Counting K-mers on distributed memory efficiently with sorting and task-based parallelism

Yifan Li

Tsinghua University

May 20th, 2024

**Work done during exchange at Cornell University, with Prof. Giulia Guidi*

Table of Contents

1. **Introduction**
2. Methodologies
   - The Common *k*-mer Counting Pipeline
   - Sorting Based Approach
   - Supermer Strategy
   - Task Abstraction Layer
3. Performance Results
4. Conclusion

K-mer Counting

A *k*-mer is a substring of fixed length *k* extracted from DNA sequences.

```
Read:          CATCATCA
5-mers:        CATCA
                ATCAT
                 TCATC
                  CATCA
```

Example: 5-mers of a length-8 DNA sequence

```
Result:
(CATCA, 2)
(ATCAT, 1)
(TCATC, 1)
```

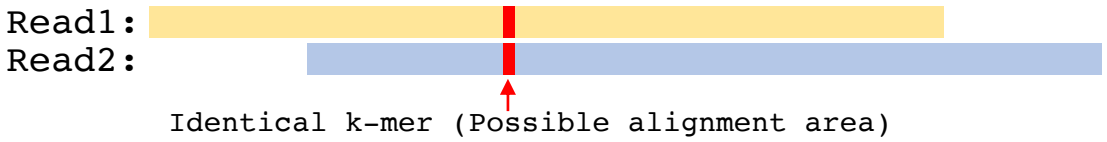
Example: Counting results of the sequence

k-mer counting involves

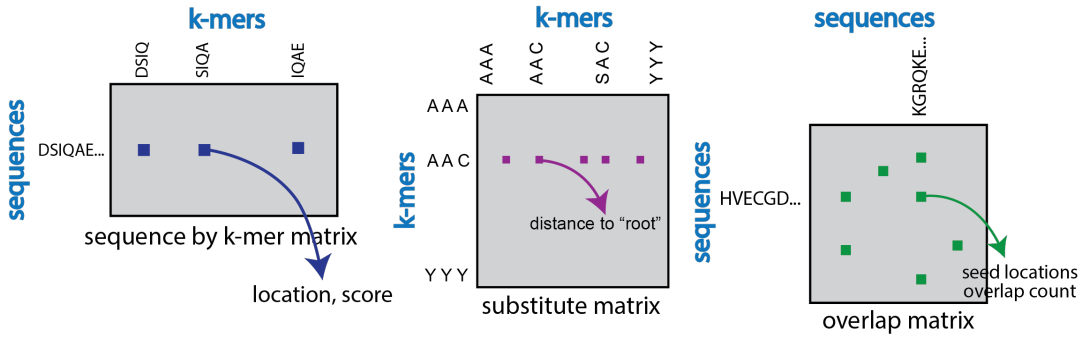
- Counting the **frequency** of *k*-mers in DNA sequences.
- Collecting the **distribution** of *k*-mers, or ***k*-mers within a certain range.**

Some uses of *k*-mer Counting

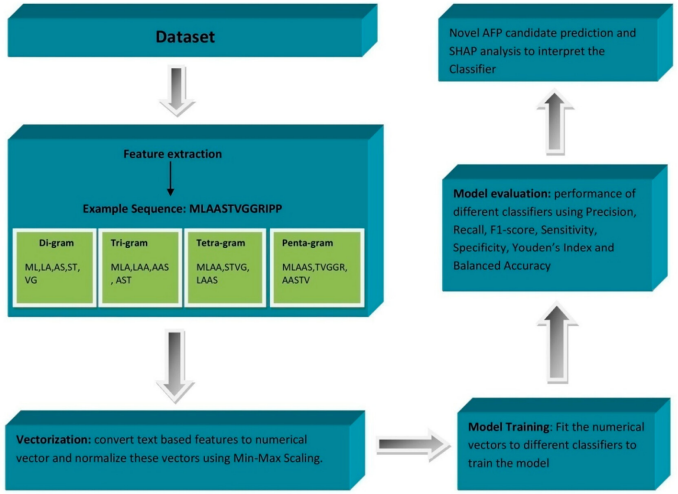
- **Seed and Extend** for genome assembly[1]



- Protein similarity search[2]



- *k*-mers as features of datasets for learning and inference purposes[3]



[1] G. Guidi et al., Bella: Berkeley efficient long-read to long-read aligner and overlapper.
[2] O. Selvitopi et al., Extreme-Scale Many-against-Many Protein Similarity Search
[3] S. Dhibar et al., Accurate prediction of antifreeze protein from sequences through natural language text processing and interpretable machine learning approaches.

Counting *k*-mers on a large scale

Counting *k*-mers for **very large datasets** are required due to

- high-throughput sequencing technologies
- applications like pangenome graph building

However, **counting *k*-mers on a single machine is RAM consuming and compute intensive.**

- The option to filter the input data is not available.
- It is impossible to treat the *k*-mer counting in an embarrassingly parallel manner.
- Utilizing disks will result in severe speed down.

Therefore, counting *k*-mers in distributed memory is (hard but) necessary!

Background

What is MPI?

- Standard for efficient communication **between processes.**
- The de facto standard for message passing in parallel computing.

What is OpenMP?

- A widely-used API that supports **shared memory** multiprocessing programming.

Where do we run the programs?

- Anywhere, but faster on supercomputers / clusters with **fast interconnect.**

Preview

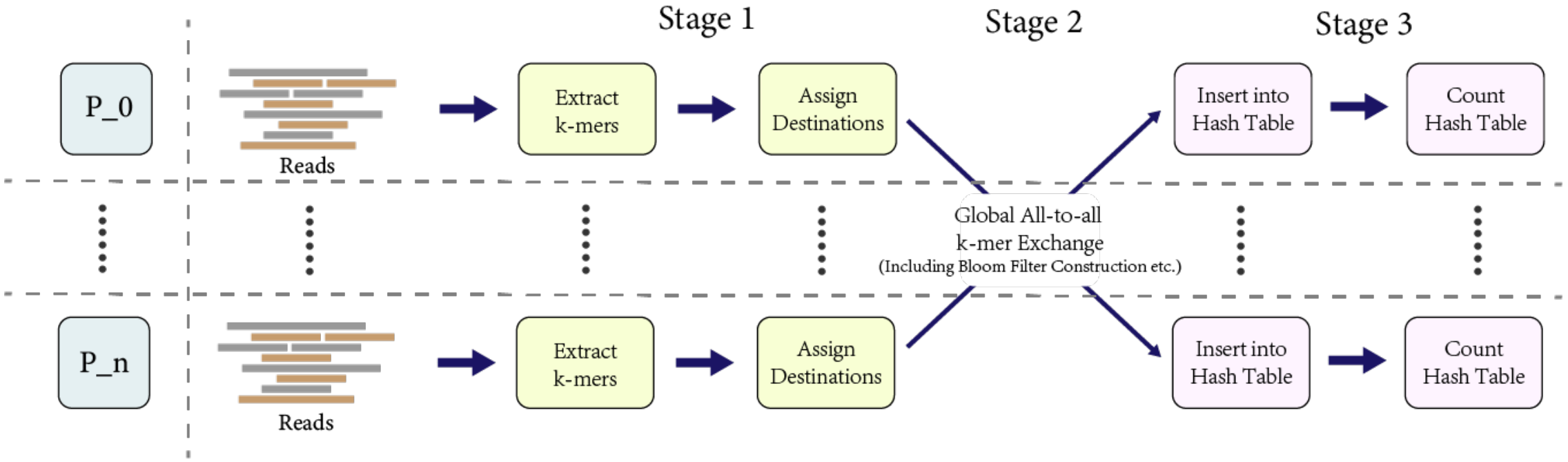
- Use **sort algorithms** instead of hash tables for distributed memory *k*-mer counting.
- Adopt and improve the **Supermer** strategy to reduce communication volume.
- Introduce a **task abstraction layer** for better performance, pipeline integration and load balance.
- Process a human 52x dataset in just **6 seconds** using 64 nodes, compared to the **410 seconds** with the state-of-the-art single-node k-mer counter.

Table of Contents

1. Introduction
2. **Methodologies**
   - **The Common *k*-mer Counting Pipeline**
   - Sorting Based Approach
   - Supermer Strategy
   - Task Abstraction Layer
3. Performance Results
4. Conclusion

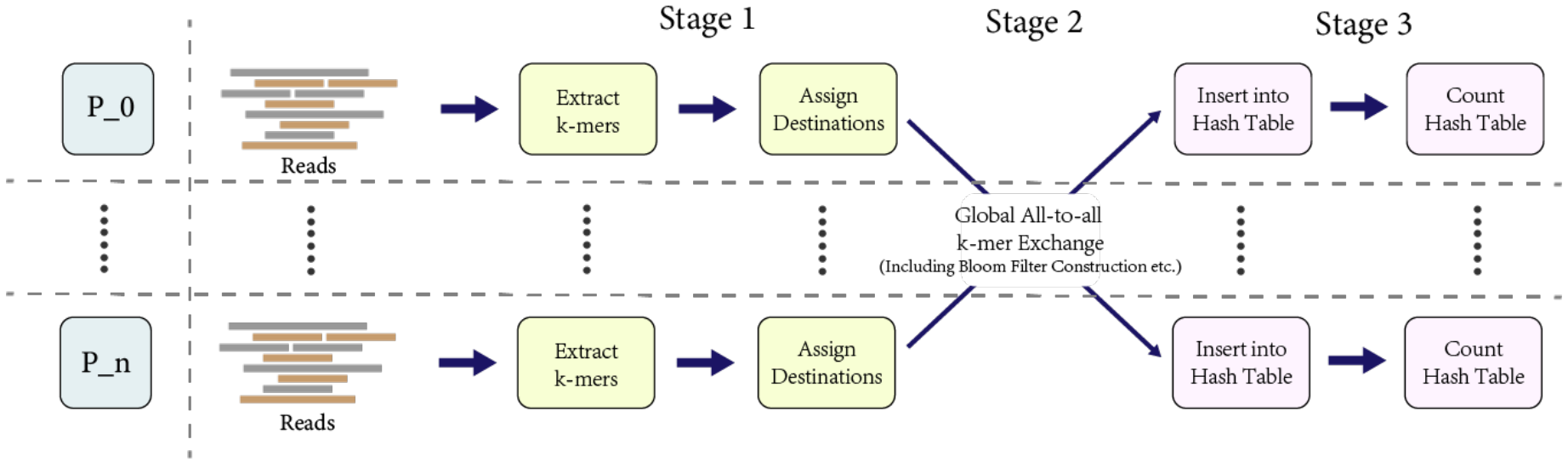
Previous work

The common *k*-mer counting pipeline on distributed memory.



[1] Georganas et al., K. 1114 Parallel de bruijn graph construction and traversal for de novo genome assembly.

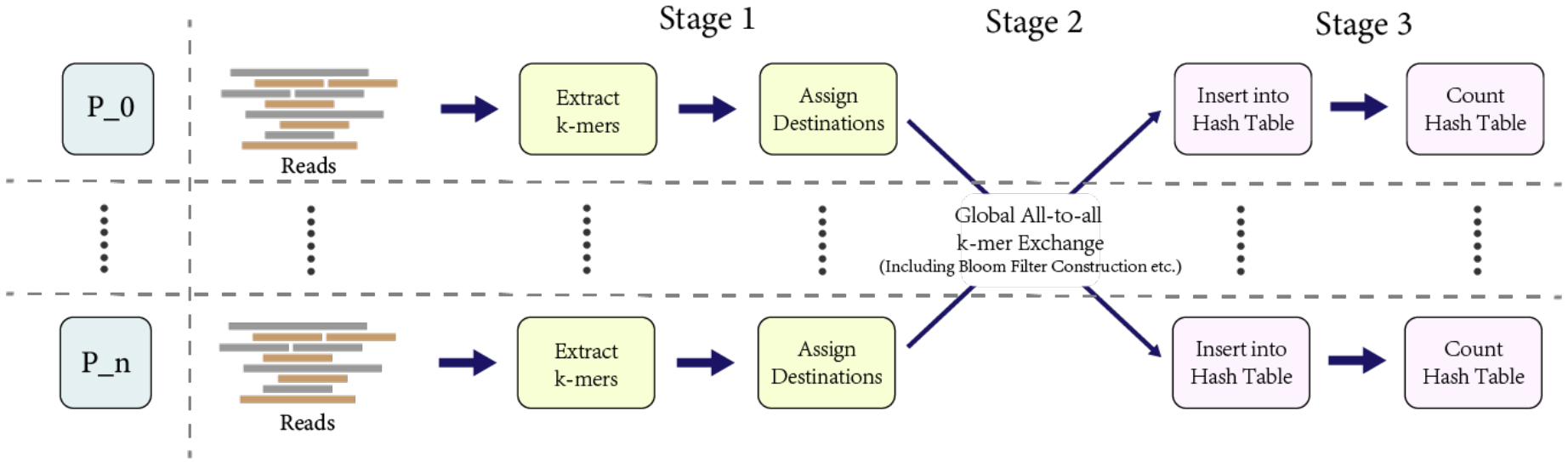
Previous work: The *k*-mer counting pipeline



Stage 1

1. The processes **independently** read and parse batches of input sequences.
2. Processes use the same hash function and mod operation to compute an ID for each *k*-mer, **dividing the local *k*-mer set into groups** based on the ID.

Previous work: The *k*-mer counting pipeline



Stage 2

1. *k*-mers are **distributed** in groups to different target processes based on their respective IDs.
2. (Possibly a 2-Pass approach for Bloom Filters)

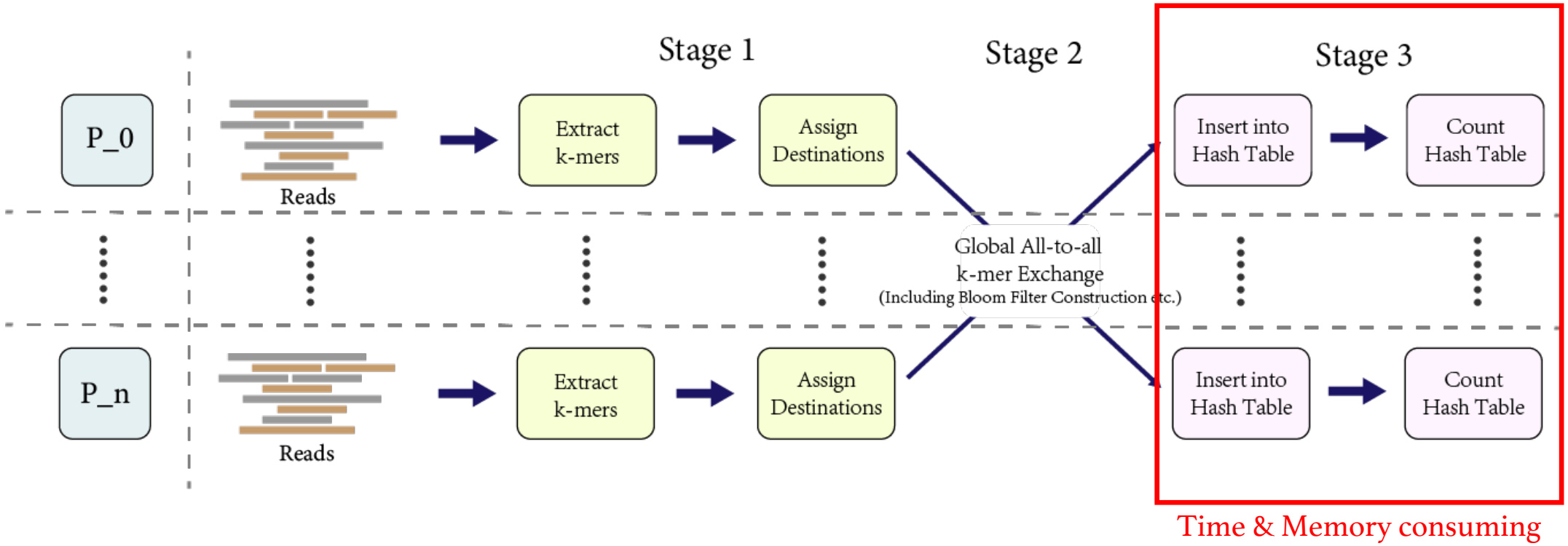
Stage 3

1. The *k*-mers are inserted into the hash table.
2. The hash table serves as a counter.

Table of Contents

1. Introduction
2. **Methodologies**
   - The Common *k*-mer Counting Pipeline
   - **Sorting Based Approach**
   - Supermer Strategy
   - Task Abstraction Layer
3. Performance Results
4. Conclusion

Methodologies

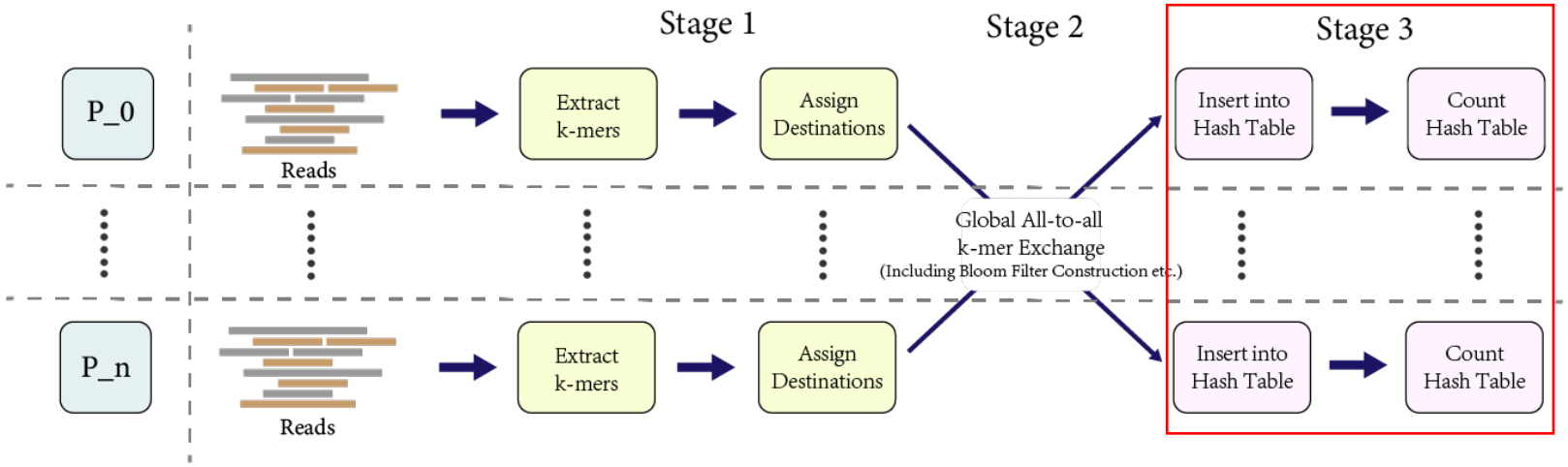


Time & Memory consuming

Methodologies

Sorting based approach

Improves Stage 3



- *k*-mers remain in the receive buffer.
- **A multithreaded sort is performed to reorder the *k*-mer instances according to the value.**
- A linear scan is then performed to count the frequency of *k*-mers.

```
Buffer:              TGA TCG TCG TGG TCG
Sorted Buffer:       TCG TCG TCG TGG TGA
Scanning results:    (TCG, 3) (TGG, 1) (TGA, 1)
```

Example: Sort based approach for stage 3

Methodologies: Sorting Based Approach

Sort Algorithm Selection

- **Radix Sort's theoretical complexity is O($n \cdot d$),** which makes it efficient for large input data.

(n: the number of items to be sorted, d: a constant dependent on the length k)

- Radix sort is particularly well suited for **multicore parallelization.**

- **PARADIS** is a parallel **in-place** radix sort algorithm known for its low memory footprint. [1]
- **RADULS**, another parallel radix sort algorithm, is **cache-friendly optimized** for modern hardware, but **requires more physical memory.** [2]

[1] Cho, M., Brand, D., Bordawekar, R., Finkler, U., Kulandaisamy, V., and Puri, R. Paradis: an efficient parallel algorithm for in-place radix sort.
[2] Kokot, M., Deorowicz, S., and Debudaj-Grabysz, A. Sorting data on ultra-large scale with raduls: New incarnation of radix sort.

Methodologies: Sorting Based Approach

Advantages over traditional Hash table based approach

- Cache-friendly and fast
- Hardly requires any additional space when memory resource is limited
- Facilitates multithreading scaling

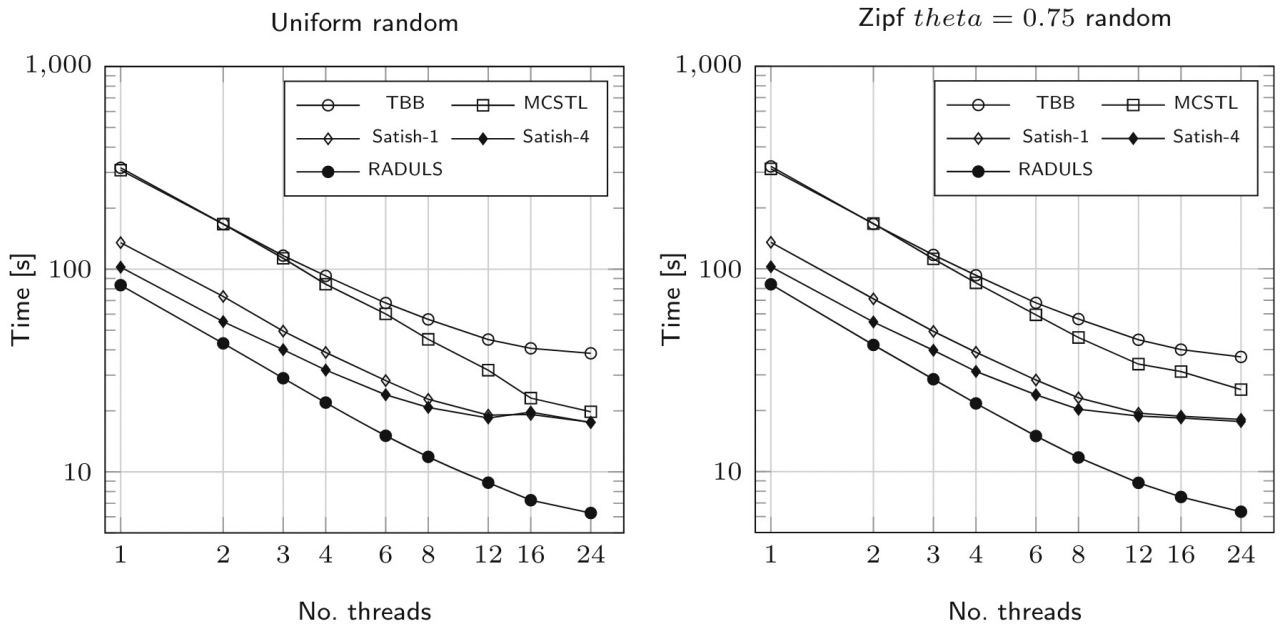
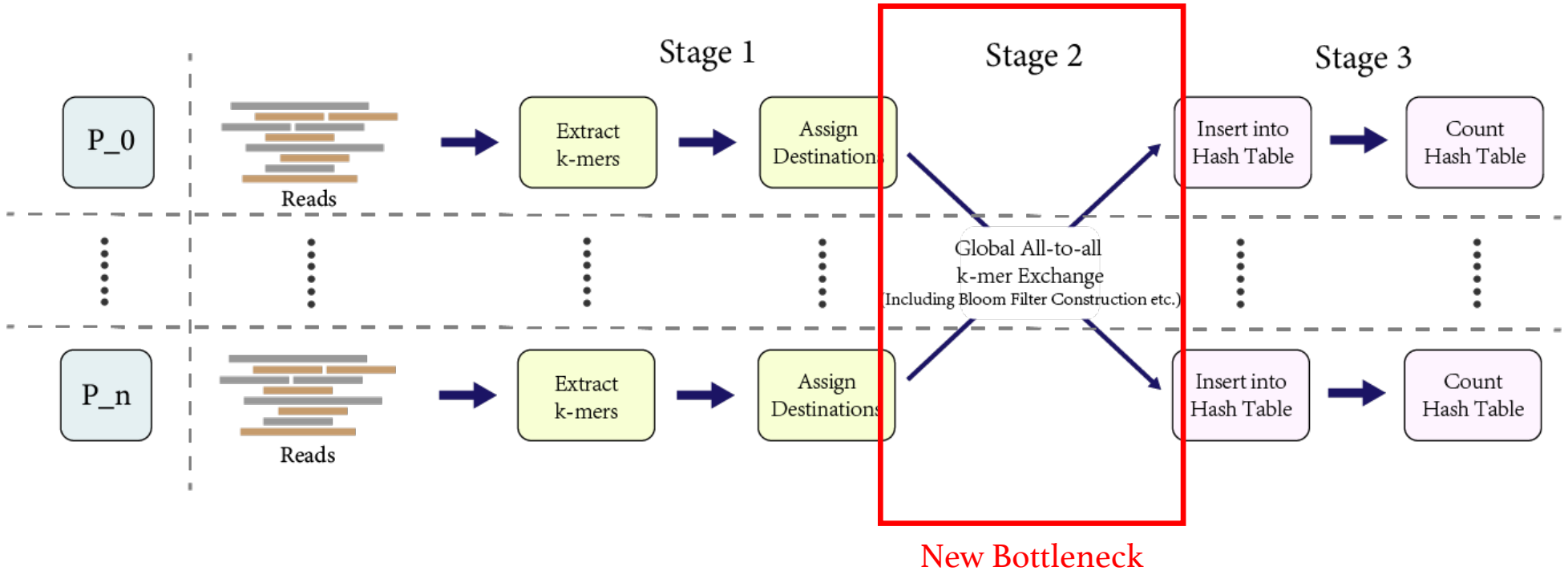


Table of Contents

1. Introduction
2. **Methodologies**
   - The Common *k*-mer Counting Pipeline
   - Sorting Based Approach
   - **Supermer Strategy**
   - Task Abstraction Layer
3. Performance Results
4. Conclusion

Methodologies



Methodologies

The Original Supermer Strategy

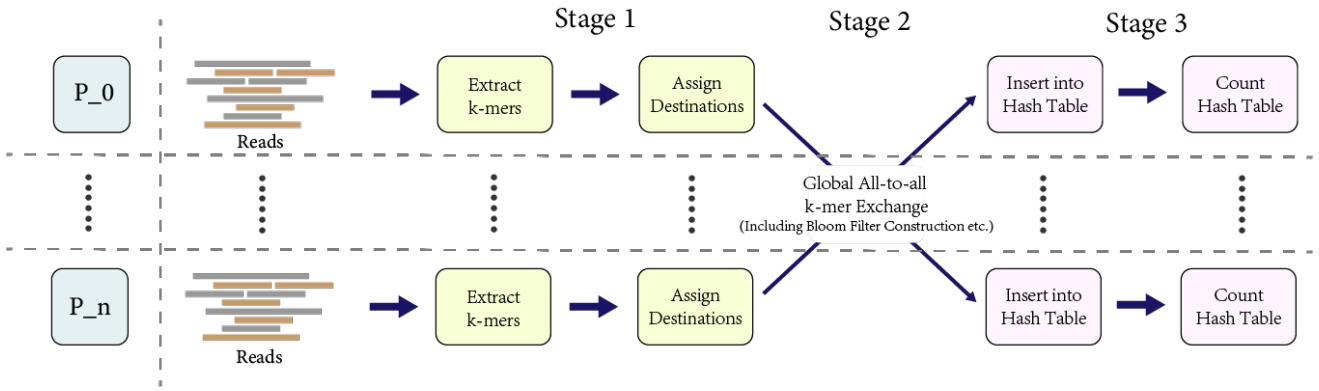
Improves stage 2

- Reduces communication volume

Optimized Supermer Strategy

Improves stage 1, 2

- Using Hash function as scoring function
- An efficient method of finding minimizers



Methodologies: Supermer Strategy

The Original Supermer Strategy

- A *supermer*, or super-*k*-mer, is a contiguous sequence of DNA bases.
- *k*-mers extracted from a supermer should have the same target process.
- The overlapping subsequence of these *k*-mers is not exchanged repeatedly.

```
Read:          AATCGATA
5-mers:        AATCG                  Target: 1
                ATCGA                 Target: 1
                 TCGAT                Target: 1
                  CGATA               Target: 2

Supermers:     AATCGAT                Target: 1  Length: 7
               CGATA                  Target: 2  Length: 5
```

Example: *Supermers* of a DNA sequence

[1] Li, Y., et al. Mspkmercounter: a fast and memory efficient approach for k-mercounting.

Methodologies: Supermer Strategy

The Original Supermer Strategy

- However, the naïve way of assigning *k*-mers to processes **leads to a low probability that adjacent *k*-mers belong to the same process.**
- *M*-mers and minimizers are proposed to remedy the issue.

Unlikely!

```
Read:          AATCGATA
5-mers:        AATCG                  Target: 1
                ATCGA                 Target: 1
                 TCGAT                Target: 1
                  CGATA               Target: 2

Supermers:     AATCGAT                Target: 1  Length: 7
               CGATA                  Target: 2  Length: 5
```

Example: *Supermers* of a DNA sequence

Methodologies: Supermer Strategy

The Original Supermer Strategy

- An m-mer is defined as a length m subsequence of DNA bases ($m < k$) .
- **A minimizer is the m-mer of a k-mer with the lowest score for a function f .**
- The target process of the k-mer is determined **by the hash value of the minimizer.**

```
Read:          AATCGATA      (k=5, m=3)
5-mers:        AATCG         minimizer:    TCG          target: 1
                ATCGA                      TCG          target: 1
                 TCGAT                     TCG          target: 1
                  CGATA                    CGA          target: 3

Supermers:     AATCGAT              Target: 1  Length: 7
               CGATA                Target: 2  Length: 5
```

Example: *Supermers* and *minimizers* of a DNA sequence, simple lexical order T<C<G<A for scoring function

Methodologies: Supermer Strategy

Optimized Supermer Strategy

Using Hash function as scoring function

- Guarantees randomness and ensures load balance (to some extent)
- Does not cause much overhead

Methodologies: Supermer Strategy

Optimized Supermer Strategy

Efficient method of finding minimizers

Algorithm 2 Supermer–based distributed k-mer counting

```
1:  procedure BUILDSUPERMER                          ▷ Parse k-mers and build supermer
2:      for r ∈ reads do
3:          for i = 0 to LEN(r) − k + 1 step window do
4:              kmer = EXTRACTKMER(r, i, k)
5:              minimizer = MINIMIZER(kmer)
6:              prev ← minimizer
7:              P = HASH(minimizer, nProc)           ▷ Find processor
8:              supermers[P].PUSHBACK(kmer)
9:              slens[P].PUSHBACK(k)
10:             INCREMENT(nSmer[P])
11:         for w = 1 to window do
12:             kmer = EXTRACTKMER(r, i + w, k)
13:             minimizer = MINIMIZER(kmer)
14:             if minimizer ≠ prev then
15:                 P = HASH(minimizer, nProc)
16:                 supermers[P].PUSHBACK(kmer)
17:                 slens[P].PUSHBACK(k)
18:                 INCREMENT(nSmer[P])
19:             else
20:                 supermers[P][nSmer].ADDCHAR(kmer, k − 1)
21:                 INCREMENT(slen[P][nSmer[P]])
22:             prev ← minimizer
```

$n * k$ calculations

DEDUKT[1]

Expected $2n$ calculations

Algorithm 1 Minimum Substring Partitioning

```
Input: String s = s1 s2 ... sn, integer k, p.
min_s = the minimum p-substring of s[1, k]
min_pos = the start position of min_s in s
for all i from 2 to n − k + 1 do
  if i > min_pos then
    min_s = the minimum p-substring of s[i, i + k − 1]
    update min_pos accordingly
  else
    if the last p-substring of s[i, i + k − 1] < min_s then
      min_s = the last p-substring of s[i, i + k − 1]
      update min_pos accordingly
    end if
  end if
end for
```

MSPKmerCounter[2]

[1] Nisa, I., Pandey, P., Ellis, M., Oliker, L., Buluç, A., and Yelick, K. Distributed-memory k-mer counting on gpus.
[2] Li, Y., et al. Mspkmercounter: a fast and memory efficient approach for k-mercounting.

Methodologies: Optimized Supermer Strategy

Finding minimizers for consecutive *k*-mers is actually a sliding window problem.

```
Read:          TCTAGCCA      (k=5, m=3)
5-mer:         TCTAG         Deque: [TCT, TAG]           Minimizer: TCT
                CTAGC        Deque: [~~TCT~~, TAG, AGC]  Minimizer: TAG
                 TAGCC       Deque: [TAG, ~~AGC~~, GCC]  Minimizer: TAG
                  AGCCA      Deque: [~~TAG~~, ~~GCC~~, CCA]  Minimizer: CCA
```

Example: Finding minimizers of consecutive *k*-mers with a deque

Methodologies: Optimized Supermer Strategy

Finding minimizers for consecutive *k*-mers is actually a sliding window problem.

- A **deque** is kept. The elements in the deque are ordered monotonically, i.e. the *m*-mers in the deque have increasing scores.

```
Read:          TCTAGCCA      (k=5, m=3)
5-mer:         TCTAG         Deque: [TCT, TAG]           Minimizer: TCT
                CTAGC        Deque: [~~TCT~~, TAG, AGC]  Minimizer: TAG
                 TAGCC       Deque: [TAG, ~~AGC~~, GCC]  Minimizer: TAG
                  AGCCA      Deque: [~~TAG~~, ~~GCC~~, CCA]  Minimizer: CCA
```

Example: Finding minimizers of consecutive *k*-mers with a deque

Methodologies: Optimized Supermer Strategy

Finding minimizers for consecutive *k*-mers is actually a sliding window problem.

- A **deque** is kept. The elements in the deque are ordered monotonically, i.e. the *m*-mers in the deque have increasing scores.
- Rule 1: To remove an expiring *m*-mer in the deque, we check the front part of the deque. If the front element is the expiring *m*-mer, it is removed; otherwise nothing is done.

```
Read:          TCTAGCCA      (k=5, m=3)
5-mer:         TCTAG         Deque: [TCT, TAG]           Minimizer: TCT
                CTAGC        Deque: [~~TCT~~, TAG, AGC]  Minimizer: TAG
                 TAGCC       Deque: [TAG, ~~AGC~~, GCC]  Minimizer: TAG
                  AGCCA      Deque: [~~TAG~~, ~~GCC~~, CCA]  Minimizer: CCA
```

Example: Finding minimizers of consecutive *k*-mers with a deque

Methodologies: Optimized Supermer Strategy

Finding minimizers for consecutive *k*-mers is actually a sliding window problem.

- A **deque** is kept. The elements in the deque are ordered monotonically, i.e. the *m*-mers in the deque have increasing scores.
- Rule 1: To remove an expiring *m*-mer in the deque, we check the front part of the deque. If the front element is the expiring *m*-mer, it is removed; otherwise nothing is done.
- Rule 2: To insert a *m*-mer, we remove elements at the end of the deque until the score of the end element is lower than that of the new *m*-mer or the deque is empty, and then insert the new *m*-mer at the end.

```
Read:          TCTAGCCA      (k=5, m=3)
5-mer:         TCTAG         Deque: [TCT, TAG]            Minimizer: TCT
                CTAGC        Deque: [~~TCT~~, TAG, AGC]   Minimizer: TAG
                 TAGCC       Deque: [TAG, ~~AGC~~, GCC]   Minimizer: TAG
                  AGCCA      Deque: [~~TAG~~, ~~GCC~~, CCA]   Minimizer: CCA
```

Example: Finding minimizers of consecutive *k*-mers with a deque

Methodologies

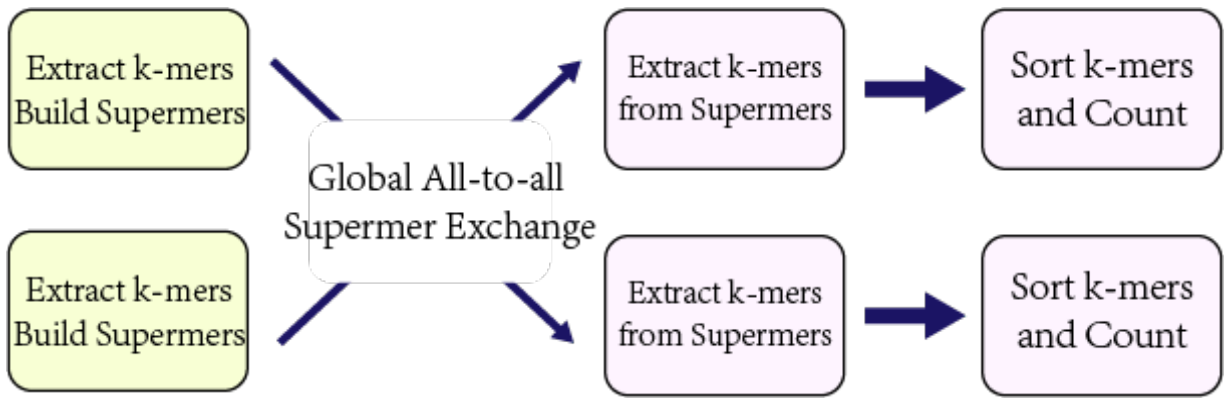


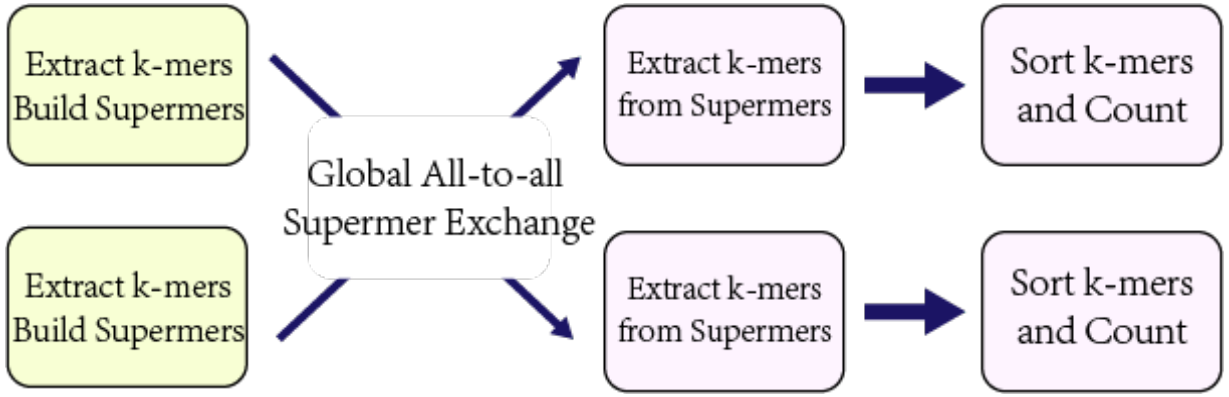
Table of Contents

1. Introduction
2. **Methodologies**
   - The Common *k*-mer Counting Pipeline
   - Sorting Based Approach
   - Supermer Strategy
   - **Task Abstraction Layer**
3. Performance Results
4. Conclusion

Methodologies

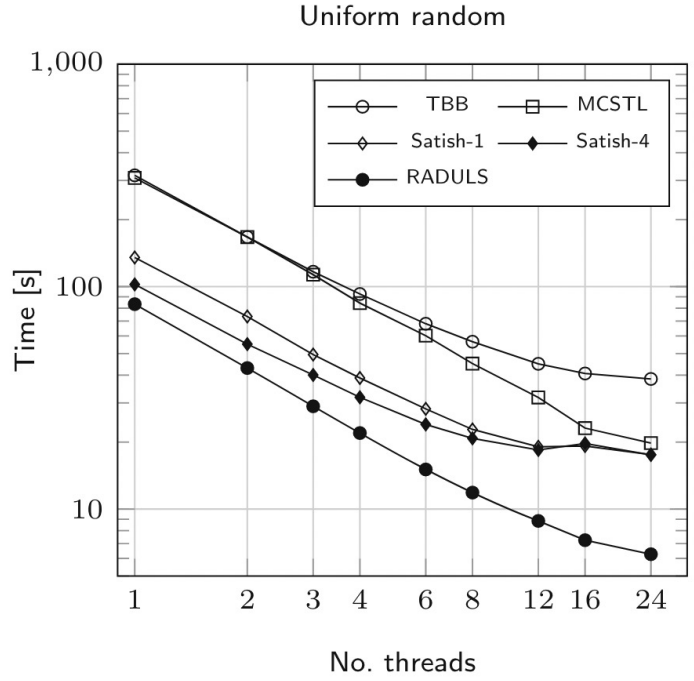
Task Abstraction Layer

Improves stage 2 and 3



- Problem 1: Both RADULS and PARADIS exhibit poor weak scaling performance once the number of threads exceeds 16.

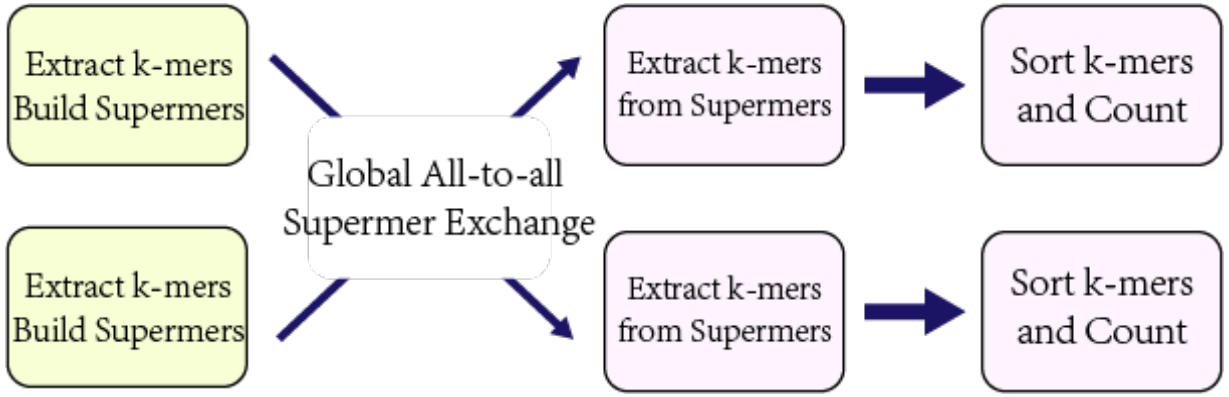
- Problem 2: Modern CPUs have many cores and more than 1 non-uniform memory access (NUMA) domains.



Methodologies

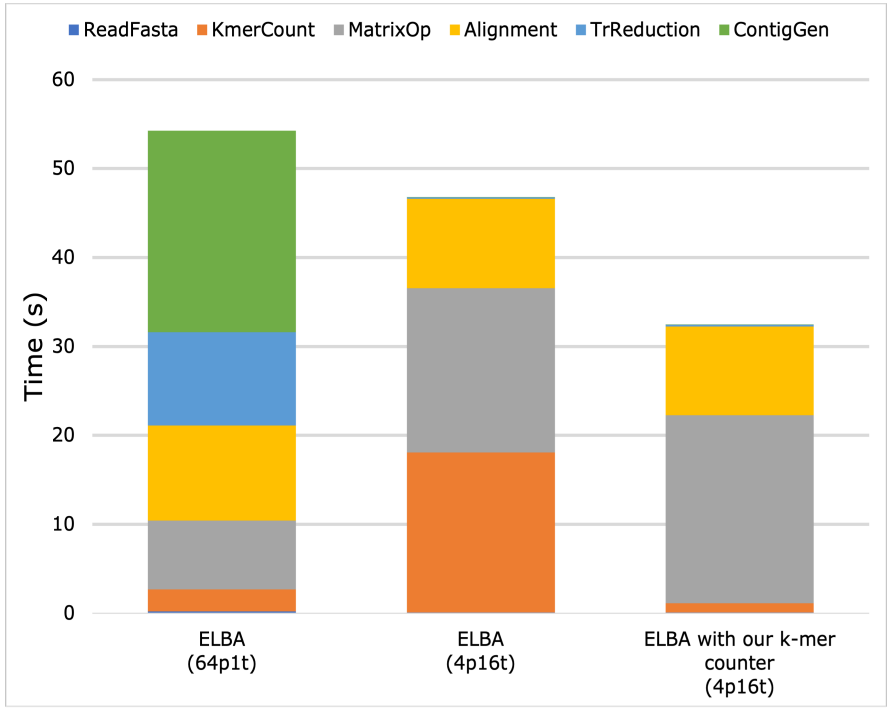
Task Abstraction Layer

Improves stage 2 and 3



- Problem 3: Some bioinformatics pipelines have preference for process-level or thread-level parallelization settings.

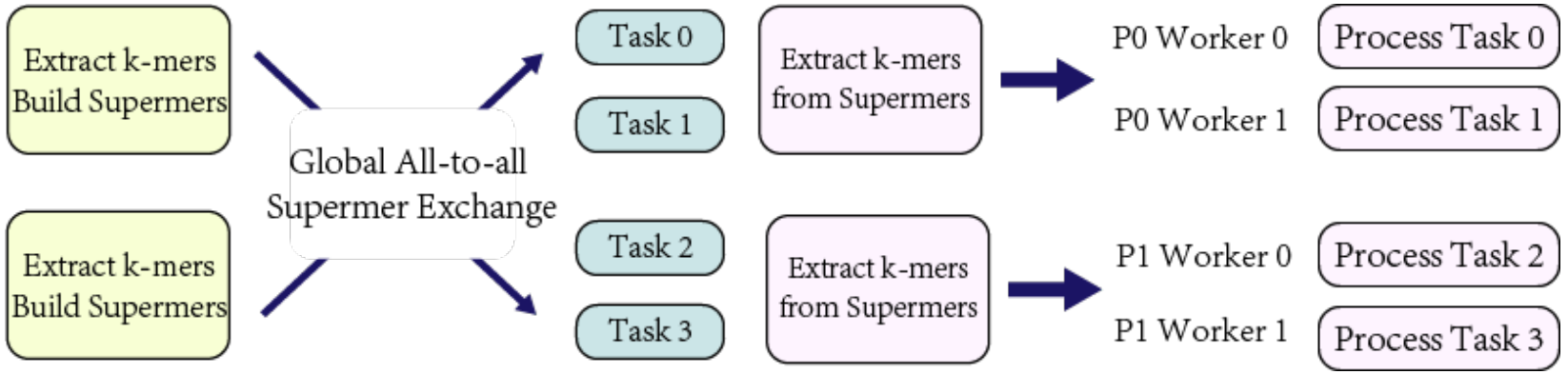
- Problem 4: Some datasets are imbalanced.



Methodologies: Task Abstraction Layer

- Distributed memory k-mer counters usually partition k-mers into num_proc batches.
- We partition the k-mers into s batches(tasks), $\boldsymbol{s > k}$.

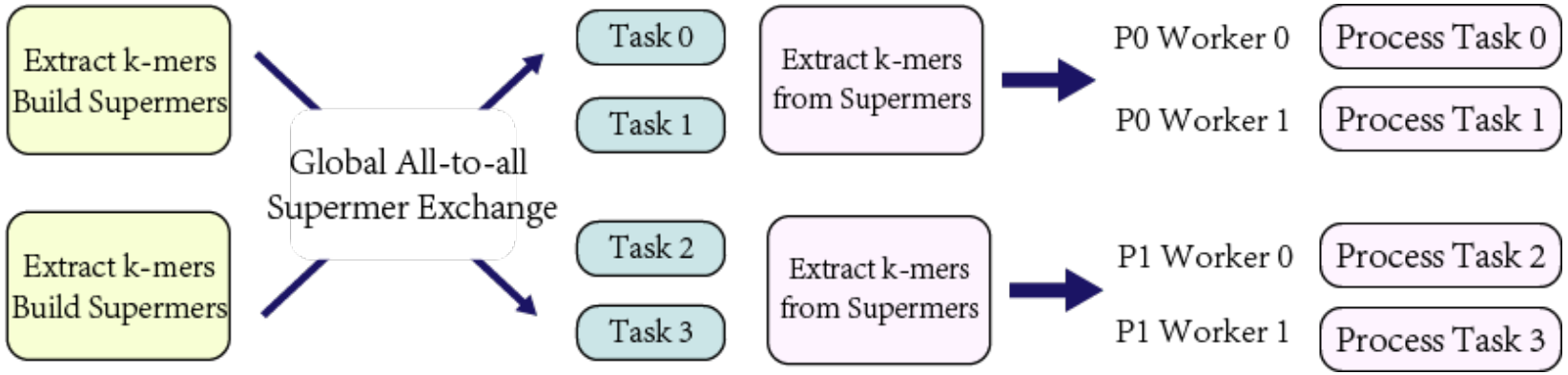
- In the third stage, available computing resources of a process are divided among **several workers.**
- Each worker is assigned some computing resources and several tasks.



Methodologies: Task Abstraction Layer

Benefits:

- We can utilize the **numerous physical cores** of modern CPUs.
- Limiting processes per node and threads per process **reduces scheduling overhead.**
- The layer opens up more opportunities for **load balancing.**



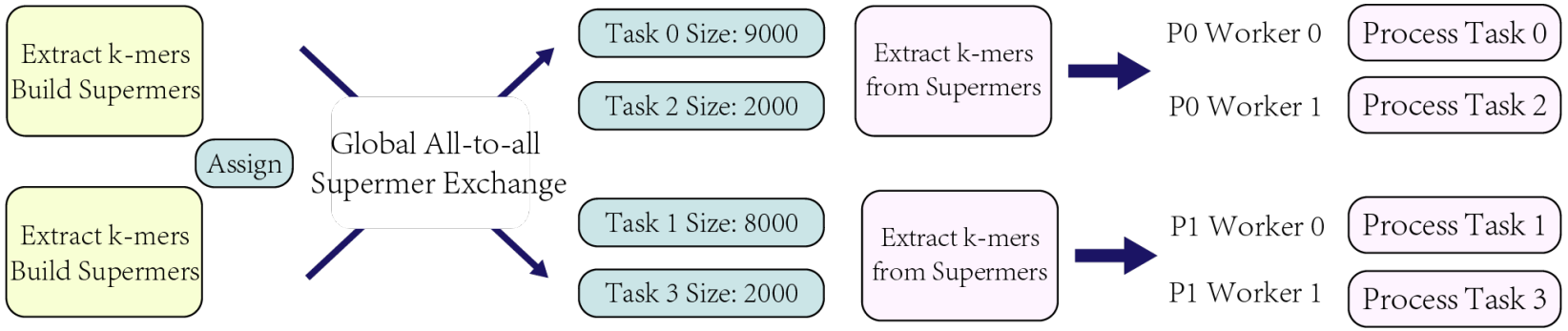
Methodologies: Task Abstraction Layer

Load Balance Strategy

Load imbalance is a nightmare for bulk-synchronous parallel programming model like MPI.

We proposed a load balance strategy based on the task abstraction layer:

- The root process retrieves data about the size of each task **before assigning** it to a target process.
- The goal is to **minimize the largest sum** of task sizes for a single process.



Methodologies: Task Abstraction Layer

Load Balance Strategy

However, there's still **severe load imbalance issue** for some datasets.

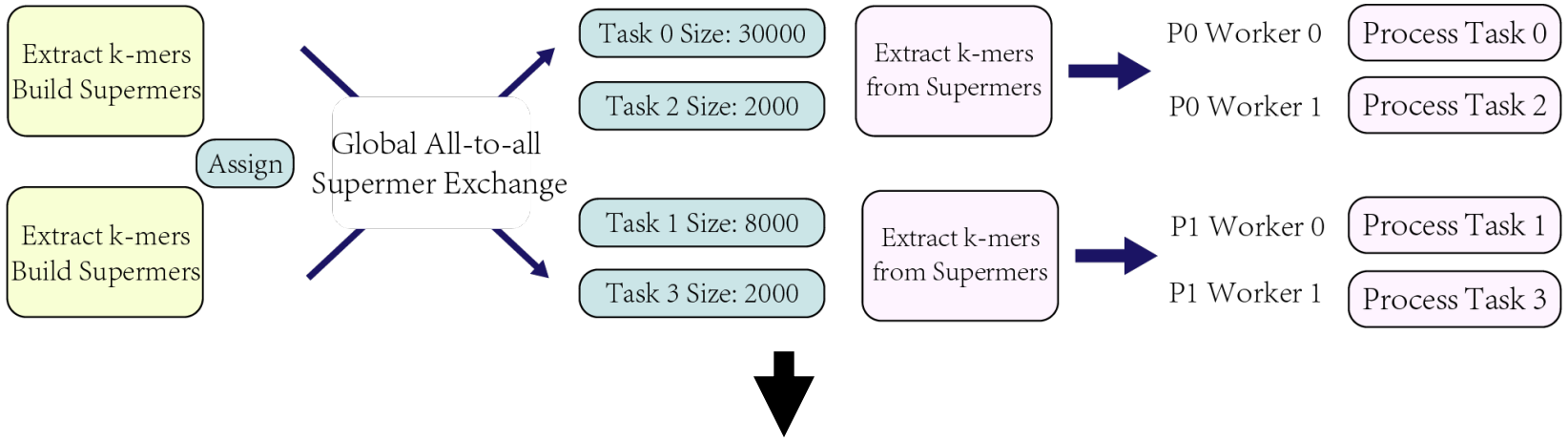
For example, the human genome contains numerous **repeats** of $(AATGG)n$

Both Supermer (including scoring function selection) and Load Balance Strategy **are unable** solve this problem.

```
Read:          TGGAATGGAATGGAATGGAATGGAATGGAATGGAATGGAATGGAATGGAATGGAA
```

Example: When counting 7-mers, at least 10 k-mers from the sequence will be assigned the same id.

Methodologies: Task Abstraction Layer



Advanced Load Balance Strategy

- We use statistics of the task to determine if it includes heavy hitters.
- For heavy hitter tasks, we **transform them locally** before communication happens.

```
Original Task:          [AATGGA, AATGG, AATGG, AATGG, GAATGG]
Transformed Task:       [(AATGG, 4), (ATGGA, 1), (GAATG, 1)]
```

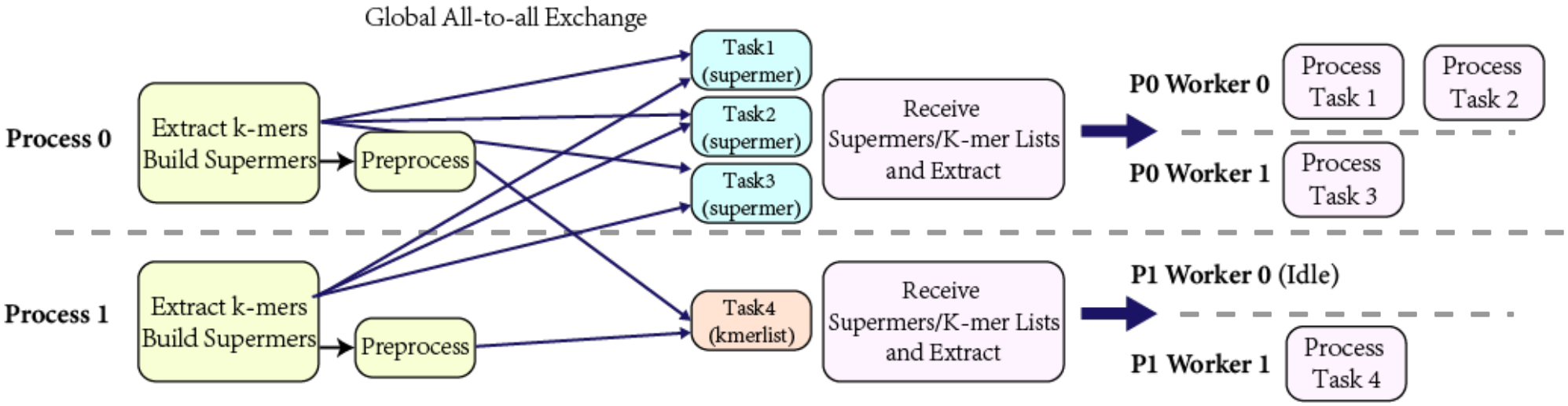
Example: Heavy hitter task transformed

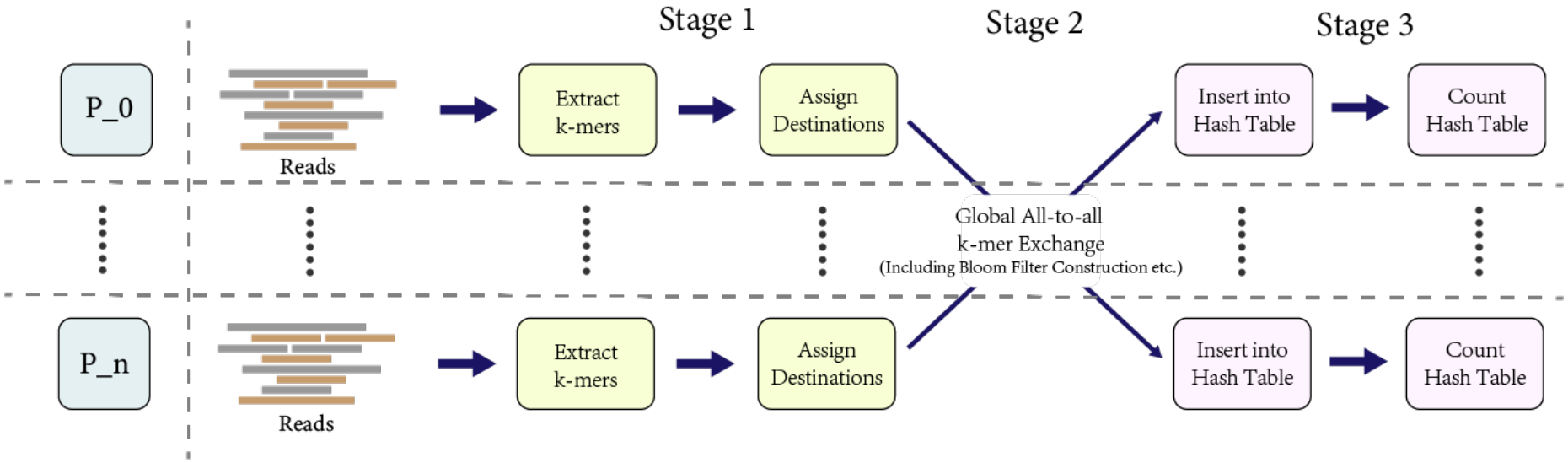
Methodologies: Task Abstraction Layer

Advanced Load Balance Strategy

- We use statistics of the task to determine if it includes heavy hitters.
- For heavy hitter tasks, we transform them locally before communication happens.

Reduces load imbalance for both stage 2 and stage 3





Overview

- **Supermer and Optimized Supermer strategy**
- **Sorting Based Approach → Task Abstraction Layer → Advanced load balance**

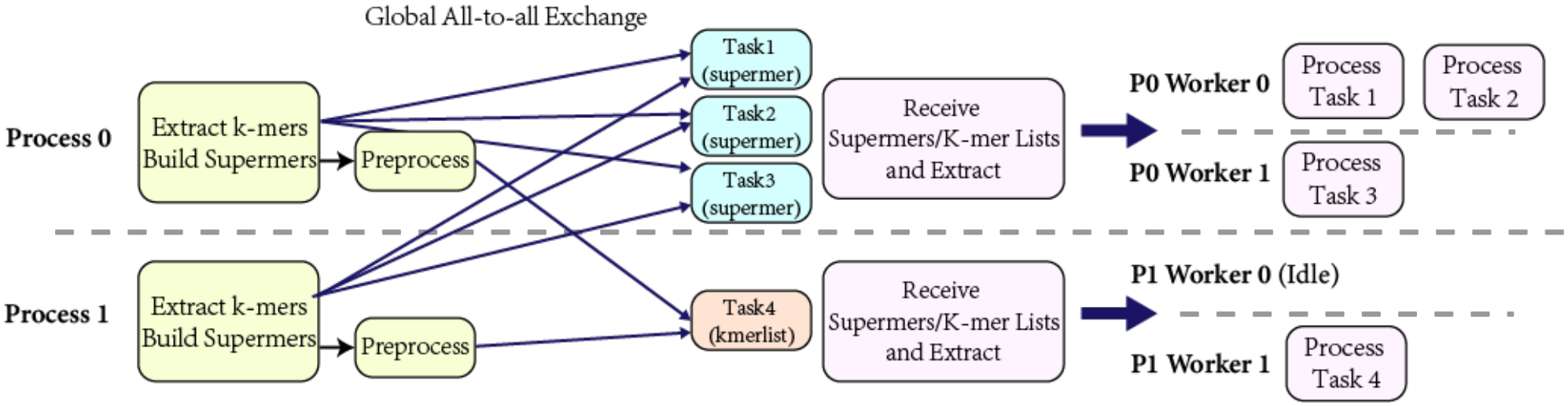
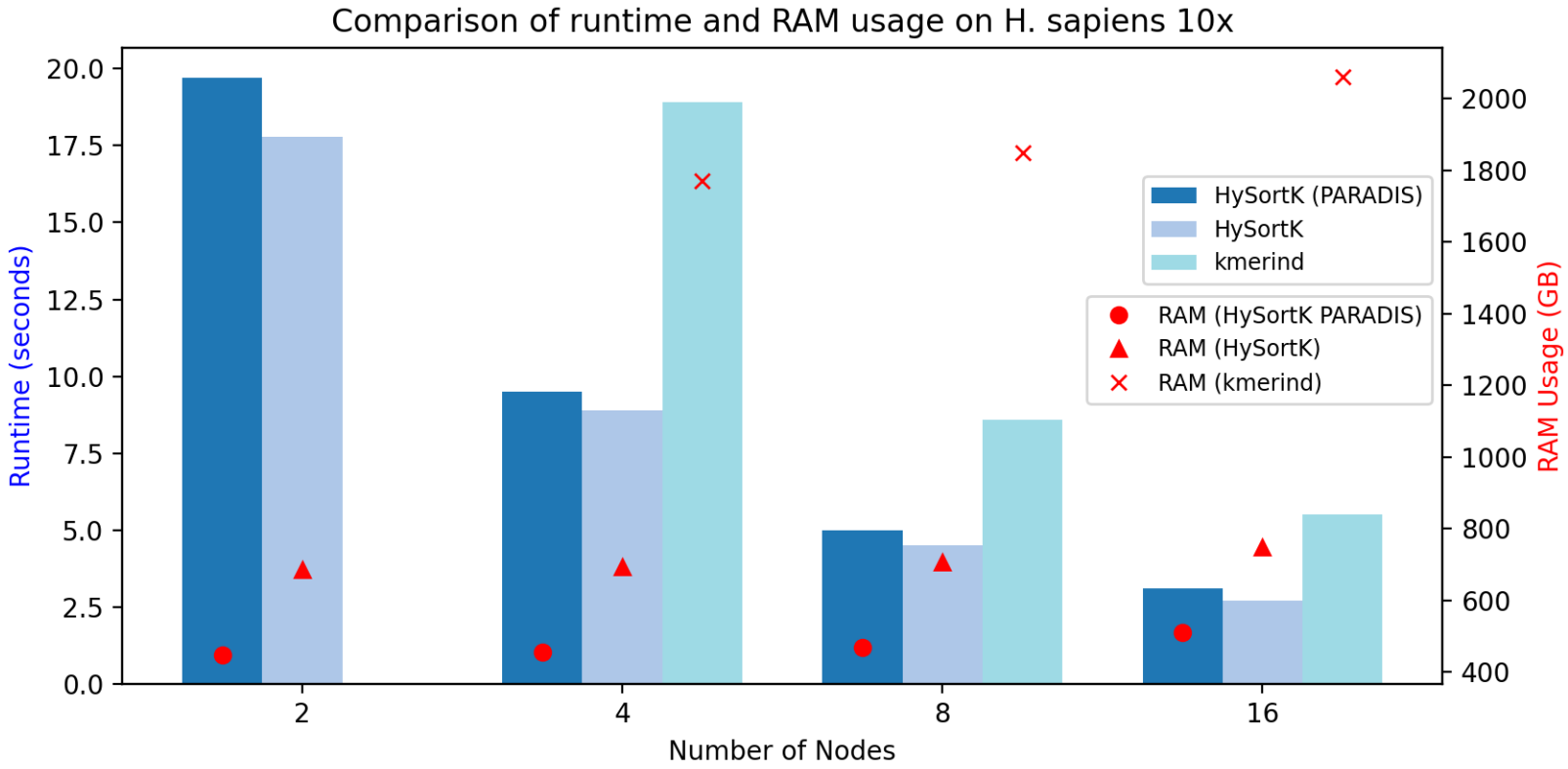


Table of Contents

1. Introduction
2. Methodologies
   - The Common *k*-mer Counting Pipeline
   - Sorting Based Approach
   - Supermer Strategy
   - Task Abstraction Layer
3. **Performance Results**
4. Conclusion

Results

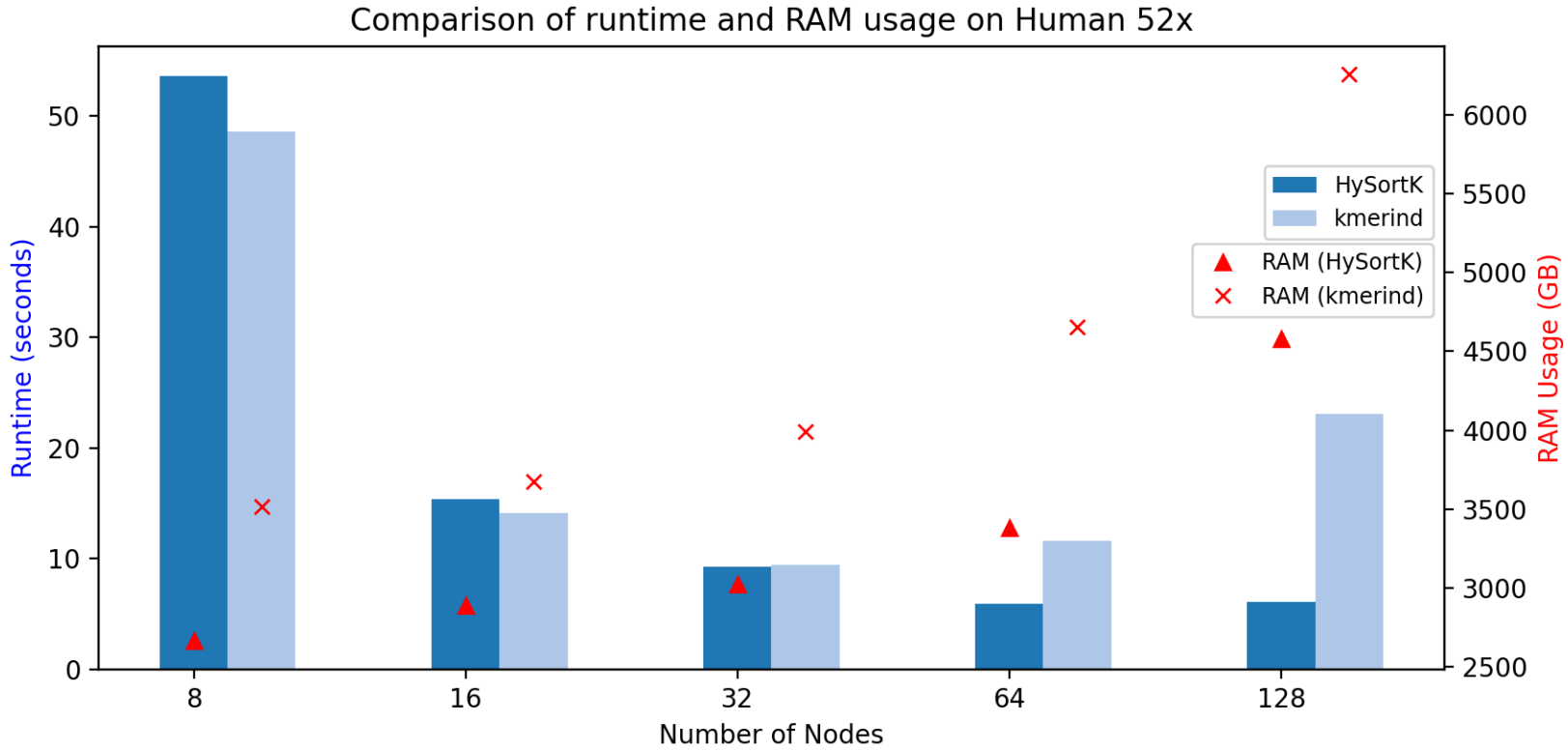
- **Time & Memory Comparison with kmerind**, H. sapiens 10x, 31GB FASTA, IO Excluded
- On NERSC's Perlmutter CPU Node (2 EPYC 7763, 512 GB RAM, 1 NIC per Node)



(Up to **2x** faster and reduces RAM usage by **70%**)

Results

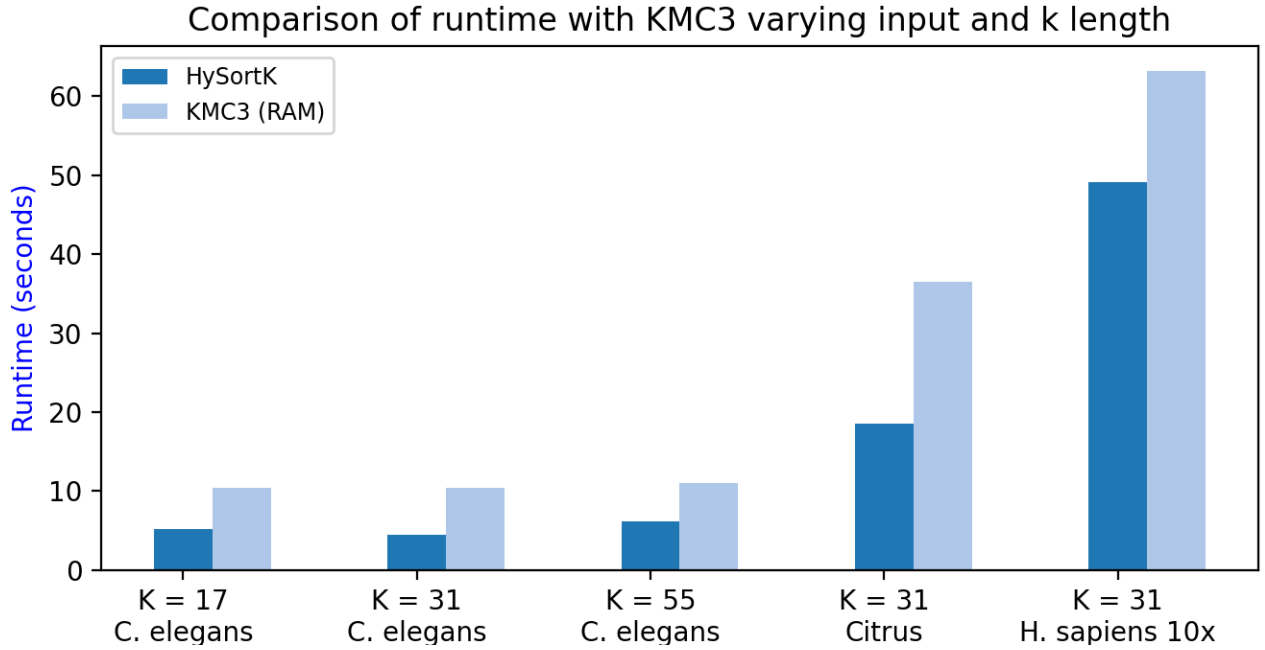
- **Time & Memory Comparison with kmerind**, H. sapiens 52x, 156GB FASTA, IO Excluded
- On NERSC's Perlmutter CPU Node (2 EPYC 7763, 512 GB RAM, 1 NIC per Node)



(**2x** faster on 64 nodes)

Results

- **Single Node Comparison with KMC3**, IO Excluded
- On NERSC's Perlmutter CPU Node (2 EPYC 7763, 512 GB RAM, 1 NIC per Node)



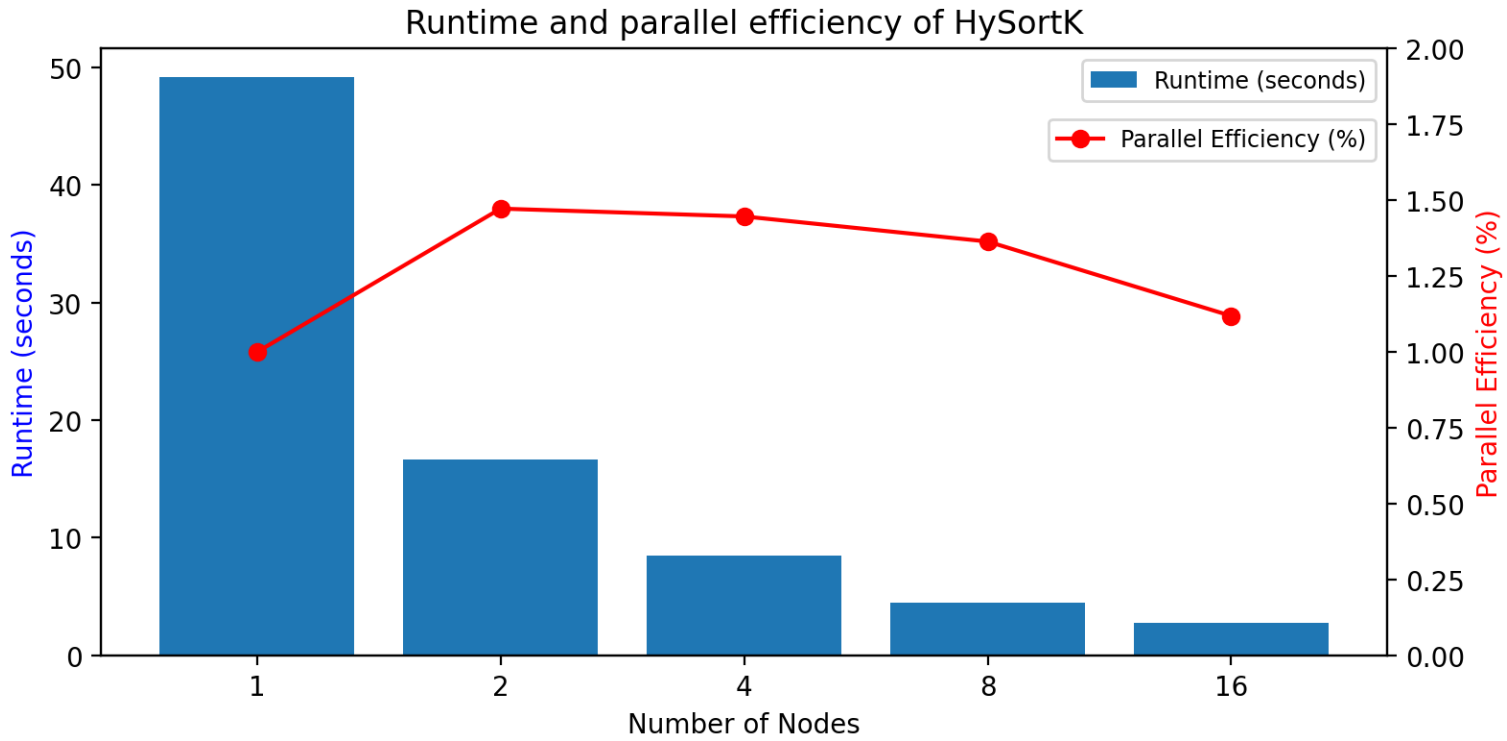
(faster on a single node)

- Counting H. Sapiens 52x with KMC3 needs 412s;

  Our counter finishes the task on 64 nodes in 5.9s, which is a **70**× speedup.

Results

- **Strong Scaling**, H. sapiens 10x, 31GB FASTA, IO Excluded
- On NERSC's Perlmutter CPU Node (2 EPYC 7763, 512 GB RAM, 1 NIC per Node)



$$\text{Parallel efficiency} = \frac{time\_base}{time} \cdot \frac{node\_base}{node} \qquad \text{Here } node\_base = 1 \text{ and } time\_base = 48s\ .$$

Conclusion

- Proposed a highly efficient *k*-mer counter, up to **2× faster** than existing software
- Reduced memory usage by **more than 30%.**
- Improved speed, memory consumption and **flexibility** benefits many bioinformatics pipelines.
- Enables *k*-mer counting to be applied in **a wider range of scenarios.**
- Methods such as the task abstraction layer and load balancing strategy can be **applied to other related applications.**

Links and information

- Github Repo: https://github.com/CornellHPC/HySortK Or Scan QrCode

- Email: yf-li21@mails.tsinghua.edu.cn

- The paper is currently under review. If you're interested in it, please send me an email. I'll send you a copy as soon as it's available.