Distributed-Memory Parallel Algorithms for Graph Traversal & Genome Assembly

Aydın Buluç
Berkeley Lab (LBNL)

Stony Brook University
December 3, 2014
Outline and Acknowledgments

1. Parallel De Bruijn Graph Construction and Traversal for de novo Genome Assembly, SC’14
   Joint work with Evangelos Georganas, Jarrod Chapman, Leonid Oliker, Daniel Rokhsar, and Katherine Yelick

   Joint work with Scott Beamer, Kamesh Madduri, Krste Asanović, and David Patterson

3. Standards for Graph Algorithm Primitives, HPEC’13
   Joint work with David Bader, Jeremy Kepner, Timothy Mattson, and John Gilbert

My work is funded by U.S. Department of Energy Office of Science
Graphs for Scientific Discovery

Matching in bipartite graphs: Permuting to heavy diagonal or block triangular form

Graph partitioning: *Dynamic load balancing* in parallel simulations

Problem size: as big as the sparse linear system to be solved or the simulation to be performed

Picture (left) credit: Sanders and Schulz
Graphs for Scientific Discovery

Matching in bipartite graphs: Permuting to heavy diagonal or block triangular form

The case for distributed memory

Graph partitioning: Dynamic load balancing in parallel simulations

Problem size: as big as the sparse linear system to be solved or the simulation to be performed

Picture (left) credit: Sanders and Schulz
Graph Theoretical analysis of Brain Connectivity

Whole genome assembly

A Read Layout
B Overlap Graph

C de Bruijn Graph

Vertices: reads
Vertices: k-mers

26 billion (8B of which are non-erroneous) unique k-mers (vertices) in the hexaploid wheat genome W7984 for k=51

Schatz et al. (2010) Perspective: Assembly of Large Genomes w/2nd-Gen Seq. Genome Res. (figure reference)
Large Graphs in Scientific Discoveries

Whole genome assembly

A Read Layout

B Overlap Graph

Vertices: reads

C de Bruijn Graph

Vertices: k-mers

The case for distributed memory

26 billion (8B of which are non-erroneous) unique k-mers (vertices) in the hexaploid wheat genome W7984 for k=51

Schatz et al. (2010) Perspective: Assembly of Large Genomes w/2nd-Gen Seq. Genome Res. (figure reference)
1. Three copies of the same novel.

2. Some text from the novel. All pages will be randomly cut into strips of characters. Random **typos (errors)** throughout each novel.

   For all men tragically great are made so through a certain morbidity... all mortal greatness is but disease.

3. A few strips of characters from one page.

   For all men tragically great are made so

   all men tragically g

   great are made so

4. All of the strips of characters from the 3 novels.

5. Every strip must be assembled as shown here to create a single copy of the novel.

   For all men tragically great are made so

   For a great are made so

   all men tragically g

   ally great

1. Three copies of the same DNA.

2. Some part of the DNA sequence. It will be read into strips. There are random **errors** throughout the sequence.

   ACCGTAGCAAAACCGGGTAGTCATACTACTACGTACTCATCT

3. The sequence is read into smaller pieces (**reads**). Can not read whole DNA sequence **in one go**.

   ACCGTAGCA AACC CGGT A TAGT CACT ACTACGTACTCATCT

4. All reads

   AACC CGGT A ACTACGTACT

5. Reconstruct original DNA sequence from the read set.

   ACCGTAGCAAAACC CGGT A TAGT CACT ACTACGTACTCATCT

   ACCGTAGCA GTAGT CACT ACTACGTACTCATCT

   AACC CGGT A CTACTACGTAC

   CGTACTCATCT
De Novo Genome Assembly is hard

- There is no genome reference!
  - In principle we want to reconstruct unknown genome sequence.

- Reads are significantly shorter than whole genome.
  - Reads consist of 20 to 30K bases
  - Genomes vary in length and complexity – up to 30G bases

- Reads include errors.

- Genomes have repetitive regions.
  - Repetitive regions increase genome complexity.
Genomes vary in size

- Switchgrass: 1.4 Giga-base pairs (Gbp)
- Maize: 2.4 Gbp
- Miscanthus: 2.5 Gbp
- Human genome: 3 Gbp
- Barley genome: 7 Gbp
- Wheat genome: 17 Gbp
- Pine genome: 20 Gbp
- Salamander: 20-30 Gbp
De novo genome assembly is expensive!

• Human genome (3Gbp) “de novo” assembled:
  – SGA assembler: 140 hours
  – Meraculous assembler: 75.6 hours

• Wheat genome (17 Gbp) “de novo” assembled (2014):
  – Meraculous assembler: 170 hours – required a machine with 512 GB RAM

• Pine genome (20 Gbp) “de novo” assembled (2014):
  – Masurca assembler: 3 months on a machine with 1 TB RAM
Basic Vocabulary for Genome Assembly

- **Read**: short fragment of DNA sequence that can be read by a DNA sequencing technology (cannot read whole genomes in one go)
- **k-mer**: a DNA sequence of bases with length k
- **De Bruijn graph of k-mers**: a graph with nodes as k-mers and edges as length k-1 overlaps between the suffix of the source node and the prefix of the destination node
- **Contig**: a contiguous sequence of DNA longer than a k-mer (non-branching subgraph of the de Bruijn graph)
- **Scaffold**: one or more contigs linked together by some unknown sequence
Input: Reads that may contain errors

1. Chop reads into k-mers, process k-mers to exclude errors

2. Construct & traverse de Bruijn graph of k-mers, generate contigs

3. Leverage read information to link contigs and generate scaffolds.

Genome Assembly a la Meraculous
1. Chop reads into overlapping k-mers. Pick k such that:
   • Eventually get unique k-mers
   • Most reads have multiple overlapping \textbf{error-free} k-mers

2. Each k-mer has forward and backward single-base extensions
   • Preprocess the k-mers
     - Keep only those with high-quality (unique) extensions (UU k-mers).

3. Store UU k-mers in a hash table
   • “key” is the k-mer
   • “value” is a two-letter code [ACGT][ACGT] that indicates the extensions
   • This is the UU graph (de Bruijn graph)

4. Traverse the graph of UU k-mers (de Bruijn graph)
   • Find linear chains of UU k-mers (de Bruijn graph)
1. Chop reads into overlapping k-mers. Pick k such that:
   - Eventually get unique k-mers
   - Most reads have multiple overlapping error-free k-mers

2. Each k-mer has forward and backward single-base extensions
   - Preprocess the k-mers
   - Keep only those with high-quality (unique) extensions (UU k-mers).

3. Store UU k-mers in a hash table
   - “key” is the k-mer
   - Value is a two-letter code [ACGT][ACGT] that indicates the extensions
   - This is the UU graph (de Bruijn graph)

4. Traverse the graph of UU k-mers (de Bruijn graph)
   - Find linear chains of UU k-mers (UU contigs)
1. Chop reads into overlapping k-mers. Pick k such that:
   • Eventually get unique k-mers
   • Most reads have multiple overlapping error-free k-mers

2. Each k-mer has forward and backward single-base extensions
   • Preprocess the k-mers
   • keep only those with high-quality (unique) extensions (UU k-mers).

3. Store UU k-mers in a hash table
   • “key” is the k-mer
   • value is a two-letter code [ACGT][ACGT] that indicates the extensions
   • This is the UU graph (de Bruijn graph)

4. Traverse the graph of UU k-mers (de Bruijn graph)
   • Find linear chains of UU k-mers (UU contigs)
1. Chop reads into overlapping k-mers. Pick k such that:
   - Eventually get unique k-mers
   - Most reads have multiple overlapping error-free k-mers

2. Each k-mer has forward and backward single-base extensions
   - Preprocess the k-mers
   - keep only those with high-quality (unique) extensions (UU k-mers).

3. Store UU k-mers in a hash table
   - “key” is the k-mer
   - value is a two-letter code [ACGT][ACGT] that indicates the extensions
   - This is the UU graph (de Bruijn graph)

4. Traverse the graph of UU k-mers (de Bruijn graph)
   - Find linear chains of UU k-mers (UU contigs)
Parallel k-mer analysis in a nutshell

- Binary SeqDB format for read input files
  - Exploits parallel HDF5 and effectively parallelizes the I/O

- Use of **Bloom filters** to eliminate erroneous k-mers and decrease memory footprint (savings ~7X)
  - Bloom filter: Probabilistic data structure for membership queries

- Use of Hyperloglog algorithm to estimate the Bloom filter size.
  - Hyperloglog is a cardinality estimation algorithm.
  - Bloom filters are much faster when we know their size a-priori.
  - Merging in parallel Hyperloglog counts requires “almost-zero” communication.
Why use a Bloom filter?

Bloom filter is a *probabilistic* data structure used for membership queries

- Given a bloom filter, we can ask: “Have we seen this k-mer before?”
- **No false negatives.**
- May have false positives (in practice 5% false positive rate)

k-mers with frequency =1 are guaranteed to be an error, and can safely be eliminated.
Parallel k-mer analysis: pass 1

Reads → Parse to k-mers

\( P_0 \) → Hyperloglog
\( P_1 \) → Hyperloglog
... → Hyperloglog
\( P_n \) → Hyperloglog

k-mer cardinality estimation

Parallel Reduction

Create local Bloom filters

G = Global cardinality estimation

Bloom Filter of size \((G/n)\)
Bloom Filter of size \((G/n)\)
Bloom Filter of size \((G/n)\)
Parallel k-mer analysis: pass 2

Reads

Parse to k-mers

Hash k-mers & find owners

All-to-all communication of k-mers

Store a k-mer in the local set only if it was seen before

The local set does the actual counting of the k-mers

$P_0$

$P_1$

\vdots

$P_n$
Parallel k-mer analysis: pass 3

Reads

Parse to k-mers

Find extensions of k-mers, hash them & find their owners

All-to-all communication of k-mers and extensions

Use a threshold & find the high quality extensions of k-mers

Received k-mers & extensions

Hash k-mers

Received k-mers & extensions

Local set

Local set

Keep track of the number of occurrences of each extension for each k-mer
In Meraculous, the de Bruijn graph is represented as a hash table. K-mers are both nodes in the graph & keys in the hash table. An edge in the graph connects two nodes that overlap in k-1 bases. The edges in the graph are put in the hash table by storing the extensions of the k-mers as their corresponding values.
Parallel De Bruijn Graph Construction

**Input:** k-mers and their high quality extensions

- ACCCA CT
- CTTAG CF
- AACCT TG
- CGCAT XA
- AGGCA AT
- GGTAG FF
- AAAAT TG
- CCCAT XX
- ...
- TTTCA GT
- TTTGC CA
- AACTT GG
- CTTTT CA

**Read k-mers & extensions**

- $P_0$
- $P_1$
- ...
- $P_n$

**Store k-mers & extensions**

- Distributed Hash table

**Fine-grained communication & fine-grained locking required**
Aggregating stores optimization

1 aggregate remote transfer
Full local buffer

Local buffer for $P_0$
Buffer local to $P_0$

Local buffer for $P_1$

Local buffer for $P_n$
Aggregating stores optimization

P₀ stores the k-mers & extensions in its local buckets in a lock-free & communication-free fashion
Goal:
- Traverse the de Bruijn graph and find UU contigs (chains of UU nodes), or alternatively
- find the connected components which consist of the UU contigs.

Main idea:
- Pick a seed
- Iteratively extend it by consecutive lookups in the distributed hash table
Assume *one* of the UU contigs to be assembled is:

CGTATTGCCAATGCAACGTATCATCGGCCAATCCGAT
Parallel De Bruijn Graph Traversal

Processor $P_i$ picks a random k-mer from the distributed hash table as seed:

```
CGTATTGCCAATGCAACGTATCATGGCCAATCCGAT
```

$P_i$ knows that forward extension is A

$P_i$ uses the last k-1 bases and the forward extension and forms: CAACGTATCA

$P_i$ does a lookup in the distributed hash table for CAACGTATCA

$P_i$ iterates this process until it reaches the “right” endpoint of the UU contig

$P_i$ also iterates this process backwards until it reaches the “left” endpoint of the UU contig
Multiple processors on the same UU contig

However, processors $P_i$, $P_j$ and $P_t$ might have picked initial seeds from the same UU contig

- Processors $P_i$, $P_j$ and $P_t$ have to collaborate and concatenate subcontigs in order to avoid redundant work.

- **Solution**: lightweight synchronization scheme based on a state machine
Parallel k-mer analysis

Human dataset
Cray XC30
(NERSC Edison)
Parallel contig generation

Human dataset
Cray XC30
(NERSC Edison)
Overall performance

Human dataset
Cray XC30
(NERSC Edison)
Improvement in End-to-End Assembly times

- **Human genome**
  - k-mer analysis + contig generation: 60 hours → 2 minutes

- **Wheat genome**
  - k-mer analysis + contig generation: 140 hours → 15 minutes

- We incorporated our work into the Meraculous assembler
  - Replaced the k-mer analysis and contig generation steps
  - Kept the original scaffolding step.

- The result of this work is an end-to-end improvement in Meraculous assembly times:
  - 6.6X for wheat (from 165.4 hours to 25 hours)
  - 5X for human (from 75.6 to 15.15 hours)
Parallel Graph Traversal

**Top500 Benchmark:**
Solve a large system of linear equations by Gaussian elimination

\[ P A = L \times U \]

**Graph500 Benchmark:**
Breadth-first search in a large power-law graph
Floating point vs. Graphs (Nov 2014)

33.8 Petaflops

\[ \mathbf{P} \mathbf{A} = \mathbf{L} \times \mathbf{U} \]

23.7 Terateps

33.8 Peta / 23.7 Tera is about 1400
Floating point vs. Graphs (Nov 2014)

33.8 Petaflops

\[
P \cdot A = L \times U
\]

23.7 Terateps

Nov 2014: 33.8 Peta / 23.7 Tera \( \sim 1,400 \)

Nov 2010: 2.5 Peta / 6.6 Giga \( \sim 380,000 \)

We are rapidly learning how to do graphs well!
Breadth-first search in the language of linear algebra

\[\begin{array}{c}
\text{from} \\
1 & 7 \\
\end{array}
\]

\[\begin{array}{c}
\text{to} \\
A^T \\
\end{array}
\]
Particular semiring operations:

**Multiply:** select2nd

**Add:** minimum

parents:

\[
\begin{array}{c|c|c|c}
1 & \text{from} & 7 \\
\hline
1 & \text{to} & 7 \\
\hline
\end{array}
\]

\[
\begin{align*}
A^T & \quad X \\
\rightarrow & \quad A^T X
\end{align*}
\]
Multiple traverses outgoing edges
Add chooses among incoming edges
Select vertex with minimum label as parent.
Result: Deterministic breadth-first search
ALGORITHM:

1. Find owners of the current frontier’s adjacency [computation]
2. Exchange adjacencies via all-to-all. [communication]
3. Update distances/parents for unvisited vertices. [computation]
ALGORITHM:
1. Gather vertices in *processor column* [{communication}]
2. Find owners of the current frontier’s adjacency [{computation}]
3. Exchange adjacencies in *processor row* [{communication}]
4. Update distances/parents for unvisited vertices. [{computation}]
Matrix/vector distributions, interleaved on each other.

Default distribution in Combinatorial BLAS.

Scalable with increasing number of processes

- 2D matrix layout wins over 1D with large core counts and with limited bandwidth/compute
- 2D vector layout sometimes important for load balance

Performance observations of level-synchronous BFS

When the frontier is at its peak, almost all edge examinations "fail" to claim a child.
Bottom-up BFS algorithm

Classical (top-down) algorithm is optimal in worst case, but \textit{pessimistic} for low-diameter graphs (previous slide)

\begin{itemize}
  \item \textbf{Direction-optimizing approach:}
    \begin{itemize}
      \item Switch from top-down to bottom-up search
      \item When the majority of the vertices are discovered.
    \end{itemize}
\end{itemize}

\textbf{Top-Down}

\begin{itemize}
  \item for all $v$ in frontier attempt to parent \textit{all} neighbors($v$)
\end{itemize}

\textbf{Bottom-Up}

\begin{itemize}
  \item for all $v$ in unvisited find \textit{any} parent (neighbor($v$) in frontier)
\end{itemize}
Bottom-up + 2D partitioning

- Adoption of the 2D algorithm created the *first quantum leap*
- The *second quantum leap* comes from the bottom-up search

- Can we just do bottom-up on 1D?
- Yes, if you have *in-network* fast frontier membership queries
  - IBM by-passed MPI to achieve this [Checconi & Petrini, IPDPS’14]
  - Unrealistic and counter-productive in general

- 2D partitioning reduces the required frontier segment by a factor of $p_c$ (typically $\sqrt{p}$), without fast in-network reductions
- **Challenge:** Inner loop is serialized
Solution: Temporally partition the work

- *Temporal Division* - a vertex is processed by *at most one processor* at a time
- *Systolic Rotation* - send completion information to next processor so it knows what to skip
Solution: Temporally partition the work

- *Temporal Division* - a vertex is processed by *at most one processor* at a time
- *Systolic Rotation* - send completion information to next processor so it knows what to skip
Solution: Temporally partition the work

- *Temporal Division* - a vertex is processed by \textit{at most one processor} at a time
- *Systolic Rotation* - send completion information to next processor so it knows what to skip
**Solution:** Temporally partition the work

- *Temporal Division* - a vertex is processed by *at most one processor* at a time

- *Systolic Rotation* - send completion information to next processor so it knows what to skip
Direction-optimizing BFS on distributed memory

- Kronecker (Graph500): 16 billion vertices and 256 billion edges.
- Implemented on top of Combinatorial BLAS (i.e. uses 2D decomposition)

Beamer, B., Asanović, and Patterson, "Distributed Memory Breadth-First Search Revisited: Enabling Bottom-Up Search", IPDPSW’13
Direction-optimizing BFS reduces communication

Communication volume reduction based on analytical model derived in the paper, assuming 4 bottom-up steps (typical for power-law graphs)

![Graph](image)
Direction-optimizing BFS saves energy

For a fixed-sized real input, direction-optimizing algorithm completes at the same time using $1/16^{th}$ of processors (and energy)
A moral of these two stories: One algorithm does not fit all

Low diameter graph (R-MAT) vs. Long skinny graph (genomics)

Genetic linkage map, courtesy Yan et al.
Graphs as middleware

Continuous physical modeling

Linear algebra

Computers

Discrete structure analysis

Graph theory

Computers
Graphs as middleware

By analogy to numerical scientific computing...

What should the Combinatorial BLAS look like?

Basic Linear Algebra Subroutines (BLAS):
Ops/Sec vs. Matrix Size

\[ C = A \cdot B \]

\[ y = A \cdot x \]

\[ \mu = x^T y \]
Many irregular applications contain coarse-grained parallelism that can be exploited by abstractions at the proper level.

<table>
<thead>
<tr>
<th>Traditional graph computations</th>
<th>Graphs in the language of linear algebra</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data driven, unpredictable communication.</td>
<td>Fixed communication patterns</td>
</tr>
<tr>
<td>Irregular and unstructured, poor locality of reference</td>
<td>Operations on matrix blocks exploit memory hierarchy</td>
</tr>
<tr>
<td>Fine grained data accesses, dominated by latency</td>
<td>Coarse grained parallelism, bandwidth limited</td>
</tr>
</tbody>
</table>
The Combinatorial BLAS implements these, and more, on arbitrary semirings, e.g. \((\times, +), (\text{and, or}), (+, \text{min})\)
# Examples of semirings in graph algorithms

<table>
<thead>
<tr>
<th>Semiring</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real field: $(R, +, \times)$</td>
<td>Classical numerical linear algebra</td>
</tr>
<tr>
<td>Boolean algebra: $({0, 1},</td>
<td>, &amp;)$</td>
</tr>
<tr>
<td>Tropical semiring: $(R \cup {\infty}, \min, +)$</td>
<td>Shortest paths</td>
</tr>
<tr>
<td>$(S, \text{select, select})$</td>
<td>Select subgraph, or contract nodes to form quotient graph</td>
</tr>
<tr>
<td>(edge/vertex attributes, vertex data aggregation, edge data processing)</td>
<td>Schema for user-specified computation at vertices and edges</td>
</tr>
<tr>
<td>$(R, \max, +)$</td>
<td>Graph matching &amp; network alignment</td>
</tr>
<tr>
<td>$(R, \min, \times)$</td>
<td>Maximal independent set</td>
</tr>
</tbody>
</table>

- **Shortened semiring notation:** $(\text{Set, Add, Multiply})$. Both identities omitted.
- **Add:** Traverses edges, **Multiply:** Combines edges/paths at a vertex
- **Neither add nor multiply needs to have an inverse.**
- **Both add and multiply are associative, multiply distributes over add**
Can we standardize a “Graph BLAS”?

No, it’s not reasonable to define a universal set of building blocks.

- Huge diversity in matching graph algorithms to hardware platforms.
- No consensus on data structures or linguistic primitives.
- Lots of graph algorithms remain to be discovered.
- Early standardization can inhibit innovation.

Yes, it *is* reasonable to define a common set of building blocks...

... for graphs as linear algebra.

- Representing graphs in the language of linear algebra is a mature field.
- Algorithms, high level interfaces, and implementations vary.
- But the core primitives are well established.
The Graph BLAS effort

Standards for Graph Algorithm Primitives

Tim Mattson (Intel Corporation), David Bader (Georgia Institute of Technology), Jon Berry (Sandia National Laboratory), Aydin Buluc (Lawrence Berkeley National Laboratory), Jack Dongarra (University of Tennessee), Christos Faloutsos (Carnegie Melon University), John Feo (Pacific Northwest National Laboratory), John Gilbert (University of California at Santa Barbara), Joseph Gonzalez (University of California at Berkeley), Bruce Hendrickson (Sandia National Laboratory), Jeremy Kepner (Massachusetts Institute of Technology), Charles Leiserson (Massachusetts Institute of Technology), Andrew Lumsdaine (Indiana University), David Padua (University of Illinois at Urbana-Champaign), Stephen Poole (Oak Ridge National Laboratory), Steve Reinhardt (Cray Corporation), Mike Stonebraker (Massachusetts Institute of Technology), Steve Wallach (Convey Corporation), Andrew Yoo (Lawrence Livermore National Laboratory)

Abstract-- It is our view that the state of the art in constructing a large collection of graph algorithms in terms of linear algebraic operations is mature enough to support the emergence of a standard set of primitive building blocks. This paper is a position paper defining the problem and announcing our intention to launch an open effort to define this standard.

- The Graph BLAS Forum: [http://istc-bigdata.org/GraphBlas/](http://istc-bigdata.org/GraphBlas/)
An extensible distributed-memory library offering a small but powerful set of linear algebraic operations specifically targeting graph analytics.

- Aimed at graph algorithm designers/programmers who are not expert in mapping algorithms to parallel hardware.
- Flexible templated C++ interface; 2D data decomposition
- Scalable performance from laptop to 100,000-processor HPC.
- Open source software (v1.4.0 released January, 2014)
Performance of Linear Algebraic Graph Algorithms

Combinatorial BLAS fastest among all tested graph processing frameworks on 3 out of 4 benchmarks in an independent study by Intel.

The linear algebra abstraction enables high performance, within 4X of native performance for PageRank and Collaborative filtering.

Satish, Nadathur, et al. "Navigating the Maze of Graph Analytics Frameworks using Massive Graph Datasets", in SIGMOD’14
Conclusions

• Graph processing has many applications in sciences
• Massive graphs require *distributed-memory parallelism*
• Best algorithm depends on the graph characteristics
• We covered graphs with skewed degree distribution and de Bruijn graphs of genome assembly (almost linear)

• Combinatorial BLAS was an good proof of concept.
• A more elegant, simpler, complete system to emerge that is developed *concurrently with the Graph BLAS standard*
• Your input is solicited. If interested, join the mailing list here: [http://istc-bigdata.org/GraphBlas/](http://istc-bigdata.org/GraphBlas/) or email me
Thank you!