Parasol

Toward Exascale Programming with STAPL

Lawrence Rauchwerger

http://parasol.tamu.edu/~rwerger

Dept of Computer Science and Engineering, Texas A&M

Exa – Peta … where is PDT now ?

Parasol

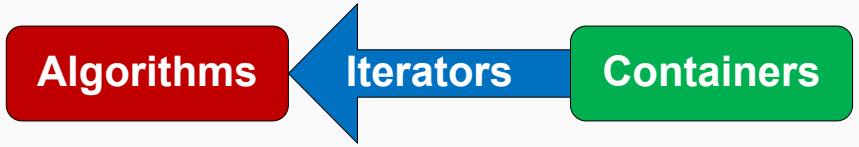
- Previously developed PDT using PTTL library
  - Scale to 128k cores
- Currently developing PDT using STAPL library
  - STAPL is general purpose parallel library
  - 75 K LOC in PDT only
  - Scales up to 393k processors on BG/Q (Vulcan)
- Exploit space (geometry) level parallelism
  - Sequence of parallel sweeps across the *rectangular* grids with Pipelined directions
  - Asynchronous (step wise) communication
  - No fault tolerance
  - Homogeneous computer system (BG/Q) – no accelerators
- Our software environment : STAPL …

STAPL: Standard Template Adaptive Parallel Library

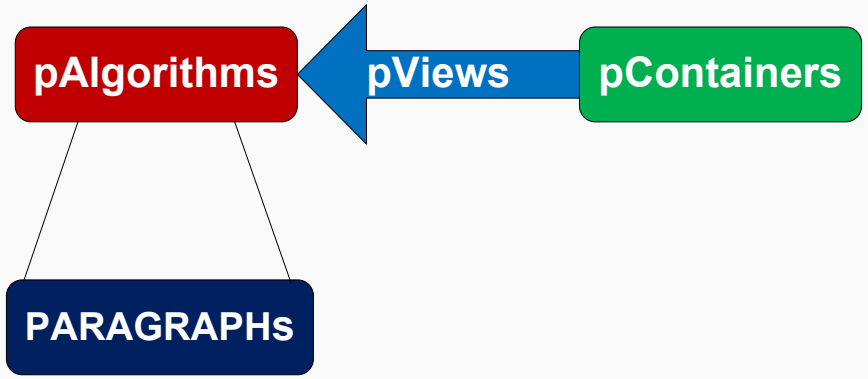
Parasol

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*** spec-ed by **Skeletons**, represented at run time as ***PARAGRAPHs***, parallel task graphs that transform the input data.
    - Can use existing Skeletons, defined in collection of common ***parallel patterns***.
    - ***Extensible*** - users can define new patterns.



Programming Model with STAPL

Parasol

- STAPL Programming Model.
  - High Level of Abstraction ~ similar to C++ STL
  - ***Fine grain*** expression of parallelism – can be coarsened
  - Implicit parallelism – Serialization is explicit
  - Distributed Memory Model (PGAS)
  - Algorithms defined by
    - Data Dependence Patterns (Library)
    - Distributed containers
    - Execution policies (scheduling, data distributions, etc.)
  - Algorithm run-time representation: Task Graphs (PARAGRAPHs)

Some features for Exa-scalable STAPL

Parasol

- Asynchronous Algorithms (a bit later)
- Nested/Hierarchical parallelism (parallel algorithms)
- Extension to heterogeneous architectures - GPUs
- Special support for
  - AMR (space/angle)
  - Arbitrary grids, sparse data structures
- Adaptive behavior
  - Granularity control of tasks (data + work)

    Fine ←→ Coarse Grain Parallel Algorithms Morphing
  - Communication/Synch aggregation AND Customization(pt2pt)
- Algorithmic Composition for Productivity & Performance (skeleton library + composition operators)

Asynchronous Algorithms & Communication

Parasol

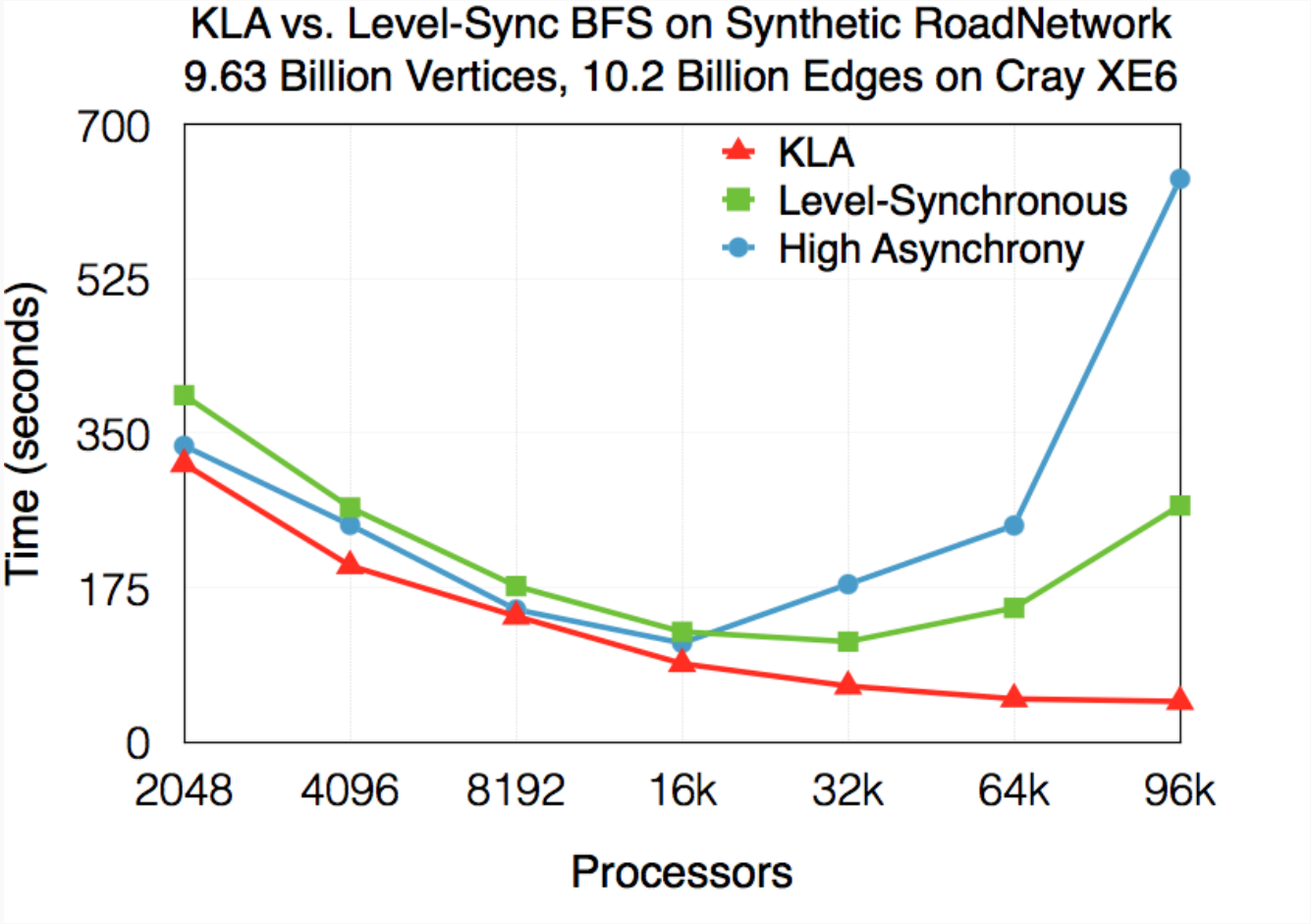
- Asynchrony ➔ Latency Hiding

- Asynch communication: STAPL: ARMI comm. Library
  - Asynch active messages – never waits for a return value.
  - Futures – place holders for return values not yet computed but needed for current evaluation (increases asynchrony).
  - Recursively nested communication subgroups (and subcontainer registration) ➔ locality, load balancing + affinity, work reduction (efficiency)

- Asynchronous Algorithms – not an easy task …

Asynch Algos: K-level Asynchronous BFS

Parasol

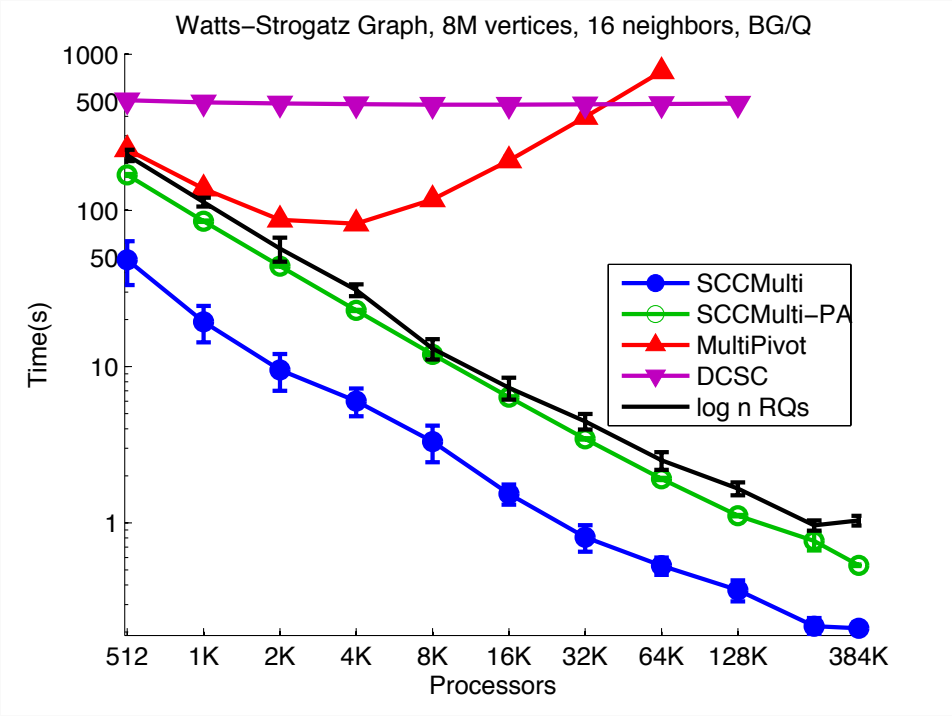


- Removing synchs more important at higher proc. counts

Asynch Algos: Strongly Connected Components (SCC)

Parasol

- SCCMulti algorithms use KLA paradigm
- Improved performance over
  - DCSC (McLendon et al., 2000)
  - MultiPivot (Schudy, 2007)
- SCCMulti
  - Exponentially increasing number of random pivots chosen on each iteration
- SCCMulti-PA
  - Every node is a pivot with random priority
  - Uses priority-scheduler to block lower priority traversals



Adam Fidel will present Asynchronous Graph Algorithms

Containers in the STAPL Programming Model

Parasol

- Provide shared object view of data to application
  - Data partition and distribution can be specified.
  - Implementation of distribution encapsulated in Container.
- Container: Data storage + Abstract Data Type
  - In STAPL, ADT is View (LCPC 2010)
  - Sequential Containers used as storage
    - STL (vector, list, map, set, hash), MTL[1] (matrix), BGL[2] (graph), etc.
- Container properties
  - Distributed non-replicated storage
  - Concurrent, thread safe methods

1] Matrix Template Library 2] Boost Graph Library

pContainer Interfaces

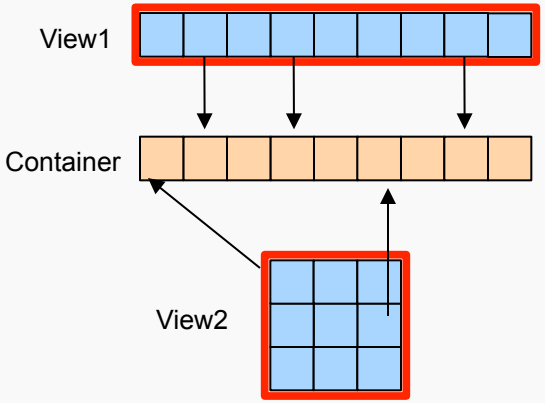
Parasol

- Collective Methods
  - Default constructors
  - Users may specify a desired data distribution
- Concurrent Methods
  - Sequentially equivalent methods (sync)
  - Methods optimized for parallelism (async, split-phase)
- Adam Fidel will present Containers

Views in the STAPL Programming Model

Parasol

- A View defines an abstract data type (ADT) for the collection of elements it represents.



Example: Matrix View of the elements in stapl::vector

- Allows storage independent data access
  - Views provide data access operations to Algorithms
  - View domains can be partitioned independent of data storage in Container
- User focused on application instead of data access and distribution details.
- Timmie Smith will present Views

View Example

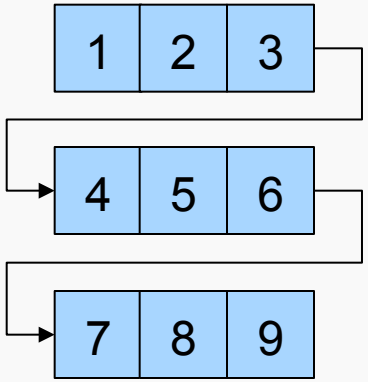
Parasol

Matrix

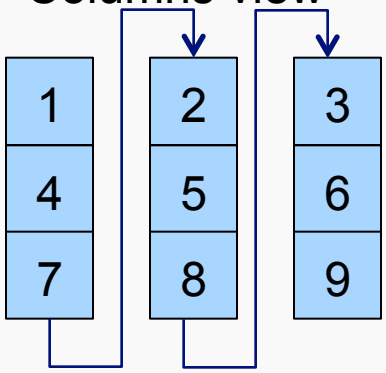
1	2	3
4	5	6
7	8	9

- Print elements of Matrix
  - row-wise or column-wise?
  - Implement several print methods…
  - Use one generic print method with different views

Rows view



Columns view




```
print(View v)
  for i=1 to v.size() do
    print(v[i])
```

Output
1,2,3,4,5,6,7,8,9

Output
1,4,7,2,5,8,3,6,9

Nested, Hierarchical Algos and RT

Parasol

- Nested parallelism:

  While{forall (reduce (sweep (….))))}

- Hierarchical parallelism (algos): nested and mapped onto the machine memory hierarchy

  forall (view_i, forall (view_j, wf{})) where view_i = U{view_j}

  mapped hierarchically on machine hierarchy (Locales)

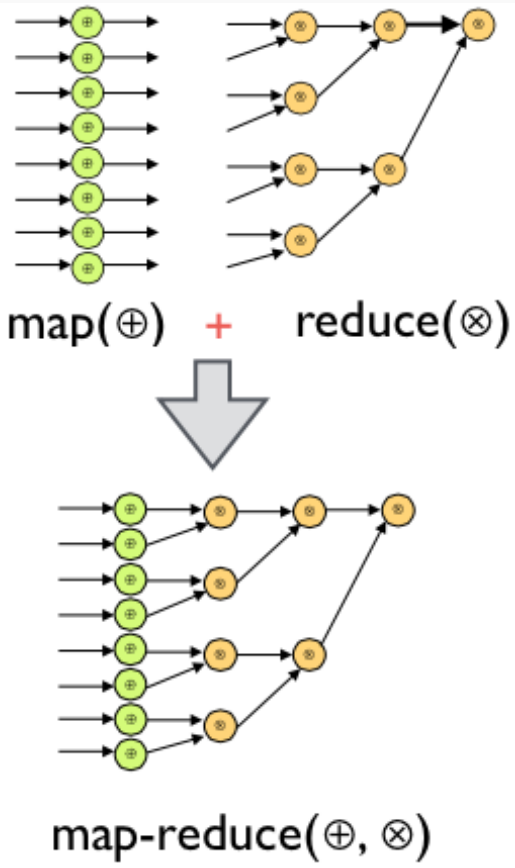
- Support for various Runtime Systems (MPI/OpenMP/ Pthreads…recursive constructs)

- **Nested/Hierarchical ➔ Latency reduction (locality) + Expressivity (and productivity)**

Algorithm Specification with Skeletons

Parasol

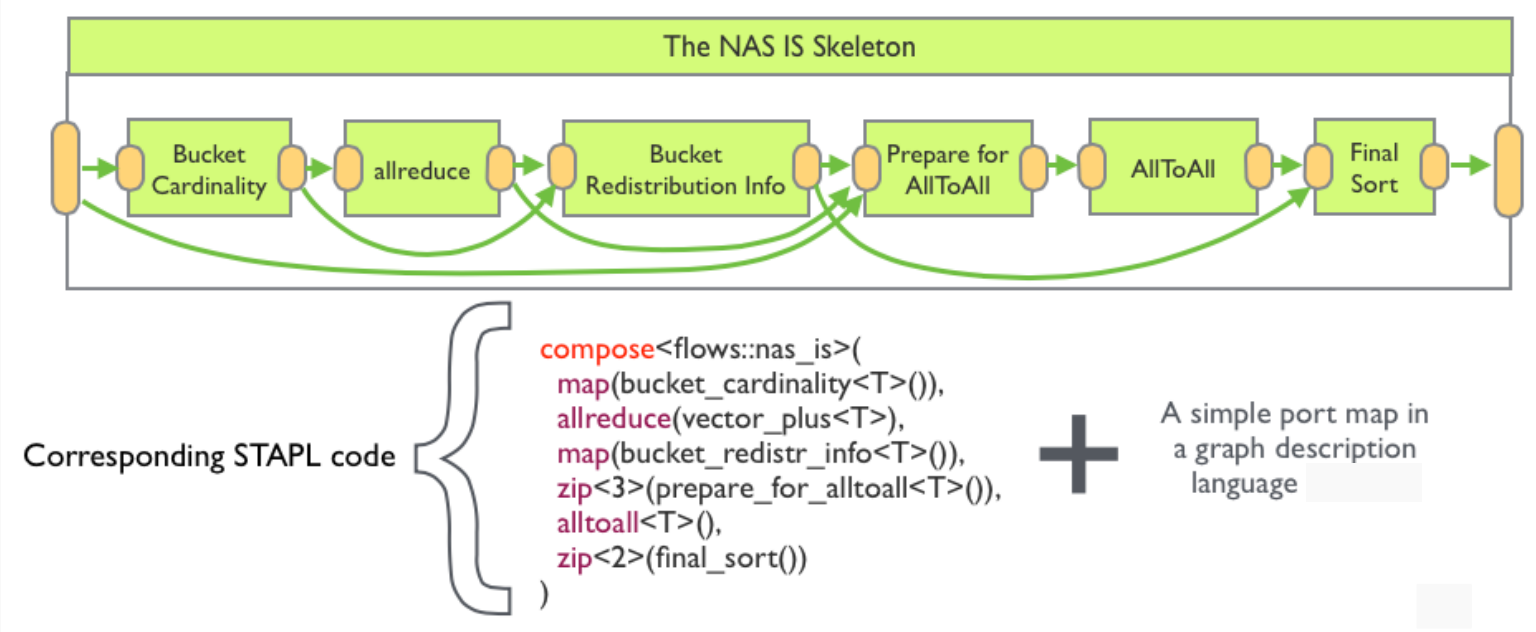
- Algorithmic skeletons
  - Portable and implementation-independent representation
  - Polymorphic higher-order functions
    - map f $[a_1, a_2, \ldots, a_n] = [f(a_1), f(a_2), \ldots, f(a_n)]$
  - Inherently composable
  - Allow formal analysis and transformation
  - Current libraries don't provide composition as first-class feature
- STAPL Skeleton Library
  - Agnostic to shared and distributed system
    - Algorithmic skeletons (e.g., map and map-reduce)
    - Data flow programming
      - Internal Representation of skeletons
      - Explicit parallelism in the representation
    - Composition a first-class feature
      - Composition = point-to-point dependencies
      - No need for reimplementation
      - No need for global synchronization
- Mani Zandifar will present Algorithms and Skeletons



Case Study – NAS IS

Parasol

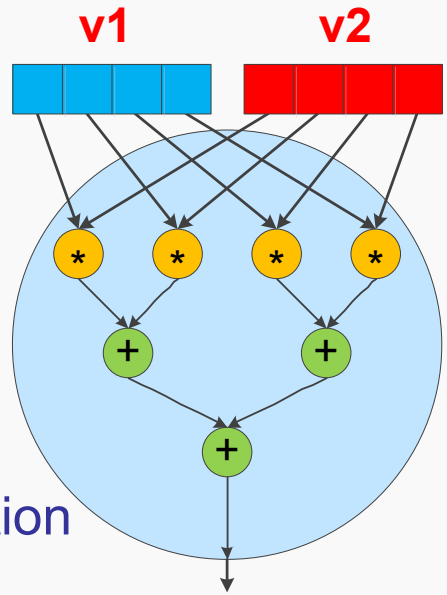
- NAS Integer Sort Benchmark
  - Input = a list of N uniformly distributed numbers
  - Output = ranks for each value in Input



STAPL PARAGRAPH

Parasol

- Runtime representation of task dependence graph
- Vertices represent computation on input views
- Edges represent dependences that enforce an ordering on execution
- PARAGRAPHs can be recursively defined
  - Invocation of STAPL Algorithm inside task computation



- PARAGRAPH construction and execution is asynchronous
  - Only specified point-to-point dependencies are enforced

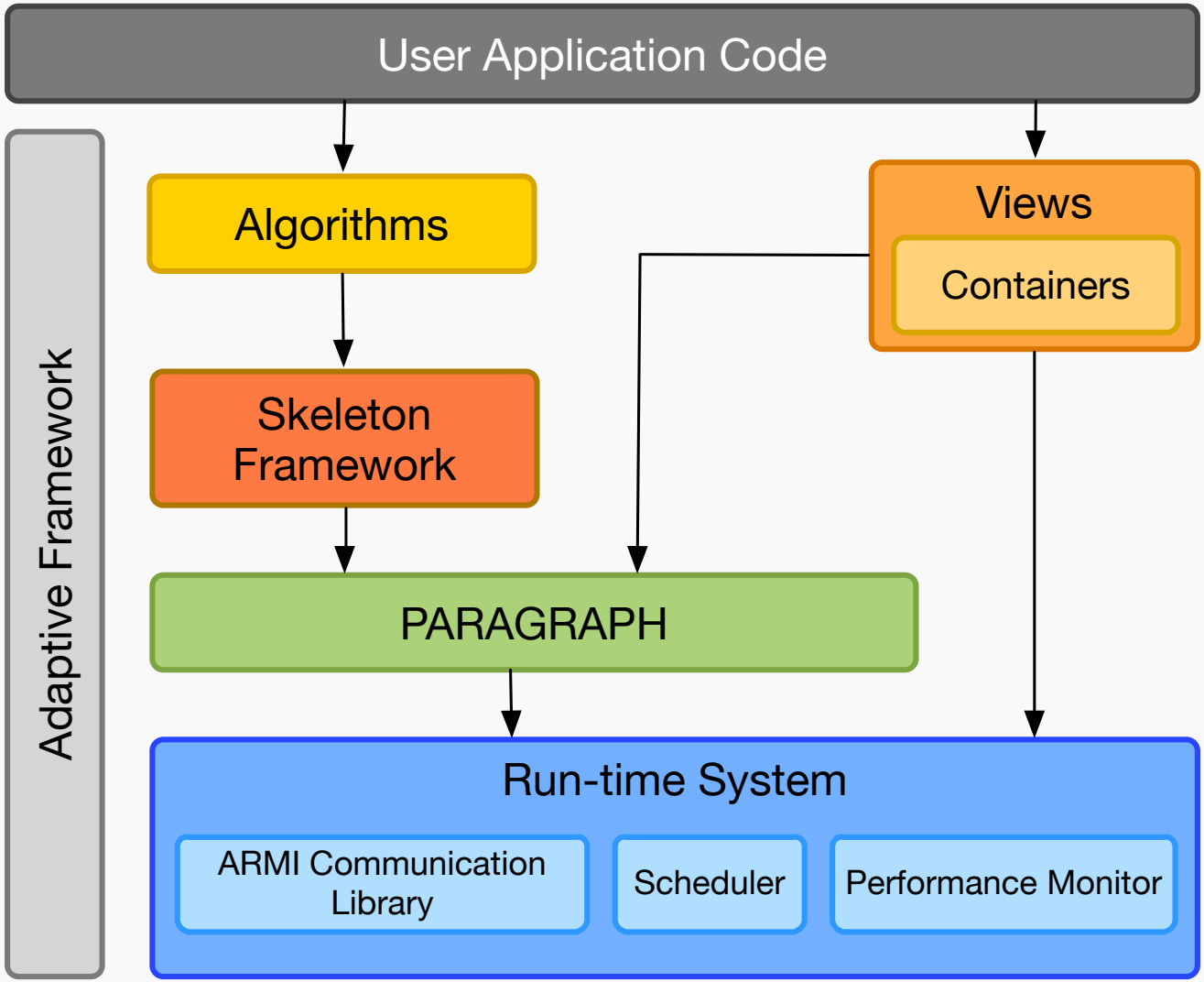
STAPL Runtime

Parasol

- ARMI: Adaptive Remote Method Invocation
  - Unified communication model over distributed and shared memory
  - Asynchronous communication through RMI methods
    - RMI can send work, data, or both to another location for processing
  - Copy semantics enable asynchrony
- Scheduler
  - Customizable component to control execution of PARAGRAPHs and tasks within PARAGRAPHs
  - Hierarchical scheduling and execution for nested parallel algorithms
- Performance Monitors
  - Abstract interfaces of system-specific performance libraries
  - Enables user code to be easily ported
- Nathan Thomas will present the PARAGRAPH and Runtime

STAPL Components

Parasol



Presentation Outline

- Overview
- **Containers**
- Views
- Algorithms & Skeletons
- PARAGRAPH and Runtime

Outline

- STAPL overview
- STAPL containers
- STAPL views
- STAPL algorithms and skeletons
- STAPL runtime system

stapl::containers

Adam Fidel

Parasol Laboratory
Department of Computer Science and Engineering
Texas A&M University

Outline

- STAPL Container Framework
  - Provided Containers
  - Distributions
  - Interfaces
- Example Usage
- Parallel Graphs

STAPL Containers

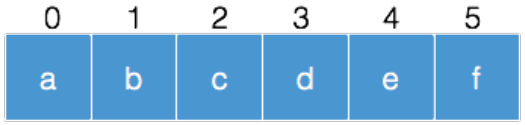
- Provide shared object view of data to application
  - Data partition and distribution can be specified.
  - Implementation of distribution encapsulated in STAPL container
- STAPL container properties
  - Distributed non-replicated storage
  - Concurrent methods
  - Shared object view
  - Support for user customization (e.g., data distributions)

- Concepts and methodology for developing parallel containers
  - STAPL Containers - a collection of base containers and support for parallelism management
- Improved user productivity
  - Base classes providing fundamental functionality
    - Inheritance
    - Specialization
  - Composition of existing containers
- Scalable performance
  - Parallel random access to base containers
  - Low overhead relative to the base container counterpart

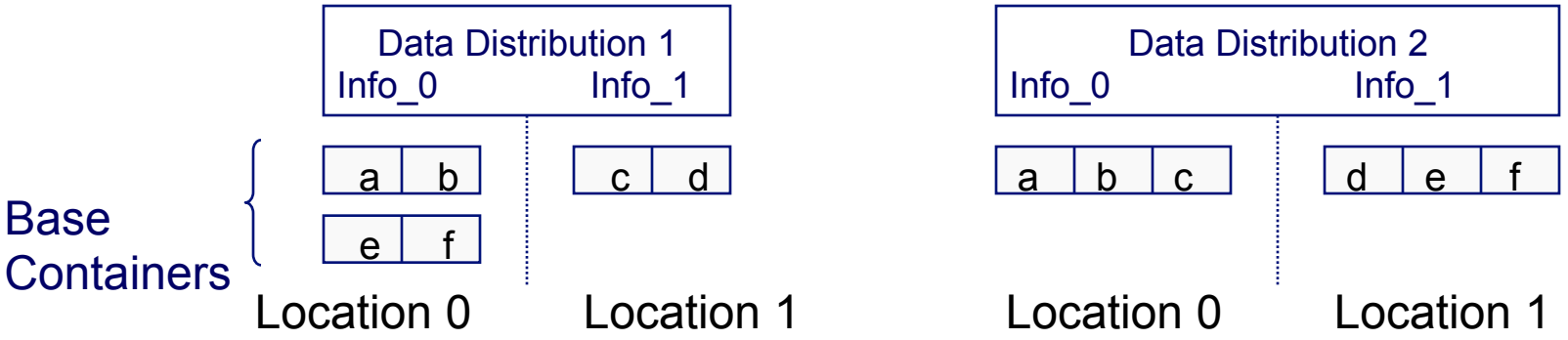
Container Framework Concepts

- Base container: data storage
  - sequential containers (e.g., STL, MTL, BGL)
  - parallel containers (e.g., Intel TBB)
- Distribution information
  - shared object view
  - distributed directory
  - local base container managers

```
stapl::array<char>
a(6)
```

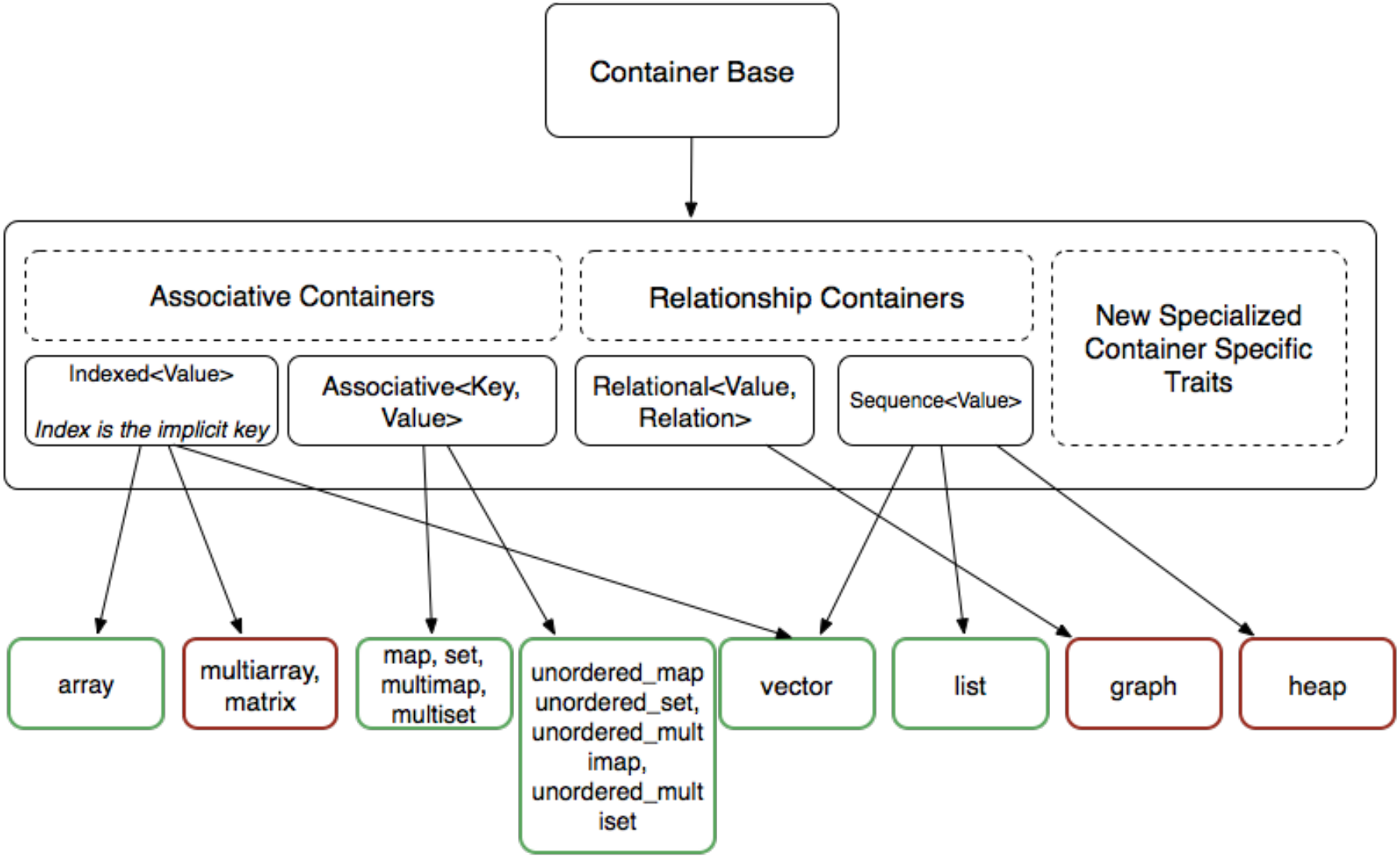


User Level



Parasol

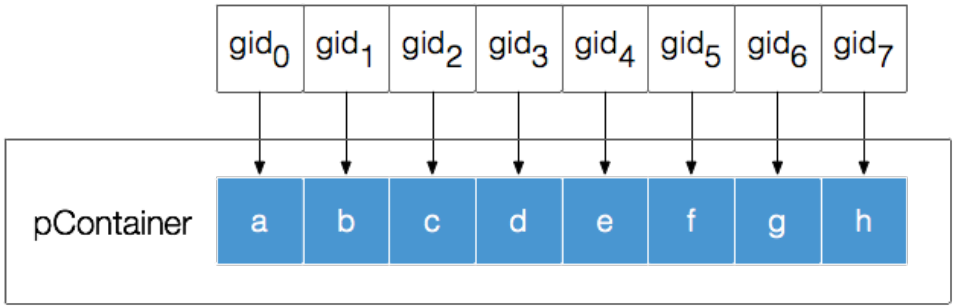
STAPL Container Taxonomy



Parasol

Global Unique Identifier

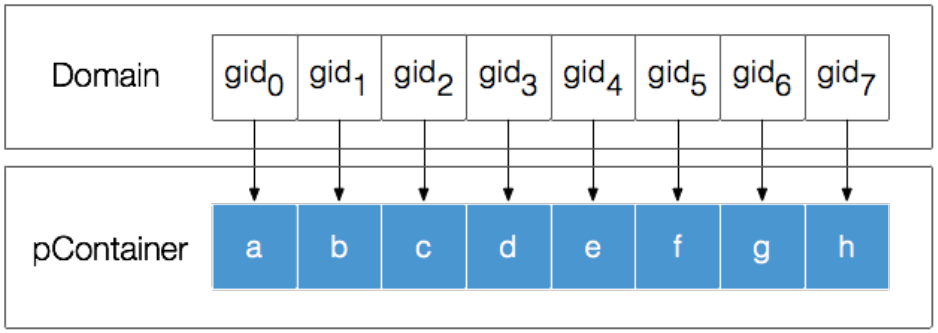
- GID - Each element is uniquely identified by its GID
  - Allows implementing a shared memory programming model on a distributed memory architecture
  - Abstracts the underlying memory architecture
  - Allows data representation at the logical (abstract) level
  - GID examples
    - Index for array and vector
    - Vertex descriptor for graph
    - Key for associative containers
    - Tuples for multiarray (i, j, k)



Parasol

Domain

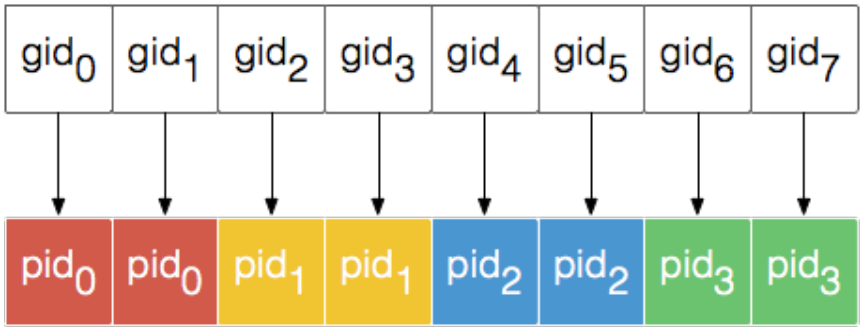
- Domain - A set of GIDs corresponding to the elements of the container
  - Used when defining pContainers or views
    - `array<int> arr(domain(50,99)))`
- Domain Types
  - Finite Domain :
    Indexed(0,99) ~ [0,99]
  - Infinite Domain: Continuous<string>('a', 'z');



Parasol

Partition

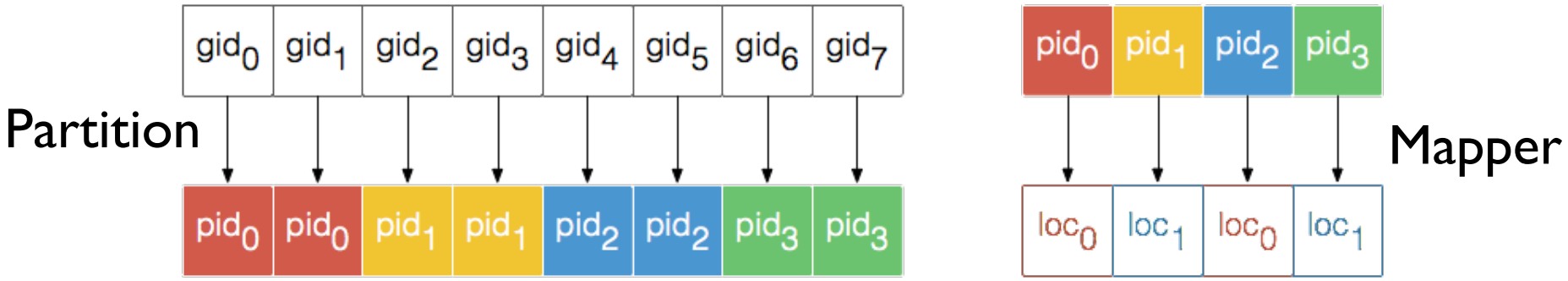
- Specify a decomposition of a domain in sub-domains
  - For a given domain D, P={D_0, D_1,…D_{n-1}} such that
    - $D = D_0 \cup D_1 \cup \ldots \cup D_{n-1}$
    - $D_i \cap D_j = \varnothing$, $\forall$ i,j, 0 <= i,j < n, i != j
  - Example: blocked, balanced, cyclic, arbitrary
- Interface
  - For each GID, compute partition ID
    - P(gid) → partition id



Parasol

Partition Mapping

- Specify the mapping of partitions on to locations
  - For a given set of partitions P = {D_0, D_1,…,D_{n-1}}, compute location for P_i
  - Example: blocked, cyclic, arbitrary
- Interface
  - For each partition ID, compute location ID
    - M(partition id) → location



Container Interfaces

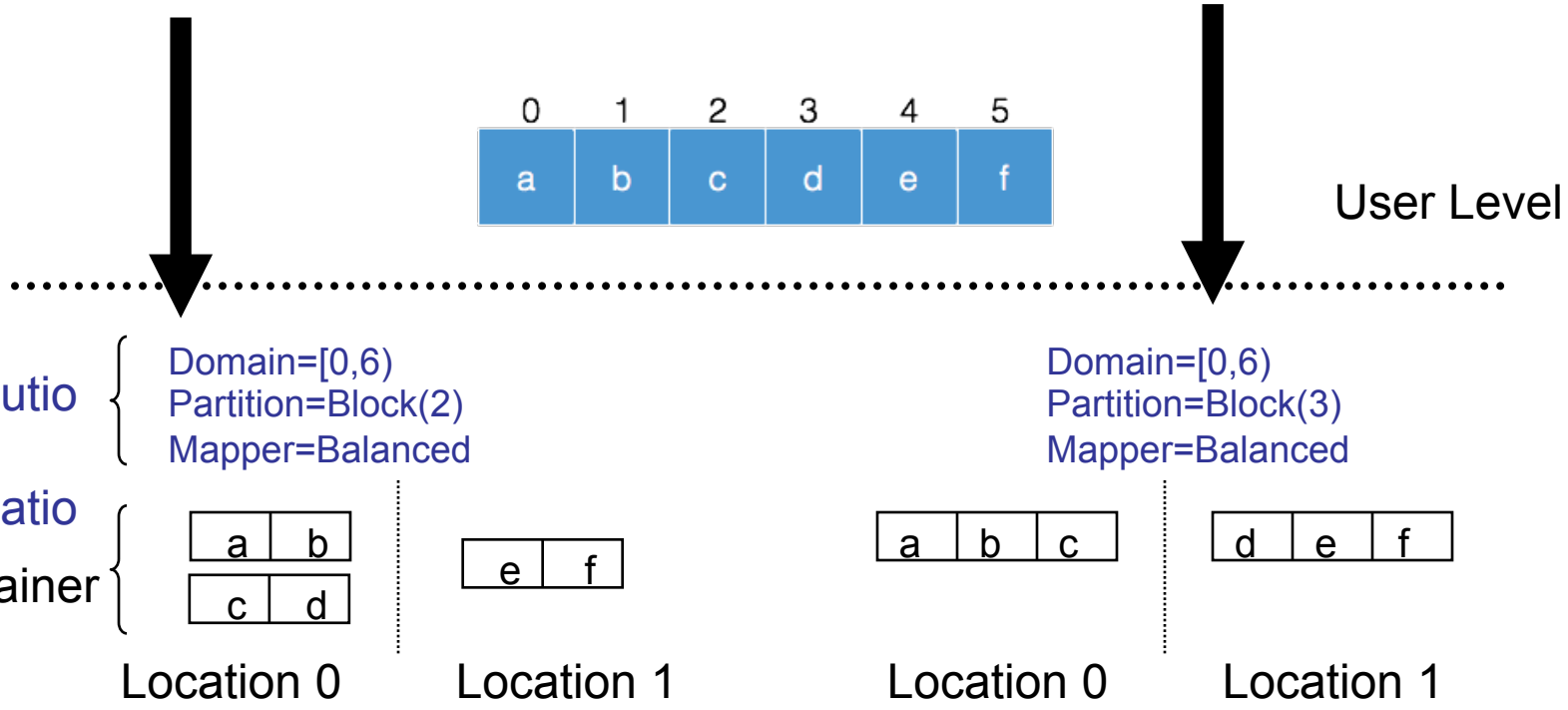
- Collective Methods
  - Container constructors
  - Users may specify a desired data distribution
- Concurrent Methods
  - Sequentially equivalent methods (sync)
  - Methods optimized for parallelism (async, split-phase)

Container Interfaces

- Collective Methods
  - `stapl::array<int> a(size);`
- Element wise
  - synchronous
    - `x = ct.get_element(i);`
  - asynchronous
    - `ct.set_element(i, 100);`
- split phase
  - `stapl::future<int> f = ct.get_element_split(i);`
  - `int x = f.get();`
  - `f.then(my_function);`
- Global properties
  - `size(), empty();`

```
array<int>
arr(6, blocked(2));
```

```
array<int>
arr(6, blocked(3));
```



stapl::array

Usage

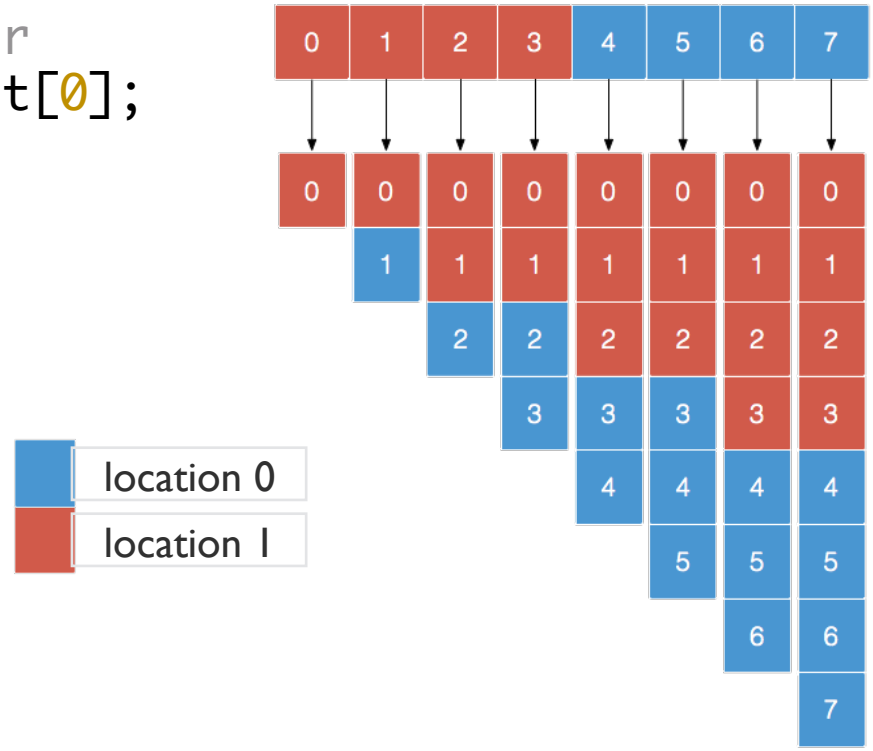
```
array<int> arr(n, blocked(2));

const int size = arr.size();

arr[i] = 5;

arr.set_element(0, 10);

future<int> f = arr.get_element_split(0);
cout << f.get();
```

Container Composition

- The element type of a container can be a container
  - e.g.: `array<vector<double>>`
- Opportunity for nested parallelism
- Better expressivity for certain applications
  - Kripke
- Opens new interesting research directions
  - Optimal data distribution for container at different levels of nesting
  - Optimize data access using views defined on composed pContainers
    - E.g., flatten views, matrix views

Composed Containers

```
array<int> sizes_ct = {1, 2, 3, 4, 5, 6, 7, 8};

array<array<double>> nested_ct(sizes_ct);

// returns a nested container
auto first_nested = nested_ct[0];
```



Graph Container

Highlights

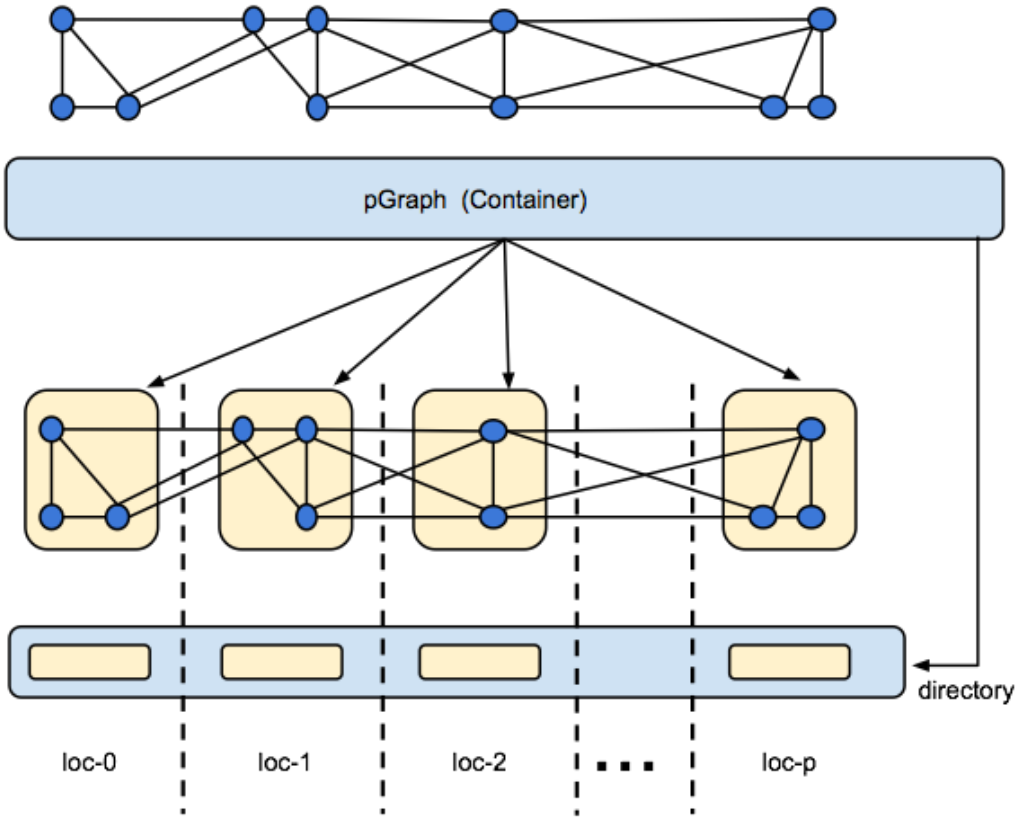
- Built using STAPL container framework
  - Shared-Object View
  - Global address-resolution via 2-level distributed-directory
  - Handles data-distribution and communication
- Asynchronous migration of elements
  - While algorithms are being executed
  - Requests made to vertex being migrated remain valid
- Scalable Rebalancing
  - Based on user-provided vertex-weights
  - Easily invoked, takes care of data-migration

Parasol

Graph Container

Overview

- Graph is partitioned into sub-graphs
- Sub-graphs distributed across the machine



stapl::graph

- Container of vertices and edges
- stapl::graph<DIRECTED/UNDIRECTED,
  MULTI/ NONMULTIEDGES,
  VertexProperty,
  EdgeProperty>
- Unique descriptors
  - No ghost nodes - no data replication
- Support for out-of-core storage [IPDPS 2015]
  - Disk paging at the subgraph level


```
graph<DIRECTED, MULTIEDGES, int> g(n);

g.add_vertex(n+1, 10);

const int size = g.num_vertices();

for (int i = 0; i < 10; ++i)
  g.add_edge(rand() % size, rand() % size);

const int degree = g[3].edges().size();

bool success = g.delete_edge(3,2);
```

Redistribution

- Specify a new distribution of a container
  - Migration of elements internally handled

```
auto dist = stapl::balance(n);

vector<int> v(n, dist);

auto new_dist = stapl::block(10);

v.redistribute(new_dist);
```

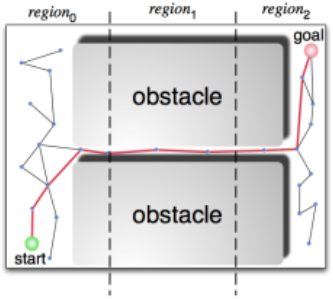
- Also repartitioning algorithms specifically for graphs

Parasol

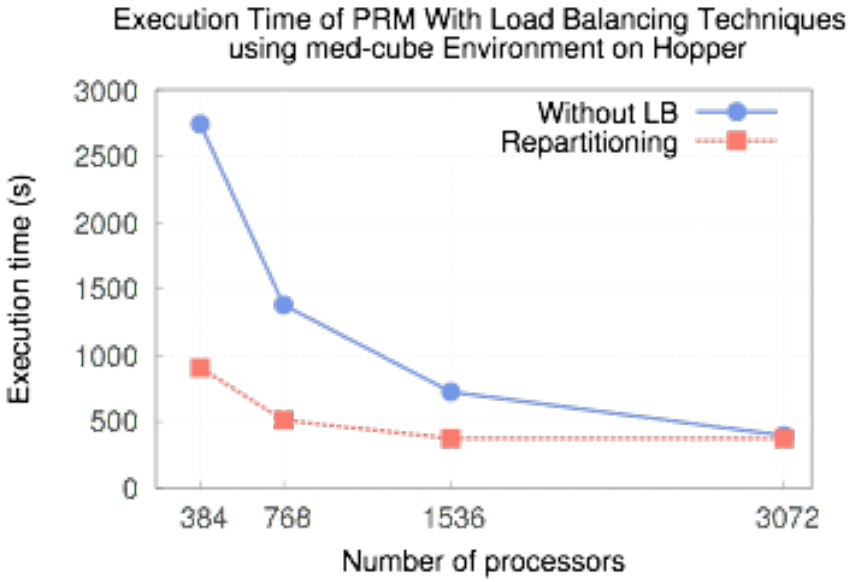
Redistribution

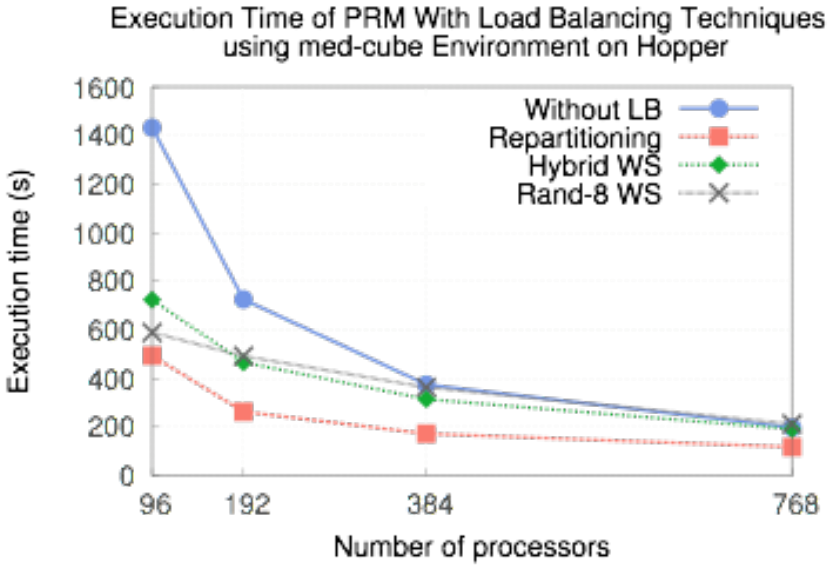
Motion Planning Case Study

- Parallel motion planning application
  - Computes feasible paths through an environment for a movable object



- Use redistribution to balance the application





Summary

- STAPL Container Framework
  - Suite of available containers
  - Simplify parallel container development
    - Inheritance, specialization and composition
  - Scalable performance

- Using a STAPL container almost as easy as using a sequential container

Presentation Outline

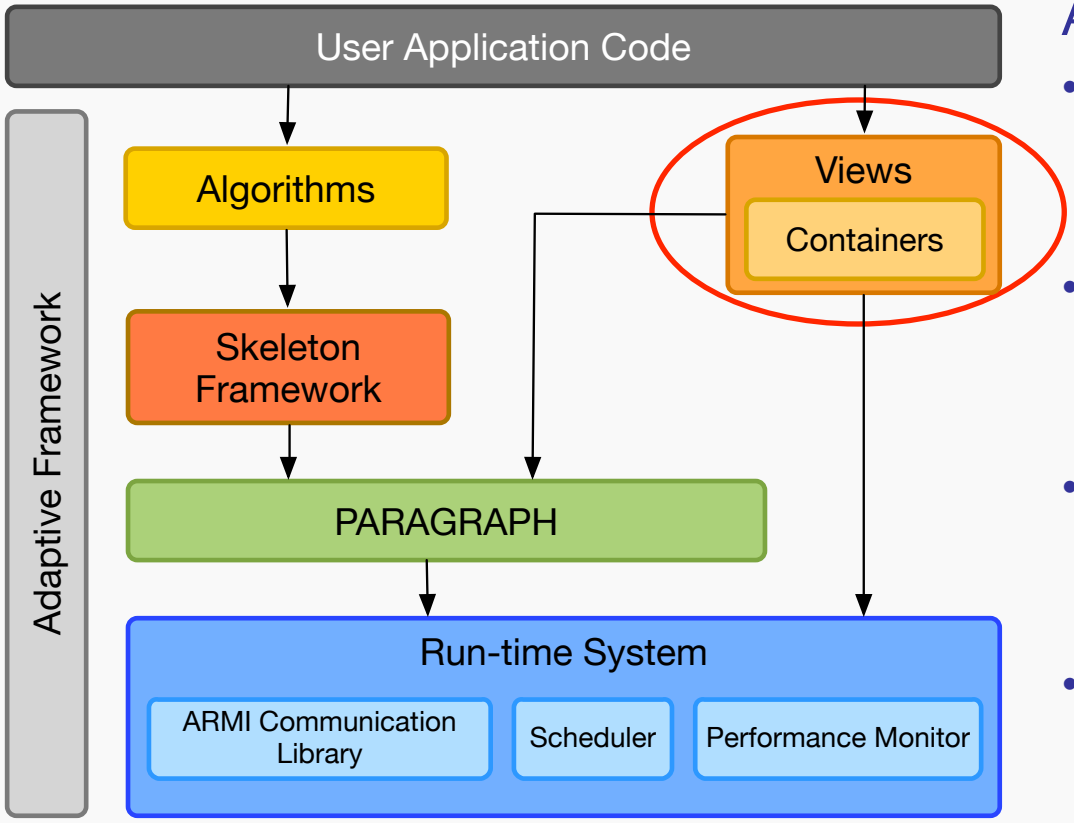
- Overview
- stapl::containers
- **stapl::views**
- stapl::algorithms
  stapl::skeletons
- stapl::paragraph
  stapl::runtime

Parasol

stapl::views

Timmie Smith

Views in STAPL



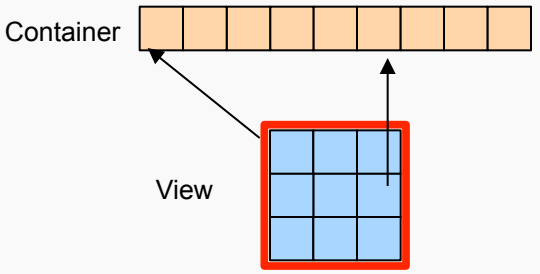
Abstract Data Types

- Adapt data structures to program requirements
- Increase level of abstraction used to express algorithms
- Enable parallelism by providing random access to distributed data
- Encapsulate information related to access pattern

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

View Specification

Parasol

- Partitioned collection of labeled elements

$$(c, d, f_v^c, o)$$

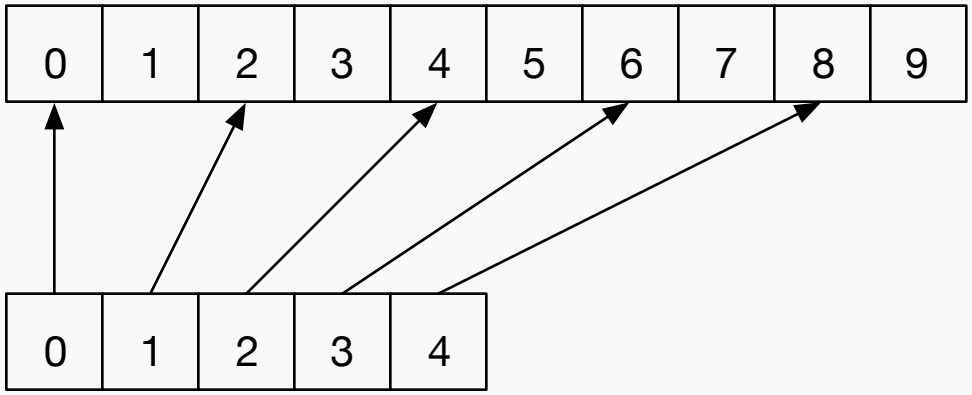
- Defined as tuple
  - Reference to a partitioned collection of elements (c)
  - Set of element identifiers (d)
  - Mapping function from View element identifiers
    to underlying collection identifiers
  - Set of operations

View Specification Example

(c, d, f_v^c, o)

Parasol

```
// collection of 10 elements
array c(10);

// set of identifiers [0,4]
indexed_domain d(5);

// function that multiplies input by two
multiply_by f(2);

// view[i] returns reference to c[i*2]
array_view view(c, d, f);

// subscript operation inherited from base class
class array_view : subscript<array_view>
```



Composed Views Example

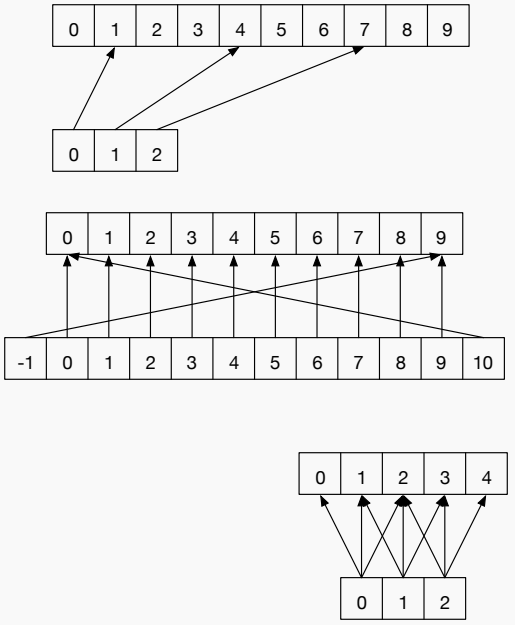
Parasol

- V-cycle multigrid algorithm to solve

$$\nabla^2 u = v$$

- Express computation with multidimensional Views

- strided_view(view, strides, offsets)
  - strided subset of elements
  - first element at offset
- periodic_boundary_view(view)
  - first element is last element of input
  - last element is first element of input
- overlap_view(view, overlaps)
  - same shape as input
  - element is original view element in center, surrounded by neighboring elements specified by overlaps



Composed Views Example

Parasol

```
void vcmg(View coarse, View fine)
{
  auto strided_stencil_view =
    strided_view(
      overlap_view(
        periodic_boundary_view(fine), tuple(1,1,1)
      ),
      tuple(2, 2, 2), tuple(0, 0, 0)
    );
  map([](double& point, View stencil) {
      point = map_reduce(multiply(), plus(),
        stencil,
        coefficient_view(0.5, 0.25, 0.125, 0.0625)
    )},
    coarse, strided_stencil_view
  );
}
```

vcmg without Views

Parasol

```
auto pb = [&](long i) {  return i == -1 ? n-1 : i == n ? 0l : I;  }
for (long j3 = 0; j3 < m; j3++) {
  long i3 = 2*j3;
  for (long j2 = 0; j2 < m; j2++) {
    long i2 = 2*j2;
    for (long j1 = 0; j1 < m; j1++) {
      long i1 = 2*j1;
      s(j1,j2,j3) = 0.5 * r(pb(i1),pb(i2),pb(i3))
+0.25 * (r(pb(i1-1),pb(i2),pb(i3))+r(pb(i1+1),pb(i2),pb(i3))
+r(pb(i1),pb(i2-1),pb(i3))+r(pb(i1),pb(i2+1),pb(i3))
+r(pb(i1),pb(i2),pb(i3-1))+r(pb(i1),pb(i2),pb(i3+1)))
+0.125 * (r(pb(i1),pb(i2-1),pb(i3-1))+r(pb(i1),pb(i2-1),pb(i3+1))
+r(pb(i1),pb(i2+1),pb(i3-1))+r(pb(i1),pb(i2+1),pb(i3+1))
+r(pb(i1-1),pb(i2-1),pb(i3))+r(pb(i1-1),pb(i2+1),pb(i3))
+r(pb(i1-1),pb(i2),pb(i3-1))+r(pb(i1-1),pb(i2),pb(i3+1))
+r(pb(i1+1),pb(i2-1),pb(i3))+r(pb(i1+1),pb(i2+1),pb(i3))
+r(pb(i1+1),pb(i2),pb(i3-1))+r(pb(i1+1),pb(i2),pb(i3+1)))
+0.0625 * (r(pb(i1-1),pb(i2-1),pb(i3-1))+r(pb(i1-1),pb(i2-1),pb(i3+1))
+r(pb(i1-1),pb(i2+1),pb(i3-1)+r(pb(i1-1),pb(i2+1),pb(i3+1)+r(pb(i1+1),pb
(i2-1),pb(i3-1))+r(pb(i1+1),pb(i2-1),pb(i3+1))
```

Composed Views Example

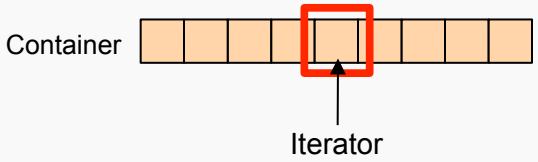
Parasol

```
void vcmg(View coarse, View fine)
{
  auto strided_stencil_view =
    strided_view(
      overlap_view(
        periodic_boundary_view(fine), tuple(1,1,1)
      ),
      tuple(2, 2, 2), tuple(0, 0, 0)
    );
  map([](double& point, View stencil) {
      point = map_reduce(multiply(), plus(),
        stencil,
        coefficient_view(0.5, 0.25, 0.125, 0.0625)
    )},
    coarse, strided_stencil_view
  );
}
```

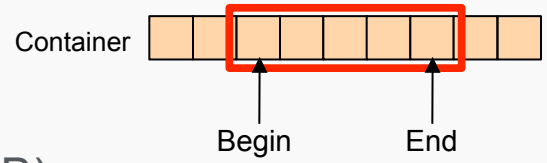
Iterators, Ranges, and Views

Parasol

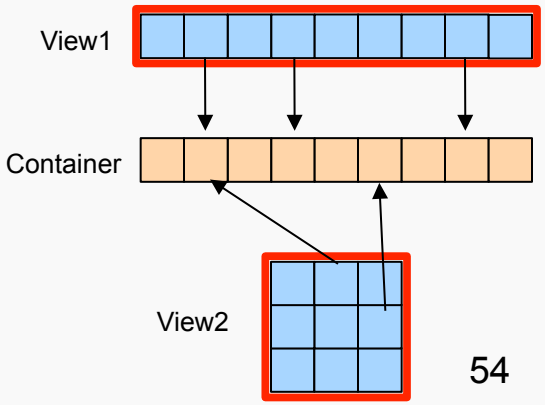
- **Iterators :** Abstraction of pointers
  - Provide element access and traversal.
  - Algorithms decoupled from containers (e.g., STL)
  - Naturally sequential, limited use in parallel environments



- **Ranges:** Combine two iterators (start and end)
  - Linear representation of data space
  - Split range and process subranges in parallel (e.g., TBB)
  - Some algorithms (e.g., blocked matmul) not easy/intuitive to implement



- **Views:** define an abstract data type
  - Generalize ranges to an arbitrary structure
  - Allow decoupling of container interface and storage
  - Provide container behavior (interface)
  - Work well in parallel environments



View Properties

Parasol

- Domain
  - Enumerable set of discrete element identifiers
  - Identifiers may be multi-dimensional
  - Operations on domains include set operations (union, intersection, etc.)
- Operations
  - Must be supported by the operations provided by underlying collection
- Inclusion
  - All elements accessed through View must be valid elements of c
  - View's domain defines set of elements that can be accessed
  - Element access requires mapping identifier to collection identifier

$$f_v^c[\text{dom}(v)] \subseteq \text{dom}(c)$$

View Properties

- Mutating Operations
  - Modify cardinality of underlying collection
  - Specific to collection being referenced
    - vector : insert, erase, push_back, pop_back
    - graph : add_vertex, remove_vertex, add_edge, remove_edge
  - Requires mapping function be injective (one-to-one)

Chaining and Coalescing

Parasol

- Define View over View
  - Chain of referenced collections created
  - Implies operations are forwarded until data collection is reached
  - Chain of indirection and forwarding may be costly

- Coalescing Transformation
  - Reduce overhead to that of a single View
  - Applies a substitution over the referenced underlying collection
  - Applies a substitution over the mapping function

Views in STAPL

- Most common source of partitioned collections is Container
  - View to Container relation is not one-to-one
  - View composition introduces many-to-one relationship
- View composition has compile-time and run-time components
  - Metaprogram substitutes mapping function types with composed mapping function type
  - In cases of arbitrary mapping functions, static substitution isn't possible
- Views may be invalidated
  - Operation performed directly on underlying collection
  - Operation performed on a second View over the same collection
  - Attempts to use invalid Views in a Algorithm result in runtime error

STAPL View Operations

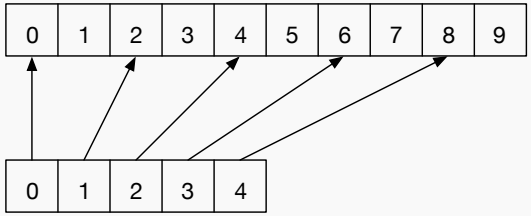
- Common operations
  - Implemented as base classes
  - Inherited by View classes
- Operations unique to a View are implemented as class member function

	read	write	[ ]	seq.	insert erase
array_view	●	●	●	●	
list_view	●	●		●	●
matrix_view	●	●	●		
graph_view	●	●			●
balanced_view	●		●	●	
native_view	●		●	●	
transform_view	●		○	○	
filter_view	●	●	●	●	
repeated_view	●		●	●	
overlap_view	●		●	●	
strided_view	●	●	●	●	
counting_view	●		●	●	
array_ro_view	●		●	●	
static_list_view	●			●	

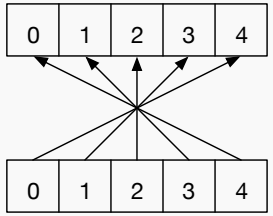
View Classification

- Modeling ADTs
  - array_view, list_view, etc.
- Modifying Access Pattern
  - strided_view, reverse_view, filter_view, overlap_view
- Generating Data
  - repeated_view, counting_view
- Modifying Data
  - transform_view
- Grouping Elements
  - balanced_view
  - native_view
- Restricted Operations
  - array_ro_view
  - static_list_view

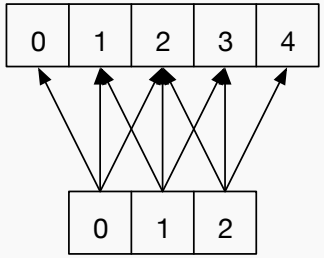
Strided View



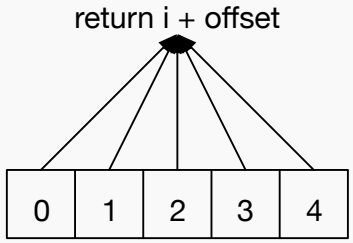
Reverse View



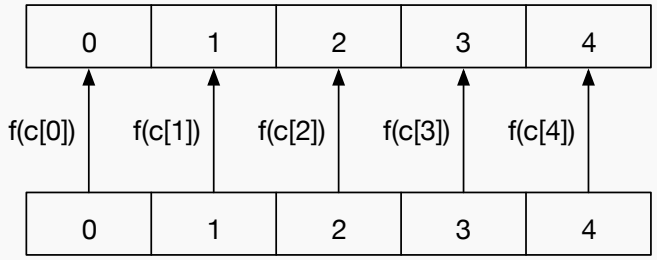
Overlap View



Counting View



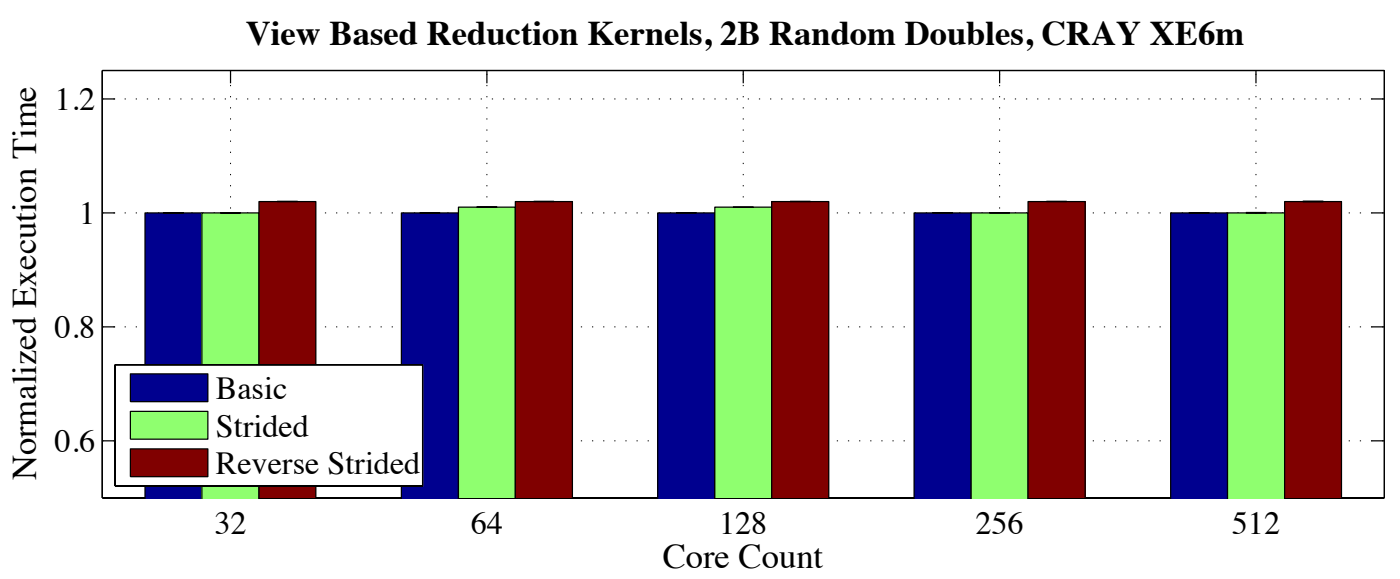
Transform View



Synthetic Evaluation of Views

Parasol

- Observes overhead introduced by use of Views
- Sum elements in array using three different traversals
  - array_view<array<double>>
  - strided_view<array_view<array<double>>
  - reverse_view<strided_view<array_view<array<double>>>>
- Reference is same access pattern on Container

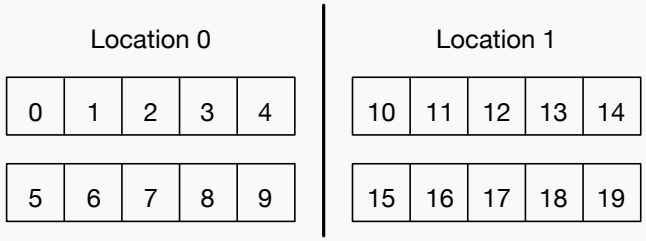


View Application: Specifying Data Distribution

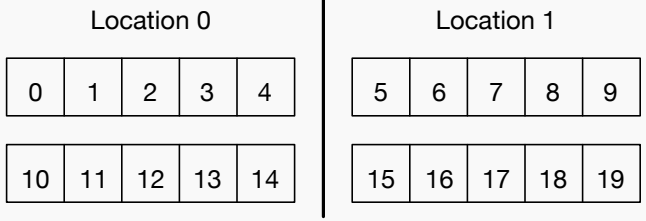
Parasol

- Data distribution
  - Assignment of container elements to locations for storage
  - Two step process
    - Assign elements to partitions
    - Assign partitions to locations
- Common distributions
  - block, block-cyclic, balance
  - could reuse code to perform assignment
- Arbitrary distributions
  - ad hoc implementation for each one, or
  - use a common framework

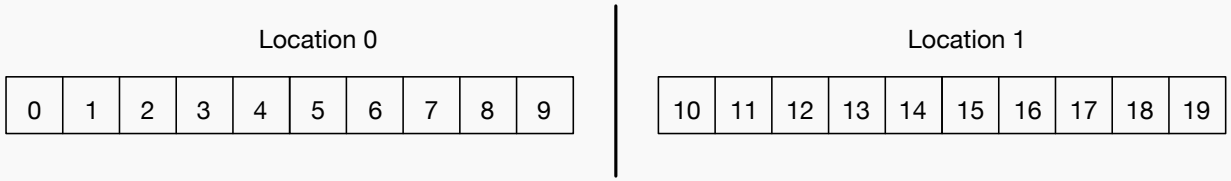
Block



Block Cyclic



Balance



Data Distribution using View Terms

Parasol

- Three sets of labeled elements
  - Element Ids of Container – GIDs
  - Partition Ids – CIDs
  - Location Ids – LIDs
- Two mapping operations
  - GID ➔ CID
  - CID ➔ LID

- Data distribution can be expressed using chained views

Views for Data Distribution

Parasol

(c, d, f_v^c, o)

- Partitioning View
  - c = set of partition ids (CIDs)
  - d = container element ids (GIDs)
  - f = changes across distributions

- Mapping View
  - c = set of location ids (LIDs)
  - d = set of partition ids (CIDs)
  - f = changes across distributions

Block Distribution

Parasol

```
block(n, block_size, lvl){

  system_view sys_view(lvl);

  nb = n/block_size;
  bpl = nb/sys_view.size();

  return distribution_spec (
      mapping_view_type(sys_view, indexed_domain(nb), block_map(bpl)),
      indexed_domain(n),
      block_map(block_size));
}
```

Block Cyclic Distribution

Parasol

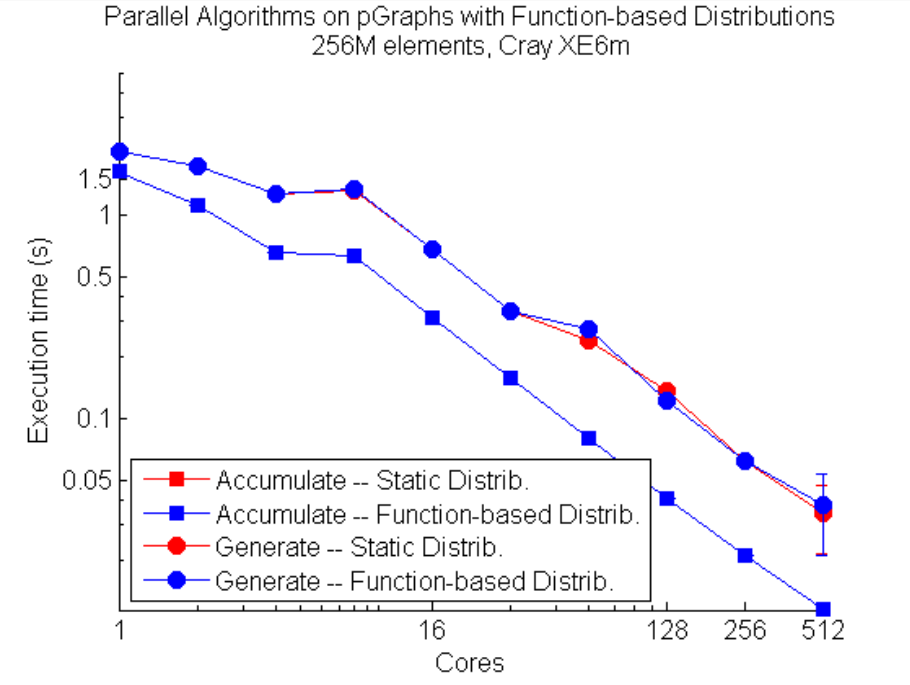
```
block_cyclic(n, block_size, lvl){

  system_view sys_view(lvl);

 nb = n/block_size;



  return distribution_spec(
      mapping_view(sys_view, indexed_domain(nb), cycle_map(sys_view.size())),
      indexed_domain(n),
      block_map(block_size));
}
```

Distribution changed by changing mapping function used for CID➔LID.

View-based Data Distribution Performance

Parasol

- Container construction has overhead
  - View-based distributions currently assumed to be arbitrary
  - Requires a distributed computation to determine which bContainers a location should construct
- After construction Container performs as well as statically distributed Containers



View Summary

- Views as ADTs
  - Allow Containers to be adapted to meet Algorithm requirements
- Increase level of abstraction used to express parallel algorithms
- Used to express data distributions of Containers

Presentation Outline

Parasol

- Overview
- stapl::Containers
- stapl::Views
- **stapl::Algorithms**
  stapl::skeletons
- stapl::paragraph
  stapl::runtime

stapl::algorithms

Adam Fidel and Mani Zandifar
Texas A&M University, Feb. 2015

Outline

- Algorithms
  - STL algorithms in STAPL
  - graph algorithms
- The STAPL Skeleton Framework
  - Algorithm Specification
  - Algorithm Execution
- Experimental Results
- Conclusion

STL Algorithms in STAPL

STL and STAPL Algorithms

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


iota(c.begin(), c.end(), 0);
fill(c.begin(), c.end(), 1);
generate(c.begin(), c.end(), gen());

int r1 = accumulate(c.begin(), c.end(), 0);

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

int r2 = inner_product(c.begin(), c.end(),
                       c.begin(), 0);
```

STL Algorithms

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

iota(view, 0);
fill(view, 1);
generate(view, gen());

int r1 = accumulate(view, 0);

partial_sum(view, view);

int r2 = inner_product(view,
                       view, 0);
```

STAPL Algorithms

Parasol Algorithms Coverage

	Algorithm	STL	STAPL
Generalized Numeric Algorithms	accumulate	✔	✔
	adjacent_difference	✔	✔
	inner_product	✔	✔
	iota	✔	✔
	partial_sum	✔	✔
	weighted_inner_product	-	✔
	weighted_norm	-	✔
Non-modifying Sequence Operations	all_of, none_of, any_of	✔	✔
	for_each	✔	✔
	count, count_if	✔	✔
	mismatch	✔	✔
	equal	✔	✔
	find, find_if, find_if_not	✔	✔
	find_end, find_first_of	✔	✔
	adjacent_find	✔	✔
	search, search_n	✔	✔

Parasol Algorithms Coverage

	Algorithm	STL	STAPL
Mutating Sequence Operations	copy, copy_n, copy_if	✔	✔
	copy_backward	✔	✔
	move, move_backward	✔	-
	fill, fill_n	✔	✔
	transform	✔	✔
	generate, generate_n	✔	✔
	keep_if	-	✔
	remove, remove_if	✔	✔
	remove_copy, remove_copy_if	✔	✔
	replace, replace_if	✔	✔
	replace_copy, replace_copy_if	✔	✔
	swap, swap_ranges	✔	✔
	iter_swap	✔	N/A
	reverse, reverse_copy	✔	✔
	rotate, rotate_copy	✔	✔
	random_shuffle, shuffle	✔	✔
	unique, unique_copy	✔	✔

Algorithms Coverage

	Algorithm	STL	STAPL
Sorting Operations	is_sorted, is_sorted_until	✔	✔
	sort	✔	✔
	partial_sort, partial_sort_copy	✔	✔
	stable_sort	✔	✔
	nth_element	✔	✔
	n_partition	-	✔
	sample_sort	-	✔
	radix_sort	-	✔
Binary Search Operations	lower_bound	✔	✔
	upper_bound	✔	✔
	binary_search	✔	✔
	equal_range	✔	✔

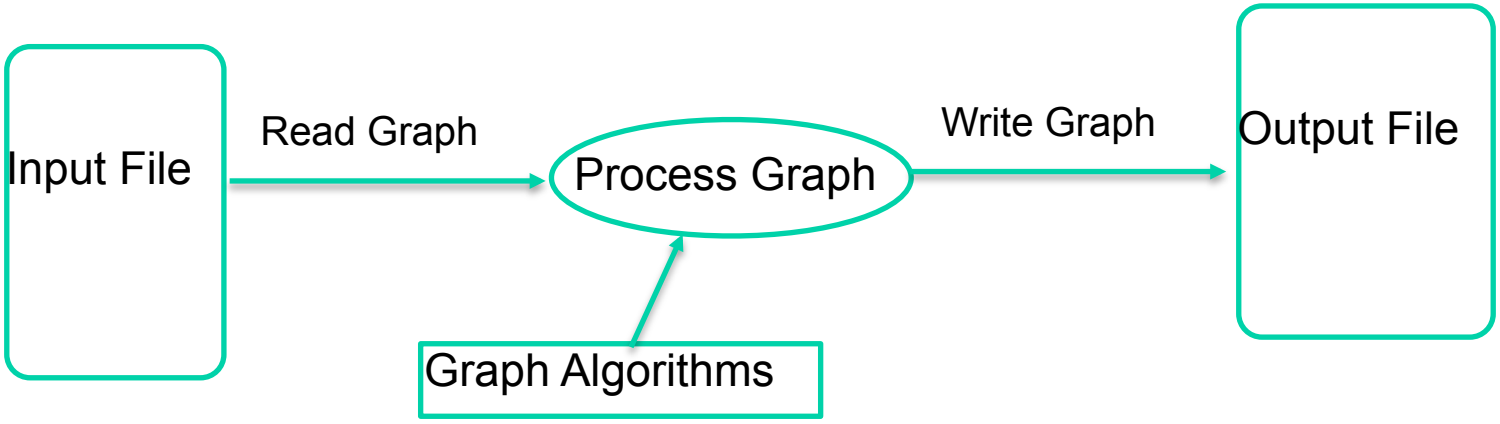
Algorithms Coverage

	Algorithm	STL	STAPL
Partitioning Operations	is_partitioned	✔	✔
	partition, partition_copy	✔	✔
	stable_partition	✔	✔
	partition_point	✔	✔
Minimum/Maximum Operations	max, min	✔	✔
	max_element, min_element	✔	✔
	minmax, minmax_element	✔	✔
	lexicographical_compare	✔	✔
	is_permutation	✔	✔
	next_permutation, prev_permutation	✔	✔

Graph Algorithms

pGraph Algorithms

```
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();
```



Current Approaches

- Many domains model their data as graphs
- Current strategies:
  - Level-synchronous
    - Bulk-synchronous model, good for few iterations
  - Asynchronous
    - Would be ideal, except for redundant work

- Observation
  - Performance highly dependent on graph structure

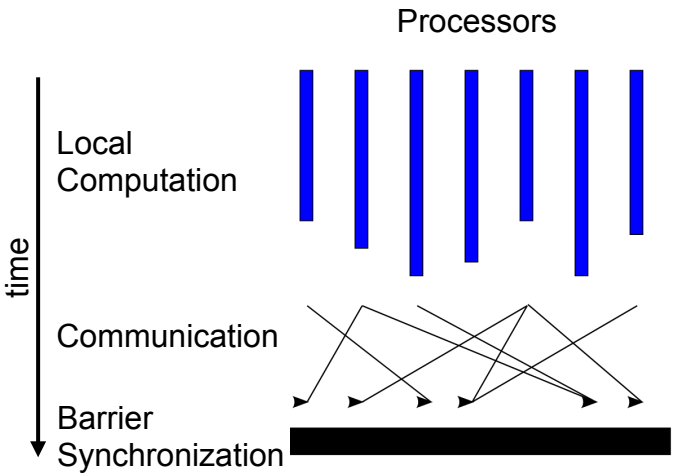
Parallelization of Graph Algorithms

Level-Synchronous Approach

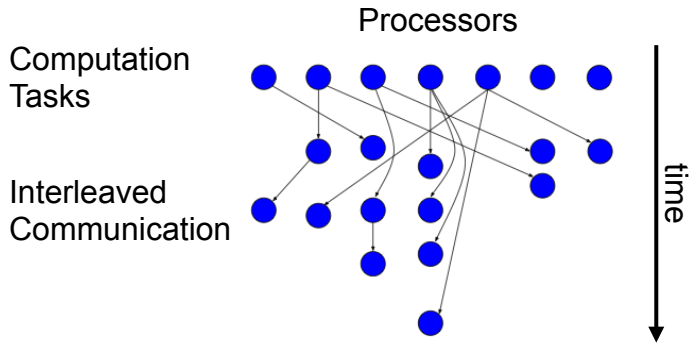
- BSP-model iterative computation
- Global synchronization after each level, no redundant work

Asynchronous Approach

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

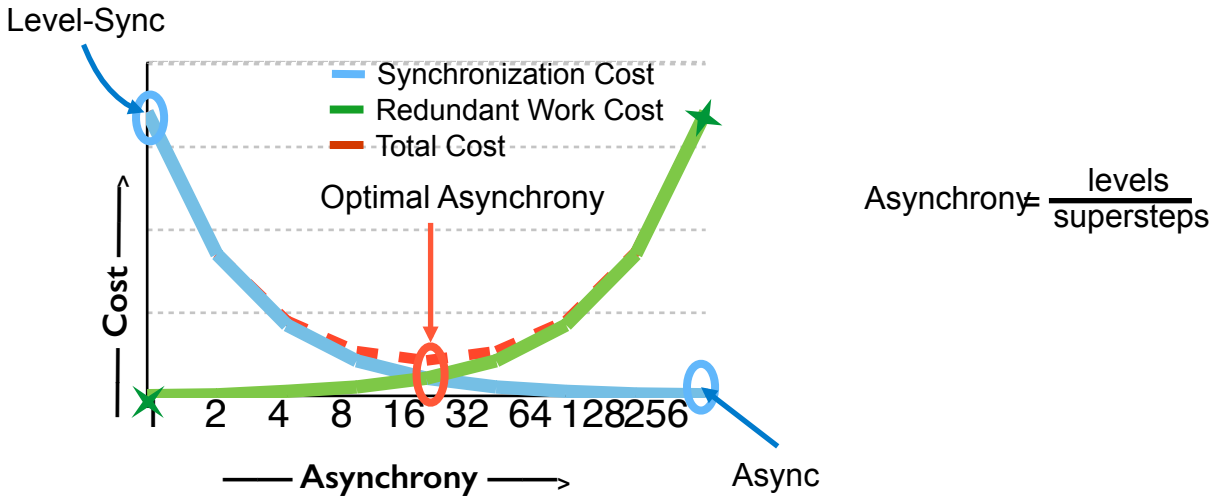


A single BSP Superstep



Asynchronous Task Execution

Parallelization of Graph Algorithms



k-Level Asynchronous Model[1]

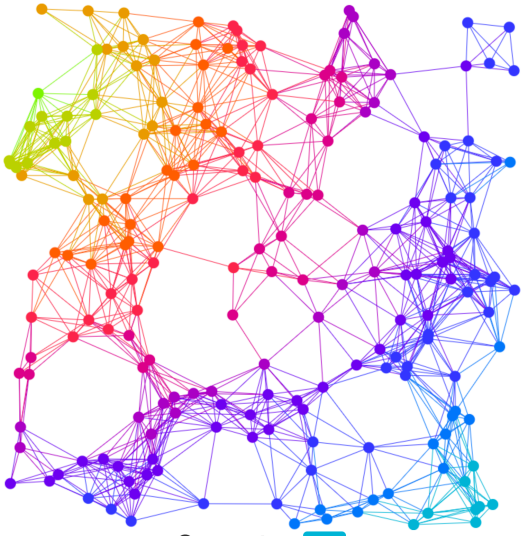
- k defines depth of superstep (KLA-SS)
- Unifies existing models
  - k=1: Level-synchronous
  - k=d: Asynchronous

[1] *KLA: a new algorithmic paradigm for parallel graph computations*, Harshvardhan, Adam Fidel, Nancy M. Amato, Lawrence Rauchwerger, PACT 2014, **Best Paper Award**

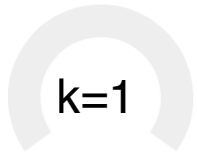
The *k*-Level Asynchronous (KLA) Model

- $k \geq d$ controls the asynchrony by controlling the depth of the superstep
- Diameter = 11

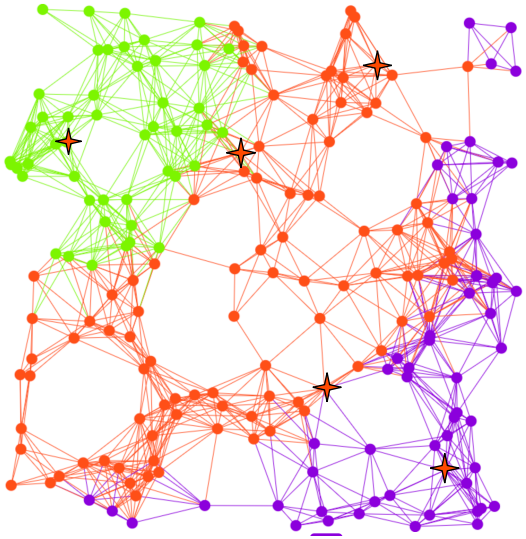
Level-synchronous



Supersteps: **11**

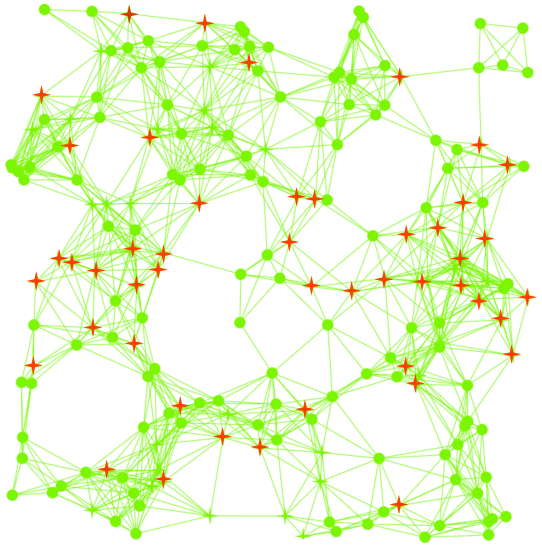


KLA

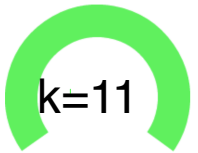


Supersteps: **3**

Asynchronous



Supersteps: **1**



Expressing Algorithms in KLA

Example - Breadth-First Traversal

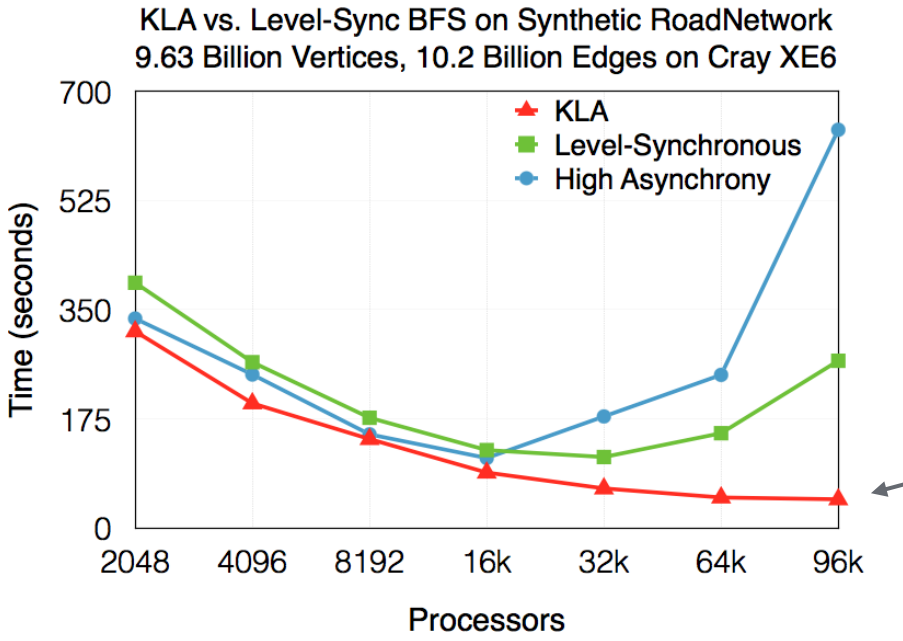
- KLA algorithms are expressed with two operators:
  - Process a vertex:

```
bool bfs_vertex_op(Vertex v)
  if (v.color == GREY)
    v.color = BLACK;
    visit_all_neighbors(v, bfs_neighbor_op(v.distance+1));
    return true;
  else
    return false;
```

  - Visit and update its neighbors:

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

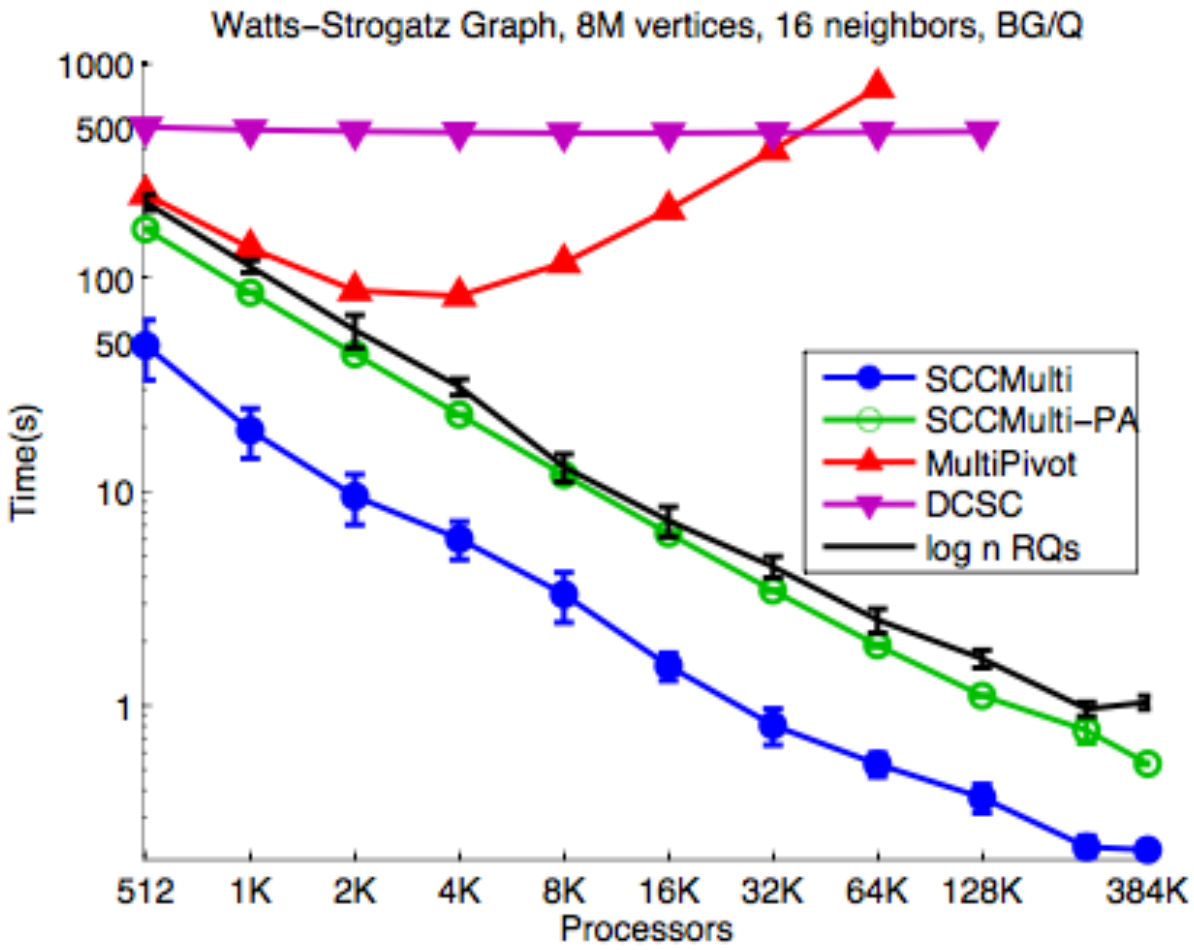
Parasol k-Level Asynchronous BFS



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

- Usage of the KLA strategy in SCCMulti



STAPL Skeleton Framework

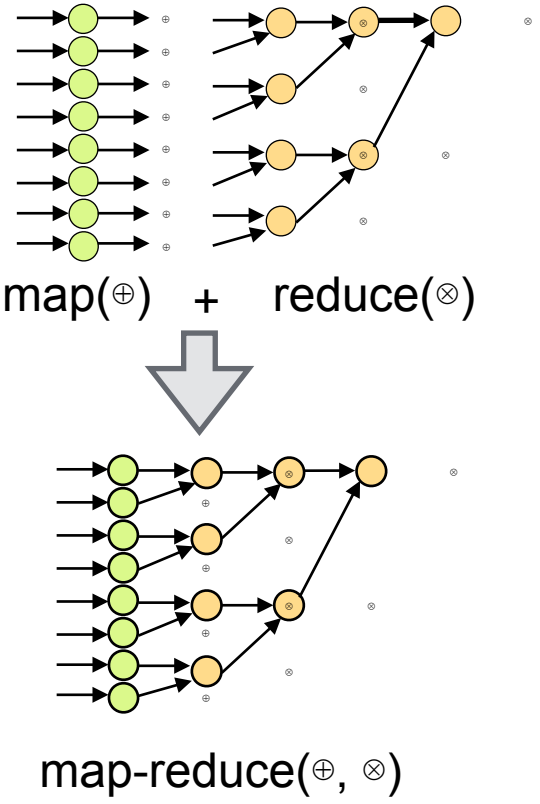
Algorithmic Skeletons

- Algorithmic skeletons[1][2] (e.g., map, map-reduce)
  - portable and implementation-independent representation
  - polymorphic higher-order functions
    - map f $[a_1, a_2, \ldots a_n] = [f(a_1), f(a_2), \ldots, f(a_n)]$
  - inherently composable
  - allow formal analysis and transformations

- Limitation of current skeletons libraries implementations
  - composition is not a first-class feature, e.g., map-reduce($\oplus$, $\otimes$) = reduce($\otimes$) $\circ$ map($\oplus$)
    - requires reimplementaion or global synchronization
  - nesting composition is not supported, e.g., map(map($\oplus$))
  - mostly limited to shared-memory systems

[1] *Algorithmic Skeletons: Structured Management of Parallel Computation*, Murray I. Cole, 1991
[2] *Patterns and Skeletons for Parallel and Distributed Computing*, Fethi A. Rabhi, Sergei Gorlatch, 2002

- Agnostic to shared and distributed system
  - algorithmic skeletons (e.g., map and map-reduce)
  - data flow graphs[1]
    - Internal Representation (IR) of skeletons
    - explicit parallelism in the representation



- Makes composition a first-class feature
  - flow-based programming[2]
    - internal representation of skeleton composition
    - composition = point-to-point dependencies
    - no need for reimplementation
    - no need for global synchronization

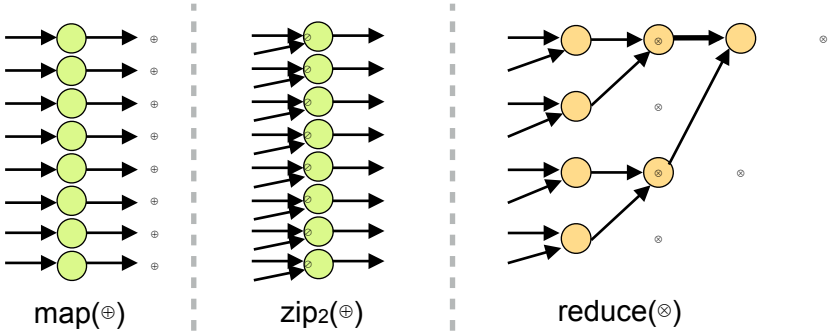
    - no need for global synchronization

[1] *Dataflow Supercomputers,* Jack B. Dennis, 1980
[2] *Flow-based Programming, 2nd Edition, J. Paul Morrison,* 2011

Algorithm Specification

New IR for Skeletons

$map(\oplus)\ [a_1, \ldots, a_n] = [\oplus(a_1), \ldots, \oplus(a_n)]$
$zip_2(\oslash)\ [a_1, \ldots, a_n][b_1, \ldots, b_n] = [\oslash(a_1,b_1), \ldots, \oslash(a_n, b_n)]$
$reduce(\otimes)\ [a_1, \ldots, a_n] = a_1 \otimes \ldots \otimes a_n$
$scan(\otimes)\ [a_1, \ldots, a_n] = [a_1, a_1 \otimes a_2, \ldots, a_1 \otimes a_2 \otimes \ldots \otimes a_n]$

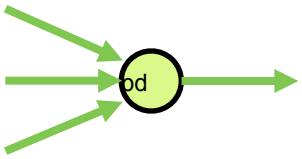


Composed from existing skeleton

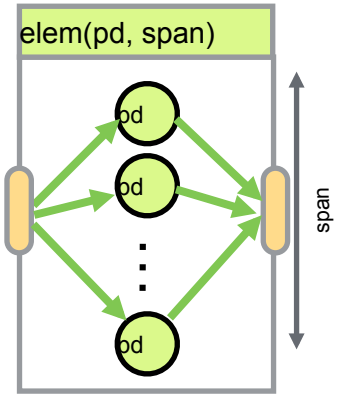
map-reduce(⊕, ⊗) = reduce(⊗) ∘ map(⊕)
zip-reduce(⊘, ⊗) = reduce(⊗) ∘ zip(⊕)
allreduce(⊗) = broadcast ∘ reduce(⊗)

Algorithm Specification

New IR for Skeletons

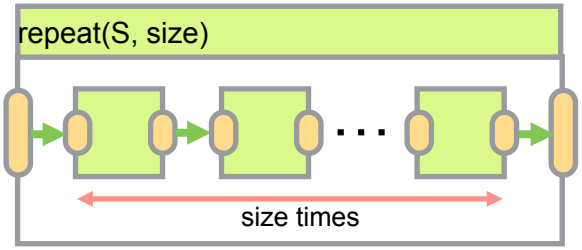


parametric dependency

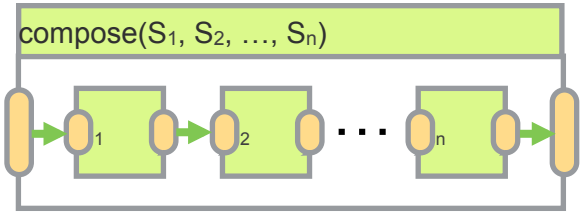


elem operator

Composed from existing skeleton

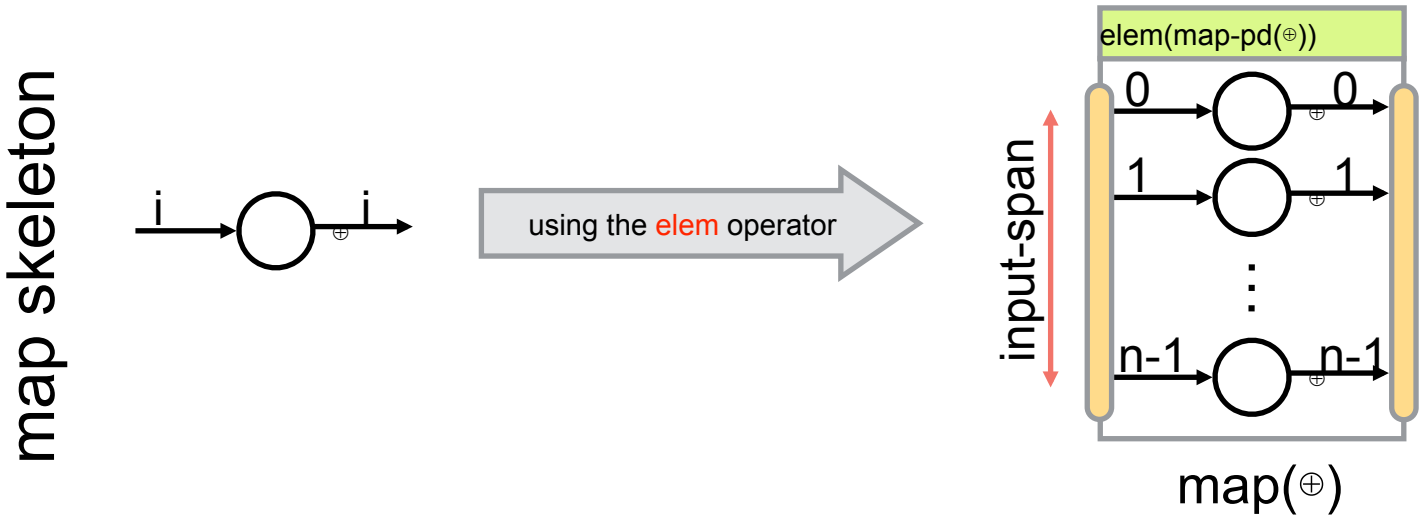


repeat operator



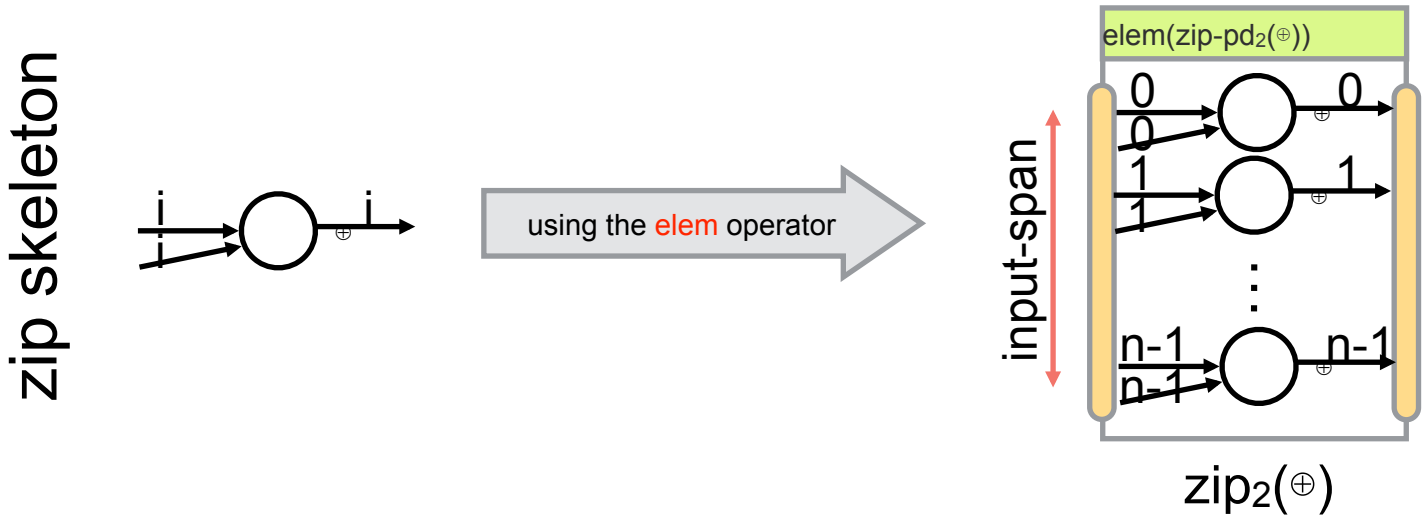
compose operator

New IR for Skeletons
Example - Map Skeleton




```
// Definition in the STAPL Skeleton Framework
template <typename Op>
auto map(Op op) {
  return elem<input_span>(map_pd(op));
}

// Usage
algorithm_executor<> exec;
exec.execute(map(increment<double>()), input_view);
```

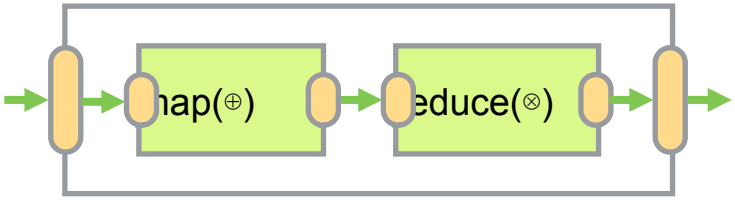
New IR for Skeletons
Example - Zip Skeleton




```
// Definition in the STAPL Skeleton Framework
template <int i, typename Op>
auto zip(Op op) {
  return elem<input_span>(zip_pd<i>(op));
}

// Usage
algorithm_executor<> exec;
exec.execute(zip<2>(plus<double>()),
             input1_view, input2_view);
```

New Skeletons from Existing Ones
Examples - Simple Functional Composition

- Examples
  - map-reduce(⊕, ⊗) = compose(map(⊕), reduce(⊗))
    = reduce(⊗) ○ map(⊕)




```
// Definition in the STAPL Skeleton Framework
template <typename MapOp, typename ReduceOp>
auto map_reduce(MapOp map_op, ReduceOp reduce_op) {
  return compose(map(map_op), reduce(reduce_op));
}
```

Inner Product Algorithm

- $[a_1, a_2, \ldots, a_n] \cdot [b_1, b_2, \ldots, b_n]^{-1} = a_1*b_1 + a_2*b_2 + \ldots + a_n*b_n$
- inner-product = zip_2-reduce(*, +)

- In STAPL

```
template <typename V1, typename V2>
auto inner_product(V1 v1, V2 v2) {
  // create the skeleton
  using T = typename V1::value_type;
  auto inner_product_s = zip_reduce<2>(multiply<T>(), plus<T>());

  // execute the skeleton on the given views
  algorithm_executor<> exec;
  return exec.execute<T>(inner_product_s, v1, v2);
}
```

New Skeletons from Existing Ones
Examples - Complex Functional Composition

- Simple functional composition becomes less expressive in complex compositions.
- Composition mini-language
  - potentially avoids repetition and improves readability[1]
  - defines the input/output mappings of skeletons in a composition
  - parsed and converted to C++ header files by our Python parser

Grammar

```
composition := input = [Var+]
               Statement+
               output = [Var+]
Statement := Var = Var [Var+]
Var := [a-Z0-9]+
```

Legends

BLUE = views
BLACK = skeleton
RED = keywords

[1] *Types and Programming Languages,* Benjamin C. Pierce, 1992

Bucket Sort in the NAS IS Benchmark

Grammar

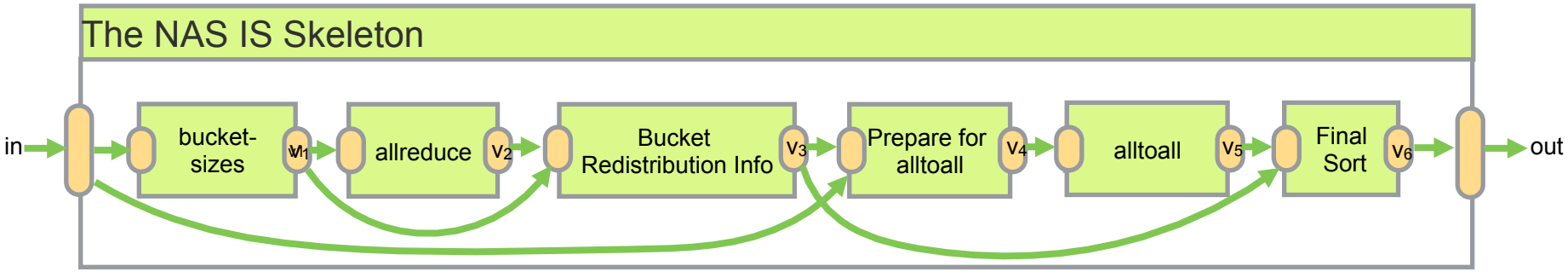
```
composition := input = [Var+]
               Statement+
               output = [Var+]
Statement := Var = Var [Var+]
Var := [a-Z0-9]+
```

Legends

BLUE = views
BLACK = skeleton
RED = keywords

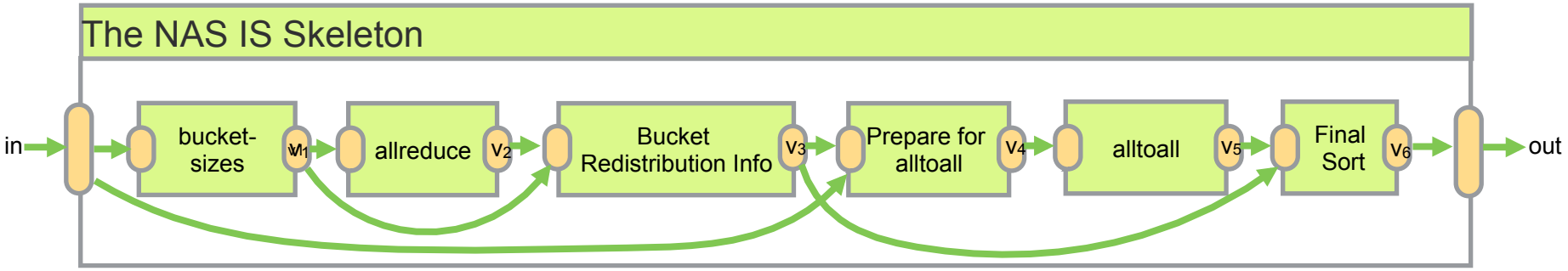
```
input = [in]
v1 = map(bucket-sizes) [in]
v2 = allreduce() [v1]
v3 = map(bucket-redistr-info) [v2]
v4 = zip<3>(prepare-for-alltoall) [v3 v2 in]
v5 = alltoall [v4]
v6 = zip<2>(final-sort) [v3 v5]
output = [v6]
```

Parasol Bucket Sort in the NAS IS Benchmark




```
input = [in]
v1 = map(bucket-sizes) [in]
v2 = allreduce() [v1]
v3 = map(bucket-redistr-info) [v2]
v4 = zip<3>(prepare-for-alltoall) [v3 v2 in]
v5 = alltoall [v4]
v6 = zip<2>(final-sort) [v3 v5]
output = [v6]
```

Parasol Bucket Sort in the NAS IS Benchmark

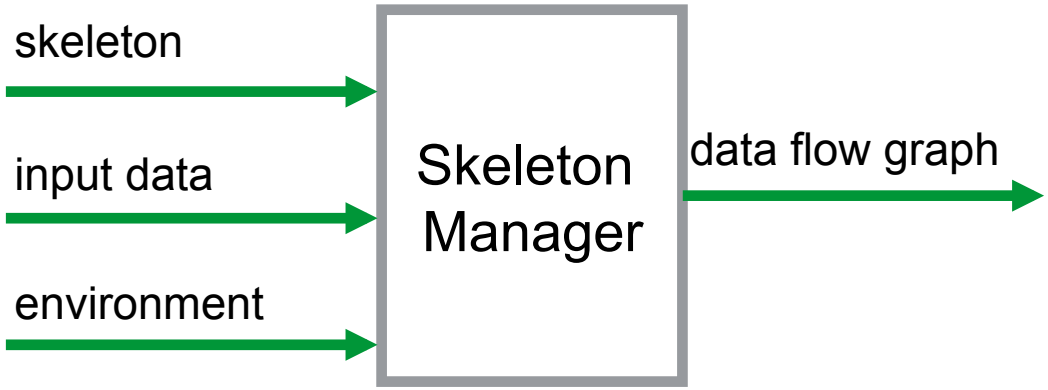


python parser

```
input = [in]
v1 = map(bucket-sizes) [in]
v2 = allreduce() [v1]
v3 = map(bucket-redistr-info) [v2]
v4 = zip<3>(prepare-for-alltoall) [v3 v2 in]
v5 = alltoall [v4]
v6 = zip<2>(final-sort) [v3 v5]
output = [v6]
```

```
compose<portmaps::bucket_sort>(
  map(bucket-sizes),
  allreduce(stapl::plus<T>()),
  map(bucket-redistr-info),
  zip<3>(prepare-for-alltoall),
  alltoall(),
  zip<2>(final-sort));
```

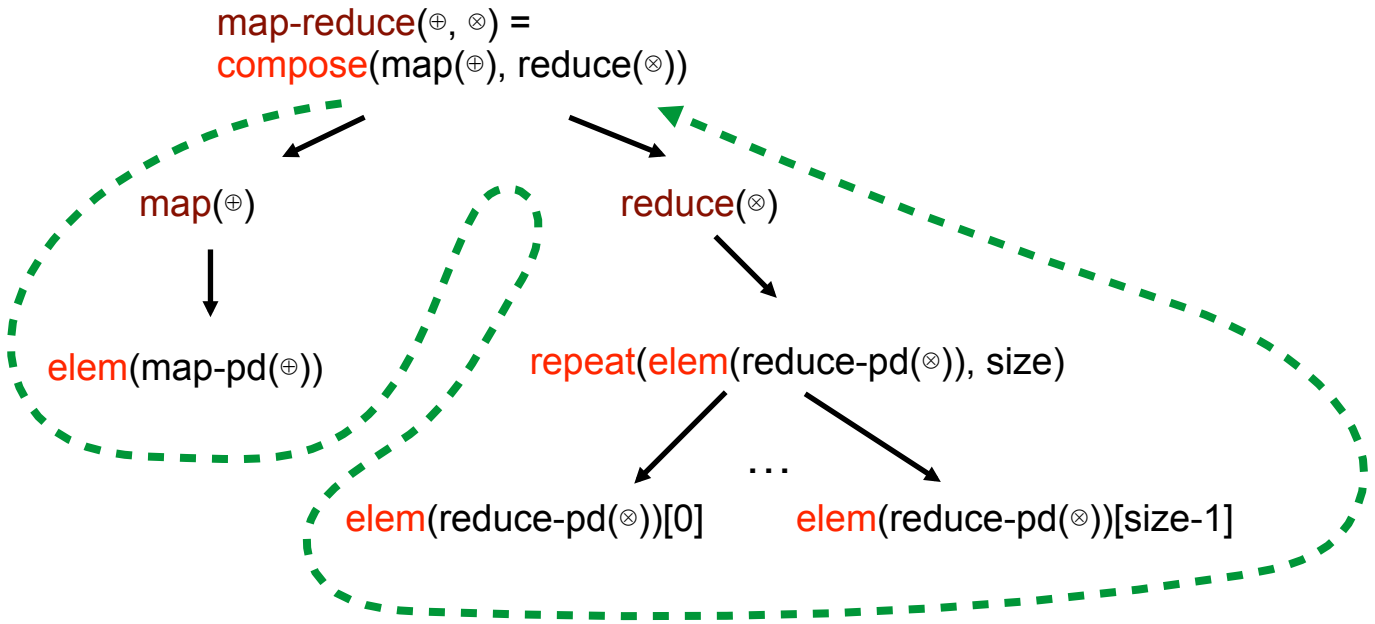
- Evaluation of skeletons as data flow graphs
- Orchestrated by the Skeleton Manager
  - called the spawning process



  - environment - defines the meaning of data flow nodes
    - e.g., taskgraph_env - creates task graphs

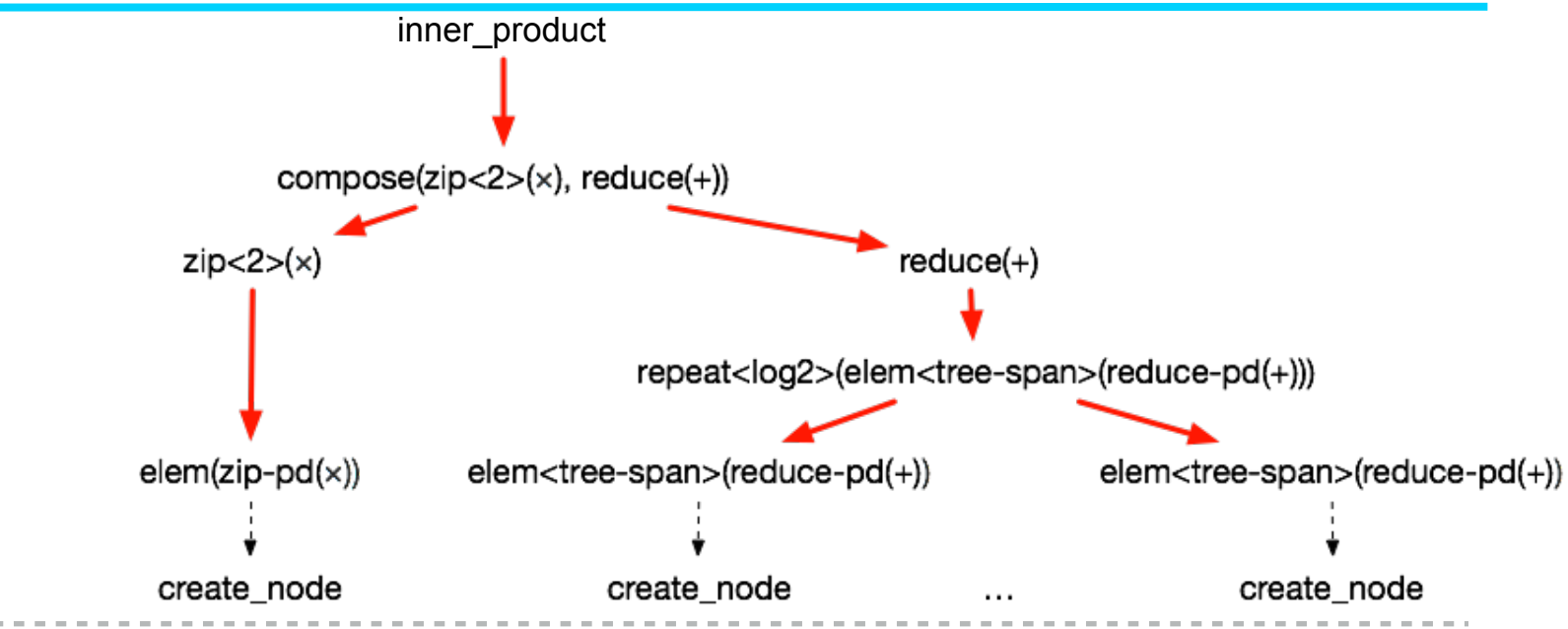
Algorithm Execution
From Specification to Execution

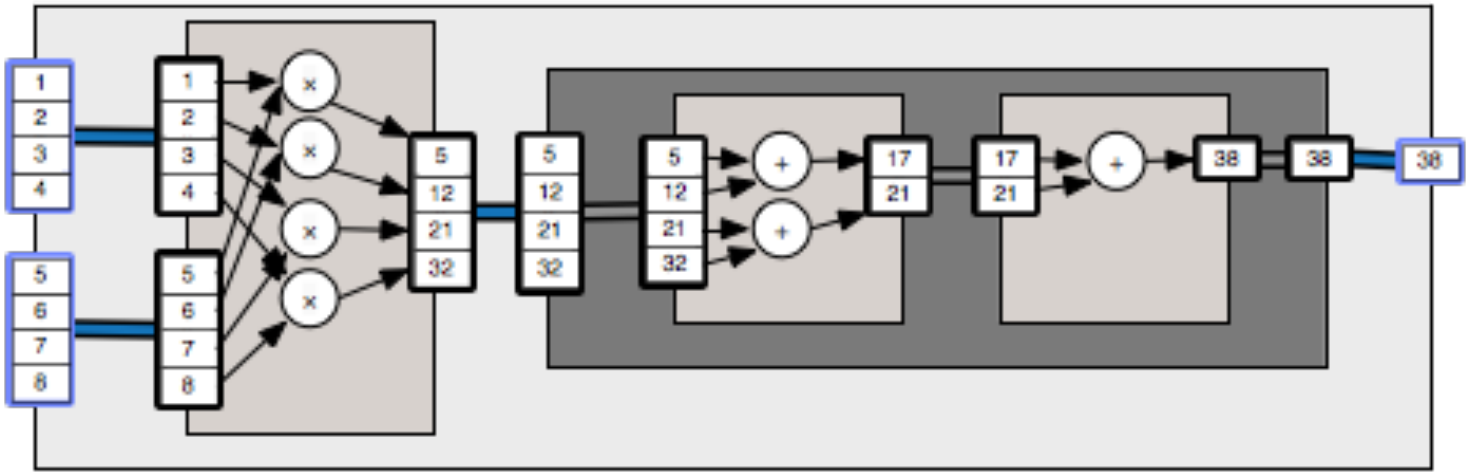
- Evaluation of skeletons as data flow graphs
- Orchestrated by the Skeleton Manager
- A pre-order depth first traversal



Parasol

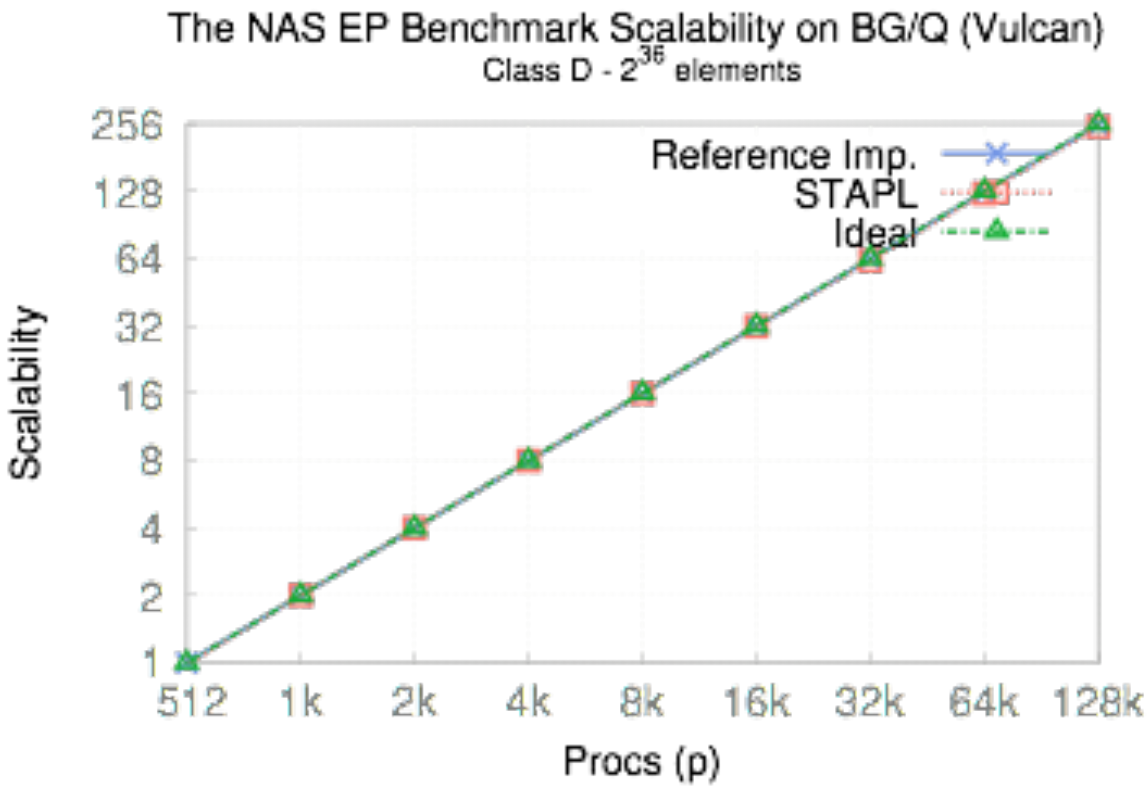
Algorithm Execution
From Specification to Execution





Experimental Results

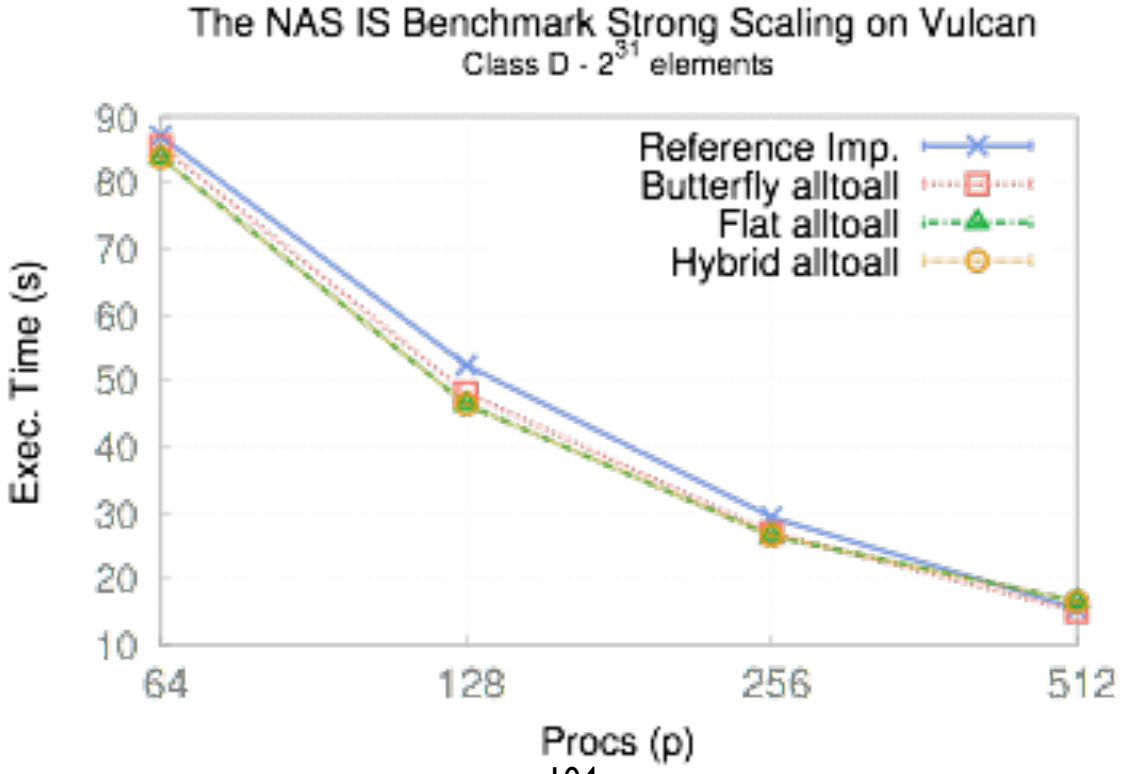
The NAS EP Benchmark (map-reduce, ~68B elements)



Parasol

Experimental Results
The NAS IS Benchmark (class D, ~2B elements)

- Skeletons - asynchronous data flow graph execution
- Reference Implementation - BSP execution



Summary

- A skeleton framework agnostic to shared and distributed-memory systems
- Skeletons as data flow graphs
  - data-size independent representation
  - compositions as point-to-point dependencies
  - no need for global synchronization
- Expressivity and composability
- Ease of extensibility
- Portability
- Efficiency and scalability

Presentation Outline

- Overview
- Containers
- Views
- Algorithms & Skeletons
- **PARAGRAPH and Runtime**

stapl::paragraph
stapl::runtime

Nathan Thomas
nthomas@cse.tamu.edu

What is a Runtime System

Parasol

- A runtime system is a software component that provides essential services to a language or a library and the applications that use the latter

  Andrew W. Appel, “A Runtime System”, 1990

- Runtime systems abstract the underlying platform
  - Architecture + Language Runtime + Operating System = platform
  - Even if the platform is the same, available hardware resources differ
- A good runtime system for parallel applications
  - Virtualizes the platform
    - e.g. provides a unified communication layer for both distributed and shared memory systems
  - Portable and extensible
    - A runtime is a collection of parts that need to be ported to a new platform
  - Configurable
    - Can take advantage of all platform capabilities
  - Adaptive
    - Available resources are not the same across application executions

Why do we need a runtime system?

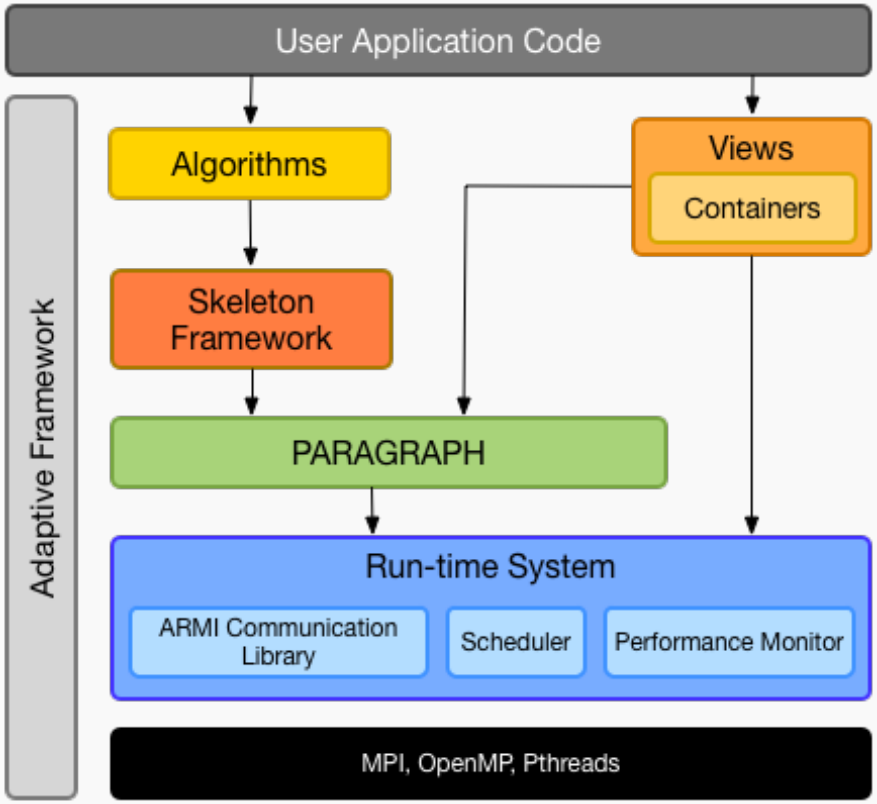
Parasol

- Portability
  - Abstract the underlying machine. Simplify higher level components' implementations.
- Virtualization
  - For nested parallelism, need to create homogeneous execution environments on portions of the system.
- Adaptive
  - Aware of the high level programming model. Provide interfaces for transfer of contextual information to drive optimizations and customizations.

The STAPL Runtime System

Parasol

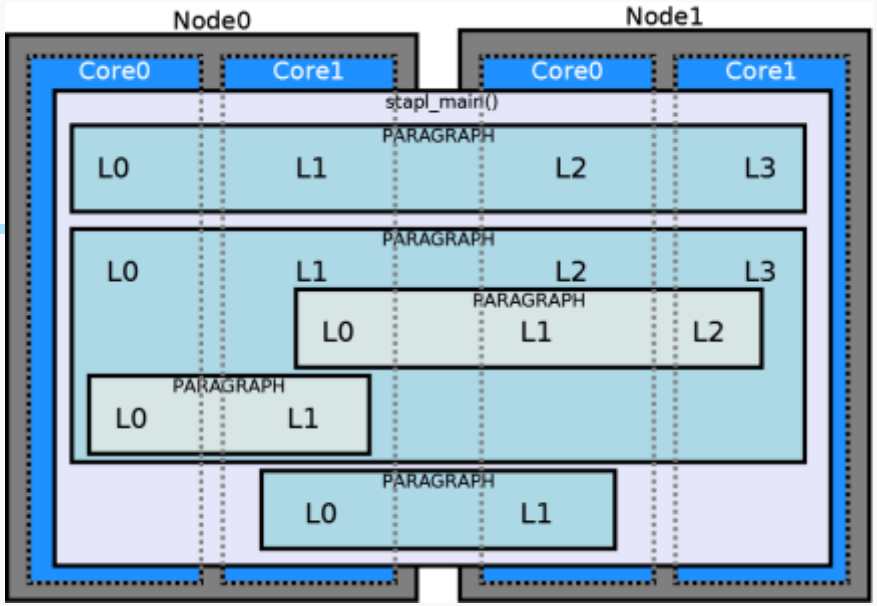
- *Most STAPL developers do not interact directly with it.*
- **PARAGRAPH** – Runtime representation of dependence graph. Coordinate creation, dependence enforcement, task placement, etc.
- **ARMI** – Communication primitives:
  - Remote method invocation
  - Data marshalling / serialization
  - future / promises.
- **Scheduler / Dispatcher** – Manage the ordering and execution of runnable tasks.



STAPL Runtime Overview

- PARAGRAPH
  - Mix of functional and imperative programming.
    - Workfunction return value propagated as data flow.
    - Task can mutated data through views, consistency guaranteed.
  - Concurrent creation and execution of task graph.
- ARMI
  - Lightweight, asynchronous RMIs aggregated.
  - Transparent use of shared memory when available.
- Scheduler
  - Customizable (PDT)

Execution Model



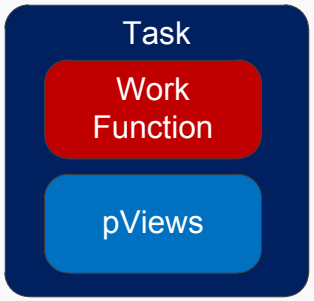
- S/MPMD model.
  - Implicit parallelism (e.g., MPI).
  - Task and Data Parallelism.
  - Nested parallelism through recursive PARAGRAPH invocation.
- Each PARAGRAPH executes on a number of locations.
  - A **location** is an isolated address space with associated execution capabilities (e.g. thread).
- Users create distributed objects (**p_objects**) on locations.
  - PARAGRAPHs, containers, and views are all p_objects.
  - Locations communicate with each other through RMIs to p_objects.
    async_rmi(destination, p_object&, member_function, arg0, ...);

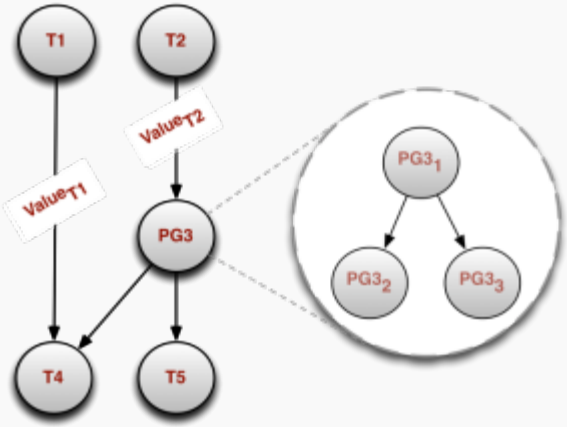
stapl::paragraph

What is the PARAGRAPH?

Parasol

- Executes *task dependence graphs* specified by the skeletons and instantiated with input views.
  - Vertices represent computation (aka *workfunction)* on input views and intermediate values flowed from other tasks
  - Edges represent dependences that enforce an ordering on execution. ***Successors*** *consume values* ***predecessors*** *produce.*



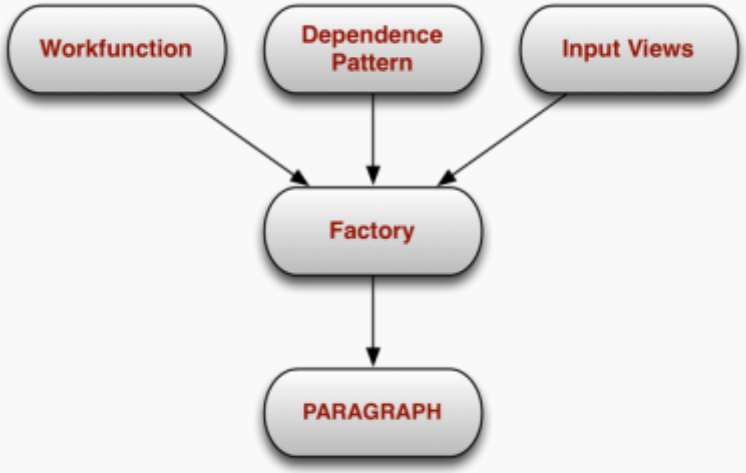


PARAGRAPHs
can be recursively defined.

Executing Algorithms with PARAGRAPHs

Parasol

- Algorithm Developer Specifies:
  1. Input Views
  2. Skeleton
  3. Function to process a single element



- Task Factory
  - Expands Dependence Pattern
  - Models Factory Method Pattern [Gamma et al 95]
  - Spawns PARAGRAPH
    - Constructs tasks
    - Establishes dependencies

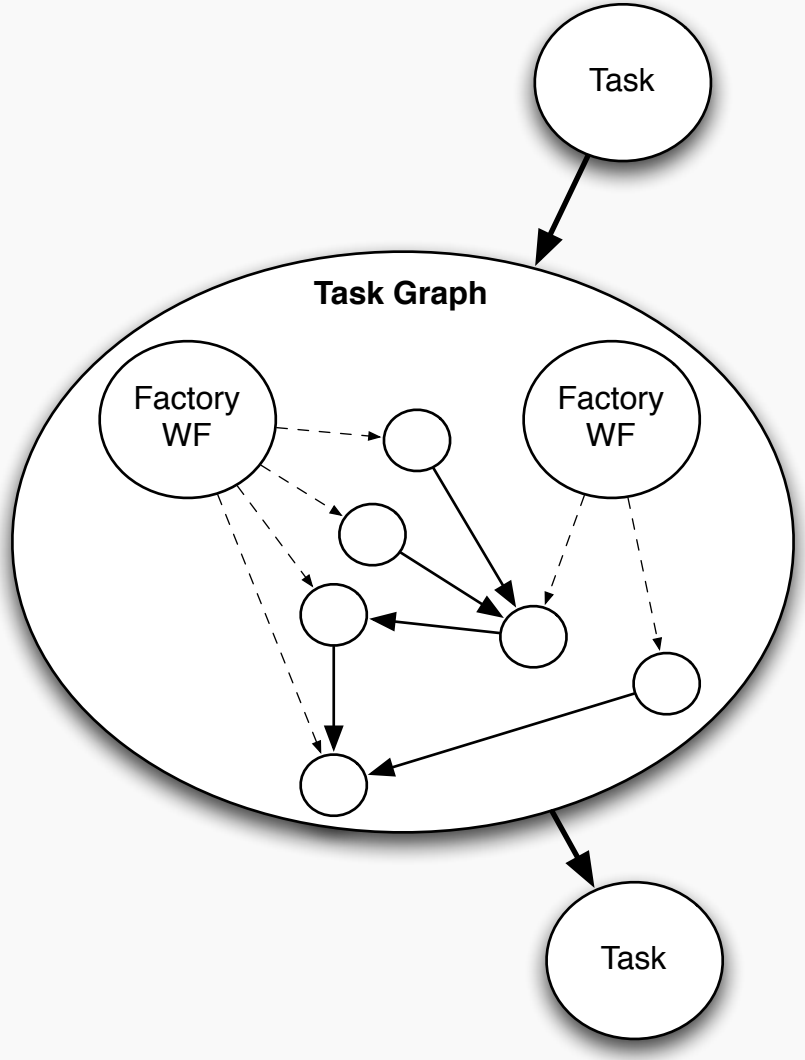
```
pg = make_paragraph(factory, view1, view2, {scheduler}, …);
return_ref = pg (); // Non-blocking, return a future.
```

Task Factories

Parasol

- ***Primary responsibility:***
  Populate a task graph.

- New Task Graph in Hierarchy
  - Single node in *outer* graph
  - First tasks in *inner* graph.
  - *Task identifier* scope.
  - Factory invoked on each location in new PARAGRAPH.



Overlapping Creation & Execution Matters

Parasol

- Global synchronization between creation and execution phases reduces scalability.
- Unordered creation enables incremental generation.
  - Storing the complete, runtime task graph of large applications may be impossible.
- Dynamic task creation supports applications who graph cannot be specified statically.

Formalizing the Problem

Parasol

Given a dependence to express: $T1 \rightarrow T2$

Divide into **five** activities:

- Task Creation T_{C1} T_{C2}
- Task Execution T_{Exec1} T_{Exec2}
- Edge Creation $T_{Edge1,2}$

Minimal Ordering Constraints:

$T_{C1} \rightarrow T_{Exec1}$

$T_{C2} \rightarrow T_{Exec2}$

$T_{Edge1,2} \rightarrow T_{Exec2}$

$T_{Exec1} \rightarrow T_{Exec2}$

Problem: How long to wait for edges to be created.

Basic Approach 1: Use Global Synchronizations

Parasol

Barriers between task creation, edge creation, and execution.

$$\text{forall}_i\ \text{forall}_j\ \text{forall}_k\ T_{Ci} \rightarrow T_{Edge\,j,k}$$

$$\text{forall}_i\ \text{forall}_j\ \text{forall}_k\ T_{Edge\,i,j} \rightarrow T_{exec\,k}$$

Drawbacks

- Global synchronizations limit parallelism.
- Static computations, cannot create new work.
- Must store entire graph.

Basic Approach 2: Deferred Task Creation

Parasol

Defer task creation until predecessor completes.
Implicit enforcement of dependence.

$$T_{Exec1} \rightarrow T_{C2}$$

Drawbacks

- Arbitrary graphs are not possible.
- The level of abstraction is lowered.
- The critical path is increased.

Our Approach

Parasol

Remove creation activity ordering requirements and allow arbitrary dependences to be specified.

- Assumes some knowledge of graph pattern.
  - STAPL skeletons provide this information.
  - Tasks mapped to unique task identifiers, used to specify edges.
- Dynamic graphs are supported.
  - Similar to deferred approach, tasks can create other tasks + edges.

Our Approach

Parasol

We choose to split the T_{Edge} activity, and merge parts with creation of both tasks

Predecessors specify out-degree: T_{C1} + OutDegree

Need to notify all successors prior to retirement.

```
pgv.add_task(tid, wf, views, num_succs);
```

Successors specify list of predecessors: T_{C2} + $Edges_{in}$

Need to wait for all predecessors prior to execution.

```
pgv.add_task(tid, wf, views, consume<T>(pred_tid), num_succs);
```

PARAGRAPH API

Primitive	*Description*
Construction / Execution	
`paragraph(factory, scheduler, views…)`	Constructor. Instantiate object. Task graph creation does not occur yet.
`result_ref_t& operator()`	Start task creation and execution. Return a placeholder (i.e., future) that will be filled with return value when available.
PARAGRAPH Population	
`void add_task(tid, wf, succ_t, views…)`	Add a task to the paragraph with the given workfunction, task identifier and views. succ_t is either out degree or defer_t.
`view_t consume<T>(tid, filter = identity)`	Return value passed to add_task, specifies single edge.
`1D_view_t consume<T>(tid_range, filter = identity)`	Return value passed to add_task, specifies aggregated edge.
`void set_num_succs(task_id, num_succs)`	Call to specify out degree if it was deferred at task creation.
`void set_result(task_id)`	Per location, invoked to select task providing return value of paragraph.
Scheduling (Customization Points)	
`add(task_entry& t) / task_entry& next()`	Add given task to user scheduler / provide next task to execute.
`locality_t preferred_location(wf, views…)`	For a task with specified workfunction and views, guide task placement.
`typedef … sched_info_type;`	Requires add_task() call to provide user scheduling metadata.
`typedef … enable_migration;`	Enable task migration. Scheduler given access to task::migrate()
`typedef … enable_persistence;`	Enable reuse via paragraph::operator(). Disables incremental destruction.

Implementation Insights

Parasol

- Data flow driven computation.

  Primary container indexed by edges.

- Progress enabled by a asynchronous activities:
  - task creation, task execution, edge specification.

  Event driven architecture.

- No assumptions can be made about task creation and execution ***location***.

  Message forwarding based on task identifiers.

stapl::runtime

Remote Method Invocation (RMI)

Parasol

- Unified communication model over distributed and shared memory.
- Asynchronous communication through Remote Method Invocation.
  - Ability to move work, data or both.
  - Fine grain RMIs are aggregated, per destination location.
- Copy semantics of RMI arguments.
  - Enables asynchrony, while eliminating data races.
  - Avoids side-effects.
  - Copy elision in shared-memory when higher level information is given.
- RMI Causal Consistency Model, guaranteed by implicit RMI ordering.

ARMI Usage

Parasol

```
struct A : public p_object {
  int m_value;

  void set(int t) { m_value = t; }
  int get() const { return m_value; }
};

foo(…) {
  A a;
  auto h = a.get_rmi_handle(); // comm. handle
  int t = 5;
  async_rmi(h, 1, &A::set, t); // set value to 5
  t = 6;
  future<int> f = opaque_rmi(h, 1, &A::get);
  int y = f.get();
  assert(y==5); // value is guaranteed to be 5
}
```

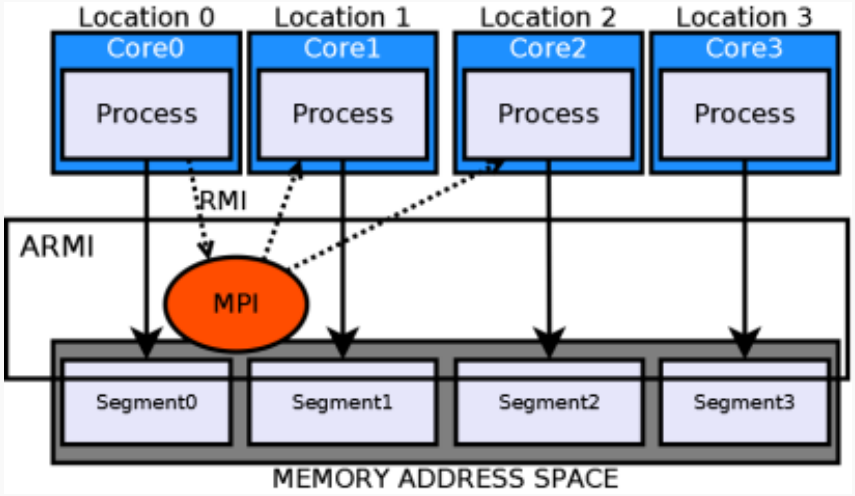
ARMI API

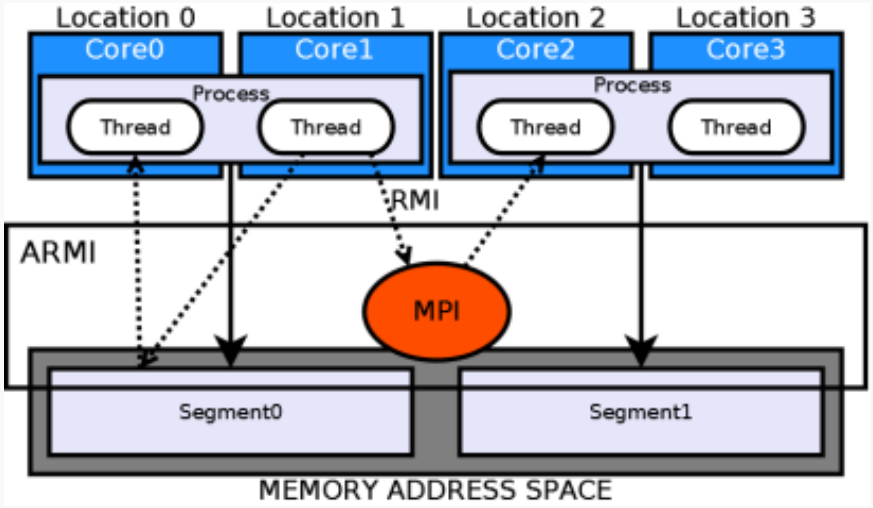
Primitive	*Description*
One-sided Primitives	
`void async_rmi(dest, h, f, args…)`	Issues an RMI that calls function f of the p_object associated with the handle h on location dest with the given arguments, ignoring the return value. Synchronization calls or other RMIs that do not ignore the return value can be used to guarantee its completion.
`future<Rtn> opaque_rmi(dest, h, f, args…)`	Returns a future object for retrieving the return value of the function.
Collective Primitives	
`futures<Rtn> allgather_rmi(h, f, args…)`	The function is called on all locations and the futures object is used to retrieve the return values.
`future<Rtn> allreduce_rmi(op, h, f, args…)`	The future is used to retrieve the reduction of the return values of f from each location.
`future<Rtn> broadcast_rmi(h, f, args…)`	The caller (root) location calls the function and broadcasts the return value to all other locations. Non-root locations have to call broadcast_rmi(root, f) to complete the collective operation.
Synchronization Primitives	
`void rmi_fence()`	Guarantees that all RMI requests invoked since the last rmi_fence() have been processed.
`void rmi_barrier()`	Performs a barrier operation.
`void p_object::advance_epoch()`	Advances the epoch of the p_object, as well as the epoch of the location. It can be used for synchronization without communication, avoiding the rmi_fence() or rmi_barrier() primitives.

Mixed-mode

Hybrid OpenMP+MPI	*STAPL Mixed-mode*
Two different programming models	Unified model over distributed and shared memory based on RMIs
Typically different algorithms / implementations for user algorithms	One algorithm and one implementation for user algorithms
Integration and performance optimization implemented by the user	Integration implemented through the runtime, which also offers adaptive performance

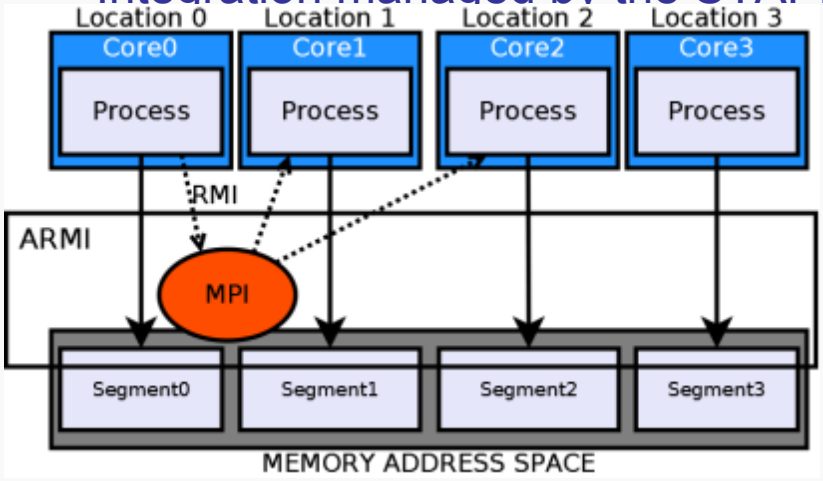
Unification through ARMI primitives

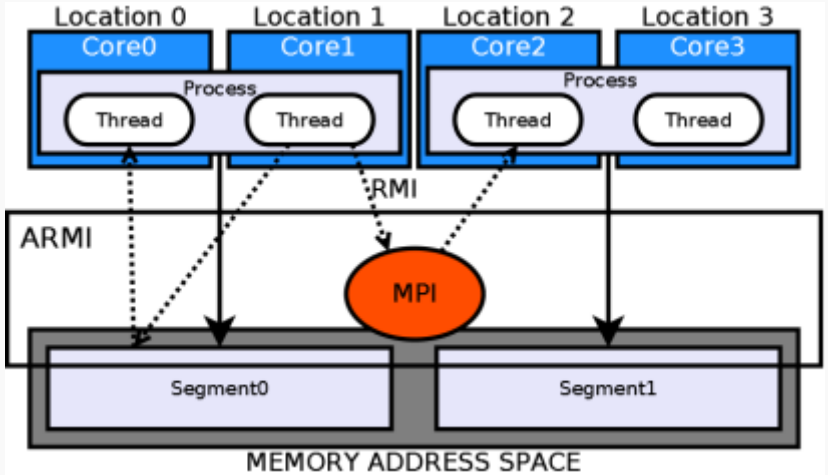




Mixed-mode

- Hybrid MPI+OpenMP
  - Two different programming models
  - Typically different algorithms/implementations for user algorithms
  - Problematic integration
- STAPL Mixed-mode
  - Unified communication model over distributed and shared memory based on RMIs
  - One algorithm and one implementation for user algorithms
  - Integration managed by the STAPL-RTS





Optimizations

Parasol

- High level, application to runtime transfer of information.
  - Maintain appropriate abstraction.

- Discuss two areas:
  - User expresses fine grain communication.

    Transparent + algorithm guided aggregation.

  - Uniform, distributed communication model.

    Runtime controlled use of shared memory.

Shared-memory optimizations

Parasol

- RMI Copy Semantics can be relaxed with appropriate high-level information:

Algorithm -> PARAGRAPH -> ARMI

- Zero Copy
  - Move semantics transfer objects between locations.

```
async_rmi(dest, obj, &A::foo, std::move(x));
```

- Immutable Sharing
  - Locations share object via read-only reference wrapper.

```
ref x_ref = immutable_shared(x);
async_rmi(dest, obj, &A::foo, x_ref);
y = x_ref
```

PARAGRAPH rules for optimization

Parasol

Task dependence graph has contextual information:

- Zero copy
  - *If a task has single consumer of its value and it is on a remote location, move value into rmi.*
- Immutable sharing
  - If task has multiple consumers, and at least one is on remote location, then immutably share value between consumers.

Experimental Setup

Parasol

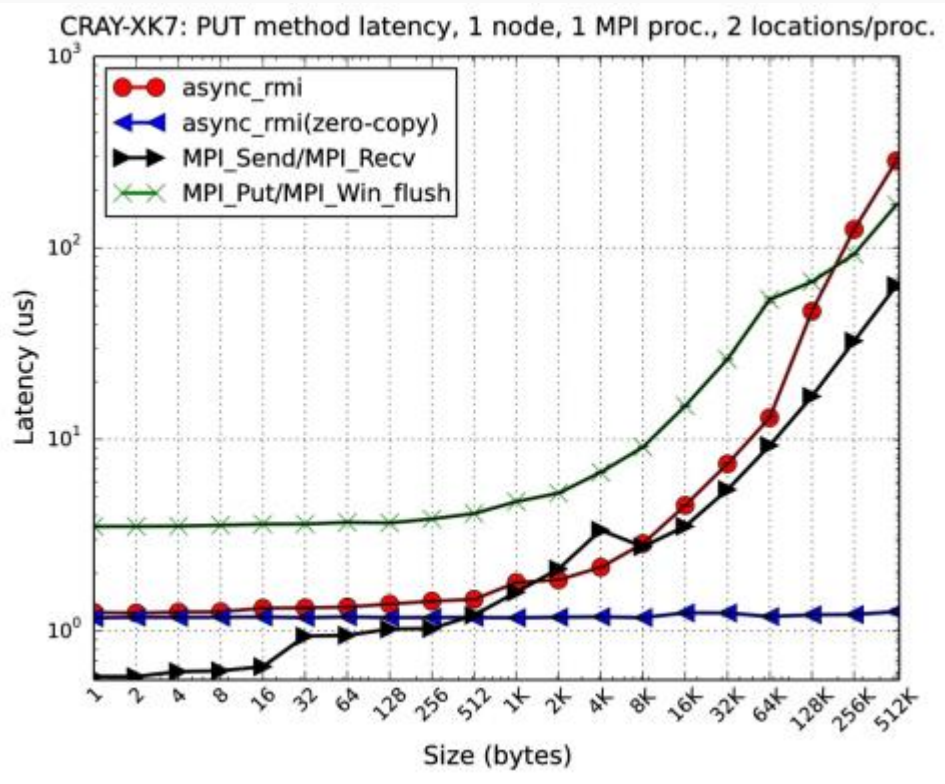
- Cray XK7m-200
  - 24 compute nodes
    - 12 single-socket nodes with GPUs and 32GB memory
    - 12 dual-socket nodes with 64GB memory
  - AMD Opteron 6272, Interlagos 16-core CPU @ 2.1GHz

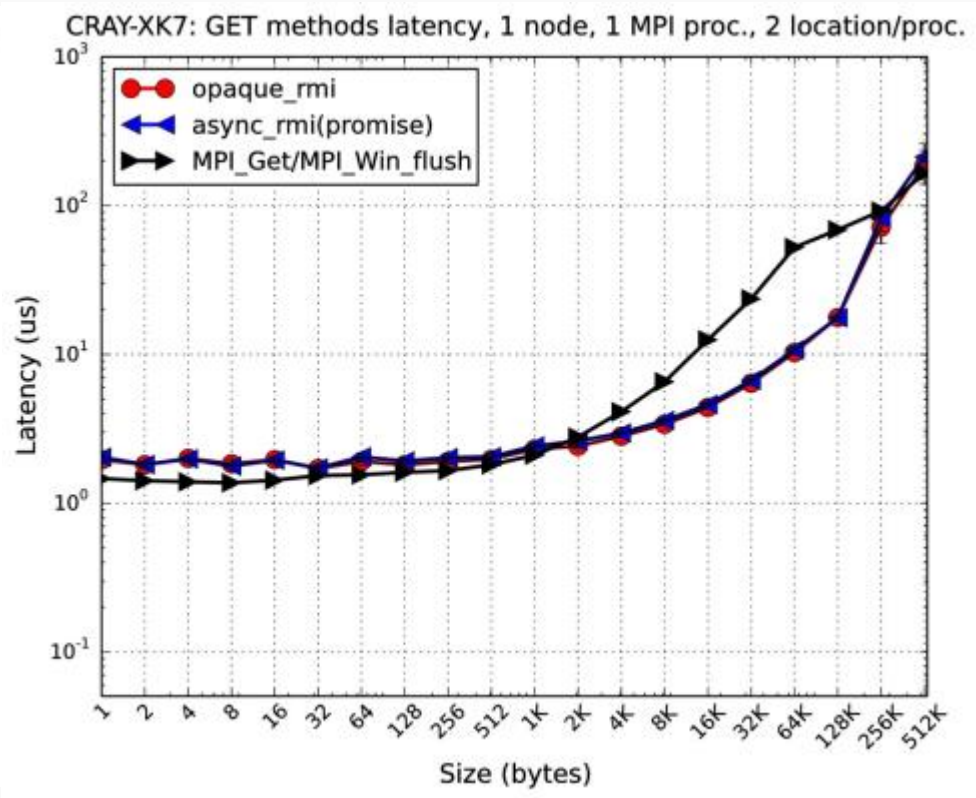
- IBM BGQ (LLNL)
  - 24,576 nodes
  - Single socket nodes with 16GB of memory
  - IBM PowerPC A2, 16-core CPU @ 1.6GHz

Zero-copy Microbenchmarks

Parasol

- OSU Put/Get Microbenchmarks on 1 node, 1 MPI process, 2 threads/ process

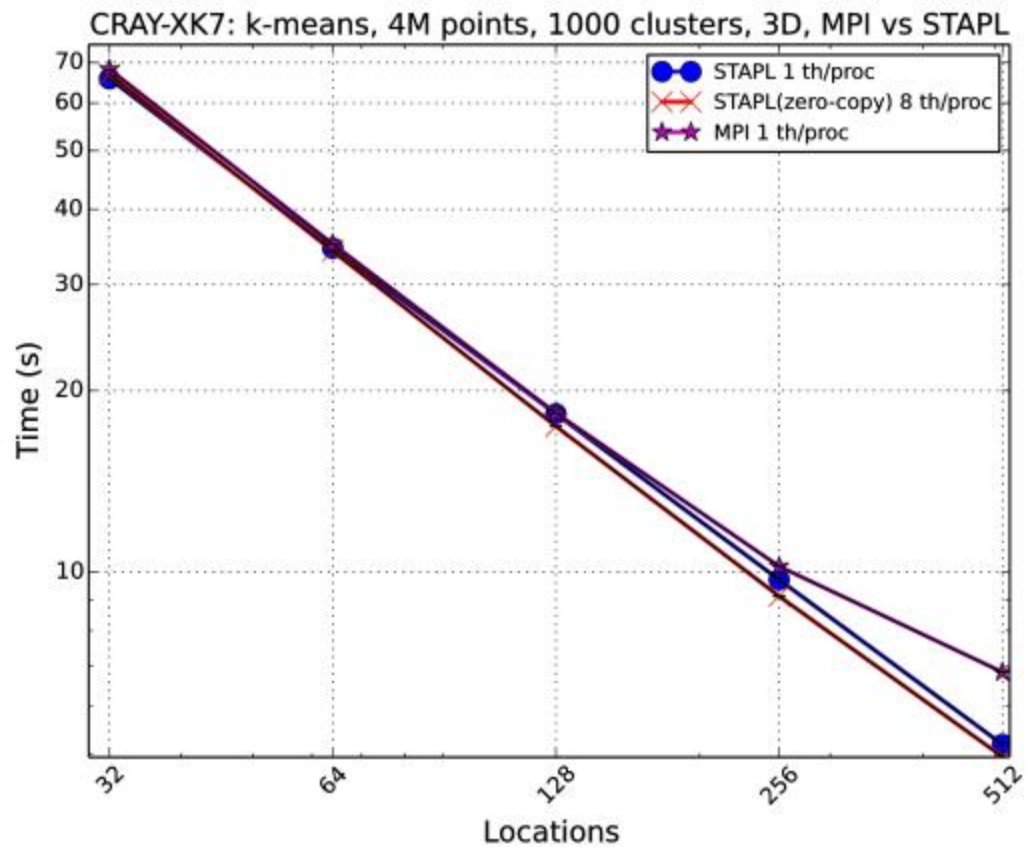




Zero copy with k-means

Parasol

- K-means clustering algorithm, primarily computation kernel.



Up to 7% gain, no change to STAPL user code.

Partial Evaluation of RMIs

Parasol

- Given basic rmi function:

```
async_rmi(location, obj, pmf, params…);
```

- What if the algorithm knows something about the pattern p_object method invocations?
  - For example, homogenous vertex updates in graph.
  - Use *partial evaluation* rmi function.

```
f = bind(async_rmi, dest, obj, &A::recv, _1);
f(10); // call A::recv(10)
f(5); // call A::recv(5)
```

- We call this an RMI tunnel.

RMI Tunnels

Parasol

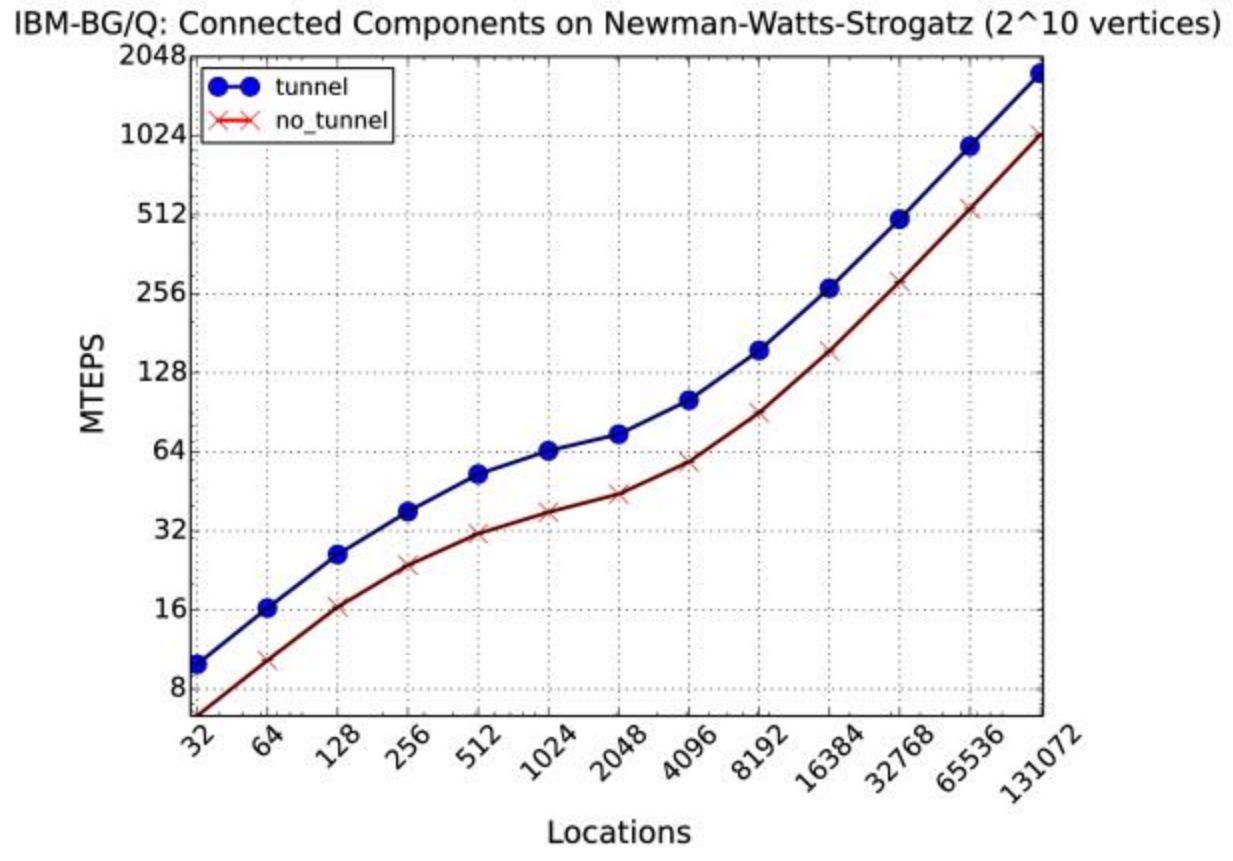
Reduces communication overhead

- Reduces required runtime checks.
- Relaxes consistency.
  - Creates a new channel for communication.
- Increases message aggregation.

Tunneling with Connected Components

Parasol

- Connected Components algorithm on Newman-Watss-Strogatz graph



From 1.5X at 32 cores to 1.7x at 128K cores. Improved scalability.

stapl::runtime::scheduler

Customizing the **PARAGRAPH** Scheduler

Parasol

- PARAGRAPH users can customize **PARAGRAPH** behaviour via the passed scheduler parameter.
  - `pg = make_paragraph(factory, view1, view2, my_scheduler());`
  - Basic patterns (e.g., map_func) pass this through to user…
- Three scheduling behaviors to customize.
  - Initial task placement.
    Default policy is majority vote election based on view locality.
  - Per location, next task to execute (local scheduler).
  - Task migration.

Customized Local Scheduling in PDT

Parasol

- PDT implements provably optimal sweep algorithm for structured meshes.
  - Sweep PARAGRAPH represents sweep for all angles in angle set
  - Sweep PARAGRAPH has a different priority on each location
- In a “transport step”
  - All sweep PARAGRAPHs placed in executor for processing
  - Priority scheduler used to select PARAGRAPH for execution
  - Strict priority enforced
    - Location will not execute ready, lower priority PARAGRAPH until higher priority PARAGRAPH has completed all tasks on the location
    - Ensures minimal number of scheduling steps to execute all PARAGRAPHS
    - Guarantee maintained at cost of missed opportunity to execute tasks

STAPL Runtime Design for Exascale

Parasol

- Asynchronous, non-blocking operations.
  - Important to overlap communication and computation.
  - PARAGRAPH creation and execution proceed concurrently.
  - Flexible mechanisms for synchronization and consistency.
- Present a uniform execution model for nested parallelism.
  - Each nested computation runs in *isolated* environment.
- Abstracts underlying protocols (e.g., MPI vs OpenMP).
- Adaptive, customization driven by higher levels.
  - Dynamic task & communication granularity, scheduling, …
  - Optimization driven from a from ***high level of abstraction***.