Parasol

The STAPL Programming Model

Lawrence Rauchwerger

http://parasol.tamu.edu

Texas A&M University

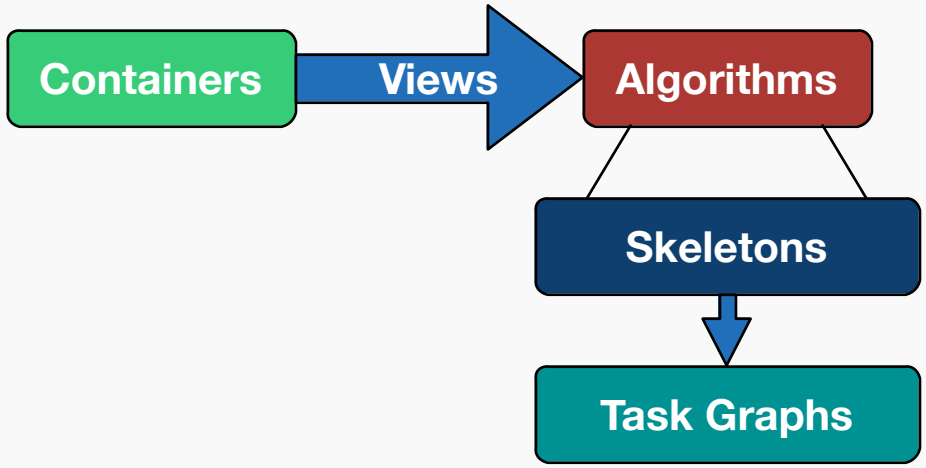
Parasol
Smarter computing.
Texas A&M University

STAPL: Standard Template Adaptive Parallel Library

Parasol

A library of parallel components that adopts the generic programming philosophy of the C++ Standard Template Library (STL).

- **STL**
  - ***Iterators*** provide abstract access to data stored in ***Containers***.
  - ***Algorithms*** are sequences of instructions that transform the data.
- **STAPL**
  - ***Views*** provide abstracted access to distributed data stored in ***Containers***.
  - ***Algorithms*** specified by **Skeletons**
    - Run-time representation is Task Graph (data flow engine)



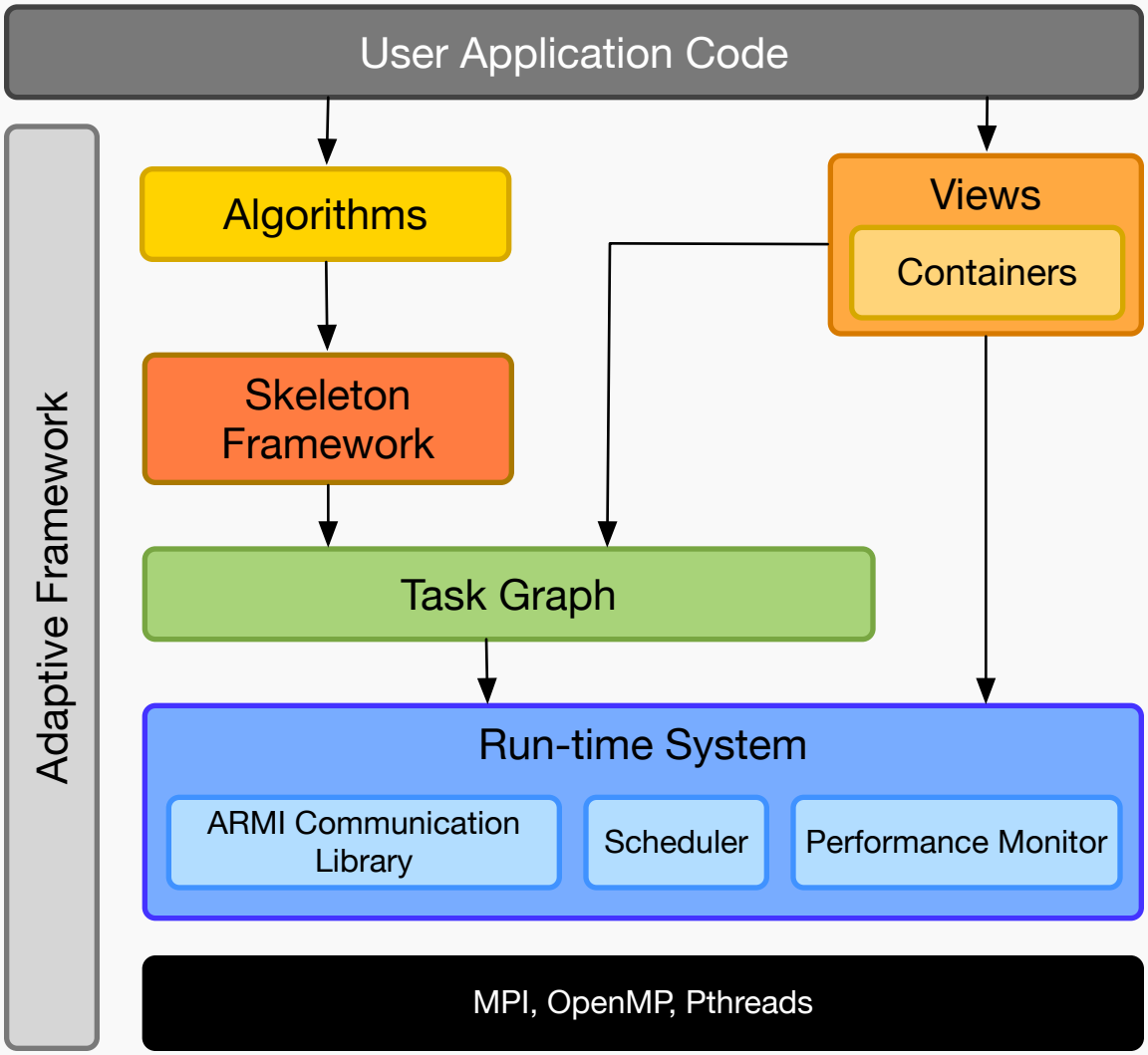
STAPL Programming Model

Parasol

- High Level of Abstraction ~ similar to C++ STL
- Task & Data parallelism:
  - Parallelism (SPMD) is implicit – Serialization is explicit
  - imperative + functional: Values flow along data flow graph and/or stored in containers
- Distributed Memory Model (PGAS)
- Programs defined by
  - Data Dependence Patterns (Library) ➔ Skeletons
    - Composition: parallel, serial, arbitrary, nested, …
  - Tasks: Work function (Parallel | Serial) & Data
    - ***Fine grain*** expression of parallelism – can be coarsened
    - Data in distributed containers
- Execution Defined by:
  - Algorithm run-time representation: Data Flow Graphs (PARAGRAPHs)
  - Execution policies (scheduling, data distributions, etc.)

STAPL Components

Parasol



Coding with STAPL vs. STL

Parasol

STL

```
using namespace std;
vector<int> c;

generate(c.begin(), c.end(), gen());

partial_sum(c.begin(), c.end(), c.begin());

int r = inner_product(c.begin(), c.end(), c.begin(), 0);
```

STAPL (synchronous)

```
using namespace stapl;
vector<int> c;
auto view = make_vector_view(c);

generate(view, gen());

partial_sum(view, view);

int r = inner_product(view,
                      view, 0);
```

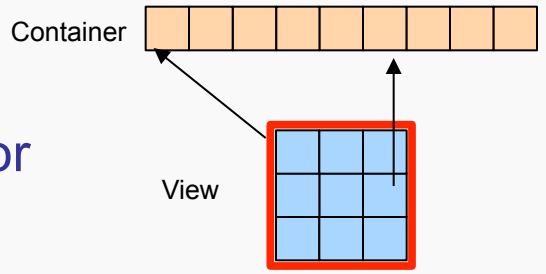
STAPL (asynchronous)

```
using namespace stapl;
vector<int> c;

skeleton<…> skel = compose(
  generate(gen()),
  partial_sum(),
  inner_product(0)
);

execute(skel, make_vector_view(c));
```

Views & Containers

Parasol

- A View defines an abstract data type (ADT) for the collection of elements it represents.
  - Example: Matrix View of the elements in a Vector



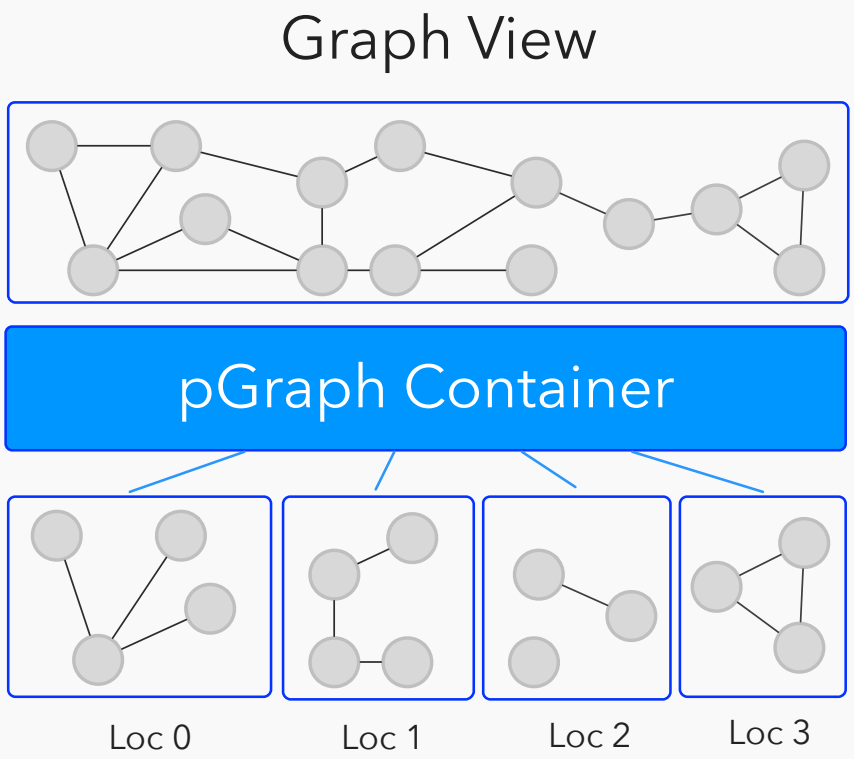
- Provides data access operations to Algorithms
- Allows element ordering independent of stored order
- Container : View + Data storage

Graph Container

Parasol

- Built using STAPL pContainer framework
  - Shared-Object View
  - Global address-resolution via 2-level distributed-directory
  - Handles data-distribution and communication
- Asynchronous migration of elements
  - While algorithms are being executed requests made to vertex being migrated remain valid
- Dynamic rebalancing

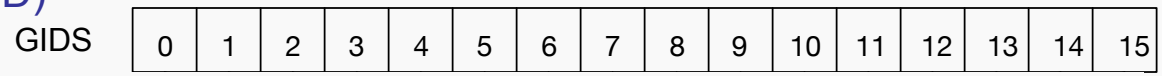
- Graph partitioned into sub-graphs
- Distributed across machine



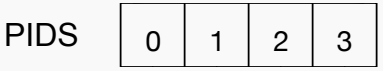
Data Distribution

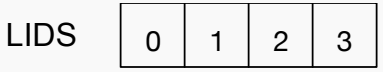
Parasol

- Three sets of identifiers
  - Container element ids (GID)
  - Partition ids (PID)
  - Location ids (LID)



- Two mapping functions needed
  - GID -> PID – maps elements to partition
  - PID -> LID – maps partitions to location





- STAPL provides mapping functions for common distributions
  - balanced, block-cyclic, block

- Arbitrary distributions supported by mapping functions that query lookup tables

View Specification

Parasol

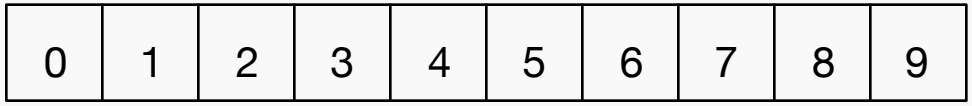
- Distributions specified using Views
- Partitioned collection of labeled elements (c, d, f_v^c, o)
- Defined as tuple
  - Reference to partitioned collection of elements (c)
  - Set of element identifiers (d)
  - Mapping function from View element identifiers to underlying collection identifiers
  - Set of operations

View Specification Example

(c, d, f_v^c, o)

Parasol

Collection of elements (c)



Set of identifiers (d)

Mapping function (f)

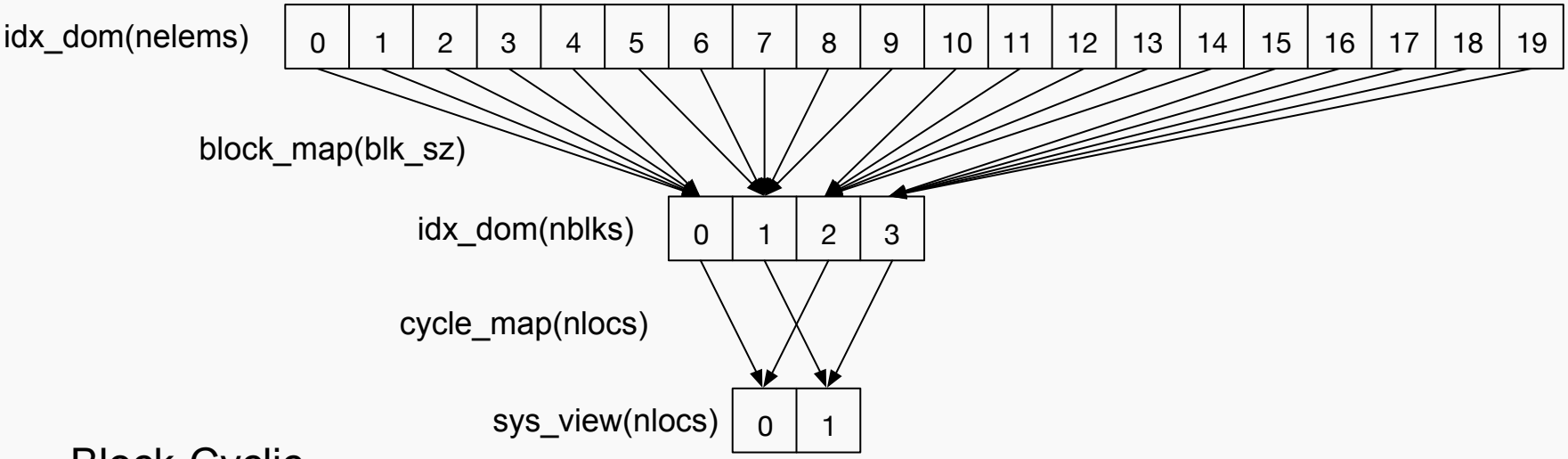
Set of operations (o)

- Read/write
- Subscript (e.g., c[i])

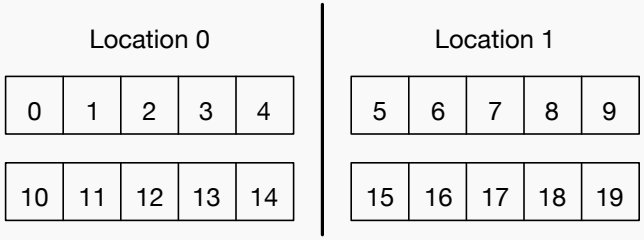
Data Distribution

Parasol

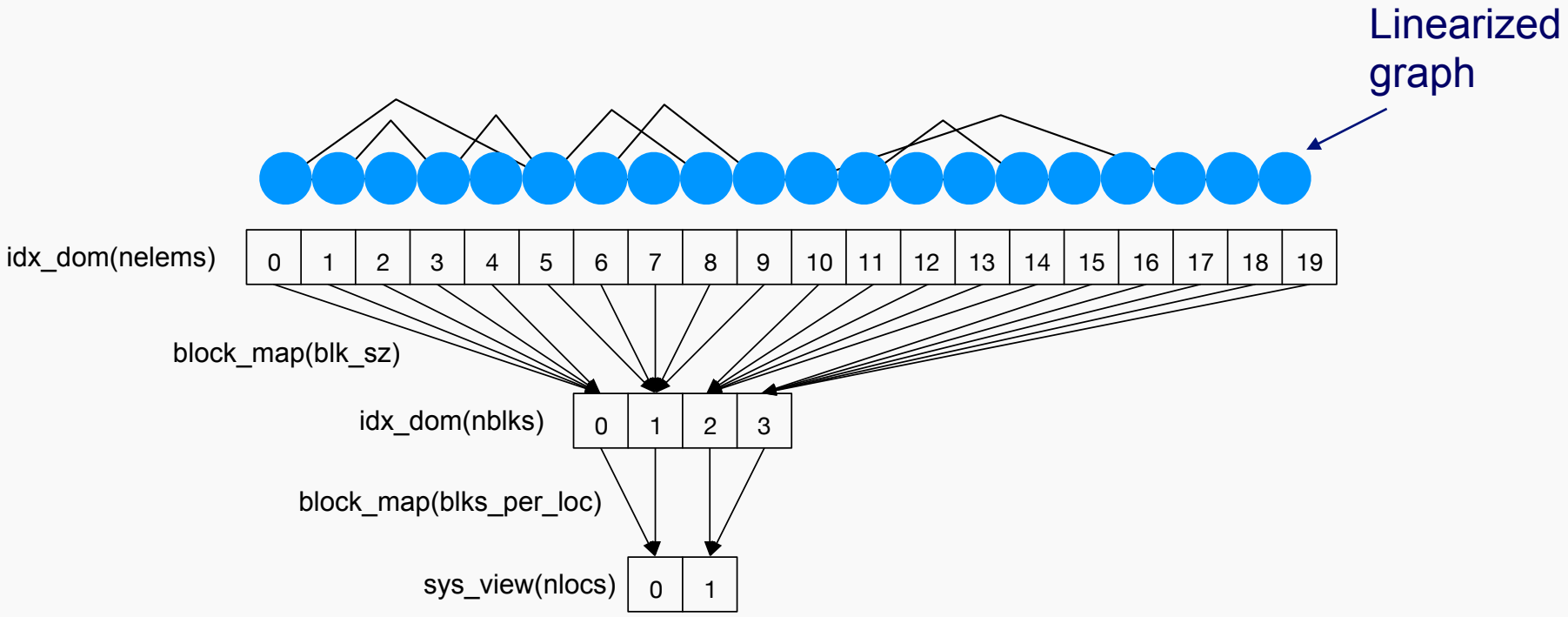
```
auto blk_cyc_spec =
partitioning_view(mapping_view(sys_view(nlocs), idx_dom(nblks), cycle_map(nlocs)), idx_dom(nelems), block_map(blk_sz));

stapl::array<int> a(blk_cyc_spec);
```



Block-Cyclic



Data Distribution (Graphs)



- Also provide graph specific partitioners using vertex weights and edge-cut

Control Model

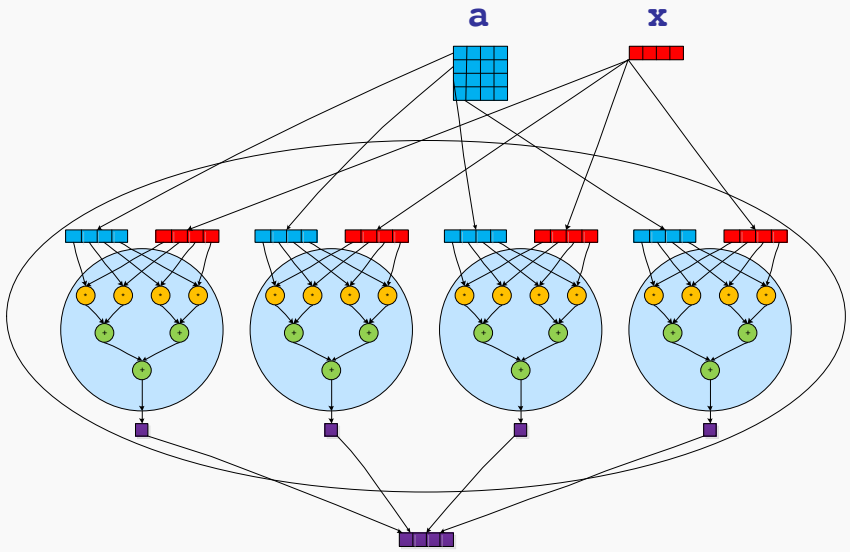
Parasol

How is parallelism achieved?

- Data Parallelism (skeleton expands algorithm pattern proportional to input data).
- Task Parallelism (skeletons composed to specify workflow).
- Implicitly parallel – all locations enter `stapl_main()`.
- Algorithm calls execute as task dependence graphs, where task can be nested, parallel sections.

How is synchronization expressed?

- Execution explicitly ordered by task graph edges, derived from data dependences specified in algorithmic skeleton.
- Distributed environment, tasks execute in isolated environment.
  No data races.




```
skeleton = zip(inner_product());
result = execute(skeleton, a.rows(), overlap(x));
```

Challenges for Parallel Graph Processing

Parasol

- Global synchronizations:
  - Asynchronous computation

    k-level asynchronous paradigm

    [PACT 2014] Best Paper
- Communication-bound:
  - Algorithm level communication aggregation

    Hierarchical communication paradigm

    [PACT 2015] Best Paper Finalist
- Very large data sets
  - **Out-of-core graph processing** [IPDPS 2015]

Graph Algorithm Suite

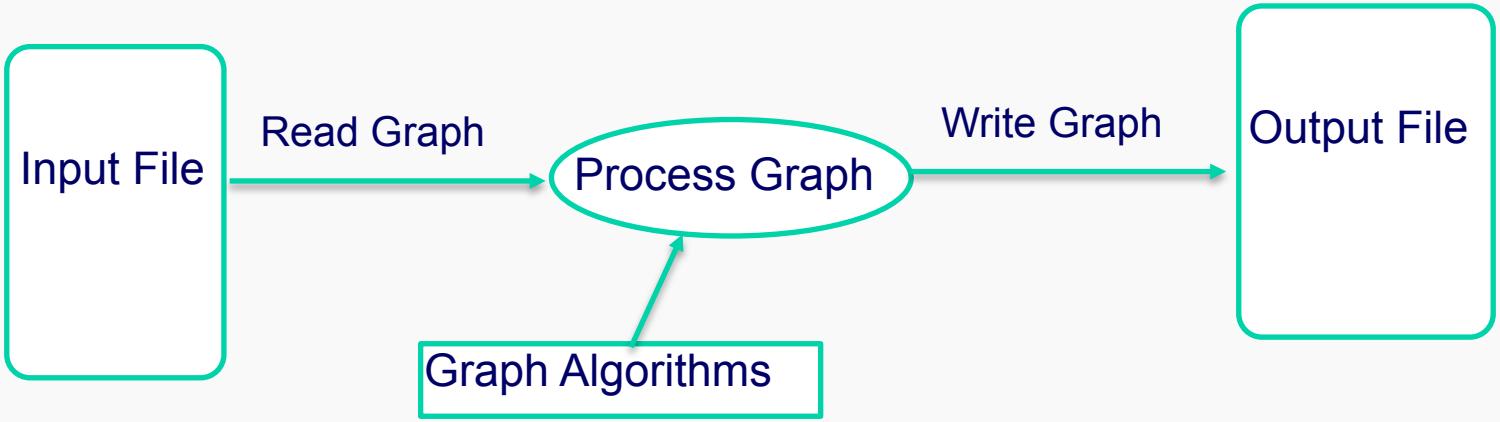
Parasol

- Traversals
  - breadth-first search, single-source shortest path, all-pairs shortest path, approximate breadth-first search, approximate diameter
- Link Analysis and Prediction
  - PageRank, BadRank, random walk, Adamic-Adar link prediction, conductance
- Clustering
  - agglomerative clustering, triangle count, clustering coefficient, label propagation community detection
- Connectivity
  - weakly connected components, strongly connected components, s-t connectivity
- Coloring
  - graph coloring, independent sets
- Spanning tree
  - minimum spanning tree
- Simulation
  - Barnes-Hut
- Network Flow
  - preflow-push max flow
- Meshing
  - Delaunay triangulation
- Matching
  - bipartite matching
- Centrality
  - betweenness centrality, closeness centrality
- DAG
  - topological sort

pGraph Algorithms

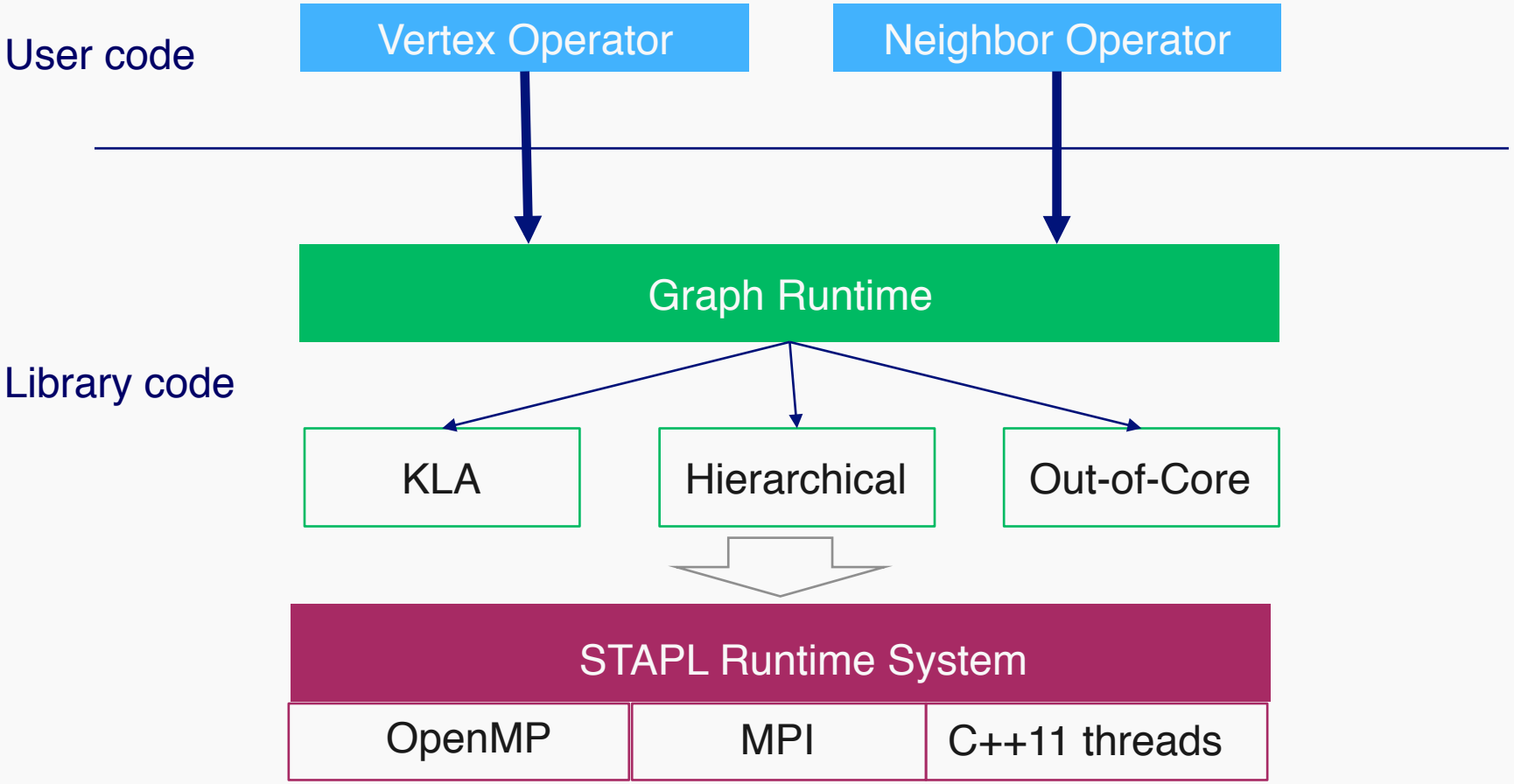
Parasol

```
graph_view g = read_graph("facebook.graph");

page_rank(g);
auto max = max_value(g, rank_comp());

size_t max_reach = breadth_first_search(g, max);
size_t friends_of_friends = count_if(g, level_equals(2));

auto ccs = connected_components(g);
size_t num_cc = ccs.size();
```



SGL Programming Model

Parasol



Algorithms in SGL: BFS - a running example

Parasol

- SGL algorithms are expressed with two operators:
  - Process a vertex and visit its neighbors:

```
bool bfs_vertex_op(Vertex v)
  if (v.color == GREY)
    v.color = BLACK;
    for (neighbor : v.edges)
      spawn(neighbor, bfs_neighbor_op(v.distance+1));
    return true;
```

  - Process a neighbor

```
bool bfs_neighbor_op(Vertex u, Distance new_distance)
  if (u.distance > new_distance)
    u.distance = new_distance;
    u.color = GREY;
    return true;
  else
    return false;
```

Algorithms in SGL: BFS -
a running example


```
bool bfs_vertex_op(Vertex v)
  if (v.color == GREY)
    v.color = BLACK;
     for (neighbor : v.edges)
       spawn(neighbor,
        bfs_neighbor_op(v.distance+1));
    return true;
```

```
bool bfs_neighbor_op(Vertex u,
                     Distance new_distance)
  if (u.distance > new_distance)
    u.distance = new_distance;
    u.color = GREY;
    return true;
  else
    return false;
```

```
void breadth_first_traversal(Graph graph, Vertex source)
  source.color = GREY
  sgl::execute(bfs_vertex_op(), bfs_neighbor_op(), graph);
```

- Expressed as a vertex-centric algorithm (Pregel-like)
- Implicit parallelism

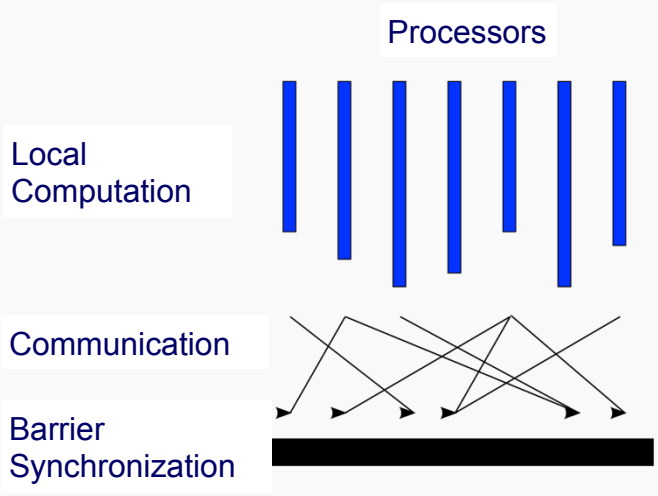
Global synchronizations limit parallelism

→ KLA Scheme

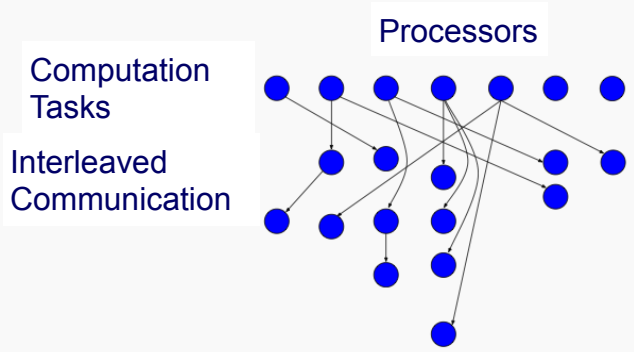
Parallel Graph Algorithms May Use:

Parasol

- Level-Synchronous Model
  - BSP-style iterative computation
  - Global synchronization after each level, no redundant work

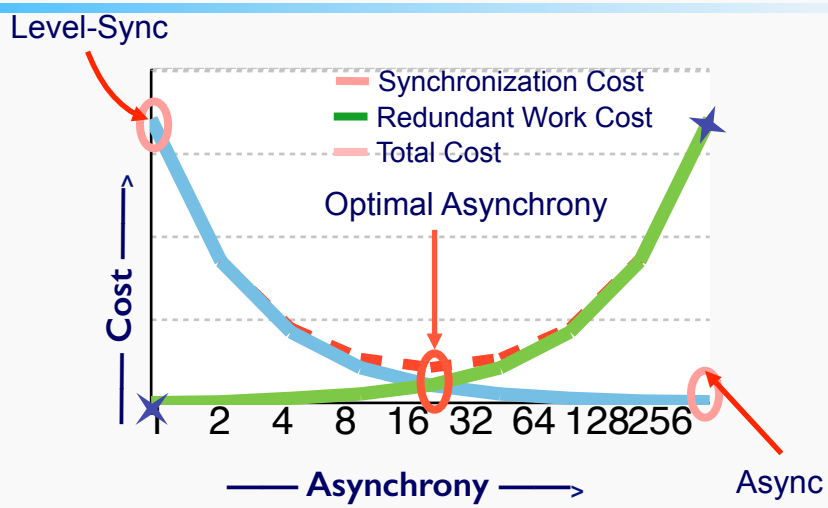


- Asynchronous Model
  - Asynchronous task execution
  - Point-to-point synchronizations, possible redundant work



Having Your Cake and Eating it Too

Parasol

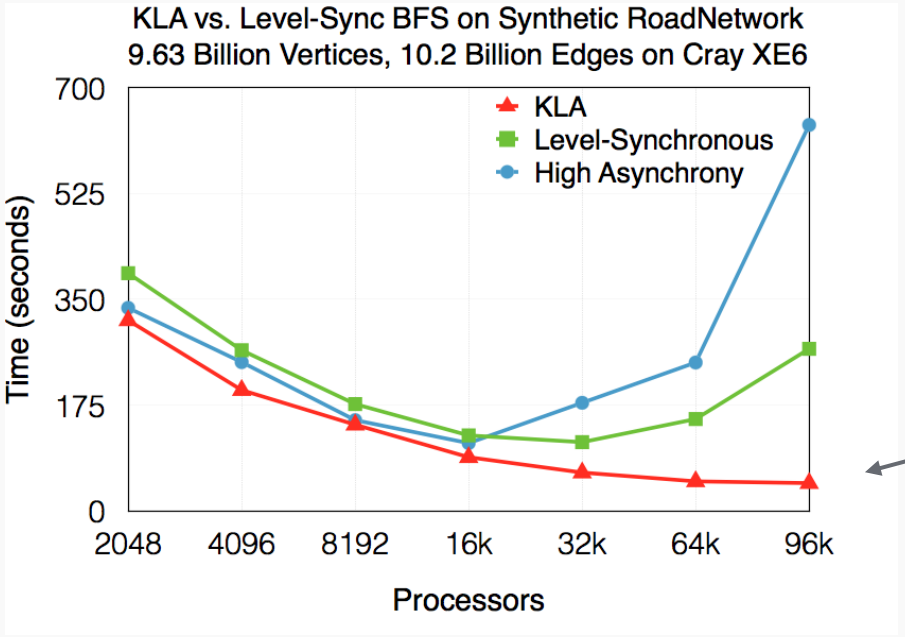


$$\text{Asynchrony} = \frac{\text{levels}}{\text{supersteps}}$$

k-Level Asynchronous Model

- k defines depth of superstep (KLA-SS)
- Unifies existing models
  - k=1: Level-synchronous
  - k=d: Asynchronous

k-Level Asynchronous BFS



Diameter = 3218
k = 9
KLA-SS = 358

- Current strategies stop scaling after 32,768 cores
- KLA strategy faster, scales better
- Adaptively change asynchrony to balance global-synchronization costs and asynchronous penalty

Algorithms in KLA: BFS

Parasol

```
bool bfs_vertex_op(Vertex v)
  if (v.color == GREY)
    v.color = BLACK;
     for (neighbor : v.edges)
       spawn(neighbor,
        bfs_neighbor_op(v.distance+1));
    return true;
```

```
bool bfs_neighbor_op(Vertex u,
                     Distance new_distance)
  if (u.distance > new_distance)
    u.distance = new_distance;
    u.color = GREY;
    return true;
  else
    return false;
```

```
void breadth_first_traversal(Graph graph, Vertex source, int k)
  source.color = GREY
  sgl::execute(bfs_vertex_op(), bfs_neighbor_op(), graph, sgl::kla(k));
```

Expressed as a vertex-centric algorithm (Pregel-like)

Implicit parallelism

User operators agnostic of ‘k’

K selection – model and/or dynamic adaptive

KLA Implementation

Parasol

```
void execute(VertexOp vertex_op, NeighborOp neighbor_op,
                  Graph graph, policy kla_policy)
  while(active)
    kla_superstep += kla_policy.k;
    active = map_reduce(kla_wf(vertex_op, visitor(neighbor_op, kla_superstep)),
                        logical_or(),
                        graph);
    global_fence();
```

KLA Paradigm

```
void visitor(NeighborOp neighbor_op, Vertex u, int k)
  bool active = neighbor_op(u);
  if (active && k < kla_ss)
    spawn(vertex_op(u));  // spawn task on neighbor
```

KLA Visitor

KLA Visitor Policies

Parasol

```
visitor(NeighborOp neighbor_op, Vertex u, int k)
  bool active = neigbor_op(u);
  if (active && k < kla_ss)
    spawn(vertex_op(u));  // spawn task on neighbor
```

KLA Visitor

```
visitor(NeighborOp neighbor_op, Vertex u)
  bool active = neighbor_op(u);
  if (active)
    spawn(vertex_op(u));
```

Async Visitor

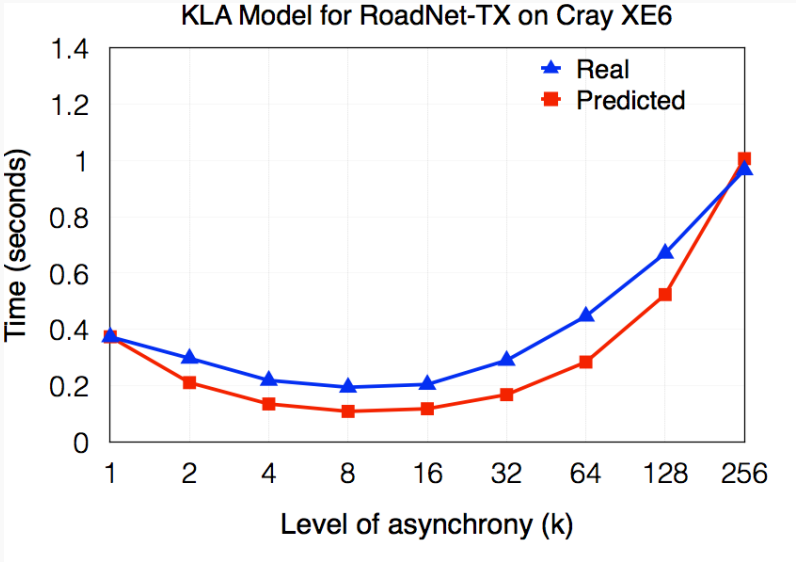
```
visitor(NeighborOp neighbor_op, Vertex u)
  bool active = neighbor_op(u);
```

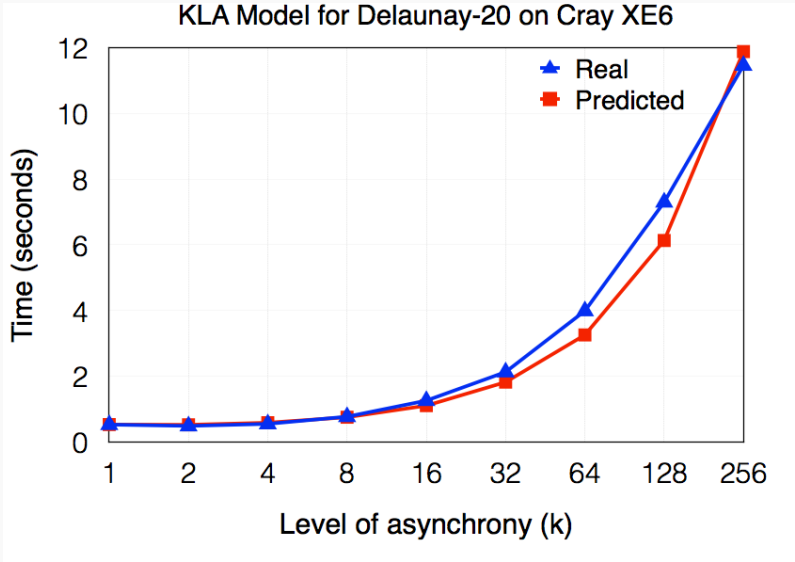
Level-Sync Visitor

Selecting a good k value

Parasol

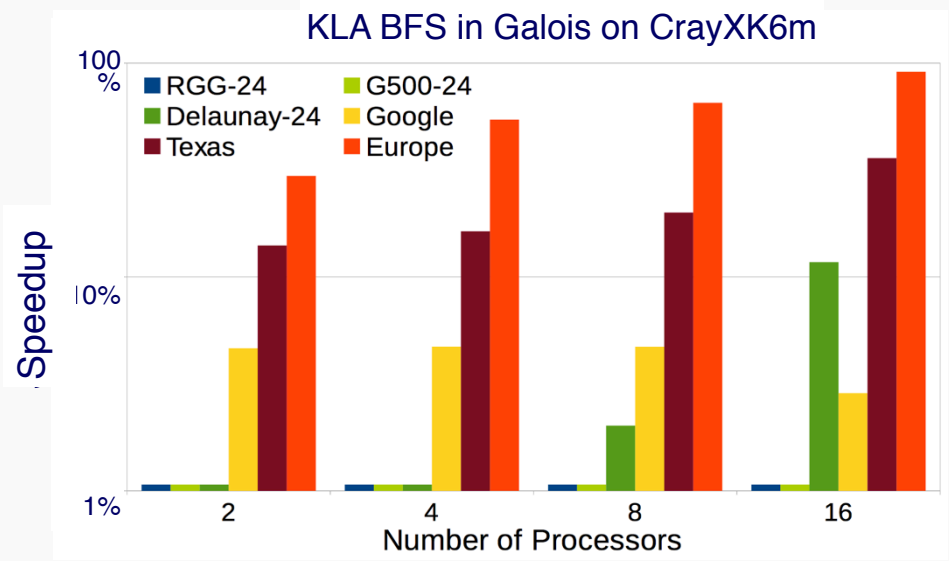
- The level of asynchrony (k) is problem & instance specific





- Model the execution time for a given k
- In practice, we provide an adaptive selection method for k

KLA in Other Frameworks



$$\text{Speedup} = \frac{\min(T_{LSYNC}, T_{ASYNC})}{T_{KLA}}$$

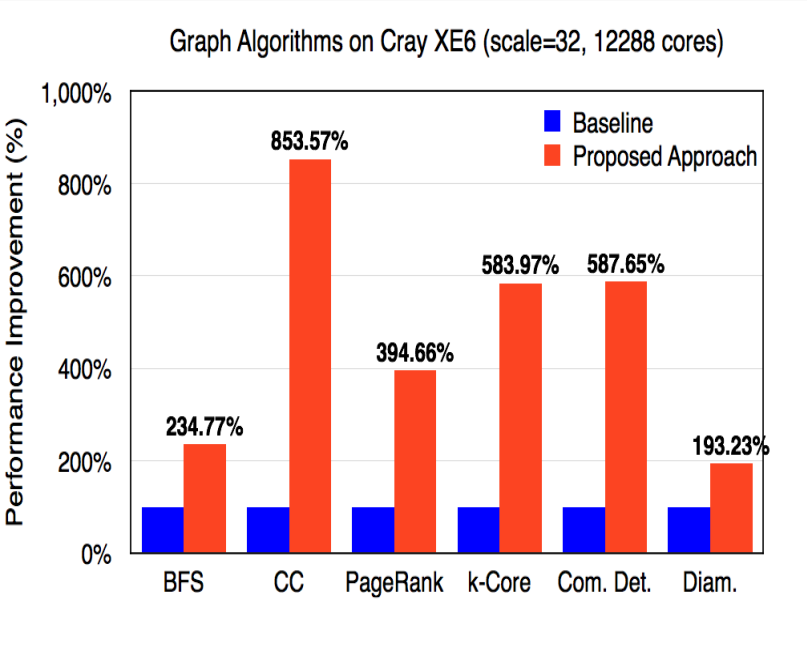
- 16 cores on a single Cray XK6m node
- Modified Level-Synchronous worklist for Galois to allow for KLA
- Improvement dependent on graph type
- Performance improves vs. level-sync and async executions

Hierarchical Paradigm

Parasol

- Reduce duplication and minimize number and volume of messages
  - Exploit Algorithmic Redundancy in outgoing and incoming messages
  - Utilize knowledge of system topology

Graph500

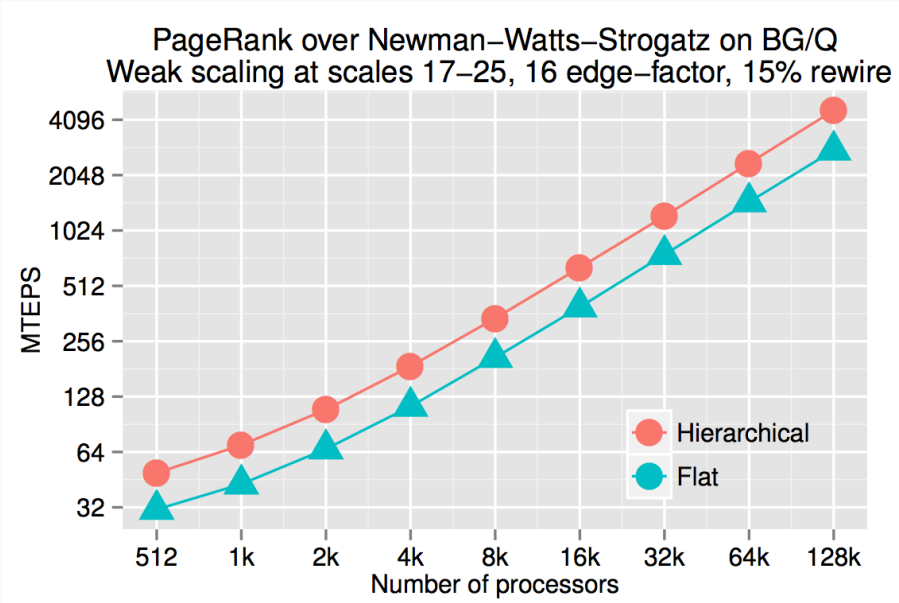



```
sgl::execute(bfs_vertex_op(), bfs_neighbor_op(), graph,
   sgl::hierarchical(h));
```

Harshvardhan, Adam Fidel, Nancy M. Amato, Lawrence Rauchwerger, "An Algorithmic Approach to Communication Reduction in Parallel Graph Algorithms," *PACT*, San Francisco, CA, Oct 2015.

Performance at Scale

Parasol

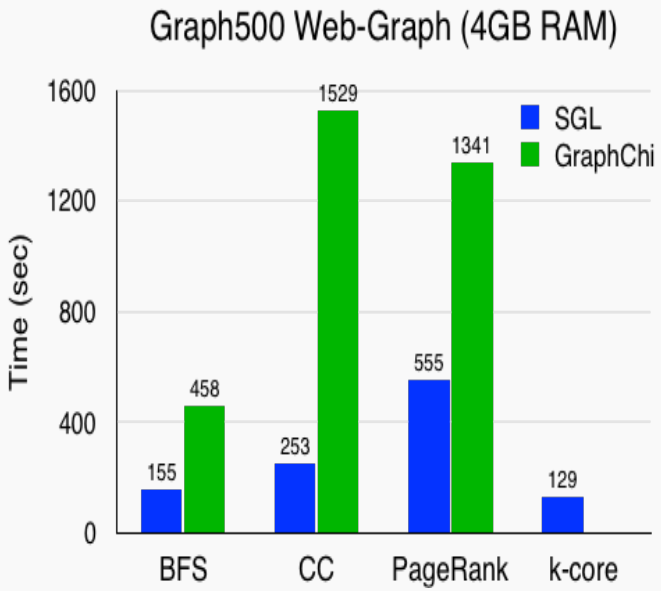


- 131,000 cores on IBM Blue Gene/Q
- Similar trend across systems, algorithms and graphs

Out of Core Graphs

Parasol

- Transparent, efficient out-of-core graph processing
- Subgraph paging
  - Logic level instead of fixed size OS paging
  - No overhead when graph fits in RAM
  - Low overhead when processing from Disk



17 Million Vertices
268 Million Edges
2-core PC with 4GB RAM

```
sgl::execute(bfs_vertex_op(), bfs_neighbor_op(), graph,
   sgl::storage(subgraph_sz));
```

Harshvardhan, Brandon West, Adam Fidel, Nancy M. Amato, Lawrence Rauchwerger, "A Hybrid Approach To Processing Big Data Graphs on Memory-Restricted Systems," In *Proc. Int. Par. and Dist. Proc. Symp. (IPDPS)*, 2015.

Conclusion

- STAPL is intended for productive parallel application development
- SGL is a collection of STAPL components designed for “easy” and efficient parallel graph processing
  - Parallel graph container, graph utilities, suite of algorithms and graph execution strategies
  - Extensibility is major goal
    - Multiple execution strategies with same model
    - Different degrees of relaxation of consistency
- Available at national labs (LLNL, LANL, Sandia)