Locality II

Sherry Yang

Outline

- NUMA Architecture
- NUMA in Graph Processing
- Graph Partitioning
- Data Placement
- Thread Placement
- Evaluation

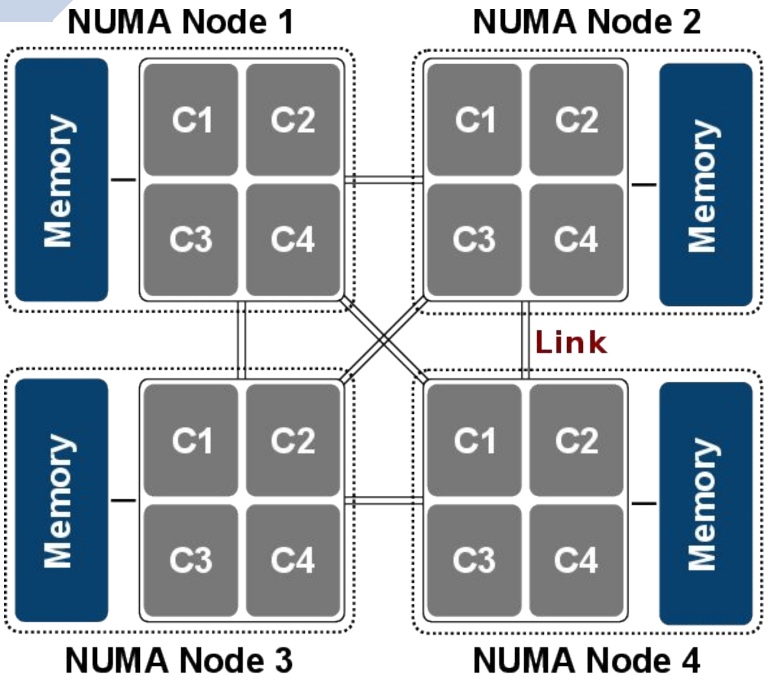
1

NUMA Architecture

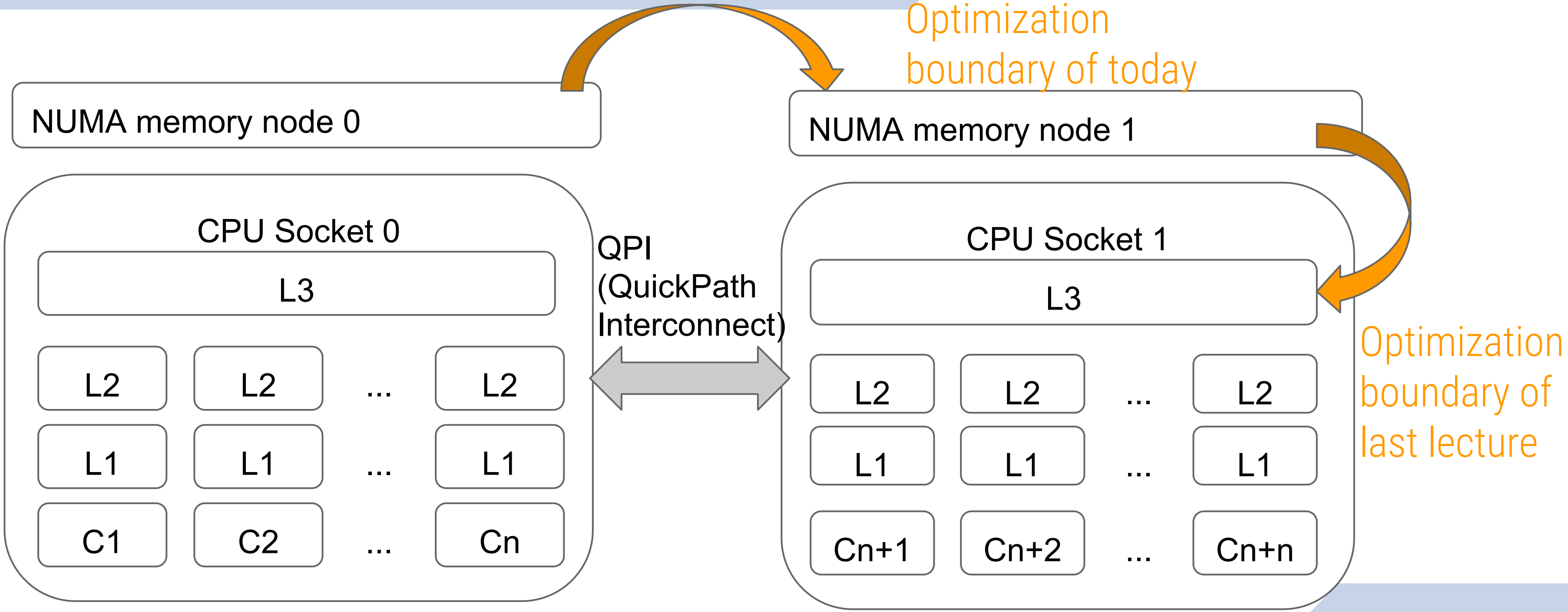
1.1 Definition

Non-uniform memory access (NUMA) architecture

- A shared memory abstraction
- Underlying memory is divided across sockets
- Memory access time is dependent on the memory location relative to the processor



1.2 The NUMA Memory Hierarchy



1.3 NUMA Characteristics

	Local	1 hop Remote
Load latency cycles	117	271
Store latency cycles	108	304
Seq BW (MB/s)	3207	2455
Rand BW (MB/s)	720	348

Polymer's microbenchmark on 80-core 8-socket Xeon

Local latency (ns)	60	100

Intel Core i7 Xeon 5500 Series Specification

Cross-socket communication is 2 to 7.5 times more expensive than intra-socket communication.

Everything You Always Wanted to Know About Synchronization but Were Afraid to Ask, SOSP '13

Remote access is a bottleneck in both latency and bandwidth

2

NUMA in Graph Processing

2.1 PageRank

```
for node : graph.vertices:
    for ngh : node.in_ngh:
        new_ranks[node] += ranks[ngh] / out_degree[ngh]
```

PageRank especially has a high QPI traffic since runtime is dominated by |E|

Potential cross-socket sequential access

Potential cross-socket random access

2.2 Connected Components

Label propagation algorithm

```
for dst : graph.vertices:
    for src : node.in_ngh:
        if IDs[dst] > IDs[src]:
            IDs[dst]= IDs[src]
```

Potential cross-socket sequential access

Potential cross-socket random access

2.3 Pull-based BFS

```
for dst : graph.vertices:
    if parent[dst] < 0:
        for src : node.in_ngh:
            if frontier[src]:
                parent[dst] = src
                next_fronter[dst] = 1
                break
```

Push-based version also has potential cross-socket random accesses

Potential cross-socket sequential access

Potential cross-socket random access

2.4 NUMA Access Summary

- Graph topology data
  - Vertex array and edge array in CSR/CSC format
- Application data
  - Ranks (PageRank), IDs (CC), Parent (BFS)
- Runtime states
  - Frontier, next_frontier

2.5 NUMA-aware Graph Processing

- Graph partitioning: preprocess the original graph into subgraphs with low duplication factor and good load balance
- Data placement: subgraphs, application data, and runtime states are allocated to specific memory nodes
- Thread placement: each CPU socket only process the subgraph belonging to the corresponding NUMA node. Intermediate results are stored in socket-local buffers
- Merge: data in socket-local buffers are merged and redistributed

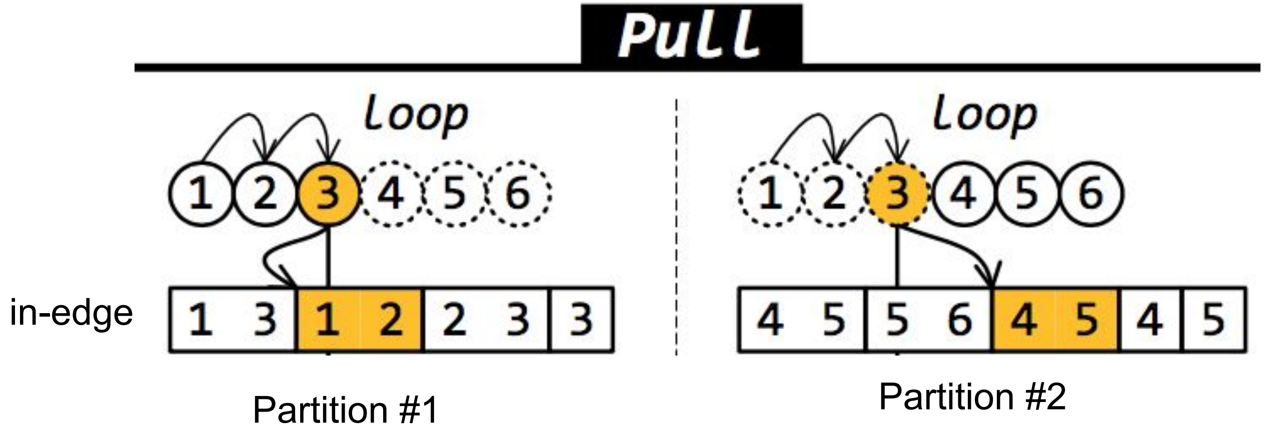
3

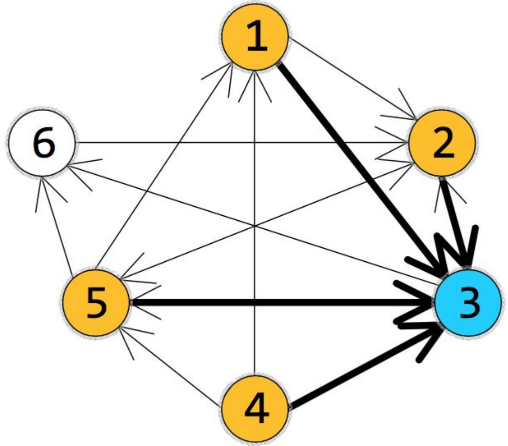
Graph Partitioning

3.1 Graph Partitioning: Polymer

NUMA-aware graph-structured analytics, PPoPP '15

- Partitions vertices into |V| / #sockets
- Assigns out-edge and in-edge by target and source
- Replicate "agent" vertex (e.g. vertex 3 in partition #2) to avoid remote access





3.1 Graph Partitioning: Polymer

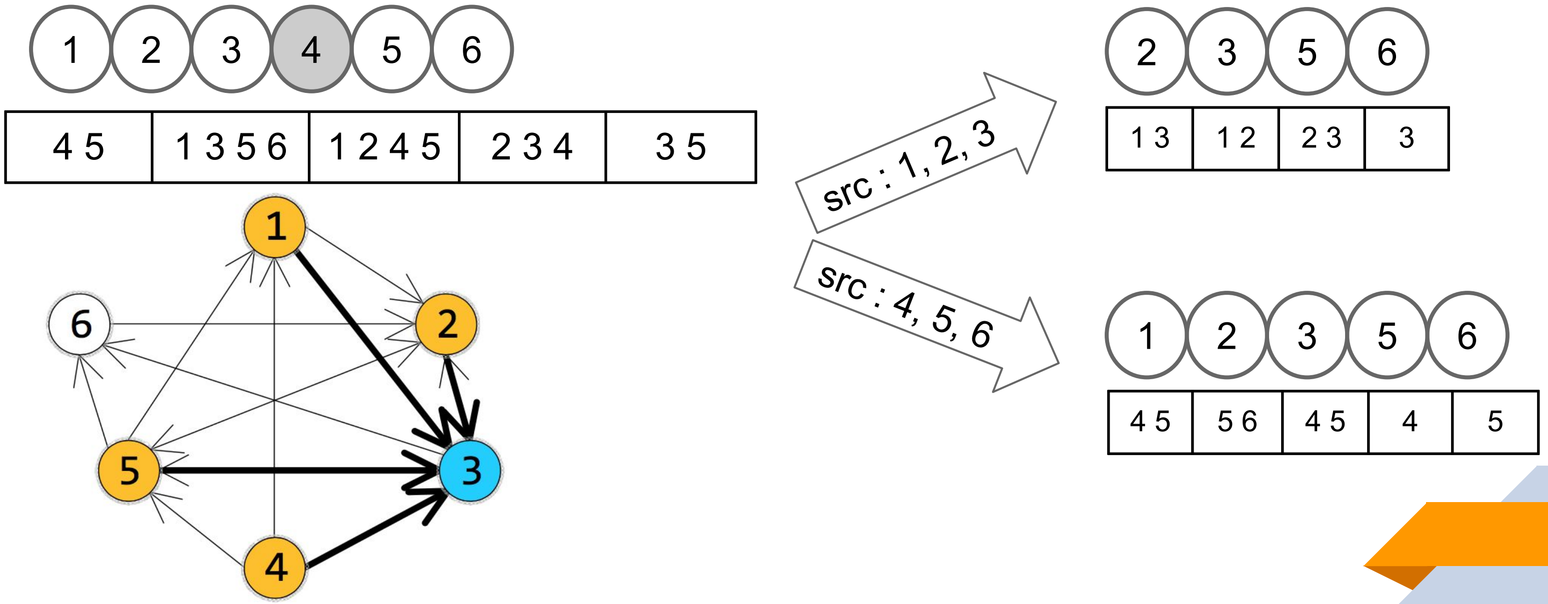
Optimization for load balancing:

- Vertex-based partitioning does not work well for skewed graphs
- Many graph analytic algorithms perform an amount of work that is proportional to the number of edges
- Edge-oriented load balancing: instead of evenly dividing vertices into |V| / #sockets partitions, uses uneven sets of V1, V2... Vs to balance edges (even longer preprocessing time)

3.2 Graph Partitioning: Gemini

Gemini: A Computation-Centric Distributed Graph Processing System, OSDI '16

- partitions the graph into #sockets subgraphs using chunk-based partitioning



3.2 Graph Partitioning: Gemini

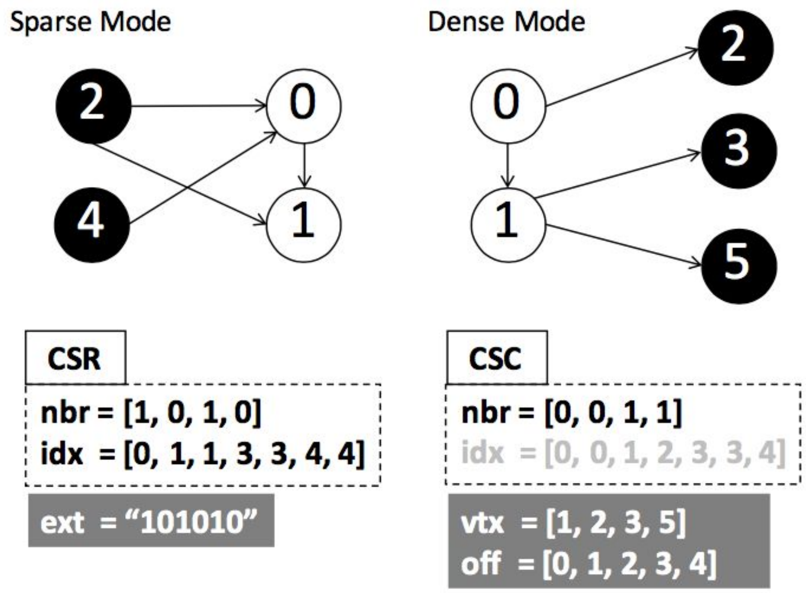
Chunk-based partitioning:

- Same as the CSR segmenting introduced during last presentation
- Different from Polymer: does not loop over all vertices in each partition (*Bitmap Assisted Compressed Sparse Row* and *Doubly Compressed Sparse Column* optimization)
- Eliminates 0 in-degree vertices: vertex 4 and vertex 1 in partition 1
- Retains the natural locality in input vertex arrays
- Can adjust the number of segment (segment range) to fit subgraphs into last level cache (the cache paper from last lecture)

3.2 Graph Partitioning: Gemini

Optimization for memory overhead:

- Dense mode: 4 edges, 7 entries in idx array (O(|V|)
- Bitmap Assisted Compressed Sparse Row: bitmap (ext) to mark vertices with outgoing edges in the partition
- Doubly Compressed Sparse Column: Stores only vertices with incoming edges (vtx) and their offsets (off)
- Offset array now is O($|V_i'|$)



3.2 Graph Partitioning: Gemini

Optimization for load balancing:

- Vertex AND edge aware
- Uses $a * |V_i| + |E_i^D|$ to choose the range of a partition

Balanced By	Runtime (s)	$\lvert V_i \rvert$	$\lvert E_i^D \rvert$
$\lvert V_i \rvert$	5.51	5.21M	957M
$\lvert E_i^D \rvert$	3.95	18.1M	183M
$\alpha \cdot \lvert V_i \rvert + \lvert E_i^D \rvert$	3.02	0.926M	423M

Table 8: Impact of locality-aware chunking (*PR* on *twitter-2010*)

8 computing nodes

Twitter-2010:

41.7M vertices

1.468B edges

$\alpha = 8 \cdot (p - 1)$

3.2 Graph Partitioning: Gemini

Gemini is one of the few frameworks that measured preprocessing time

- Long preprocessing time (many times longer than actual processing time)

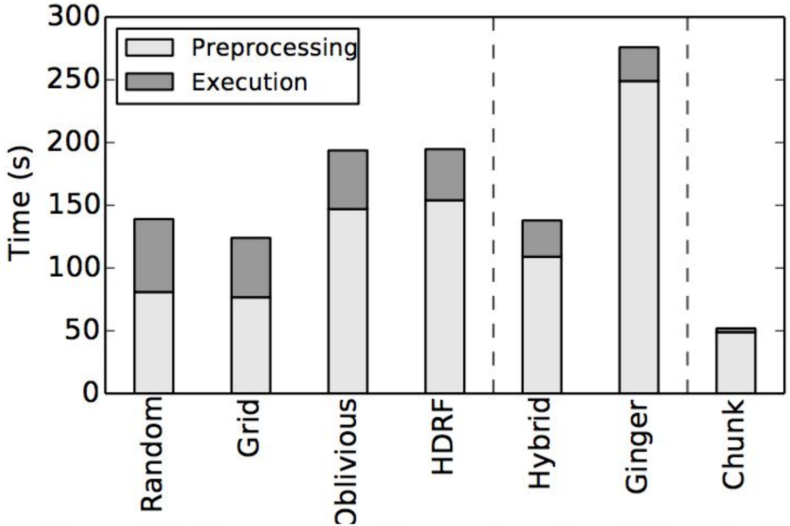


Figure 13: Preprocessing/execution time (*PR* on *twitter-2010*) with different partitioning schemes

3.3 Graph Partitioning: GraphGrind

GraphGrind: Addressing Load Imbalance of Graph Partitioning, ICS '17

Observations:

- Passes over vertices apart from passes over edges (one balance scheme does not fit all)
- Not a fixed amount of work per edge (PageRank traverses all edges but not BFS)
- Whether balancing edges or balancing vertices is better depends on algorithm

3.3 Graph Partitioning: GraphGrind

Memory Overhead

- Percentage of 0-degree vertices blows up as partition number increase

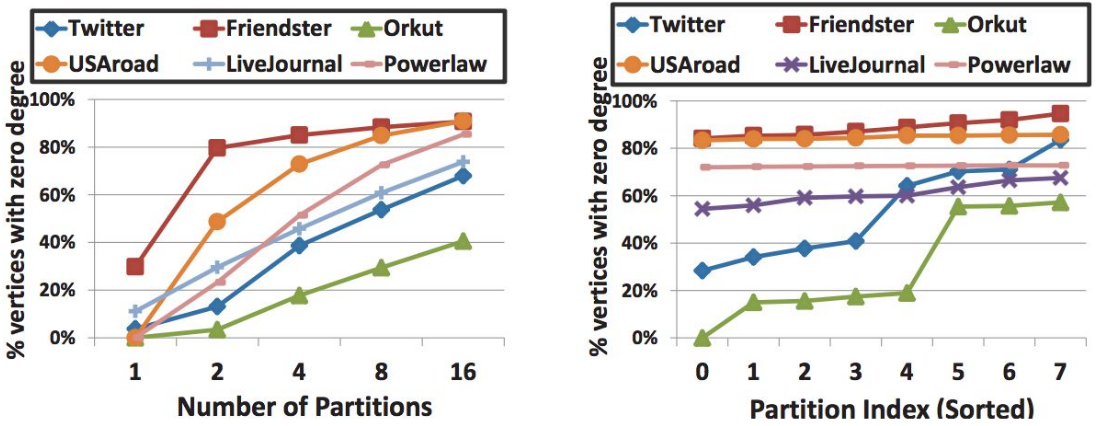


Figure 4: Percentage of vertices with zero out-degree averaged across all partitions (left) and variation across each of 8 partitions (right).

3.3 Graph Partitioning: GraphGrind

Optimization for memory overhead: eliminate 0-degree vertices

- Polymer stores all vertices on each partition O(P*|V|)
- GraphGrind stores an additional interleaved copy of the graph for sparse mode

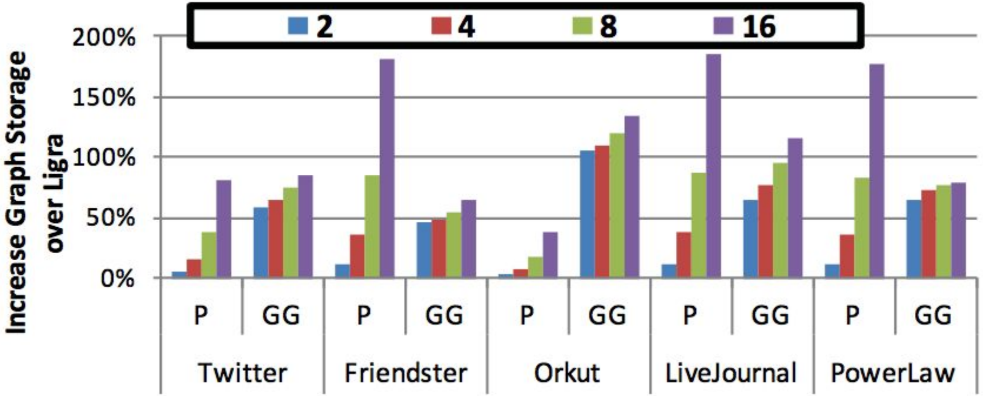


Figure 11: Increase of graph storage for Polymer (P) and GraphGrind (GG) compared to Ligra.

3.3 Graph Partitioning: GraphGrind

Over partitioning does not always work for load balancing

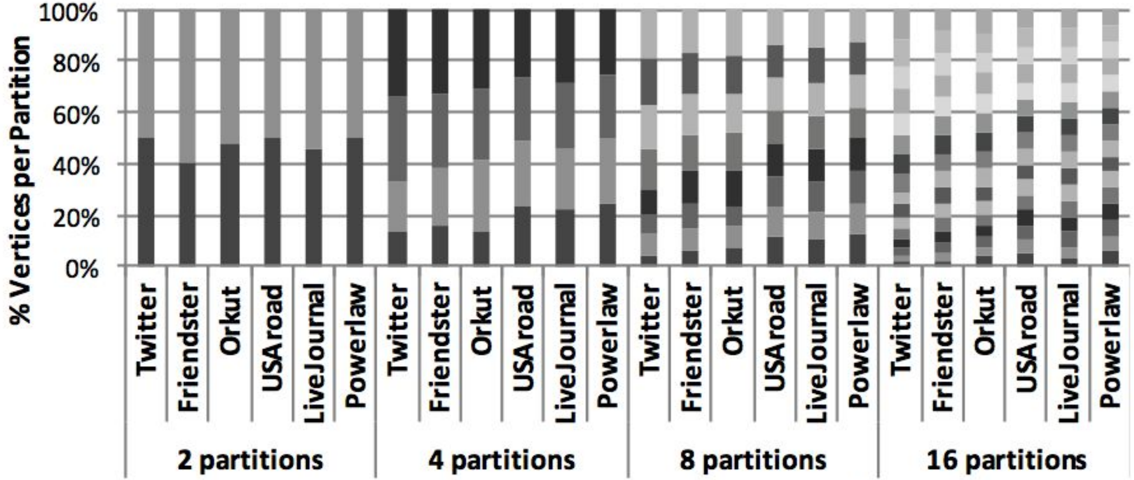


Figure 5: Relative sizes of partitions for varying degree of partitioning.

3.3 Graph Partitioning: GraphGrind

Optimization for load balancing:

- Algorithm specific partitioning strategy
- Still over-partitions

Algorithm	Description	Balance
BC	betweenness-centrality [23]	Vertices
CC	connected components using label propagation [23]	Edges
PR	simple Page-Rank algorithm using power method (10 iterations) [20]	Edges
BFS	breadth-first search [23]	Vertices
PRDelta	optimized Page-Rank forwarding delta-updates between vertices [23]	Edges
SPMV	sparse matrix-vector multiplication (1 iteration)	Edges
BF	Bellman-Ford algorithm for single-source shortest path [23]	Vertices
BP	Bayesian belief propagation [28] (10 iterations)	Edges

4

Data Placement

4.1 Linux default: first-touch policy

- A page is allocated on the memory node local to the process that first uses that page (not the process that calls malloc)
- Works well if there is good data locality
- Potential mismatch between allocation threads and processing threads
- Especially harmful in graph processing since graph loading can be single threaded

4.2 Interleaved Allocation

- Memory is allocated in a round robin fashion on the nodes specified using numactl or libnuma
- More balanced memory access time among cores
- Improves performance on NUMA-oblivious graph processing frameworks (e.g. Ligra and Galois)

4.3 User-defined Memory Bind

- Memory is allocated on a specific node or interleaved on a specific subset of nodes
- Needs the corresponding thread placement to ensure local access
- NUMA-aware graph processing frameworks use this strategy (Polymer, Gemini, GraphGrind, Grazelle)

4.4 Data Placement Strategies

	Polymer	Gemini	GraphGrind
Graph topology data	socket-local	socket-local	socket-local (dense) interleaved (sparse)
Application data	Replicated	Message passing between master and mirror	Stored on the home partition
Runtime state	Replicated	Message passing between master and mirror	Stored on the home partition

5

Thread Placement

5.1 Challenges in Thread Placement (Scheduling)

- OpenMP and Cilk don't have NUMA-aware scheduling or work stealing
- Checking thread number and binding to socket on the fly is expensive
- Manually precomputing processing range for each thread is not robust and does not guarantee good intra-socket load balance

5.2 Thread Placement: Gemini

Intra-socket fine-grained work stealing (OpenMP)

- Similar to "#pragma omp parallel for schedule(dynamic, 64)" in OpenMP but is NUMA-aware. Each thread is manually assigned begin and end on the subgraph local to the socket

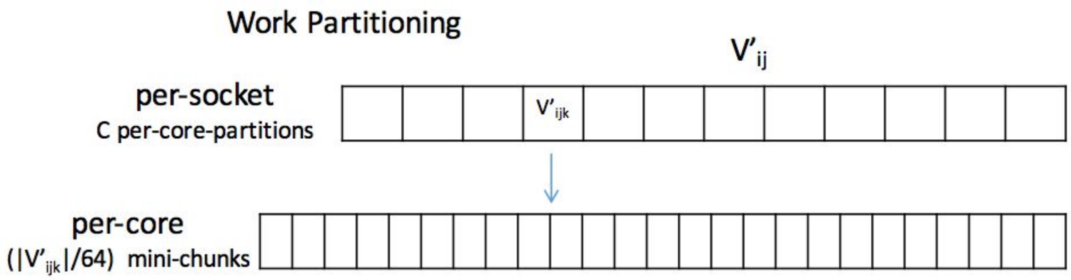
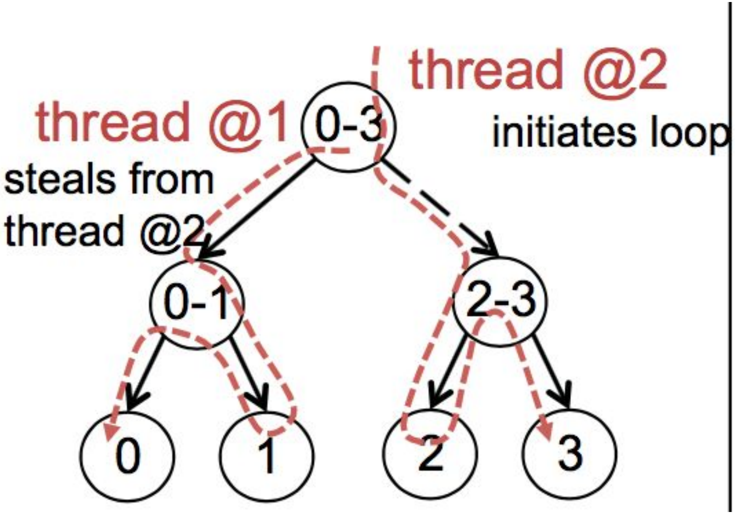


Figure 8: Hierarchical view of Gemini's chunking

5.3 Thread Placement: GraphGrind

Modified Cilk runtime

- loop iteration i should preferably be executed on cores associated to NUMA domain i
- Thread checks NUMA domain and first executes matched sub-range
- Steals from the oldest function on victim's call stack (thread 1)
- If no matched sub-range, execute on sub-optimal NUMA domain (thead 1 execute iteration 0)



(c) NUMA-aware execution order for threads 1 and 2

5.4 Thread Placement: OpenMP proc_bind

- The OMP_PLACES abstraction: sockets, cores, and threads
- Thread Affinity Policy
  - proc_bind(spread): places threads far away from each other among PLACES
  - proc_bind(close): places threads near each other among PLACES
  - proc_bind(master): same PLACE as parent thread

5.4 Thread Placement: OpenMP proc_bind

```
omp_set_nested(1);
#pragma omp parallel num_threads(num_places) proc_bind(spread) {
    int socketId = omp_get_place_num();
    auto sg = getSegmentedGraph(socketId);
    int n_procs = omp_get_place_num_procs(socketId);
   #pragma omp parallel num_threads(n_procs) proc_bind(master) {
      #pragma omp for schedule(dynamic, 64)
       for (int localVertexId = 0; localVertexId < sg->numVertices; localVertexId++) {
          int dst = sg->local_to_global_ID(localVertexId);
          int src = sg->read_source_vertex(u);
          local_new_ranks[dst] += local_ranks[src] / local_out_degree[src];
      }
   }
}
```

Socket-local sequential access **Socket-local random access**

6

Evaluation

6.1 Comparison

PageRank	Framework	Ligra
Polymer	5.28	15.03
Gemini	12.7	21.2
GraphGrind	15.979	23.66

BFS	Framework	Ligra
Polymer	0.90	1.13
Gemini	0.468	0.347
GraphGrind	0.254	0.319

CC	Framework	Ligra
Polymer	4.60	5.51
Gemini	4.93	6.51
GraphGrind	1.810	2.878

BC	Framework	Ligra
Gemini	1.88	2.45
GraphGrind	1.771	4.130

Various frameworks' reported runtime on Twitter-2010 for PageRank, BFS, CC, and BC

6.2 Performance Observations

- PageRank (most effective)
  - Traverse all edges
  - Intermediate results don't affect convergence rate
- Connected components using label propagation
  - Intermediate states matter. Socket local processing could result in a slower convergence rate
- BFS
  - Intermediate states matter (GraphGrind does not use socket-local buffer for BFS)
  - Not all edges are traversed (early break once a parent is found)

6.3 Performance of Gemini in the Distributed Setting

- 9-40 times speedup
- First framework that got reasonable running times in distributed memory
- Not showing numbers for BFS or other sparse traversal algorithms

Graph	PowerG.	GraphX	PowerL.	Gemini	Speedup (×times)
PR					
enwiki-2013	9.05	30.4	7.27	0.484	15.0
twitter-2010	40.3	216	26.9	3.02	8.91
uk-2007-05	64.9	416	58.9	1.48	39.8
weibo-2013	117	-	100	8.86	11.3
clueweb-12	-	-	-	31.1	n/a
CC					
enwiki-2013	4.61	16.5	5.02	0.237	19.5
twitter-2010	29.1	104	22.0	1.22	18.0
uk-2007-05	72.1	-	63.4	1.76	36.0
weibo-2013	56.5	-	58.6	2.62	21.6
clueweb-12	-	-	-	25.7	n/a
SSSP					
enwiki-2013	16.5	151	17.1	0.514	32.1
twitter-2010	12.5	108	10.8	1.15	9.39
uk-2007-05	117	-	143	3.45	33.9
weibo-2013	63.2	-	60.6	4.24	14.3
clueweb-12	-	-	-	56.9	n/a
GEOMEAN					19.1

Table 4: 8-node runtime (in seconds) and improvement of Gemini over the best of other systems.

6.3 Performance of Gemini in the Distributed Setting

Inter-node (cluster node) scalability:

- Near linear speedup on large graphs (weibo-2013)
- Poor scalability on small graphs (execution dominated by communication)
- Poor scalability on Twitter-2010 after 4 nodes due to duplicated mirror vertices (more partitions => higher duplication factor => more work)

$p \cdot s$	T_{PR} (s)	$\Sigma\lvert V_i\rvert/(p \cdot s)$	$\Sigma\lvert E_i\rvert/(p \cdot s)$	$\Sigma\lvert V_i'\rvert/(p \cdot s)$
$1 \cdot 2$	12.7	20.8M	734M	27.6M
$2 \cdot 2$	7.01	10.4M	367M	19.6M
$4 \cdot 2$	3.88	5.21M	184M	13.5M
$8 \cdot 2$	3.02	2.60M	91.8M	10.5M

Table 6: Subgraph sizes with growing cluster size

7

Summary

7. Summary

- Many graph applications are latency or bandwidth bounded by the QPI link
- To avoid remote memory access, a graph is partitioned and processed locally, and the results are merged across NUMA nodes
- Balanced graph partitioning is challenging:
  - fewer partitions => load imbalance => low parallelism
  - Over partitioning => higher duplication factor => more work
- NUMA-aware scheduling can be achieved through modifying the Cilk runtime, manually implementing work-stealing, or via the proc_bind API of OpenMP
- NUMA-aware graph processing trades work and parallelism for locality
- NUMA-aware graph algorithms generally perform better than NUMA-oblivious graph algorithms

Reference

- Intel Core i7 Xeon 5500 Series Specification
- NUMA-Aware Graph-Structured Analytics
- numactl(8) - Linux man page
- numa(3) - Linux manual page
- OpenMP reference page
- An NUMA API for Linux
- OpenMP API

Questions?