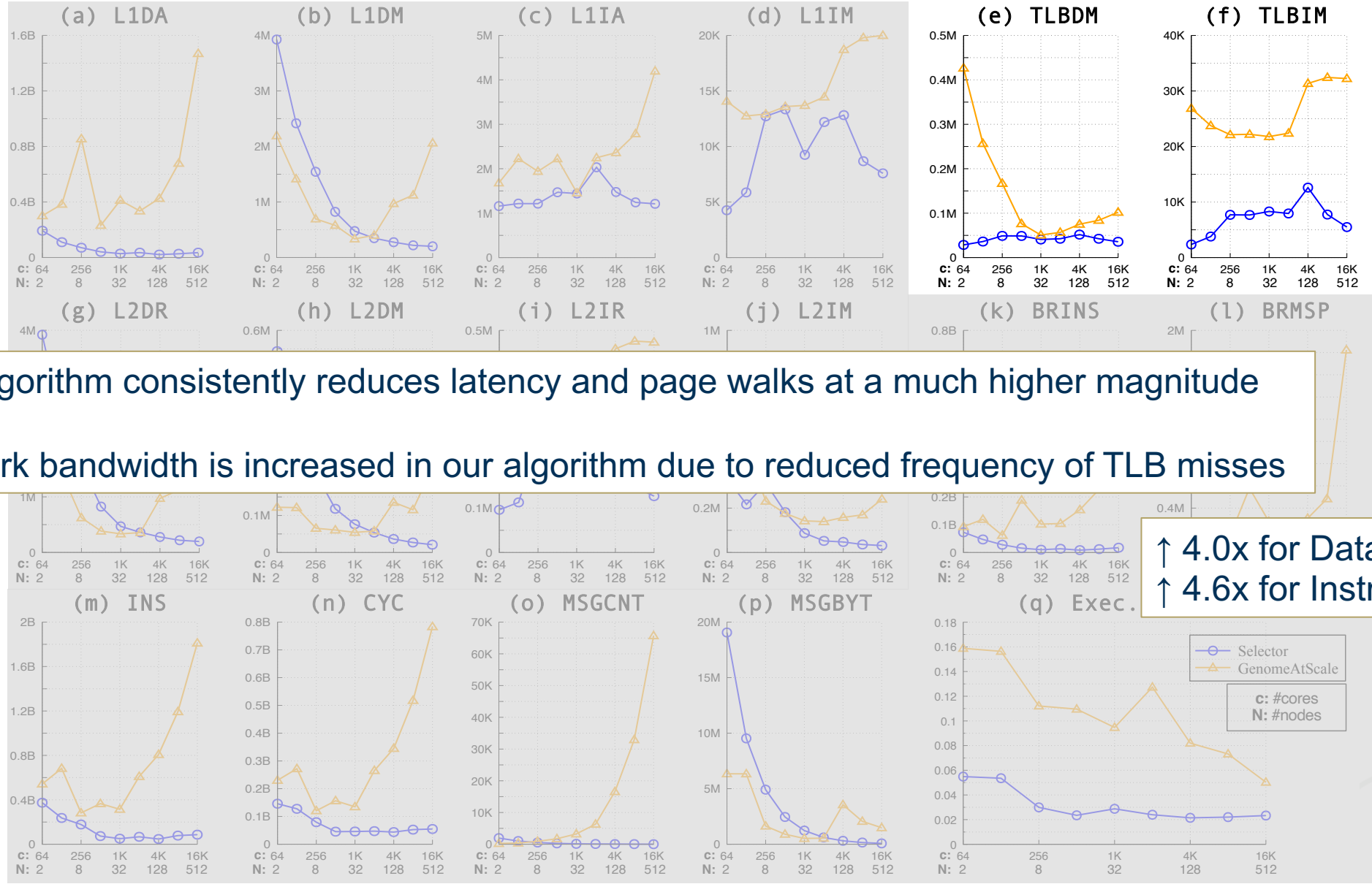
(2) The simplicity (in terms of binary file code size) of our user code and backend runtime system as compared to the SOTA CTF-based backend and external libraries

The increased performance can be attributed to the data distribution mechanisms (and both preprocessing steps) in our algorithm, increasing the possibility of reusing data stored within the same PE

↑ 1.7x for L1 icache
↑ 2.8x for L2 icache

↑ 8.7x for L1 dcache
↑ 2.5x for L2 dcache

Performance Analysis: TLB

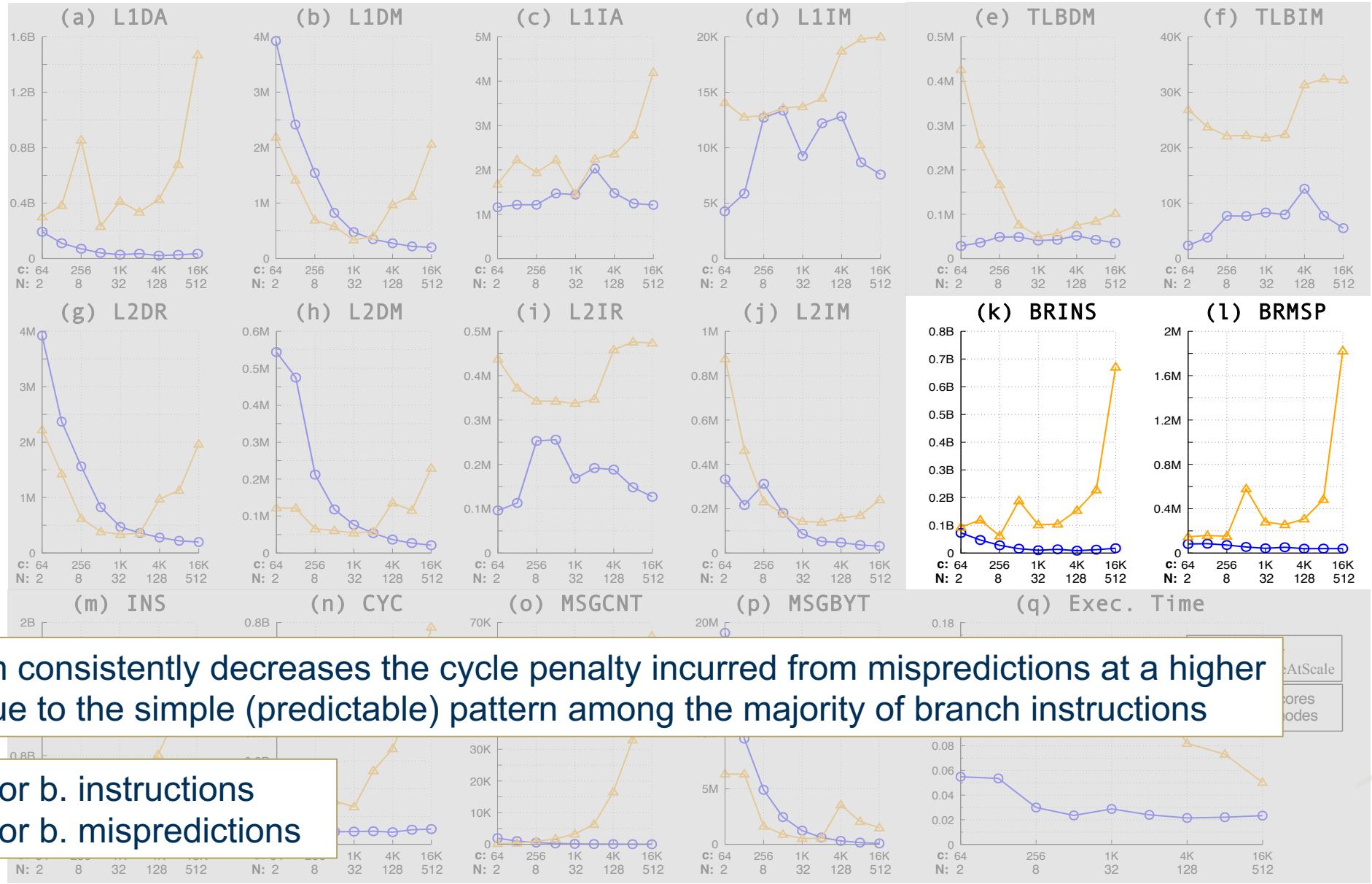


Our algorithm consistently reduces latency and page walks at a much higher magnitude

Network bandwidth is increased in our algorithm due to reduced frequency of TLB misses

↑ 4.0x for Data TLB
↑ 4.6x for Instruction TLB

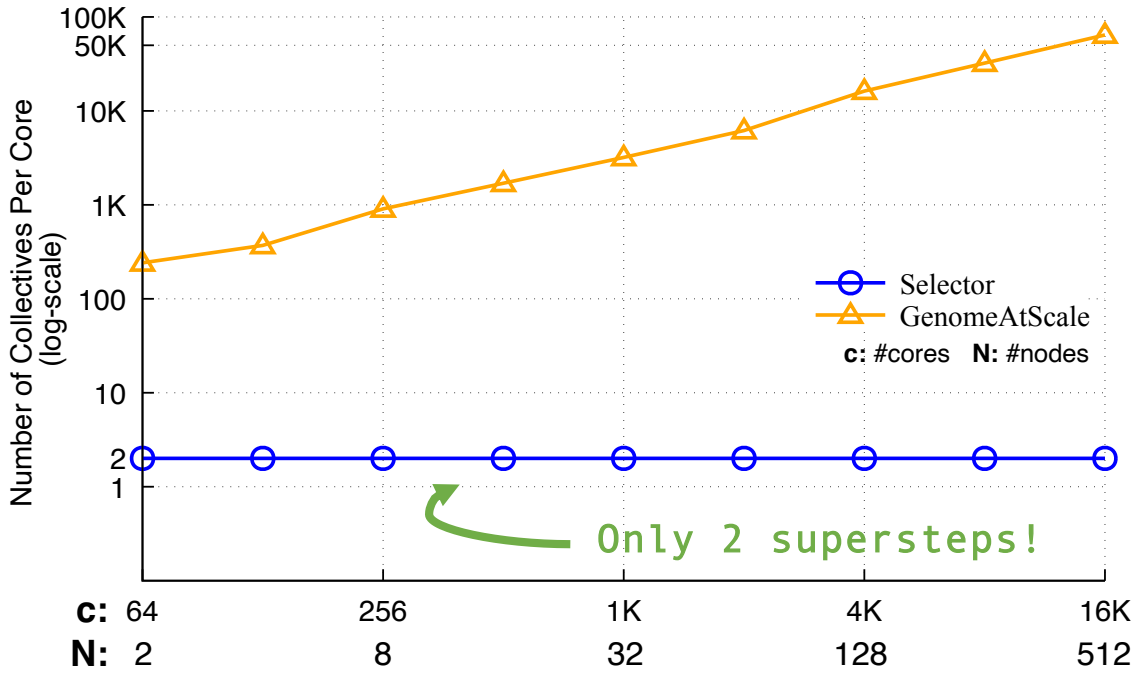
Performance Analysis: Branches



Our algorithm consistently decreases the cycle penalty incurred from mispredictions at a higher magnitude due to the simple (predictable) pattern among the majority of branch instructions

↑ 12.4x for b. instructions
↑ 10.4x for b. mispredictions

Performance Analysis: Collectives Invoked



Per-core number of collectives throughout the execution run (log-scale)

Our algorithm invokes only two barriers throughout the full execution (two superstep execution process)

Algorithm Optimizations

Row compression using bitmasking

Reduce the overhead of storing nonzeros by storing a sequence as a binary value which takes only one bit of data compared to a 32-/64-bit integer nonzero.

Matrix distribution

Reduce the need for remote atomics by distributing data in a communication-aware approach, this can be achieved by exploring other distribution strategies such as circular hash or cyclic distribution.

Message buffer size

A small amount of messages with a large message size sent across the execution of the application can be attributed to a large message buffer size, although there may be more optimal configurations.

GPU acceleration

Accelerate performance of large-scale datasets by accelerating execution of fine-grained computations.

Conclusions

Distributed, Scalable, and Asynchronous Algorithm for Computing $\mathcal{J}$ and $d_{\mathcal{J}}$

based on the Actor-based Programming system

Conclusions

Conclusions

Distributed, Scalable, and Asynchronous Algorithm for Computing $\mathcal{J}$ and $d_{\mathcal{J}}$

based on the Actor-based Programming system

Apply Algorithm to High-Scale Computations of Jaccard Similarity for Genome Comparisons and Genetic Distances

achieved 4.94× performance compared to SOTA

Conclusions

Distributed, Scalable, and Asynchronous Algorithm for Computing $\mathcal{J}$ and $d_{\mathcal{J}}$

based on the Actor-based Programming system

Apply Algorithm to High-Scale Computations of Jaccard Similarity for Genome Comparisons and Genetic Distances

achieved 4.94× performance compared to SOTA

Perform a Deep Dive HWPC Study to Realize Performance Benefits

achieved 3.6× and 5.5× performance for execution time and HWPC results compared to SOTA

Asynchronous Distributed Actor-based Approach to Jaccard Similarity for Genome Comparisons

Youssef Elmougy, Akihiro Hayashi, Vivek Sarkar

Habanero Extreme Scale Software Research Lab
Georgia Institute of **Tech**nology

Corresponding Author: yelmougy3@gatech.edu

Thank you for your attention!