High-Performance Graph Analytics in Shared Memory

Hans Vandierendonck, Jiawen Sun
Queen’s University Belfast
20 July 2018
WHAT ARE GRAPH ANALYTICS

Graphs represent interactions between people or things.

Graph analytics are algorithms that extract information from a graph.

Graphs tend to grow large, and often tend to exhibit a power-law degree distribution:
- “6 degrees of separation”
WHY SHARED MEMORY?

Because of properties of the workload

- Little computation, mostly communication/synchronisation

- Data sets not so large, e.g., Twitter’s follower graph fits in memory of a single server [Sharma PVDLB’16]

- Future memory technologies will increase capacity: High-Bandwidth Memory/die-stacking, storage-class memory

- Large-scale shared-memory systems implement a non-uniform memory access (NUMA) model
WHAT IS NUMA?

Each CPU socket is connected to local DRAM memory.

Inter-node links provide access to “remote” DRAM memory.

Local links have higher bandwidth and lower latency than inter-node links.

Difference is more pronounced for stores than for loads.

In a program optimised for NUMA, CPU cores primarily access local DRAM.
GOAL

How to map graph analytics over immutable graphs onto a NUMA architecture while minimising execution time?
AGENDA

• Context and Goal
• Preliminaries
• Graph Algorithms
• Graph Analytics Frameworks
• Elements of High-Performance Graph Analytics
• NUMA-awareness
• Graph partitioning
• Load balance
• Conclusion and outlook
PRELIMINARIES
PRELIMINARIES

Graph $G=(V,E)$ where
$V$: set of vertex labels
$E \subseteq V \times V$: set of pairs of vertices

Frontier $F$ is a set of active vertices with $F \subseteq V$
in-degree: #incoming edges
out-degree: #outgoing edges
PRELIMINARIES

Directed graph: edges have a direction (source, destination)

Undirected graph: edges have no direction

if \((u,v) \in E\) then \((v,u) \in E\)
Undirected graphs are commonly represented such that every edge occurs twice.
GRAPH ALGORITHMS
**LABEL PROPAGATION**

- Strongly connected components
- Initial label assignment of "old" label, copied to "new" label
- Update rule: for \((u,v)\) in \(E\):
  \[\text{new}[v] = \min(\text{new}[v], \text{old}[u])\]
- Copy "new" to "old" and repeat update phase until no more changes made
LABEL PROPAGATION

- Strongly connected components
- Initial label assignment of "old" label, copied to "new" label
- Update rule: for \((u,v)\) in E:
  - \(new[v] = \min(new[v], old[u])\)
- Copy "new" to "old" and repeat update rule on all edges until no more changes made
Further propagating the labels “4” and “5” held by vertices 4 and 5 incurs no changes in the labels of other vertices.
Round 0 of propagating labels has finished
Copy “new” to “old” and go again…
After round 1, label “0” has propagated to more vertices. We need to do one more round to ensure no further changes occur.
Let's return to the state at the end of round 0. If in any round the label did not change, then there is no point in trying to propagate the label again.
LABEL PROPAGATION WITH FRONTIER

Let’s return to the state at the end of round 0
Vertices 2, 4, 5 have changes in their label \((\text{old}[v] \neq \text{new}[v])\)
Frontier = \{2, 4, 5\}
Only vertices 2, 4 and 5 are visited in round 1
Let's return to the state at the end of round 0.

Vertices 2, 4, 5 have changes in their label ($\text{old}[v] \neq \text{new}[v]$).

Frontier = \{2, 4, 5\}

Only vertices 2, 4 and 5 are visited in round 1.

In round 2:

Frontier = \{1, 2, 4\}
Let’s return to the state at the end of round 0

Vertices 2, 4, 5 have changes in their label ($\text{old}[v]! = \text{new}[v]$)

Frontier = \{2, 4, 5\}

Only vertices 2, 4 and 5 are visited in round 1

In round 2:

Frontier = \{1, 2, 4\}

Nothing changes
GRAPH ANALYTICS FRAMEWORKS
LIGRA

size(U : frontier) : N
  returns |U|

EdgeMap(G : graph,
  U : frontier,
  F : (vertex × vertex) → bool,
  C : vertex → bool) : frontier

VertexMap(U : frontier,
  F : vertex → bool) : frontier

[Shun PPoPP’13]
Assume graph G=(V,E)

**EdgeMap** applies an operation F to each edge \((u,v) \in E\) where \(u \in U\) and \(C(v) = \text{true}\). It returns a frontier that contains all v where any call to F(u,v) returned true

**VertexMap** applies an operation F to each vertex \(v \in U\) and returns a frontier that contains v iff \(v \in U\) and F(v) = true

In both cases, F may have side effects, e.g., updating properties for the vertices
writeMin is an atomic “fetch_and_min” operation
Like compare-and-set, returns true if destination is successfully modified

Source: Shun PPoPP’13
interface GASVertexProgram(u) {
  // Run on gather_nbrs(u)
  gather(D_u, D_{(u,v)}, D_v) → Accum
  sum(Accum left, Accum right) → Accum
  apply(D_u, Accum) → D_u^{new}
  // Run on scatter_nbrs(u)
  scatter(D_u^{new}, D_{(u,v)}, D_v) → (D_{(u,v)}^{new}, Accum)
}

Figure 2: All PowerGraph programs must implement the stateless gather, sum, apply, and scatter functions.

Algorithm 1: Vertex-Program Execution Semantics

Input: Center vertex u
if cached accumulator a_u is empty then
  foreach neighbor v in gather_nbrs(u) do
    a_u ← sum(a_u, gather(D_u, D_{(u,v)}, D_v))
  end
end
D_u ← apply(D_u, a_u)
foreach neighbor v scatter_nbrs(u) do
  (D_{(u,v)}, Delta_a) ← scatter(D_u, D_{(u,v)}, D_v)
  if a_v and Delta_a are not Empty then a_v ← sum(a_v, Delta_a)
  else a_v ← Empty
end

[Gonzalez OSDI’12]
Similar concepts, presented differently
Vertices ‘activated’ by explicit call as opposed to recording frontier
Needs to maintain state on vertices and on edges

Distributed framework
PEGASUS

[Kang ICDM’09]
Similar concepts, presented differently
Uses connection between graphs and their adjacency matrix
Generalized matrix-vector multiplication captures ‘accumulation’ concept
Essentially says that graph algorithms may be represented as semi-rings


frontier \( F := \ldots; \)
frontier \( \text{newF} := \{ \} \);
for edge \((u,v) \in E\) do
  if \( u \in F \) then
    if \( C(v) \) and \( \text{op}(u,v) \) then
      \( \text{newF} = \text{newF} \cup \{ v \}; \)
    fi
  fi
od

- \( \text{op} \) implements the update of vertex properties
- \( \text{op}, C \) are algorithm-specific
- \( \text{op} \) returns true if destination should be considered in the next round
- \( \text{op}(u,v) \) is usually of the form
  \( \text{new}[v] = \text{new}[v] \oplus \text{old}[v] \)
  where \( \oplus \) is a commutative and associative binary operation (reduction)
- \( C(v) \) checks convergence
CONVERGENCE

frontier F := ...;
frontier newF := { };
for edge (u,v) ∈ E do
  if u ∈ F then
    if C(v) and op(u,v) then
      newF = newF ∪ { v };
    fi
  fi
od

Shun PPoPP’13:
“The function C is useful in algorithms where a value associated with a vertex only needs to be updated once (i.e. breadth-first search).”

The paper also checks convergence for betweenness centrality

Real usefulness depends on how the graph is traversed
GRAPH DATA STRUCTURES

• Compressed Sparse Rows (CSR)
  - List outgoing edges for each vertex
  - “Forward” traversal
  - “Top-down” traversal
  - “Push”
  - “Vertex-centric”

Frontier: CSR allows to skip edges for inactive vertices ($u \notin F$)

• Compressed Sparse Columns (CSC)
  - List incoming edges for each vertex
  - “Backward” traversal
  - “Bottom-up” traversal
  - “Pull”
  - “Vertex-centric”

Pruning: CSC allows to skip edges for pruned vertices ($C(v) = false$)
CSR-BASED EDGEMAP

frontier \( F \) := \cdots;
frontier \( \text{newF} \) := \{ \};

for vertex \( u \in V \) do
    if \( u \in F \) then
        for vertex \( v \in \text{out}(u) \) do
            if \( C(v) \) and \( \text{op}(u,v) \) then
                \( \text{newF} = \text{newF} \cup \{ v \} \);
        fi od
    fi od

\[ \text{• Checking frontier is compulsory: \( \text{op}(u,v) \) may be called only if } u \in F \]
\[ \text{• Pruning is not compulsory: may call } \text{op}(u,v) \text{ if } C(v) = \text{false} \]
\[ \text{• As \( \text{op}(u,v) \) is only a hand-full of instructions, there is little benefit in using } C(v) \text{ in CSR} \]
CSC-BASED EDGEMAP

frontier $F := \ldots$;
frontier $newF := \{ \}$;
for vertex $v \in V$ do
  if $C(v)$ then
    for vertex $u \in \text{in}(v)$ do
      if $u \in F$ then
        if $op(u,v)$ then
          if $C(v)$ then break; fi
          newF = newF $\cup \{ v \}$;
      fi fi fi od od
  fi fi fi od

• Pruning is highly effective in CSC
• Allows to early terminate visiting the in-edges of $u$
• Or skip in-edges altogether
EVOLUTION OF FRONTIER SIZE

Algorithms exhibit one of three primary patterns:

- flat
- shrink
- grow then shrink

Source: Shun PPoPP’13
“Sparse” frontiers
• Few bits set
• Queue of active vertex IDs

“Dense” frontiers
• Many bits set
• Bitmap or array of booleans

Dynamically switch as frontier size changes

[Hong et al, PACT’11]
CSR-BASED EDGEMAP
WITH SPARSE FRONTIER

frontier \( F \) := \ldots; \quad \text{// queue}
frontier \( \text{newF} \) := \{ \}; \quad \text{// queue}
for vertex \( u \in F \) do
  for vertex \( v \in \text{out}(u) \) do
    if \( E(v) \) and \( op(u,v) \) then
      \( \text{newF} = \text{newF} \cup \{ v \} \); \quad \text{// append}
  fi
od

• Reminder: dense frontiers:
  for vertex \( u \in V \) do
    if \( u \in F \) then
      \ldots
  fi
od

• Iteration over \( F \) is efficient when stored as a queue
• When \( F \) is stored as a queue, only CSR is efficient
• new frontier may contain duplicates!
BREADTH-FIRST SEARCH

Starting from a root vertex, identify a shortest path to all other vertices.

Construct a spanning tree.

-1 means parent unknown.

In this case we start from vertex 2.

Requires a frontier: all vertices that received a parent in the previous round.
BREADTH-FIRST SEARCH

"Top-down" traversal, “push”
Focus on out-edges

Old frontier:  1  5
New frontier:  4  5
BREADTH-FIRST SEARCH

"Bottom-up" traversal, “pull”
Focus on in-edges
Vertex complete as soon as parent updated

Old frontier: 1 5
New frontier:
IMPACT OF CONVERGENCE
[Beamer, SC12]

Fig. 3. Breakdown of edges in the frontier for a sample search on kron27 (Kronecker generated 128M vertices with 2B undirected edges) on the 16-core system.

Claimed child: parent[v] updated from -1 to a vertex ID
Failed child: parent[v] updated in same round by parent
Peer: parent[v] updated in same round by sibling
Valid parent: v was encountered in a round prior to u

Bottom-up/pull is faster in the middle rounds
In those rounds, many vertices are active
DIRECTION-OPTIMISATION

[Beamer SC’12] [Shun SPAA’13]

d = (#active vertices +
#active edges) / #edges

if d > 5% then
    # dense frontier
    if algorithm prefers
        forward then
            traverse CSR
        else
            traverse CSC
    endif
else # d <= 5%
    # sparse frontier
    traverse CSR
endif

Programmer’s choice
Little is known to guide this
We will shed some light on this
... work in progress

Requires storage of both
CSC and CSR
NUMA-AWARENESS
Remote access has higher latency, lower bandwidth than local access.

Stores are more affected than loads.

Designed a scheme using graph partitioning [Kyrola OSDI’12] and privatization of vertex properties.

We will discuss how their ideas were rehashed in GraphGrind.

[Zhang PPoPP’15]
Goal: map code and data to NUMA nodes

One type of arrays
• Properties (per vertex)

Two types of loops
• Loops over edges
• Loops over vertices

Two types of iteration
• Sparse frontier
• Dense frontier
RECAP: RACE CONDITIONS

A pair of load and store instructions, at least one of which is a store, that access the same memory location

In a concurrent program with race conditions, the outcome of the program may differ depending on the relative execution speed of threads

Typical solutions:
• mutual exclusion
• atomic memory operations
• owner-computes
Decomposition based on partitioning input/output data is referred to as the owner computes rule.

Each partition performs all the computations involving data that it owns.

- **Input data decomposition**: A task performs all the computations that can be done using these input data.
- **Output data decomposition**: A task computes all the results in the partition assigned to it.
THE “MAP” IN EDGEMAP AND VERTEXMAP

**Edgemap:**
- Iteration space: \((u,v) \in E\)
  where \(u, v \in V\)
- Dependencies are determined by graph topology

**Vertexmap:**
- Iteration space: \(v \in V\)
How to split edgemap over NUMA nodes?

- code
- data

Observation

Remote stores are more expensive than remote loads [Zhang PPoPP ‘15]

Need to co-locate code with the updated data

Edges are processed by CPUs attached to the NUMA node that holds the destination’s property
NUMA-AWARE LAYOUT FOR EDGEMAP

Goal
Determine cuts of { code, data } such that performance is maximised.

How?
Partition graph such that each partition (NUMA node) has an equal:

1. \#edges, \#cuts [PowerGraph OSDI’12] … breaks locality
2. \#sources [X-stream SOSP’13] … race conditions
3. \#edges [Polymer PPoPP ‘15]
4. (\#destinations + \#edges) [Gemini OSDI’16]
It depends! [GraphGrind ICS’17]

“Vertex-oriented” algorithms
- Best performance with equal #destinations
- Frontier density mostly below 50%
- BFS, Betweenness Centrality, Bellman-Ford

“Edge-oriented” algorithms
- Best performance with equal #edges
- Frontier density mostly close to 100%
- PageRank, SpMV, Belief Prop., PageRankDelta
NUMA-AWARE LAYOUT FOR VERTEXMAP

“Vertex-oriented” algorithms
• Trivial

“Edge-oriented” algorithms
• Need to choose between balancing compute and minimising traffic across NUMA nodes
  • Better to balance compute and incur additional inter-node traffic [GraphGrind ICS’17]
• Consequently, data is partitioned differently from compute
NUMA-AWARENESS CHOICES

- Baseline is CPU balance and memory imbalance
  - Implies remote accesses during vertex map
- CPU imbalance
  - No remote accesses during vertex map
- Memory balance
  - No remote accesses during vertex map
  - Many remote accesses during edge map
CAN WE MEET BOTH REQUIREMENTS?

Have our cake and eat it too!

VEBO
PERFORMANCE EVALUATION OF NUMA-AWARENESS

Combination of optimisations [Sun ICS’17]
- Pruned CSC/CSR representation
- Tune partitioning to edge/vertex algorithms
- NUMA-aware layout of vertex arrays
- CSC traversal: “caching” intermediate values to minimise load/stores
- Full frontier: specialised version of code that omits frontier check
- Sparse CSR traversal: no partitioning applied

Twitter

<table>
<thead>
<tr>
<th></th>
<th>Ligra</th>
<th>Polymer</th>
<th>GraphGrind (NUMA)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CC</td>
<td>80</td>
<td>20</td>
<td>0</td>
</tr>
<tr>
<td>BFS</td>
<td>6</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>BC</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>BF</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Friendster

<table>
<thead>
<tr>
<th></th>
<th>Ligra</th>
<th>Polymer</th>
<th>GraphGrind (NUMA)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CC</td>
<td>160</td>
<td>40</td>
<td>0</td>
</tr>
<tr>
<td>BFS</td>
<td>10</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>BC</td>
<td>5</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>BF</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

4-socket 2.6GHz Intel Xeon E7-4860 v2, 48 threads, 256 GiB
GRAPH PARTITIONING
MEMORY LOCALITY

- Reuse Distance Distribution

- Misses Per Kilo/Instruction (MPKI) – Twitter graph
When partitioning the edge set, a vertex may appear in multiple partitions.

Replication factor = 
\[
\frac{\text{#repeated vertices}}{\text{#unique vertices}}
\]

Replication factor tends to \(|E|/|V|\) as number of partitions grows.

Replication implies space and runtime overhead.
IMPLICATIONS OF VERTEX REPLICATION

A different view on the same effect:

If we partition the edge set P-way, then a vertex with degree $d<P$ has zero edges in at least $P-d$ partitions. It has some edges in at most $d$ partitions.

Partitions of a sparse graph are hyper-sparse
GRAPH DATA STRUCTURES

- Compressed Sparse Rows (CSR)
  - List outgoing edges for each vertex
  - “Forward” traversal (push)
  - “vertex-centric”

- Coordinate list (COO)
  - A list of edges
  - “edge-centric”

- Compressed Sparse Columns (CSC)
  - List incoming edges for each vertex
  - “Backward” traversal (pull)
  - “vertex-centric”

No performance boost from frontiers or pruning for COO
**IMPLICATIONS OF VERTEX REPLICATION**

CSC and CSR are not scalable formats
- Increased storage
- Increased execution time to traverse graph

Pruned CSC/CSR
- A.k.a. Compressed Compressed Sparse Rows/Columns
- Omits zero-degree vertices

COO is scalable to any number of partitions
- But inefficient for sparse frontiers
COO ADVANTAGE: SPACE FILLING CURVES

Space filling curves define a traversal order through a space that tends to minimise memory locality.

Map nD order onto 1D order
- Hilbert curve, Morton order (Z-order), and many others

COO allows edges to be stored in any order:
- CSR order
- CSC order
- Space filling curves

Edges are points in a 2D space:
For $u,v$ in $0,...,|V|-1$:
$$(u,v) = 1 \text{ if } (u,v) \in E,$$
$$(u,v) = 0 \text{ otherwise}$$

Edgemap: visit all $(u,v)$ in $E$
GRAPH PARTITIONING BENEFITS

- Betweenness Centrality, Twitter
- PageRank, Twitter 10 iterations

1 partition – multiple threads
1 partition – 1 thread
partitioned CSR out of memory
skips irrelevant edges
DIRECTION-OPTIMIZATION

• Ligra [Shun PPoPP’13]

\[ d = \frac{\text{#active vertices} + \text{#active edges}}{\text{#edges}} \]

```plaintext
if \ d > 5\% \ then
  \# \ dense \ frontier
  if \ \text{algorithm prefers} \ \text{forward} \ then
    \text{traverse} \ CSR
  else
    \text{traverse} \ CSC
  endif
else \ # \ d \ \leq \ 5\%
  \# \ sparse \ frontier
  \text{traverse} \ CSR
endif
```

• GraphGrind [Sun ICPP’17]

• 3-way heuristic

\[ d = \frac{\text{#active vertices} + \text{#active edges}}{\text{#edges}} \]

```plaintext
if \ d > 50\% \ then
  \# \ dense \ frontier
  \text{traverse} \ \text{partitioned COO}
else if \ d > 5\% \ then
  \# \ medium-dense \ case
  \# \ dense \ frontier
  \text{traverse} \ CSC
else \ # \ d \ \leq \ 5\%
  \# \ sparse \ frontier
  \text{traverse} \ CSR
endif
```
LOAD BALANCE
Revisting edge balance:
Two partitions with 3 edges
Which partition is processed faster?

LOAD BALANCE

Execution time/partition highly dependent on the degree of vertices

Reorder vertices
• in order of decreasing in-degree
• using list scheduling

VEBO: Vertex and Edge Balanced Partitioning

Friendster
PageRank
VEBO BENEFITS

Partitions are processed faster as a side-effect of reordering
Remote cache misses are traded for local misses
PERFORMANCE

L: Ligra, P: Polymer, Adaptive: GraphGrind with 3-way "direction-optimization"
PERFORMANCE

Comparing Ligra, Polymer (NUMA-aware), and 3 versions of GraphGrind

Twitter graph

4-socket 2.6GHz Intel Xeon E7-4860 v2, 48 threads, 256 GiB

Similar results hold for other graphs

VEBO relabels vertex IDs to achieve load balance
CONCLUSION AND OUTLOOK
CONCLUSION AND OUTLOOK

Scale-free properties of graphs make it hard to achieve high-performance
Code itself is short – devil is in the detail
Graph partitioning crucial: NUMA-locality; avoiding atomics; improving memory locality
Some open questions:
• What are the limits on memory efficiency?
• What is the cause of performance difference between CSR/CSC/COO?
• Do the principles behind GraphGrind apply to distributed memory systems?
• How well does the programming model capture graph algorithms?
REFERENCES

• Beamer, S., Asanovic, K. and Patterson, D. Direction-Optimizing Breadth-First Search. SC’12
• Besta, M., Podstawski, M., Groner, L., Solomonik, E. and Hoeﬂer, T. To Push or To Pull: On Reducing Communication and Synchronization in Graph Computations, IPDPS’17
• Gonzalez, J. E., Low, Y., Gu, H., Bickson, D. and Guestrin, C. PowerGraph: distributed graph-parallel computation on natural graphs, OSDI’12
• Hong, S., Tayo, O. and Olukotun, K. Efﬁcient parallel graph exploration on multi-core CPU and GPU. PACT’11
• Kang, U., Tsourakakis, C. E., and Faloutsos. PEGASUS: A Peta-Scale Graph Mining System - Implementation and Observations ICDM’09
• Kyrola, A., Blelloch, G. E. and Guestrin, C. GraphChi: Large-Scale Graph Computation on Just a PC. OSDI’12
• Roy, A. and Mihailovic, I. and Zwaenepoel, W. X-stream: Edge-centric graph processing using streaming partitions, SOSP’13
• Sharma, A., Jiang, J., Bommanna var, P., Larson, B. and Lin, J. GraphJet: Real-Time Content Recommendations at Twitter. PVLDB’16
• Shun, J. and Blelloch, G. E. Ligra: A Lightweight Graph Processing Framework for Shared Memory. PPoPP’13
• Sun, J., Vandierendonck, H. and Nikolopoulos, D. S. GraphGrind: addressing load imbalance of graph partitioning. ICS’17
• Sun, J., Vandierendonck, H. and Nikolopoulos, D. S. Accelerating Graph Analytics by Utilising the Memory Locality of Graph Partitioning. ICPP’17
• Zhang, K., Chen, R. and Chen, H. NUMA-Aware Graph-structured Analysis. PPoPP’15