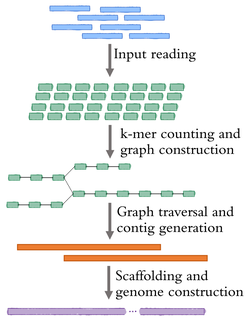
Genome Assembly using an Asynchronous Distributed Actor-based Approach

Souvadra Hati, Richard W. Vuduc (advisor)
School of Computational Science and Engineering, Georgia Institute of Technology

Problem: Genome Assembly

- We use genome assembly as a representative case to showcase the use of an actor-based programming model for data-intensive workloads.
- De novo genome assembly is the process of reconstructing a full genome of a species from short, overlapping, and potentially error-prone DNA-segments.



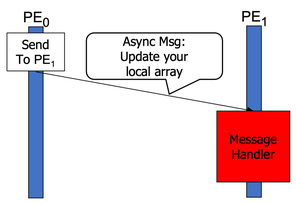
Challenges

- Data generated by the sequencer for complex organisms can be huge (~10TB).
- The pipeline involves lot of irregular communication, hence current frameworks (like MPI) are not the best solutions for programmers to write efficient yet simple software for it.

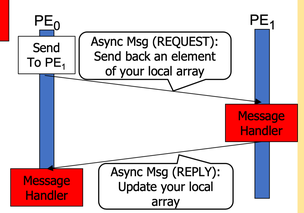
Strategy: An Actor-Based Programming System

- Actor-model a new programming system for PGAS applications that supports fine-grained-Asynchronous Bulk-Synchronous Parallelism (FA-BSP), is specifically designed for working with sparse data-structures with irregular message patterns across processing elements.
- FA-BSP offers fine-grained asynchronous point-to-point actor messages, automatic message aggregation using Conveyors library, distributed and shared memory approach, and inherent isolation to avoid mutable state sharing and data races.
- Actors are computation primitives executed sequentially within mailboxes, ensuring avoidance of concurrent contention in local data accesses. Each processing element (PE) corresponds to a physical CPU core, providing a productive and lightweight computation model for high-throughput HPC applications.

Remote GET and PUT Patterns



Remote-PUT pattern uses **one mailbox** to update a variable stored in a remote PE with information the current PE possesses.

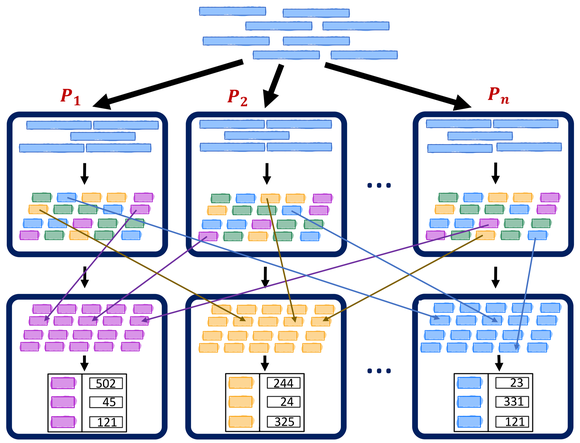


Remote-GET pattern uses **two mailboxes** to ask a remote PE to send update a local variable of the current PE with information the remote PE contains.

Kernel 1: k-mer counting

Problem Statement: Given a set of reads, chop each read into it's k-mers (overlapping sequences of length k) and then count them.

Overview



Algorithm(s)

$R_p \leftarrow$ Input set of reads for processing element p
$b \leftarrow$ buffer size

Algorithm 1 k-mer counting - Actor model

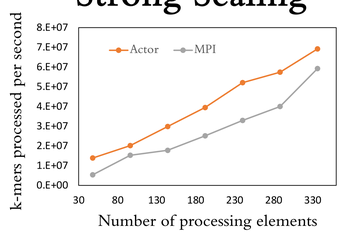
Message Handler
while k-mers left to process **do**
  Add the k-mers *kmer_list*
end while

Main Code
for each read $r \in R_p$ **do**
  for each k-mer $k_r \in r$ **do**
    actor-send(k_r, owner_pe(k_r))
  end for
end for
Global Barrier
Merge all duplicate k-mers in *kmer_list* into (k-mer, count) tuples
Return *kmer_list*

Algorithm 2 k-mer counting - MPI

for each read $r \in R_p$ **do**
  for each k-mer $k_r \in r$ **do**
    Insert k_r in *kmers_per_proc*[*owner_pe*(k)]
    num_kmers $\leftarrow$ num_kmers + 1
  end for
  if num_kmers $\geq b$ **then**
    Find total sends and receive sizes of all processes using `MPI_Alltoall`
    Update *kmer_list* by transferring k-mers using `MPI_Alltoallv`
    num_kmers $\leftarrow$ 0
  end if
end for
Merge all duplicate k-mers in *kmer_list* into (k-mer, count) tuples
Return *kmer_list*

Strong Scaling

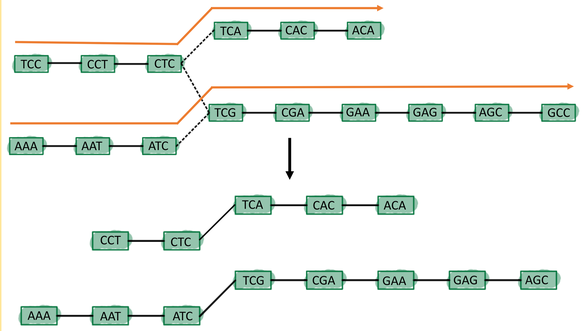


- Strong scaling performed on a simulated *E.coli* data up to 15 nodes of PACE cluster of Georgia Tech.
- The actor implementation is on average **1.6 times** faster than the MPI implementation, while also being a much simpler code for the programmer to write.

Kernel 2: de Bruijn Graph Traversal and Contig generation

Problem Statement: Given a de-Bruijn graph, find high-confidence, linear paths in that de-Bruijn graph, called contigs, which are used as building blocks for the final assembled genome.

Overview



Key (k-mer)	Value {count, [ACGT][ACGT], contig_id, contig_pos}
ATC	{134, [1000][0010], 00, 2}
TCG	{120, [1100][1000], 00, 3}
CTC	{ 78, [0100][1010], 26, 2}
TCA	{101, [0100][0100], 26, 3}
AAA	{200, [0000][0001], 00, 0}

Above: Logical representation of a de-Bruijn graph.

Left: How the same de-Bruijn graph is stored in memory as a distributed hash table

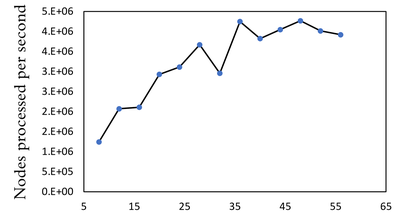
Algorithm

Algorithm 3 Contig generation in a de-Bruijn graph

$N_p \leftarrow$ set of nodes (key-value pairs) of the overall graph stored locally.
Every node $\in N_p$ asks it's neighbors to send their k-mer counts. `(remote GET)`
Every $\in N_p$ makes a priority order of its neighbors based on received k-mer counts.
while any node $X \in N_p$ has more than 2 edges **do**
  X sends pairing `REQUEST` to its most preferred neighbors. `(remote PUT)`
  X responds to its neighbors in `YES/NO/MAYBE` as `RESPONSE` to their request. `(remote PUT)`
  if `REQUEST` and `RESPONSE` matches **then**
    X deletes all other edges.
  else
    X moves on to the next neighbor in its priority order.
  end if
end while
Update the sub-contigs of a given contig to have the same *contig_id*. `(cyclic remote PUT)`
`Parallel sort` the sub-contigs based on their *contig_id*.
Combine locally present sub-contigs into the full contigs and add them in the `contig_list`.
Return: `contig_list`

- Our theoretical analysis shows that the algorithm converges in finite number of iterations of the **while** loop.
- Experimental evidence demonstrates the algorithm requires ≤ 4 iterations of the **while** loop in the worst-case. Formal proof of this claim is pending.

Strong Scaling



- Strong scaling performed on a simulated *E.coli* data up to 3 nodes of PACE cluster of Georgia Tech.
- The current implementation of the contig generation kernel is not fully optimized and load-balanced. Which is evident is poor scaling for more than 36 processors. We are current working on optimizing it.

Limitations of current approach

- Current version of the algorithm has more synchronization that required, and subsequent versions will reduce the dependencies on global synchronizations.
- We are working on proving stronger guarantees on the algorithm to make the whole kernel synchronization free.

Future Directions

- This work is a part of the genomics workflow of IARPA AGILE program that aims at designing reliable computer architectures for large-scale data-intensive applications.
- A natural progression of this work is to write the common communication patterns (remote PUT in our case) as **programmable hardware atomics** that'll allow the program to access the target memory slice directly without involving the message handler of the remote process's actor, making it faster and more scalable on our target hardware.
- We plan to incorporate **compiler optimizations** to make the code **locality aware** and minimize the number of actor messages sent across the system. From a theoretical point of view, this strategy will reduce the number of times the message buffer will fill up and send all it's messages, and hence improving the scalability and runtime.

Acknowledgement

This research is based upon work supported by the Office of the Director of National Intelligence (ODNI), Intelligence Advanced Research Projects Activity (IARPA), through the Advanced Graphical Intelligence Logical Computing Environment (AGILE) research program, under Army Research Office (ARO) contract number W911NF22C0083. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the ODNI, IARPA, or the U.S. Government.