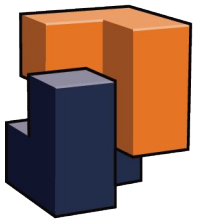


CRISP
Center for Research on Intelligent Storage and Processing in Memory

Ultra Efficient Acceleration for *De Novo* Genome Assembly via Near-Memory Computing

Minxuan Zhou[§], ***Lingxi Wu***[*], Muzhou Li[§], Niema Moshir[§], Kevin Skadron[*], Tajana Rosing[§]

University of California, San Diego[§],University of Virginia[*]

Joint First Author

PACT '21

Executive Summary

Motivation: Near-data processing (NDP) is an emerging memory-based approach that can provide scalable parallelism and memory bandwidth by integrating massive number of cores in memory devices

Problem: *De novo* genome assembly using de Bruijn graph (DBG) is essential for novel species discovery and metagenomics. However, DBG processing is memory-bound:

- Large peak memory footprint
- High L2 and L3 cache miss rate
- High memory bandwidth demand
- Benefit from high core count if memory-bound issues are addressed

Goals: Design a DBG processing solution that simultaneously meet the goals of high memory capacity, large parallelism, low memory latency, and high memory bandwidth

Our Solution: An end-to-end parallel framework leveraging NDP technology that solves the memory-bound issues of DBG processing. We leverage domain-specific knowledge to optimize its performance

- Bucket shuffling strategy → minimize inter-core communication
- Data (*K*-mer) buffering and compression technique → reduce on-chip network traffic
- Speculative contig expansion → overlap the latency of k-mer query

Key Results:

- 33X and 16X speedup over the CPU baseline for graph construction and graph traversal
- Expect to outperform a conventional platform even more with larger genome size
- Performance scales well with larger system size

CRISP

Outline

1. Near-data-processing (NDP)
2. Background
3. NDP-accelerated parallel DBG assembly
   - Parallel graph construction
   - Parallel graph traversal
4. Software & hardware support
5. Evaluation
6. Conclusion

CRISP

Near-data-processing (NDP)

Data processing unit and **storage unit** are co-located

- Most popular design: 3D-stacked memory
- Compute-enabled **logic die** underneath several **DRAM dies**
- Retrieve data through fast **through-silicon vias** (**TSVs**) ⟶ 8GB/s per vault

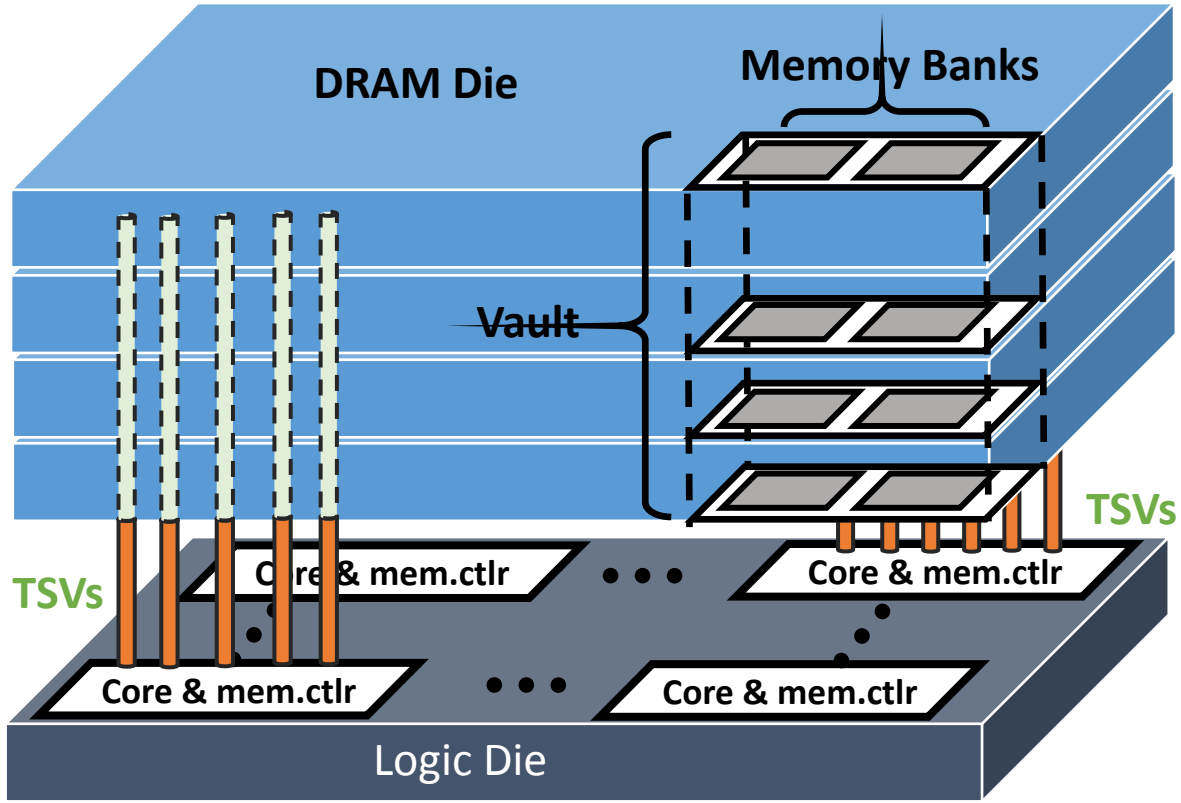


Fig. Micon's Hybrid Memory Cube (HMC)

- Integrate processing logic in memory ⟶ Low mem latency
- Vaults ⟶ High mem. level parallelism
- TSVs ⟶ High internal mem. bandwidth (480 GB/s per cube!)
- Integrate cores per Vault ⟶ High computation parallelism
- Easy to scale out ⟶ Large mem capacity

CRISP

Near-data-processing (NDP)

NDP leveraging 3D-stacked memory has accelerated many applications

- **Graph processing**: Tesseract (ISCA ’15), GraphP (HPCA ’18) ...
- **Data analytics**: Mondrian Data Engine (ISCA ’17) ...
- **MapReduce**: ISPASS ’14[1] ...
- **Pointer chasing**: ICCD ’16[2] …

There are even more aggressive forms of NDP:

- Integrate computing units directly into memory dies or using memory technology to build logic
  - Unlock huge **internal bandwidth** and **parallelism** available inside memory arrays
- E.g., Processing at memory’s subarray-level row buffers → ***in-situ*** computing
- ***In-situ*** offers **10^6 X** higher bandwidth and **10^3 X** lower energy[3]
  - General-purpose: DRISA (Micro ’17), Fulcrum (HPCA ’20), SIMDRam (ASPLOS ’21) …
  - Domain-specific bio accelerators: Pinatubo (DAC ’16), Radar (DAC ’18), Gencache (Micro ’19), Sieve (ISCA ’21) ...
- However, **limited to simple operations** such as row-wide logical **AND/OR/XOR** → not ideal for complex applications such as genome assembly

[1] *S. Pugsley et al., NDC: Analyzing the Impact of 3D-Stacked Memory þ Logic Devices on MapReduce Workloads, ISPASS 14*
[2] *K. Hsieh et al., Accelerating pointer chasing in 3D-stacked memory: Challenges, mechanisms, evaluation, ICCD 16*
[3] *R. Balasubramonian, Near-Data Processing: Insights from a MICRO-46 Workshop, Micro 14*

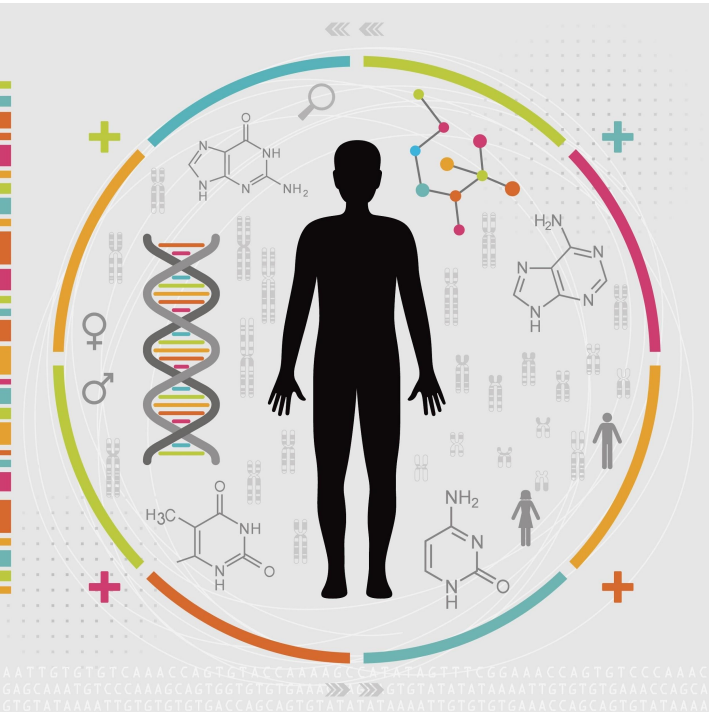

CRISP

Outline

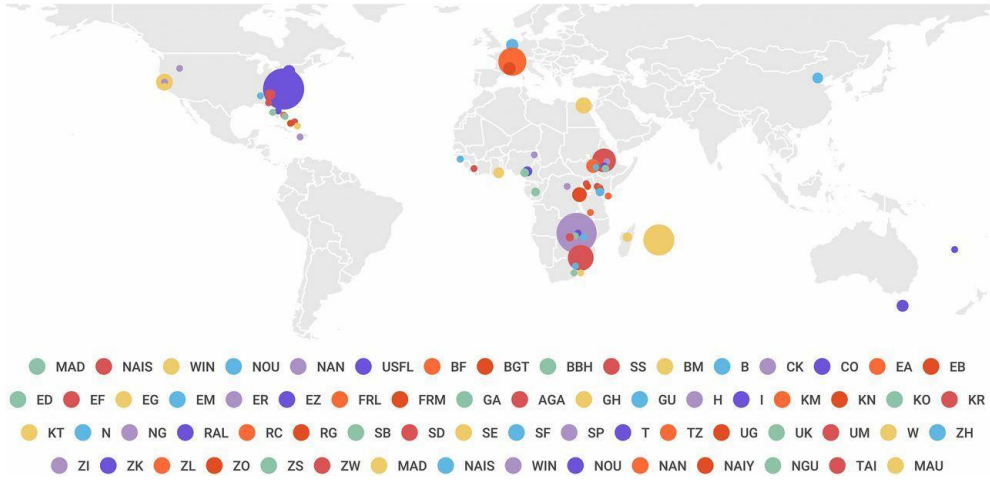
1. Near-data-processing (NDP)
2. Background
3. NDP-accelerated parallel DBG assembly
   - Parallel graph construction
   - Parallel graph traversal
4. Software & hardware support
5. Evaluation
6. Conclusion

CRISP

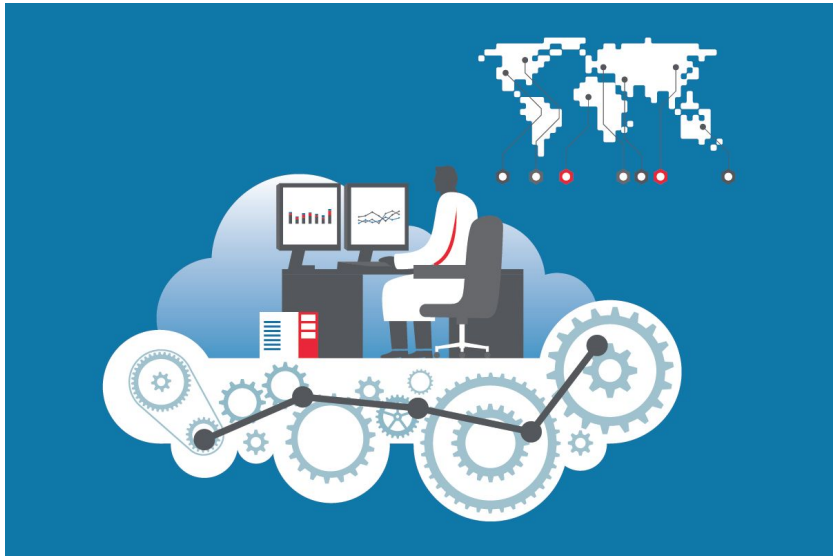
Background - bioinformatics



Precision Medicine



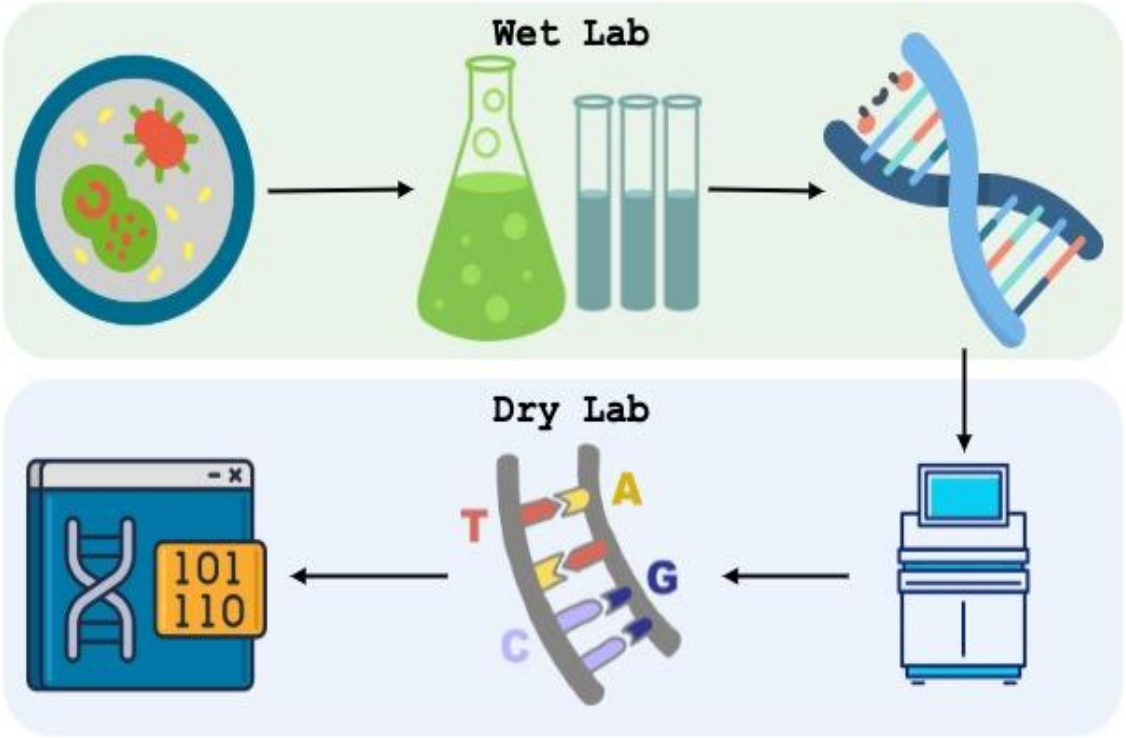
Disease Surveillance



Population Genetics

CRISP

Background - DNA (genome) sequencing

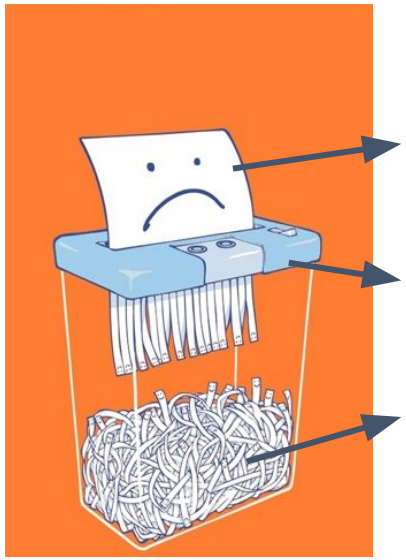


Challenge:

- **No sequencer** takes long DNA as an input, and gives the complete sequence of A, C, G, T, as output
- All sequencers chop DNA into pieces (***reads***) and identify reads but not how they fit together → **reads need to be ordered**

Task:

- Extract raw DNA sequences from living organisms
- Represent them with series of base pair **A,C,G,T**



CRISP

Background - *de Bruijn* graph genome assembly

Q: How to piece together a complete genome from its fragments?

If there is a reference genome of the same species:

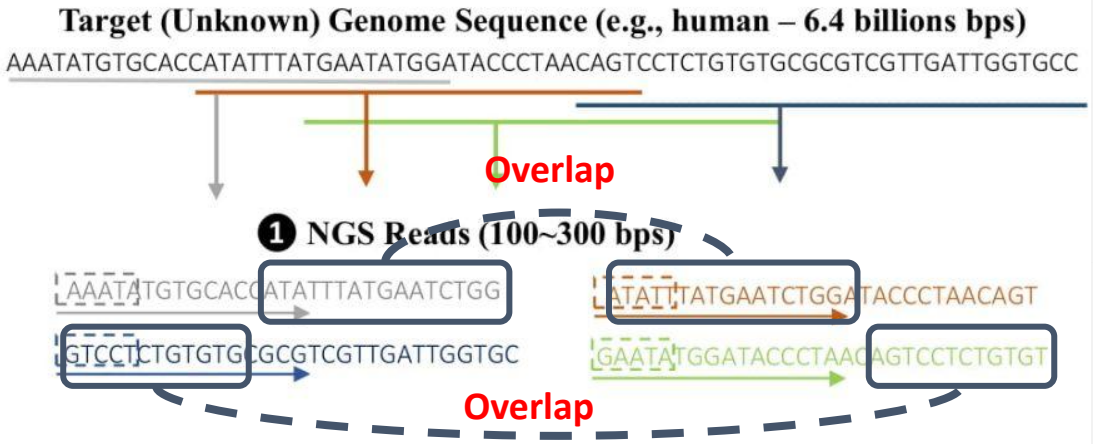
Map (align) each read to the reference

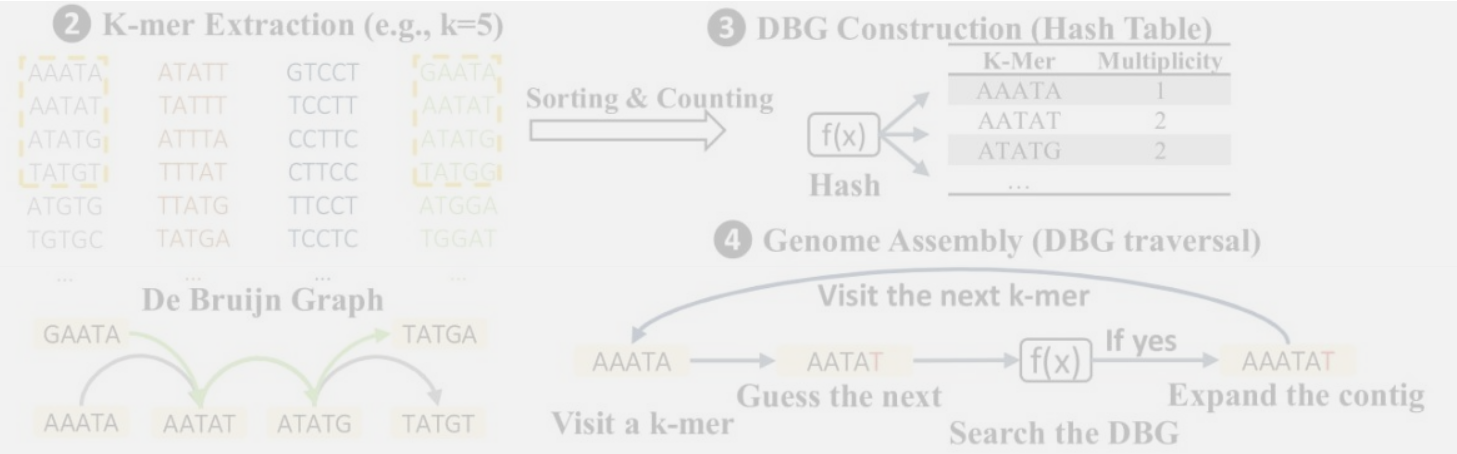
Q: What if the genome is sequenced from an unknown species?

Use ***De Novo*** assembly algorithms

Intuition: Leverage the **overlapping** portions among **reads** to figure out their relative **orders**. Key data structure: ***de Bruijn*** graph (**DBG**)

DBG: a directed multigraph that stores reads' overlapping information



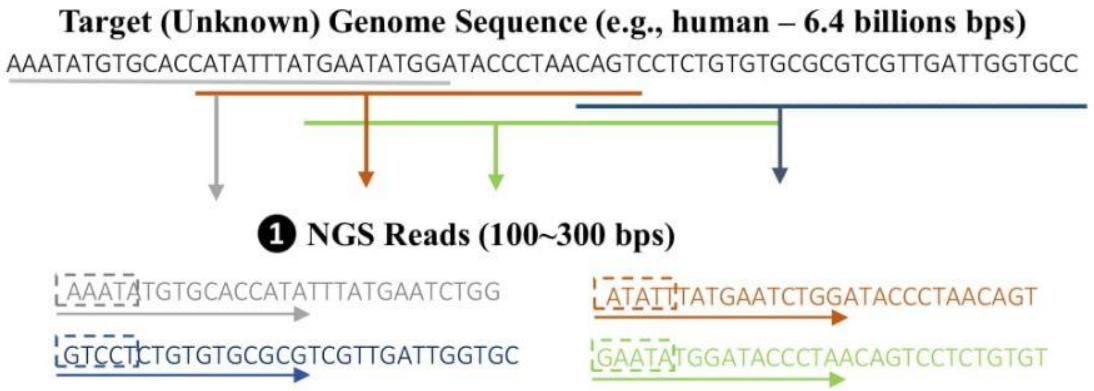


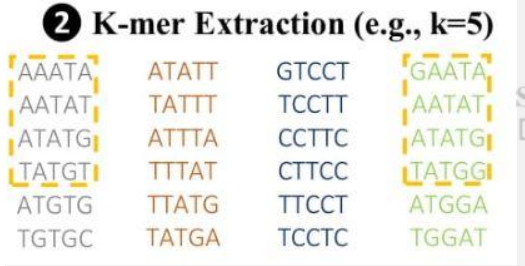
Background - *de Bruijn* graph genome assembly

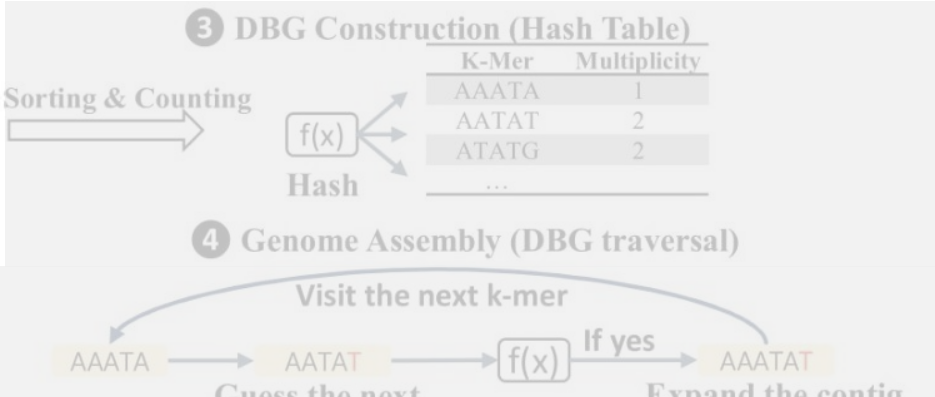
DBG: a form of directed graph that built on the overlapping relation among *k*-mers

- ***K*-mer**: a DNA subsequence of size k
- **Node**: a unique *K*-mer
- **Edge**: 'k-1' overlapping base pairs between two *K*-mers

Extracts overlapping *K*-mers from NGS short reads



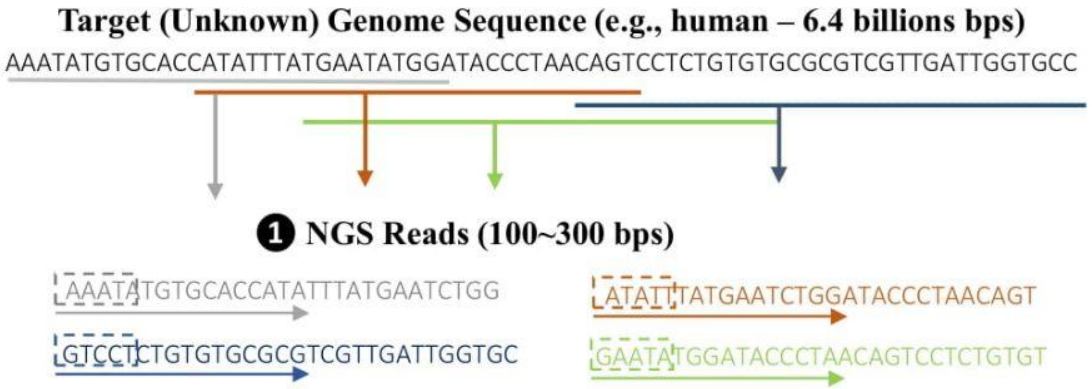


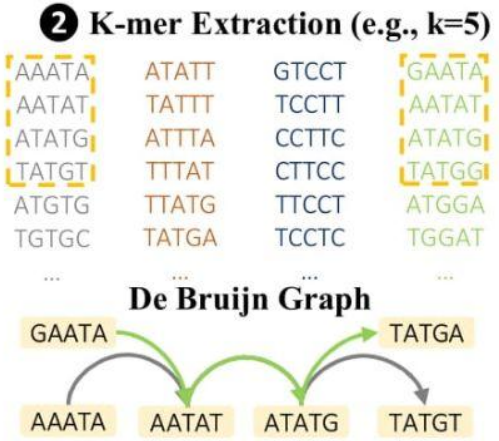


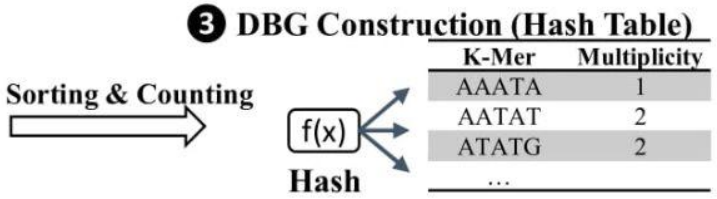
Background - *de Bruijn* graph genome assembly

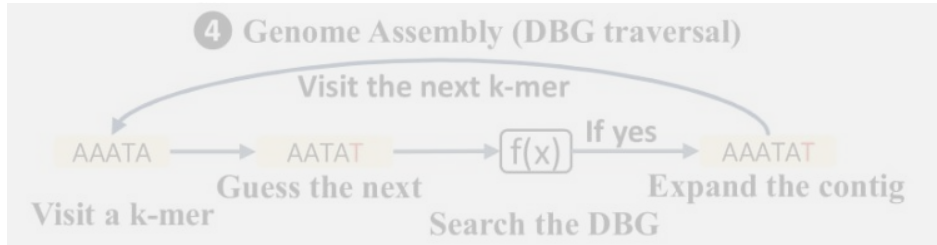
DBG Construction

- **Store unique *K*-mers (nodes)** as keys into a hash table, and its multiplicity (num. of occurrences) as values
- **Edges are stored implicitly**: query {A/C/G/T + (*K*-1) prefix} to see if a previous node exists. If so, there is an incoming edge. Similarly, query {(K-1) prefix + A/C/G/T} to check the existence of an outgoing edge





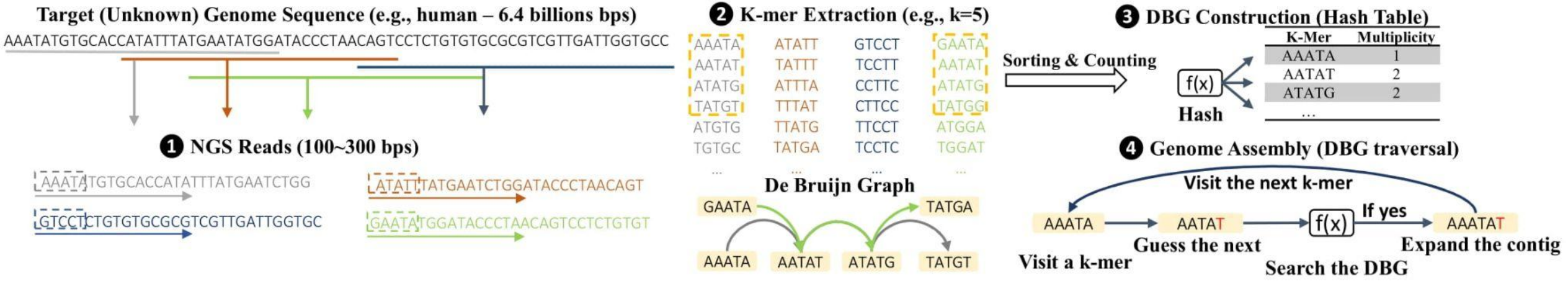




Background - *de Bruijn* graph genome assembly

DBG Traversal (contig generation):

- Traverses the graph to connect chain of *K*-mers to longer sequences called *contigs* → query the next *K*-mer in hash
- Every edge is traversed only once: Eulerian path → Has tractable solution and can finish in polynomial time with respect to graph size



Background - DBG workload analysis

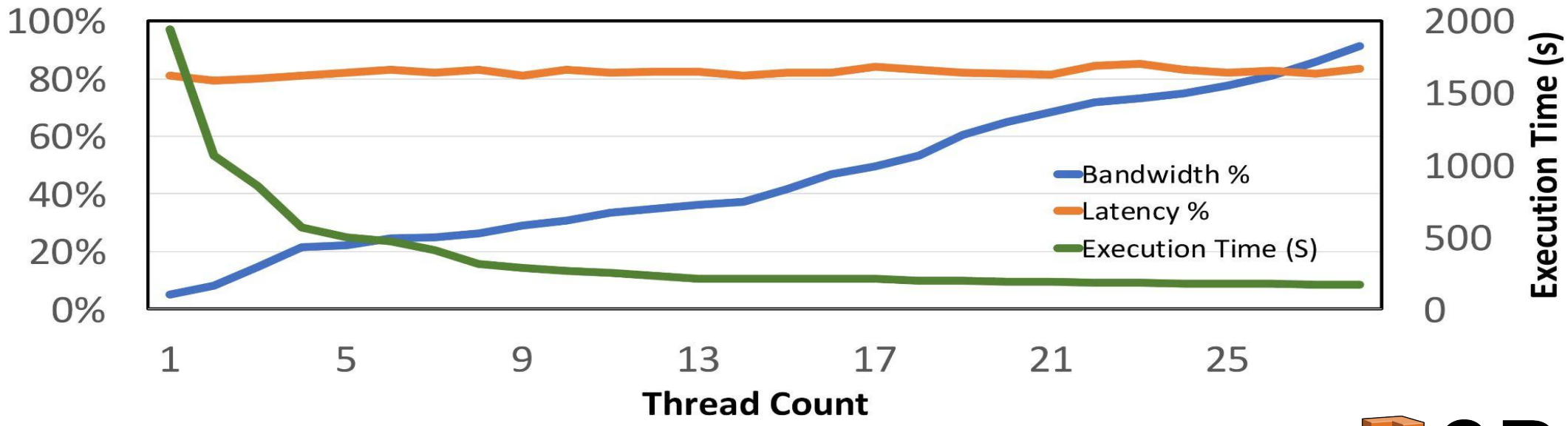
Large peak mem. footprint: Kmerizing a sequence of size L increases its size by $(L - K + 1) * K$

Bound by memory latency (**Orange line**):

- Highly random memory access pattern (45% ~ 75% L2 L3 $ miss rate) → Slow and frequent memory access
- Computation too small to mask data access latency

Increase core count improves performance, however:

- Demands scalable **mem. bandwidth** (**Blue line**)
- **Diminishing return** (**Green line**) → off-chip memory requests served by **limited memory channels**



CRISP

Background - DBG workload analysis

Large peak mem. footprint: Kmerizing a sequence of size L increases its size by $(L - K + 1) * K$

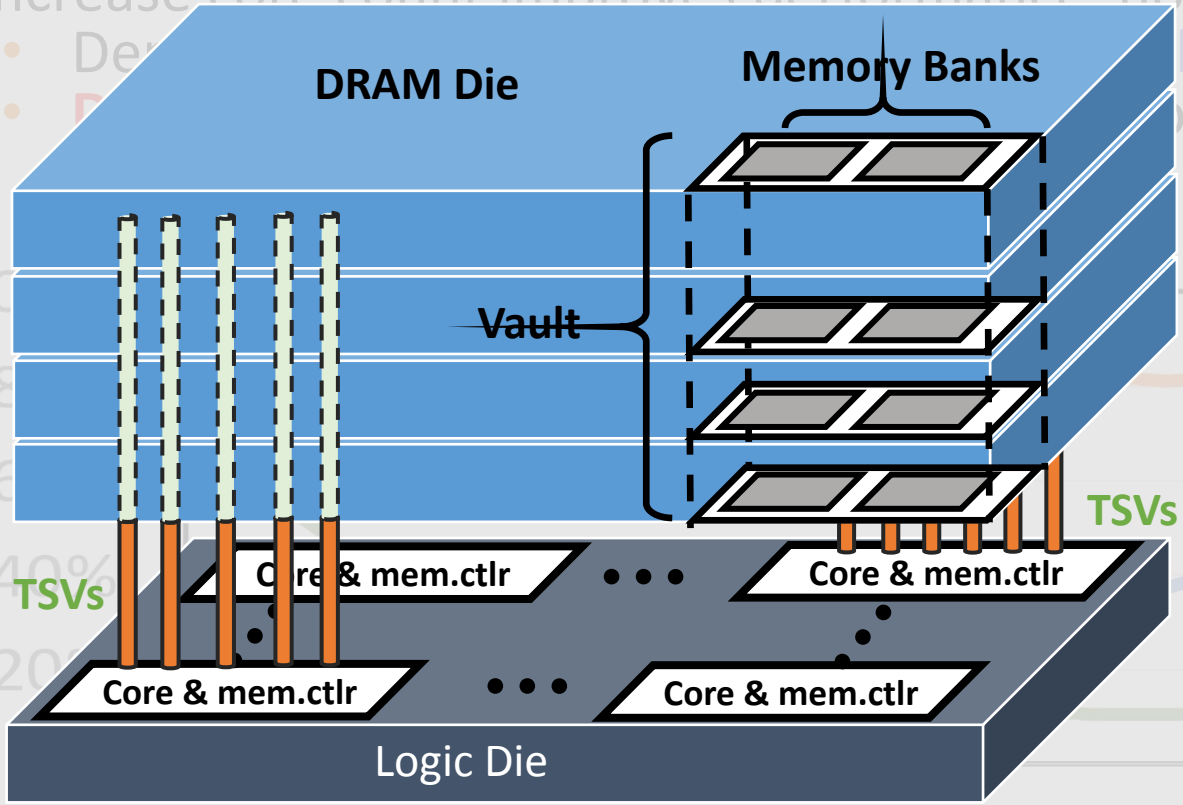
Bound by memory l

- Highly random m… d frequent memory access
- Computation too small to mask data access latency

Increase core count improves performance, however:

- De… line)
- … me…

NDP architecture checks all requirements!



- Integrate processing logic in memory → Low mem latency
- Vaults → High mem. level parallelism
- TSVs → High internal mem. bandwidth (480 GB/s per cube!)
- Integrate cores per Vault → High computation parallelism
- Easy to scale out → Large mem capacity

Outline

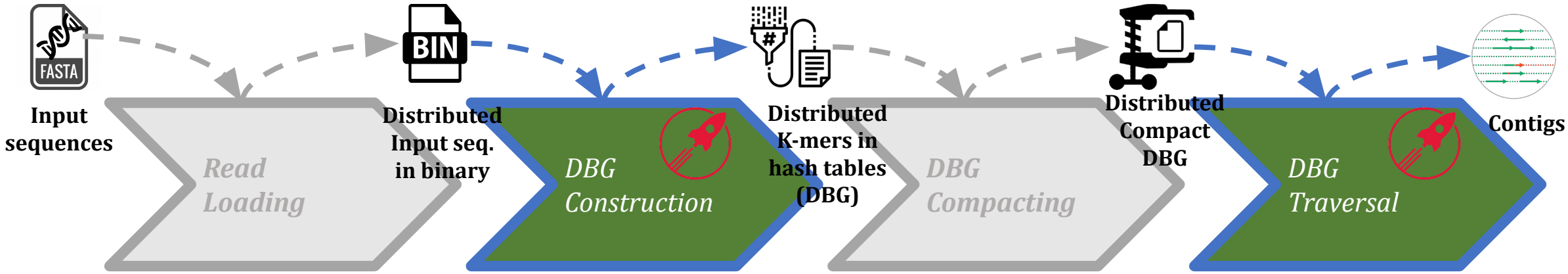
1. Near-data-processing (NDP)
2. Background
3. NDP-accelerated parallel DBG assembly
   - Parallel graph construction
   - Parallel graph traversal
4. Software & hardware support
5. Evaluation
6. Conclusion

CRISP

NDP-accelerated parallel DBG assembly

We propose an NDP-accelerated DBG assembler by modifying a widely used assembler **MEGAHIT**[1]. This is to support a general DBG assembly pipeline

- **Reuse** the interface (major steps in the pipeline) in MEGAHIT
  - Read loading, graph construction, contig assembly, etc.
- **Reuse** the MEGAHIT intermediate data structures
  - FAST* seq. input file, binarized seq. file, distributed hash table, ...
- This work **contributes** NDP-parallel **DBG Construction** and **DBG Traversal** stages
  - **Bottlenecks stages** → together make up **~ 90% exec. time**
  - They benefit from parallel implementation
  - The rest of the pipeline stages are suitable on CPU for perf. and func. reasons



[1]D. Li, et al. Megahit:an ultra-fast single-node solution for large and complex metagenomics assembly via succinct de bruijn graph, Bioinformatics, 2015

Outline

1. Near-data-processing (NDP)
2. Background
3. NDP-accelerated parallel DBG assembly
   - Parallel graph construction
   - Parallel graph traversal
4. Software & hardware support
5. Evaluation
6. Conclusion

CRISP

NDP-accelerated parallel DBG construction

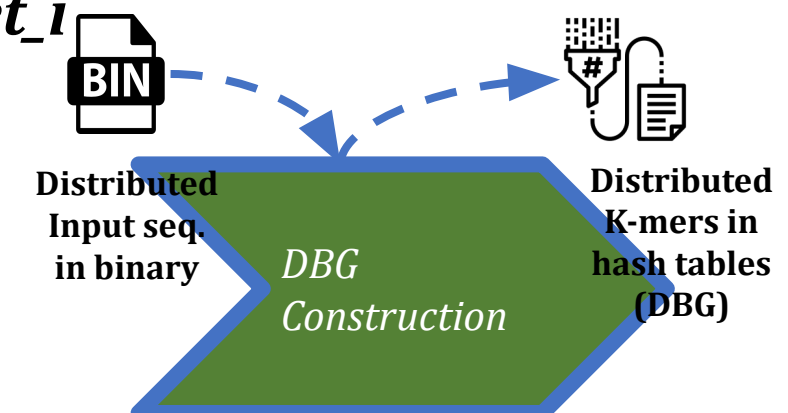
Q: How to map a DBG construction onto a NDP system?

Take inspiration from many prior parallel DBG construction implementations which have ample **task-level parallelism** to exploit:

- Each node gets a slice of input data and produce a portion of intermediate results
- DBG Construction boils down to **put *K*-mers into** appropriate **buckets** (Hash)
- **No dependency** among NDP cores
- NDP offers massive parallelism: we simulated a 16-cube NDP system with 512 NDP cores

Baseline NDP implementation (simplified for clarity):

- Partition **input reads** and **hash buckets** among NDP cores. NDP cores independently put extracted *K*-mers into hash buckets
- Put a k-mer to a bucket: NDP cores run $\boldsymbol{hash(K\text{-}mer) \rightarrow bucket\_i}$
- **Challenge**: Destination $\boldsymbol{bucket\_i}$ may be at a remote NDP core
  - Send the *K*-mers to remote cores through messages
  - Massive fine-grained communication
  - Needs optimization!!!



NDP-accelerated parallel DBG construction

We simulated a naive NDP implementation on a 16-cube 512-NDP-core system to see acceleration potential:

- **Only 7.1x faster** than CPU baseline (14 cores), not impressive consider the NDP system has 36x higher core count
- **60%** to **75%** execution time spent on fined-grained communication (remote function calls)

We propose three **optimization** techniques for the NDP-based DBG construction based on **characteristics of real-world genome data**:

Bucket Shuffling

Message Buffering

K-mer Compression

CRISP

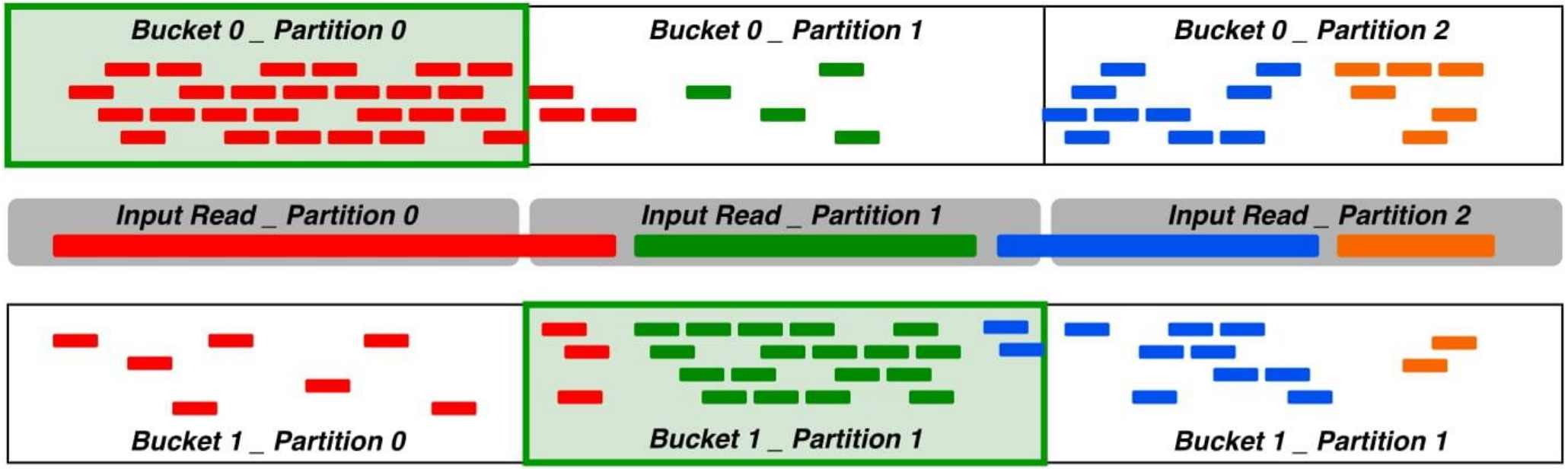
Optimization: Bucket shuffling

Key insight 1: **Co-locate** input **reads** and their preferred **buckets** as much as possible

Real **genomes** often contain **repeat regions**, hash functions inevitably fit *K*-mers from these repeats into a small group of buckets

E.g. most of *K*-mers in ***Input Read _ Partition 0*** are hashed to ***Bucket 0***, allocate both ***Input Read _ Partition 0*** (***red***) and ***Bucket 0*** at the same Vault. Similarly, co-locate ***Input Read _ Partition 1*** (***green***) and ***Bucket 1*** together, to reduce total number of remote messages (packets)

29% to **40**% reduction of messages over a naive random bucket mapping strategy



Optimization: Bucket shuffling

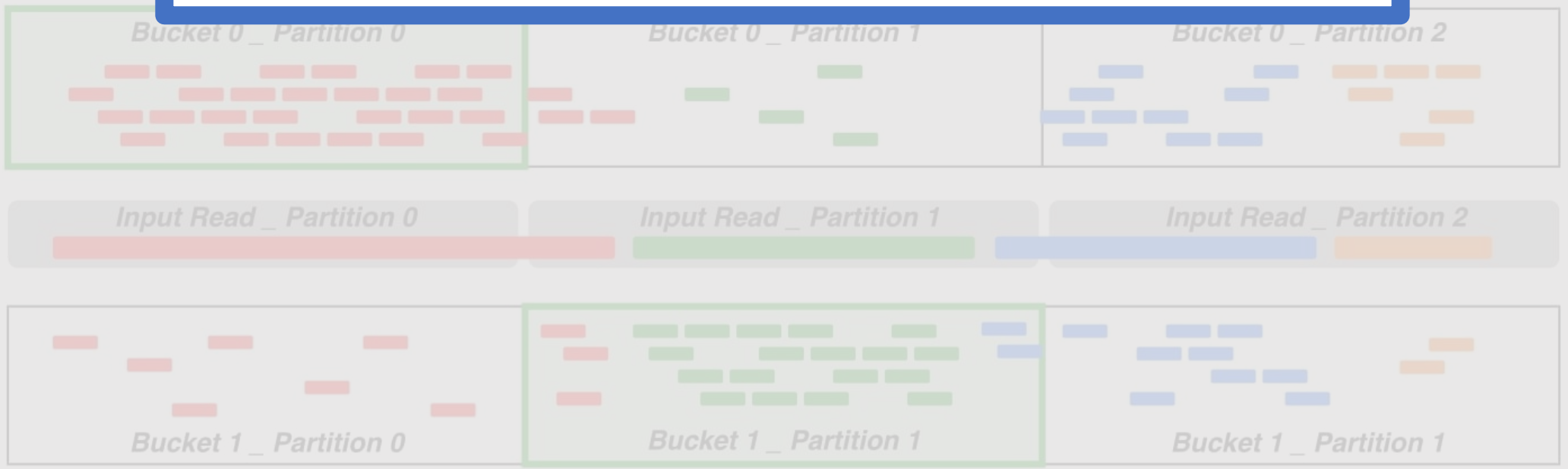
Key insight 1: Co-locate input reads and their preferred buckets as much as possible

Real genomes often contain repeat regions, hash functions inevitably fit *K*-mers from these repeats into a small group of buckets

E.g. most of *K*-mers in ***Input Read _ Partition 0*** are hashed to ***Bucket 0***, allocate both ***Input Read _ Partition 0*** (***red***) and ***Bucket 0*** at the same Vault. Similarly, co-locate ***Input Read _ Partition 1*** (***green***) and ***Bucket 1*** together, to reduce total number of remote messages (packets)

29% to 40% redu

Please read our paper for more details



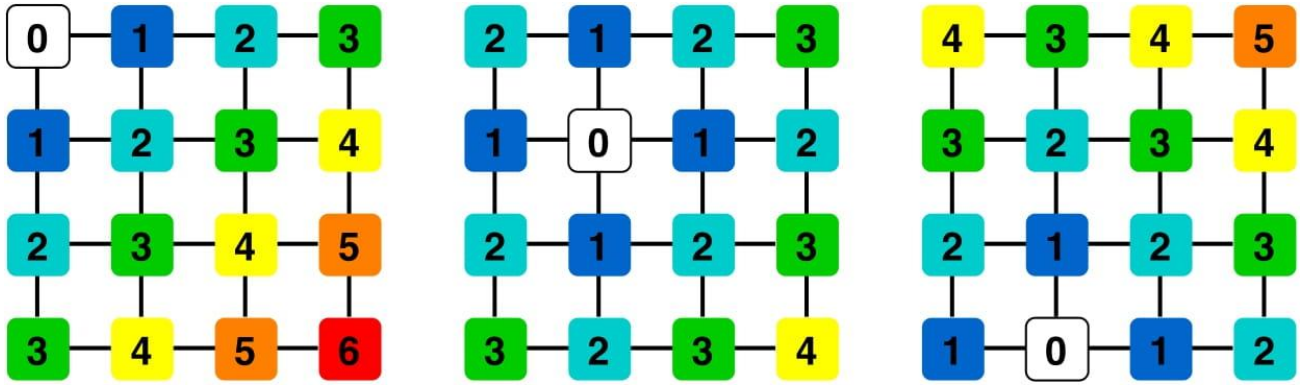
Optimization: Bucket shuffling

Key insight 2: Need to consider **non-uniform latency** of switching packets **in some network**. Simply reducing the number of messages passed among the NDP cores may not be the optimal solution

e.g., mesh-network: a packet arrives at its destination through a series of hops

Suppose an NDP core located at **vault 0** has to send 10 K-mers to all other vaults in a mesh-network. 10×1×2+10×2×3+10×3×4+10×4×3+10×5×2+10×6×1 = **480** total message hops at **vault 0**, **320** at **vault 5**, and **400** at **vault 13**

Designing a best bucket distribution scheme needs to consider **a lot of factors**: NOC, total number of messages, message hop count, …



a. Bucket at vault 0 b. Bucket at vault 5 c. Bucket at vault 13

Implementing the optimal solution is infeasible

We designed a **greedy algorithm** that adds an **insignificant** amount of **overhead** (**<1%**) and works well in our evaluation

CRISP

Optimization: Bucket shuffling

Key insight 2: need to consider **non-uniform latency** of switching packets **in some network**. Simply reducing the number of messages passed among the NDP cores may not be the optimal solution

e.g., mesh-network: a packet arrives at its destination through a series of hops

Suppose a NDP core located at **vault 0** has to send 10 K-mers to all other vaults in a mesh-network. 1 essage hops at **vault 0**, 320 at **va**

Please read our paper for more details

To design a best bucket distribution scheme need to consider **a lot of factors**: NOC, total num. of messages, hop count, …

a. Bucket at vault 0 b. Bucket at vault 5 c. Bucket at vault 13

Implementing the optimal solution is infeasible

We designed a **greedy algorithm** that adds an **insignificant** amount of **overhead** (<1%) and works well in our evaluation

CRISP

Outline

1. Near-data-processing (NDP)
2. Background
3. NDP-accelerated parallel DBG assembly
   - Parallel graph construction
   - Parallel graph traversal
4. Software & hardware support
5. Evaluation
6. Conclusion

CRISP

NDP-accelerated parallel DBG traversal

Purpose: Generate **contigs**, which are longer sequences that span as many K-mers as possible without branching

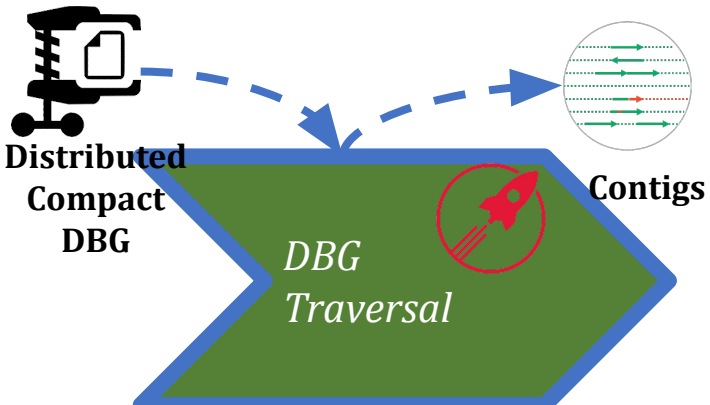
Procedure: Each NDP core selects a **seed *K*-mer** w/o incoming edges, and extends on one of four outgoing *K*-mers **{*K*-mer[1:]+A/C/G/T}** that has the highest multiplicity (**HQE**: high quality extension) → HQE might be scattered among vaults → contig expansion spends **70% of time** on **inter-core communication**

Optimization: **Speculative contig extension** → each NDP core searches multiple steps ahead, instead of only the HQE

Other challenges:

Avoid redundant contig assembly by multiple NDP cores?

Which NDP core gets the K-mer first if requested at different speculative step by different NDP core?



NDP-accelerated parallel DBG traversal

Purpose: Generate **contigs**, which are longer sequences that span as many K-mers as possible without branching

Procedure: Each NDP core selects a **seed *K*-mer** w/o incoming edges, and extend on one of four outgoing *K*-mers **{*K*-mer[1:]+A/C/G/T}** that has the highest multiplicity (**HQE**: high quality extension) → HQE might be scattered among vaults → contig expansion spends **70% of time** on **inter-core communication**

Optimization: **Speculative contig extension** → each NDP core searches multiple steps ahead, instead of only the HQE

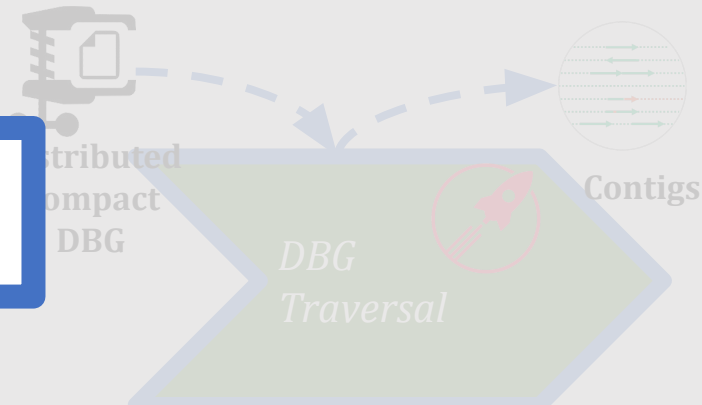
Other challenges:

Avoid

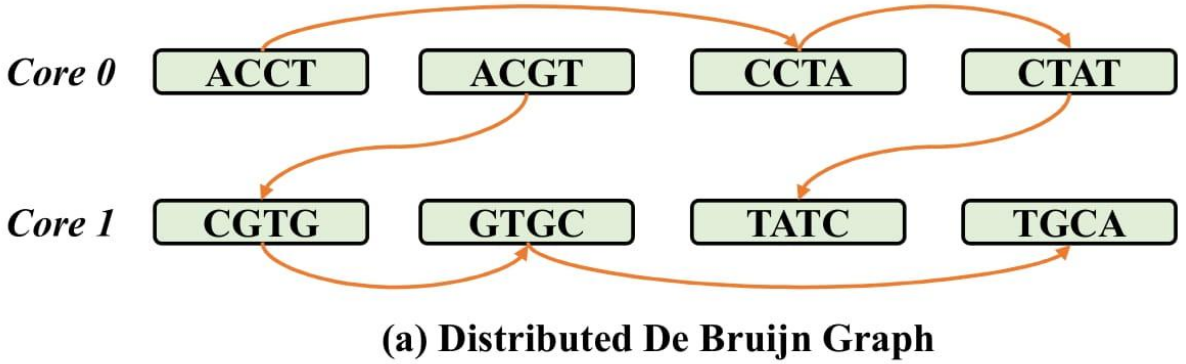
Which

different speculative step by different NDP core?

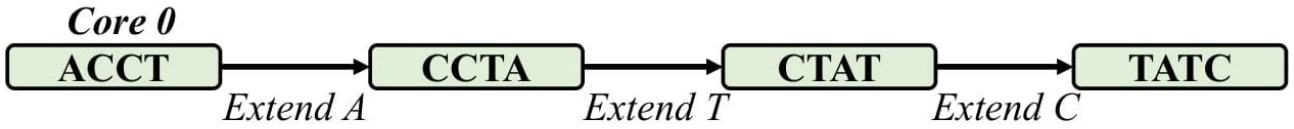
Please read our paper for more details



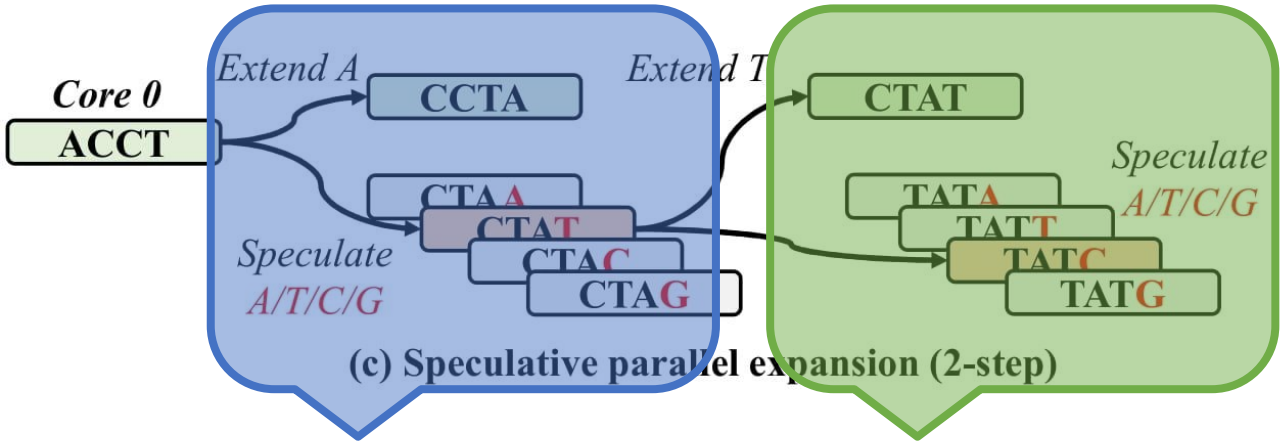
Optimization: Speculative contig extension



(a) Distributed De Bruijn Graph



(b) Original sequential expansion

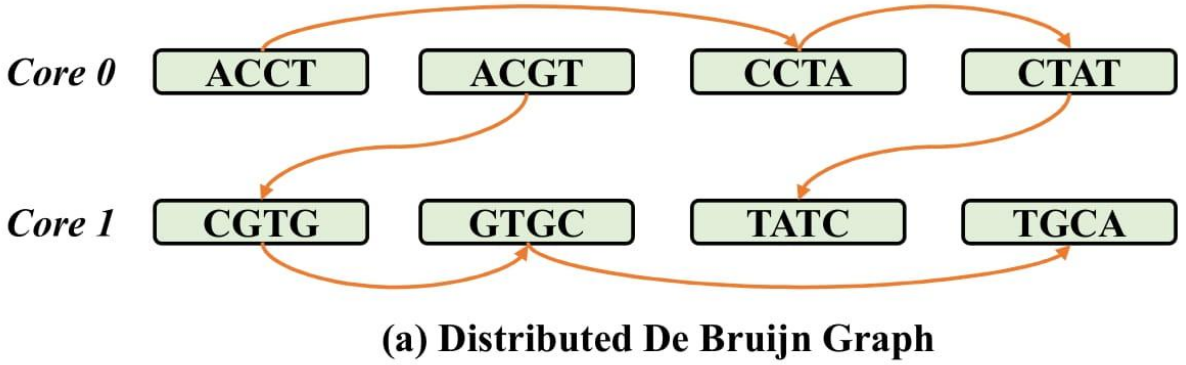


(c) Speculative parallel expansion (2-step)

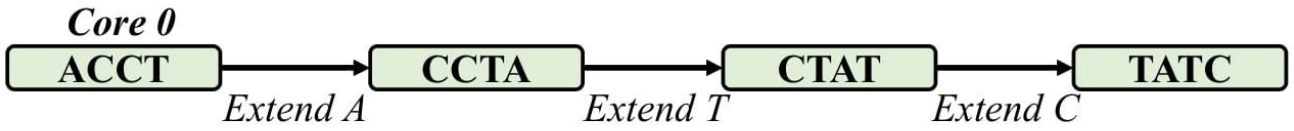
Baseline implementation: Each step **extends** on **one K-mer**, namely, the HQE *K*-mer, If the HQE is stored at a remote vault, send a message to fetch it → cannot progress forward until the current HQE is obtained

Improved implementation: Each step tentatively **extends** on **all four** candidate ***K*-mers** **{*K*-mer[1:]+A/C/G/T}**, send a message to retrieve them all, filter out non-HQEs locally → Improves performance by N x, N = spec. depth

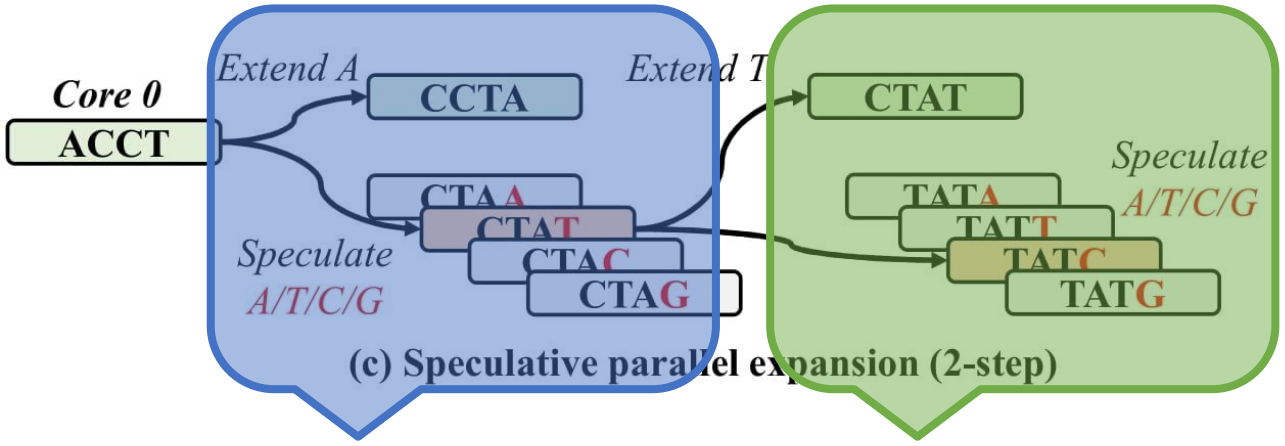
Optimization: Speculative contig extension



(a) Distributed De Bruijn Graph



(b) Original sequential expansion



(c) Speculative parallel expansion (2-step)

Performance evaluation:

spec. depth = 1 ⟶ retrieves 4 *K*-mers

spec. depth = 2 ⟶ ret. 4 * 4 = 16 *K*-mers

spec. depth = 3 ⟶ ret. 4^3 = 64 *K*-mers

…

spec. depth = n ⟶ ret. 4^n *K*-mers

Worst case: Each remote *K*-mer needs a message ⟶ **exponential** growth of num. messages but **linearly** perf. growth

Key Insight: Most significant bits of **{*K*-mer[1:]+A/C/G/T}** are the same, and ***K*-mers are sorted** in the hash buckets ⟶ continuous memory region ⟶ one message is sufficient

Outline

1. Near-data-processing (NDP)
2. Background
3. NDP-accelerated parallel DBG assembly
   - Parallel graph construction
   - Parallel graph traversal
4. **Software & hardware support**
5. Evaluation
6. Conclusion

Software & hardware support

Programming Model:

- Message passing and remote function calls are adopted from Tesseract[1]
- Blocking (***get***) and non-blocking (***put***) function calls
- **A**, **S**, and **V** represent the address type, the size type, and the value type respectively

Operation	**Remote Function Call**
Copy *K*-mer	**get**(id, **A** func, **A** addr, **A** ret, **S** ret_size)
Set Data	**put**(id, **A** func, **A** addr, **S** size)
Request *K*-mer	**put**(id, **A** func, **A** addr, **S** size)
Get Buffered *K*-mers	**get**(id, **A** func, **A** ret, **S** ret_size)
Search *K*-mer	**get**(id, **A** func, **V** hash, **A** ret, **S** ret_size)

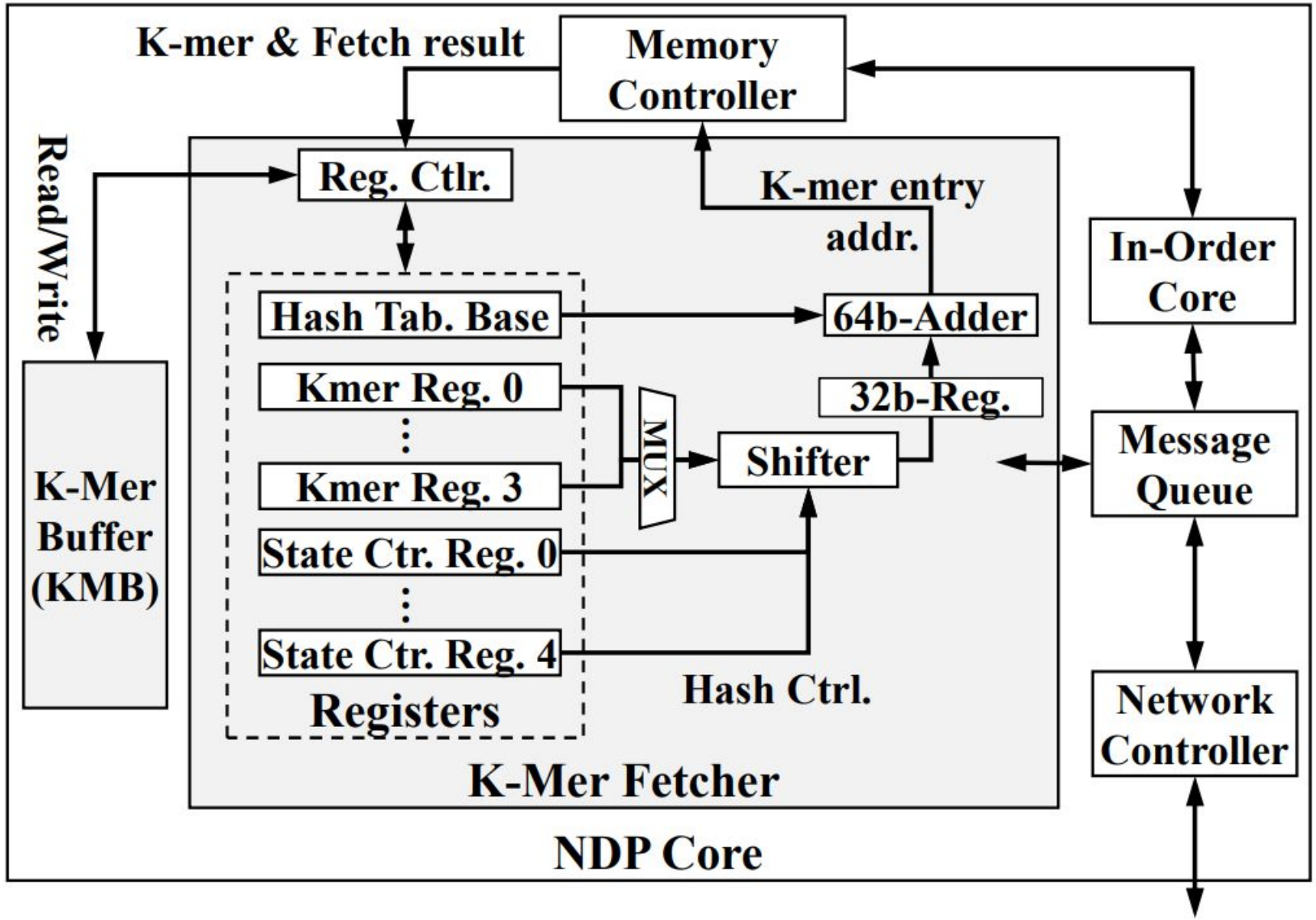
[1]J. Ahn, et, al. Scalable Processing-in-Memory Accelerator for Parallel Graph Processing, ISCA '15

CRISP

Software & hardware support

NDP core:
1 GHz, single-issue, in-order, 32 KB I$ and D$, 80 mW, **0.51 mm² <<** logic layer **226 mm²**

Custom components:
KMF: *k*-mer fetcher. Converts hash val. → mem. addr. → mem. cmd → send to mem. ctrl
KMB: *k*-mer buffer. Reconfigurable, stores *k*-mer related data for graph construction and traversal



CRISP

Outline

1. Near-data-processing (NDP)
2. Background
3. NDP-accelerated parallel DBG assembly
   - Parallel graph construction
   - Parallel graph traversal
4. Software & hardware support
5. Evaluation
6. Conclusion

CRISP

Evaluation - Experimental setup

Hardware modeling:
- KMF: **Verilog HDL** and synthesize the design on **Synopsys Design Compiler**. Placed and routed using **Synopsys IC Compile**
- KMB: **CACTI-3DD** on 22nm technology node

Simulation:
- Emulate the execution of our NDP-based DBG assembler in **Sniper** using multi-threading supported by **OpenMP** → One thread models one NDP core
- Intel **pin-tool** front-end to to generate simulation statistics for multi-core architectures
- Implement the simulation logic for different message-passing functions using Sniper's synchronous and asynchronous timing models
- Memory behavior modeled using **Ramulator**

Workloads:
- Three raw genomes, E-coli, Human, Pineapple downloaded from GenBank, simulated Illumina reads using [ ]

Baseline NDP system (per-cube):
- HMC 2.0 Organization, 8 DRAM layers, 8Gb/layer, 8GB/cube, 32 vaults/cube, internal (external) bandwidth: 512GB/s (480GB/s)
- # of cubes: 1 ~ 16 cubes
- NDP core: 1 GHz, single-issue, in-order, 32 KB I$ and D$, 80 mW, 0.51 mm2
- NOC: crossbar network
- Inter-cube network: Mesh, Fully-connected, Dragon fly

Baselines:
Multicore CPU server
NDP system (HMC w/ one wimpy core per vault) without optimizations
NDP system with software-implemented optimizations
HDP system with hardware-supported optimizations

CRISP

Evaluation - Results

DBG construction stage optimization effect:

DBG construction:

Bucket shuffling: **reduces ~ 40% messages** for a 16-cube NDP system

k-mer buffering/compression: **reduces 10% / 15% messages**

DBG traversal:

Four-step speculation works the best: smaller spec. steps do not fully explore available bandwidth and parallelism. Larger spec. steps incur too much overhead due to increased message count and overhead to resolve conflicts

Compare to software-implemented optimization:

Without HW support, our SW-implemented optimizations achieve ~10x/5x speedup over a vanilla NDP system (5~9x/4~8x faster than CPU baseline) for DBG construction/DBG traversal. However, w/ custom hardware (KMF, KMC, etc.) embedded in the NDP core, there is an additional **~ 3x** performance gain

Scalability:

For DBG construction, performance scales ~ linearly as cubes count increases

For DBG traversal, sublinearly due to difficulty to schedule balanced workloads per NDP core

Overall, the proposed NDP implementation offers **33×** and **16×** performance benefit for **DBG construction** and **DBG traversal** over the **state-of-the-art solution**

Results

DBG construction stage optimization effect:

DBG construction:

Bucket shuffling: **reduces ~ 40% messages** for a 16-cube NDP system

k-mer buffering/compression: **reduces 10% / 15% messages**

DBG traversal:

Four-step … able bandwidth

and paral… sage count and

overhead …

Compare to softw…

Without HW s… er a vanilla NDP

system (5~9x/… ever, w/ custom

hardware (KM… ormance gain

For results on **energy efficiency**, exploration on **network setup**, and comparison to **distributed-memory DBG assembler**, please **read our paper** to find out more details

Scalability:

For DBG construction, performance scales ~ linearly as cubes count increases

For DBG traversal, sublinearly due to difficulty to schedule balanced workloads per NDP core

Overall, the proposed NDP implementation improves the performance of DBG construction by 33× and DBG traversal by 16× compared to the state-of-the-art

CRISP

Outline

1. Near-data-processing (NDP)
2. Background
3. NDP-accelerated parallel DBG assembly
   - Parallel graph construction
   - Parallel graph traversal
4. Software & hardware support
5. Evaluation
6. Conclusions

CRISP

Conclusions

DBG genome assembly is an important bio application, and it is **bottlenecked** in **DBG construction** and **traversal** stages due to their **memory-intensive** nature

We propose an **NDP-based** accelerator design:

- Distribute input reads and intermediate data structures among **vaults** managed by **NDP cores**
- Proposed **four optimizations**: bucket shuffling, k-mer buffering and compression, and speculative contig assembly, which requires **domain-specific knowledge**

This work helps in invaluable life-saving tasks:

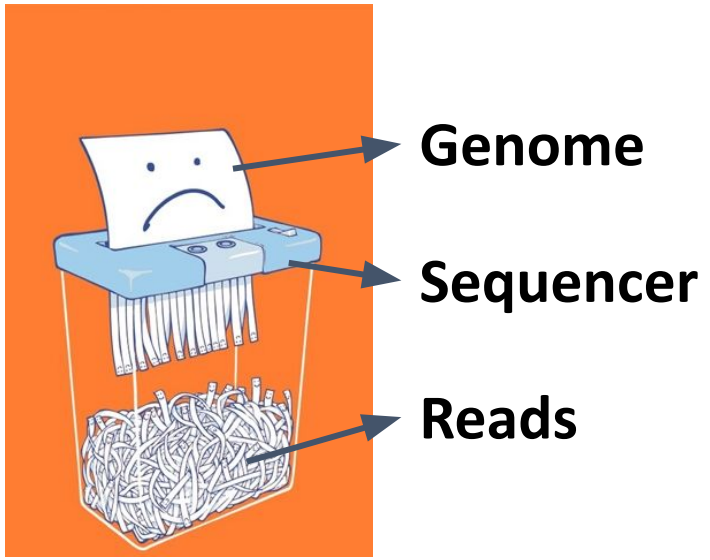
- Discovery of novel genomes
- Metagenomics studies such as surveillance of pathogens, etc
- Precision medicine, personalized treatment, etc.

CRISP


CRISP

Background - *de Bruijn* graph genome assembly

- Current-gen genome sequencers chop a whole genome to billions of short fragments (reads). To reconstruct a sequenced genome, reads need to be pieced together in correct order
- *de Bruijn* graph (DBG) genome assemblers leverage the **overlapping** portions among genome **reads** to figure out their relative **orders** to piece together a complete genome
- **DBG** is a directed multigraph that stores reads' overlapping information
- Assembling genome is similar to restore a book from its shredded pieces.
  - "book" == unknown **genomethe**
  - "shredder" is the **sequencer**
  - "shredded pieces" are the **reads**



- DBG processing is **inefficient in traditional architectures**
  - Large peak memory
  - Bounded by memory latency
  - Highly random memory access pattern
  - Low computation intensity but demands high computation parallelism
  - Bandwidth hungry

CRISP

Accelerating DBG assembly with NDP systems

We propose to accelerate DBG processing using near-data-processing (NDP):

- A 3D-stacked NDP system such as HBM offers:
  - Integrate processing logic near memory dies and access through TSVs → **low mem. latency**
  - **High mem. level parallelism**: e.g., Vaults in HBM
  - **High bandwidth**: 480 GB/s per memory cube
  - Processing unit per vault: **high computation parallelism**
  - Easy to scale out → **large memory capacity**

We propose an NDP-accelerated DBG assembler by modifying a widely used assembler MEGAHIT

- Accelerate the most time consuming steps: **graph construction** and **traversal** (**~ 90%**) time
- Propose the whole framework including **software interface** and **custom hardware**
- **33X and 16X speedup over the CPU baseline for graph construction and graph traversal**

CRISP