STAPL

Lawrence Rauchwerger, Antal Buss, Harshvardhan, Ioannis Papadopoulous, Olga Pearce, Timmie Smith, Gabriel Tanase, Nathan Thomas, Xiabing Xu, Mauro Bianco, Nancy M. Amato

Dept of Computer Science and Engineering, Texas A&M

Aydan Namdar Ghazani

Outline

Standard Template Adaptive Parallel Library

- Superset of C++ STL
- HPC computing
- 3 levels of abstraction
  - Application Developer
  - Library Developer
  - RTS Developer
- Papers 1998-2021
- Gitlab [5]

Motivation

- Productivity
- Performance
- Portability
- Maintainability

Features/ Project	Paradigm[1]	Architecture	Nested	Adaptive	Generic	Data Distribution	Scheduling	Overlap comm/comp
STAPL	S/MPMD	Shared/Dist	Yes	Yes	Yes	Auto/User	Customizable	Yes
PSTL	SPMD	Shared/Dist	No	No	Yes	Auto	Tulip RTS	No
Charm++	MPMD	Shared/Dist	No	No	Yes	User	prioritized execution	Yes
CILK	S/MPMD	Shared/Dist	Yes	No	No	User	work stealing	No
NESL	S/MPMD	Shared/Dist	Yes	No	Yes	User	work and depth model	No
POOMA	SPMD	Shared/Dist	No	No	Yes	User	pthread scheduling	No
SPLIT-C	SPMD	Shared/Dist	Yes	No	No	User	user	Yes
X10	S/MPMD	Shared/Dist	No	No	Yes	Auto	-	Yes
Chapel	S/MPMD	Shared/Dist	Yes	No	Yes	Auto	-	Yes
Titanium	S/MPMD	Shared/Dist	No	No	No	Auto	-	Yes
Intel TBB	SPMD	Shared	Yes	Yes	Yes	Auto	work stealing	No

[1] SPMD - Single Program Multiple Data, MPMD - Multiple Program Multiple Data

Image source: Rauchwerger et al. [1]

Adaptive

- Static analysis
- Cache sizes
- Periodic updates
- Quinlan’s ID3 Decision Tree

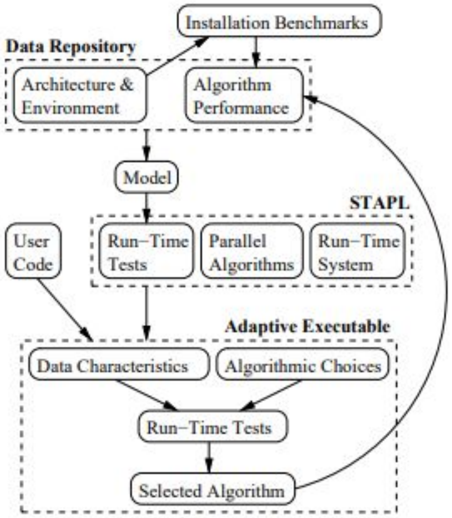


Fig. 9. Adaptive Framework

Sort	Strength	Weakness
Column	time optimal	many passes
Merge	low overhead	poor scalability
Radix	extremely fast	integers only
Sample	two passes	high overhead

Fig. 10. Parallel Sort Summary

Image source: An, P. et al. [6]

STAPL Overview

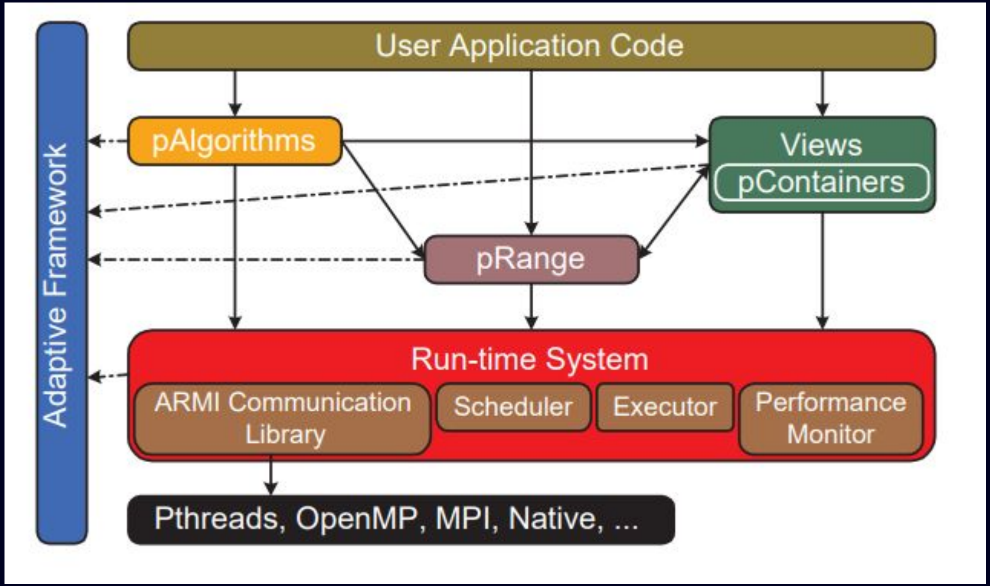


Image source: Rauchwerger et al. [1]

pContainers

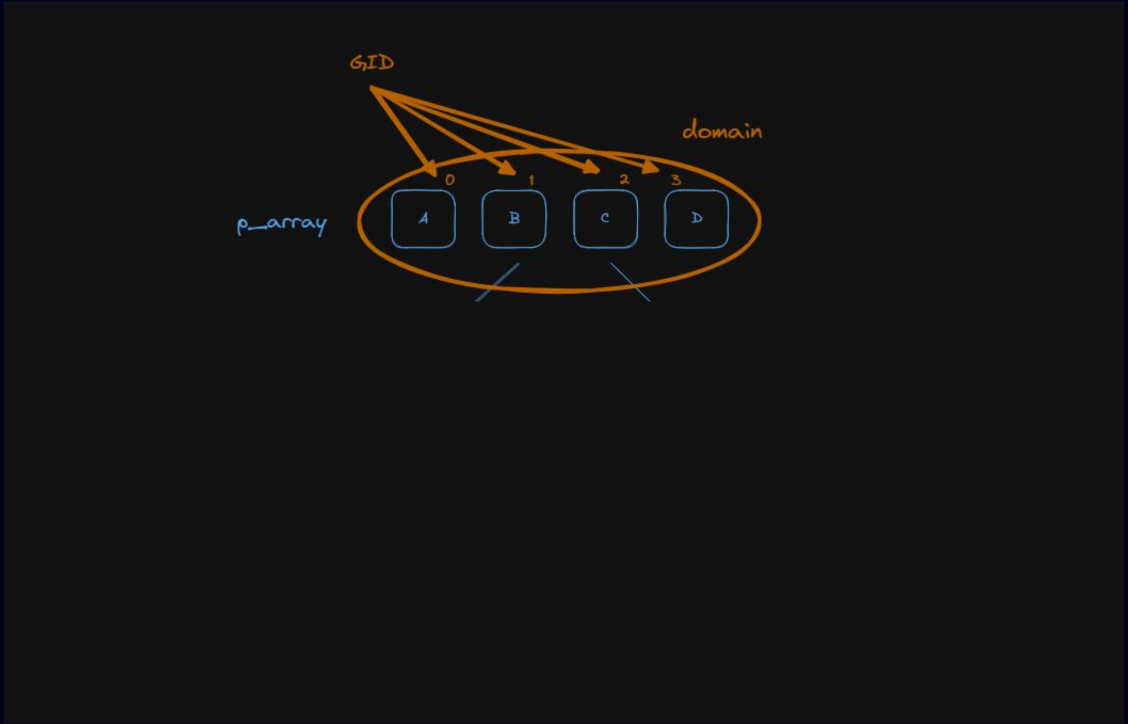
- Parallel equivalent to STL containers
- Sequential + threadsafe parallel methods
- Composable + Inheritable
- Non replicated data
- pMatrix and pGraph

pContainers

```
stapl::vector<std::string> vec_strings(5, "Howdy");
stapl::array<int> array_ints(10, 7);

f_arr = array_ints.get_element_split(0);
//code source: GitLab [5]
```

Parallel Container Framework



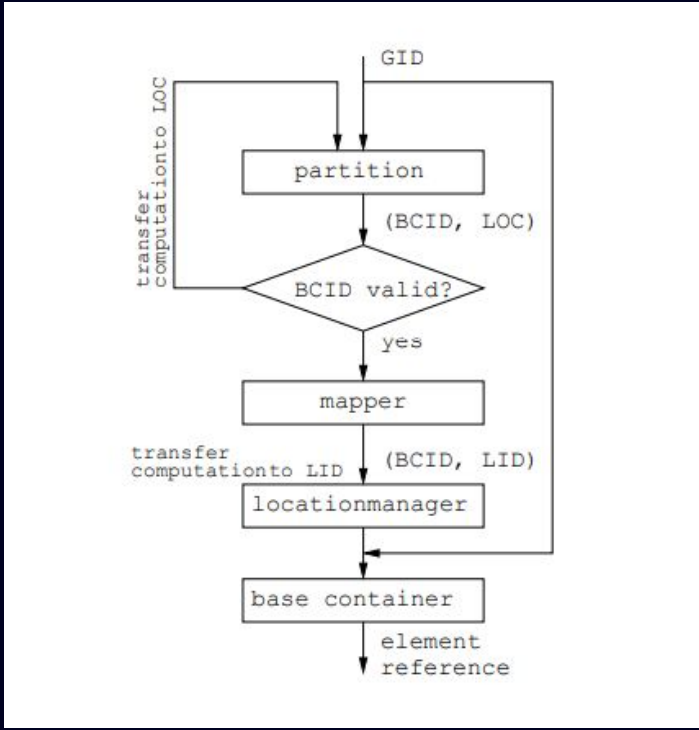


image source: Rauchwerger et al. [1]

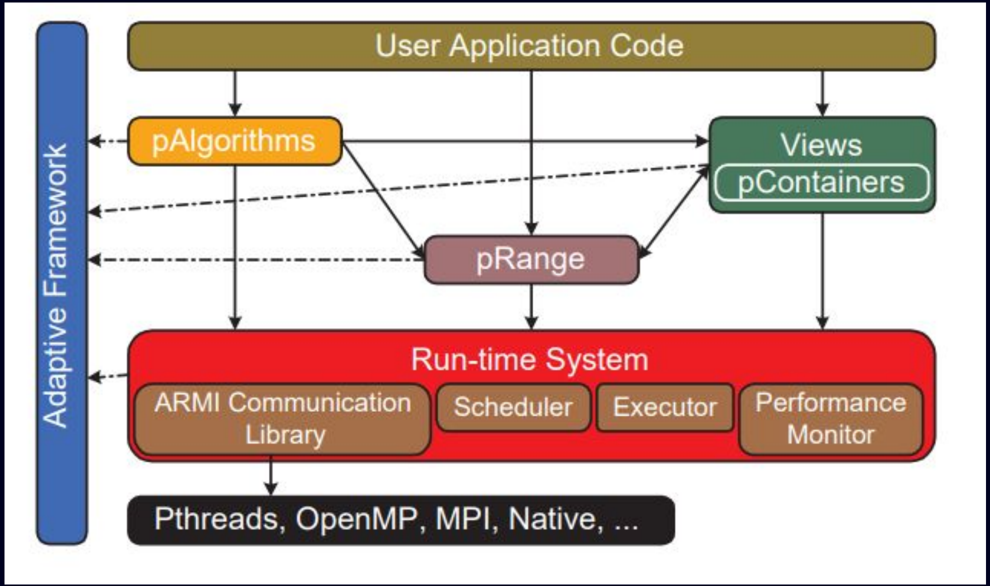
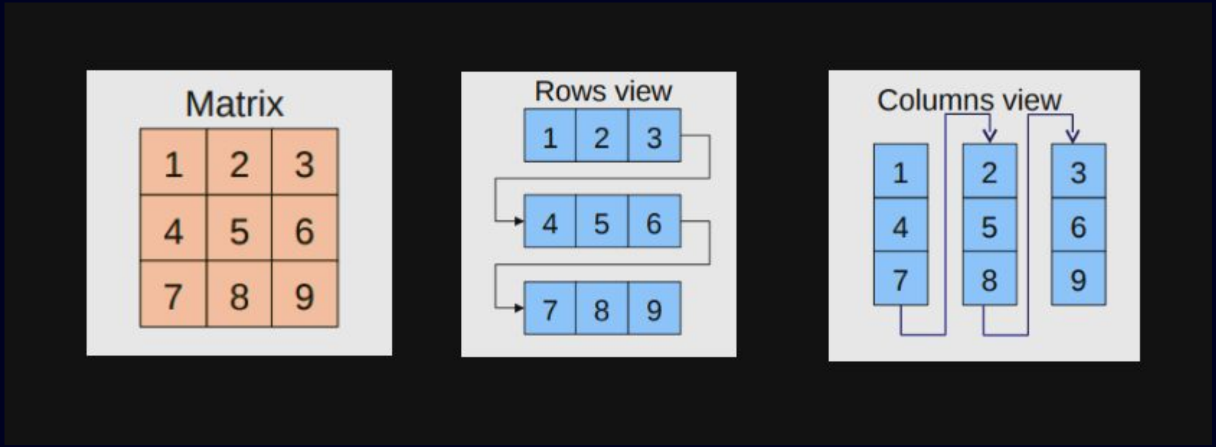


Image source: Rauchwerger et al. [1]

pViews

- A lightweight ADT
- Like an iterator
- Random access to segments



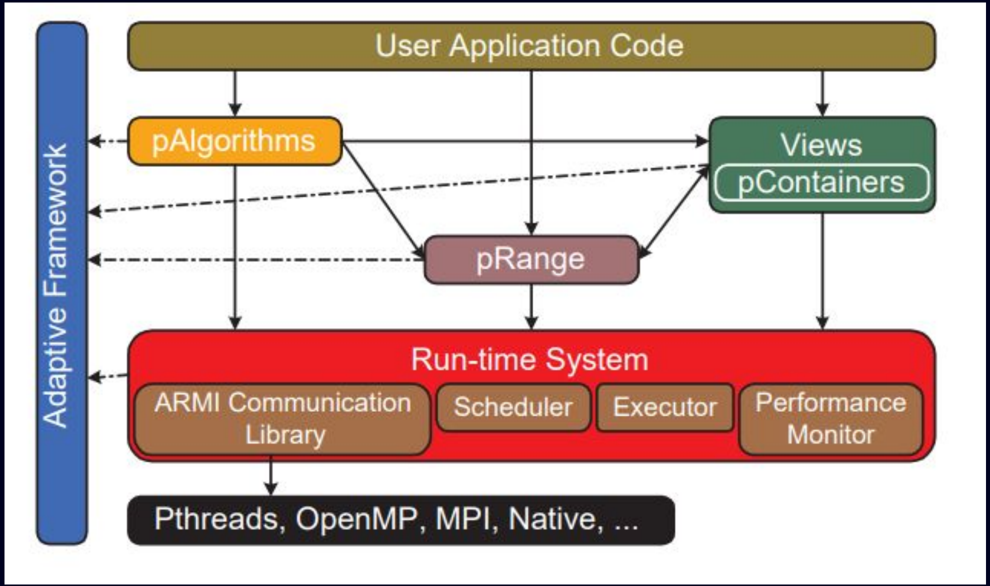


Image source: Rauchwerger et al. [1]

pAlgorithms

- STL algorithms
- Represented by pRanges

pRanges

- Binds pAlgorithms with pContainers
- A task graph
  - Vertices are tasks
  - Edges are dependencies
- A task contains work and data
- Has shared executor object

pAlgorithms

```
int mts = map_reduce(
  overlap_view(text_view, 1, 0, pattern.size()-1),
  strmatch(pattern),
  std::plus<char>
);

//code source: Rauchwerger et al. [1]
```

Data -> “abcdef”

View -> “abc”, “bcd”, “cde”, “def“

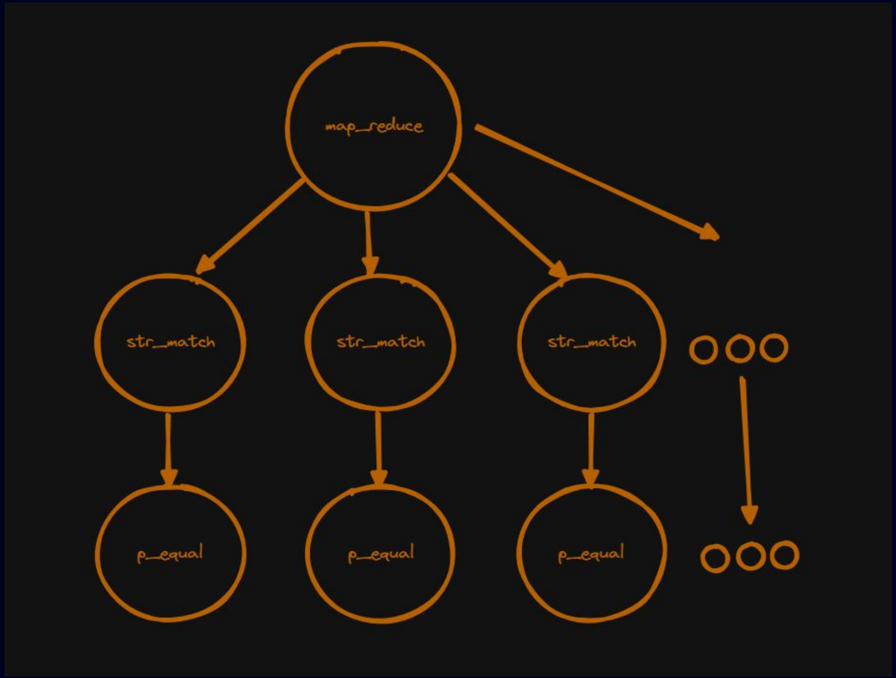
pAlgorithms

```
struct strmatch {
  typedef access_list<R> view_access_types;
  string pat;
  strmatch(string _pat) : pat(_pat) {}

  template<typename View>
  bool operator()(View x) const {
  return stapl::p_equal(pat,x);
  }
};

//code source: Rauchwerger et al. [1]
```

pAlgorithms



pAlgorithms

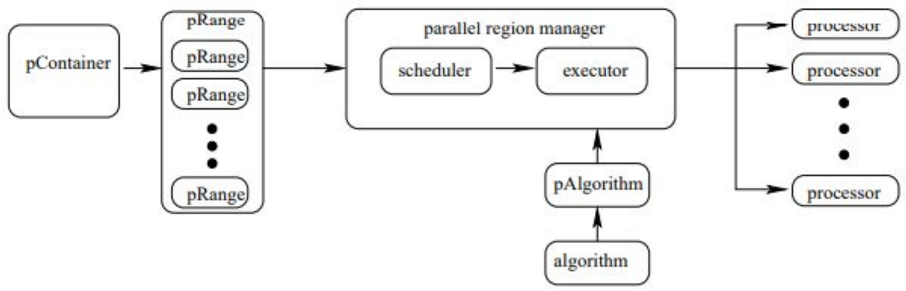


Fig. 1. STAPL Components

Image source: An, P. et al. [6]

Runtime-system

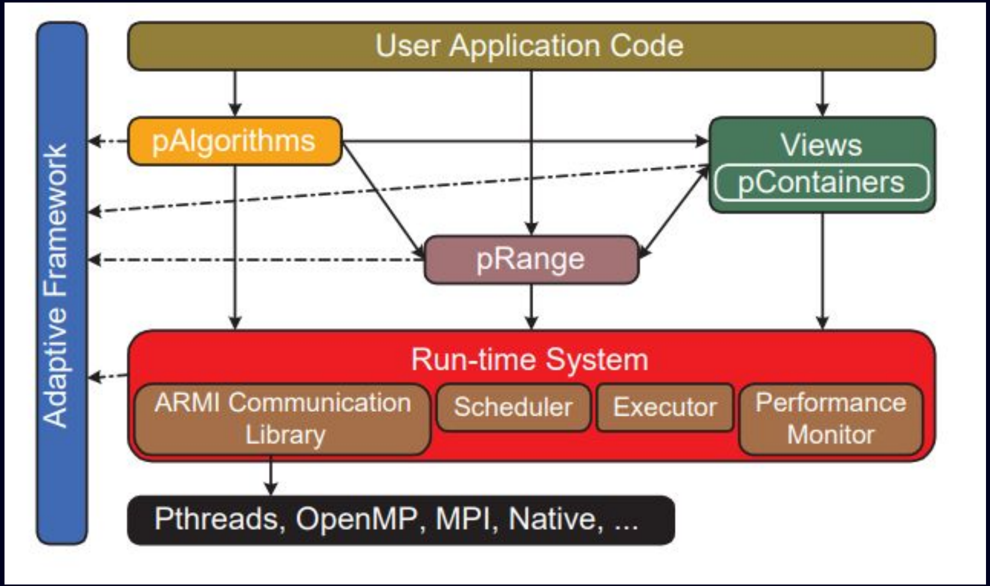


image source: Rauchwerger et al. [1]

Runtime-system

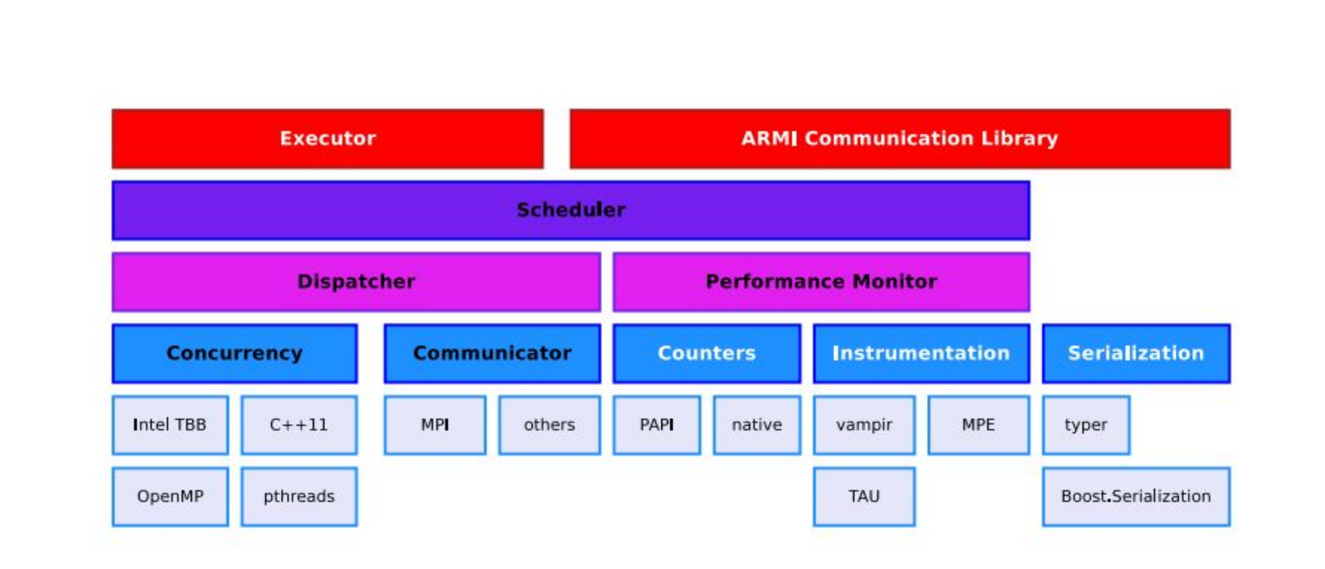


image source: Ioannis et al. [2]

ARMI

- Async/sync interprocess communication
- Communication between locations
- Collective or one-sided requests
- Machine dependent

Primitive	Description
One-Sided Primitives	
void async_rmi(dest, h, f, args ...)	Issues an RMI that calls the function `f` of the `p_object` associated with the `rmi_handle h` on location `dest` with the given arguments, ignoring the return value. Synchronization calls or other RMI requests 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.
Rtn sync_rmi(dest, h, f, args ...)	Issues the RMI and waits for the return value (blocking primitive).
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 invoked RMI requests have been processed using an algorithm similar to [36].
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.

image source: Ioannis et al. [2]

Scheduler

- Allocates resources
- Execution order of pRanges
- Policies (Diffusive, lifeline, circular, random, etc.)
- Optimization problem

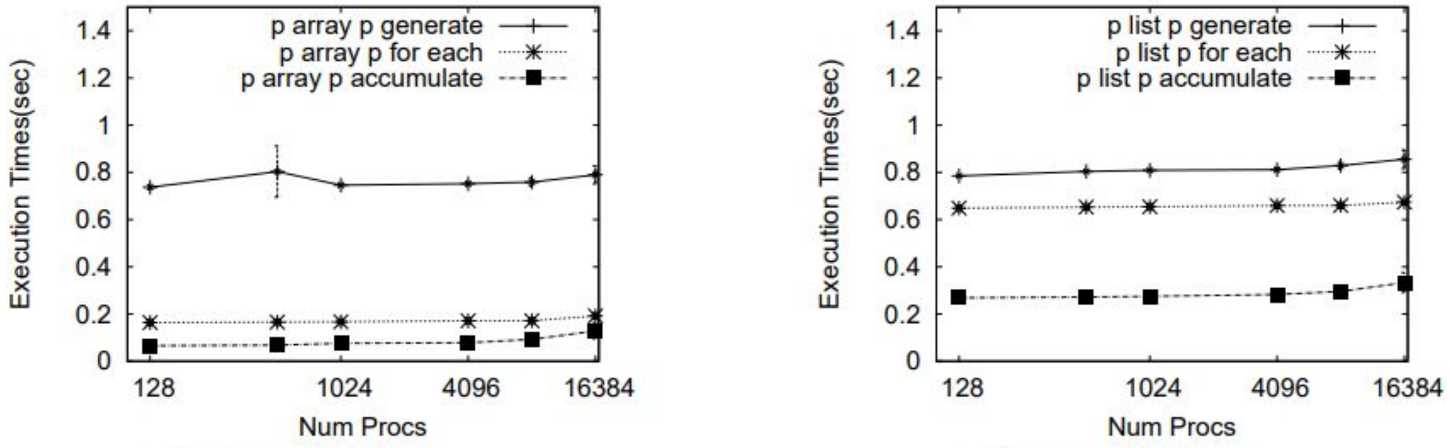
Executor

- A distributed shared object
- Responsible for parallel execution of pRanges
- Works with the scheduler
- Native or serialized nested parallelism

Performance Evaluation

- Cray XT4 & IBM P5-Cluster
- Evaluating scalability of algorithms and datastructures

Performance Evaluation



(a) `pArray`; 20M/Proc

(b) `pList`; 20M/Proc

image source: Rauchwerger et al. [1]

Performance Evaluation

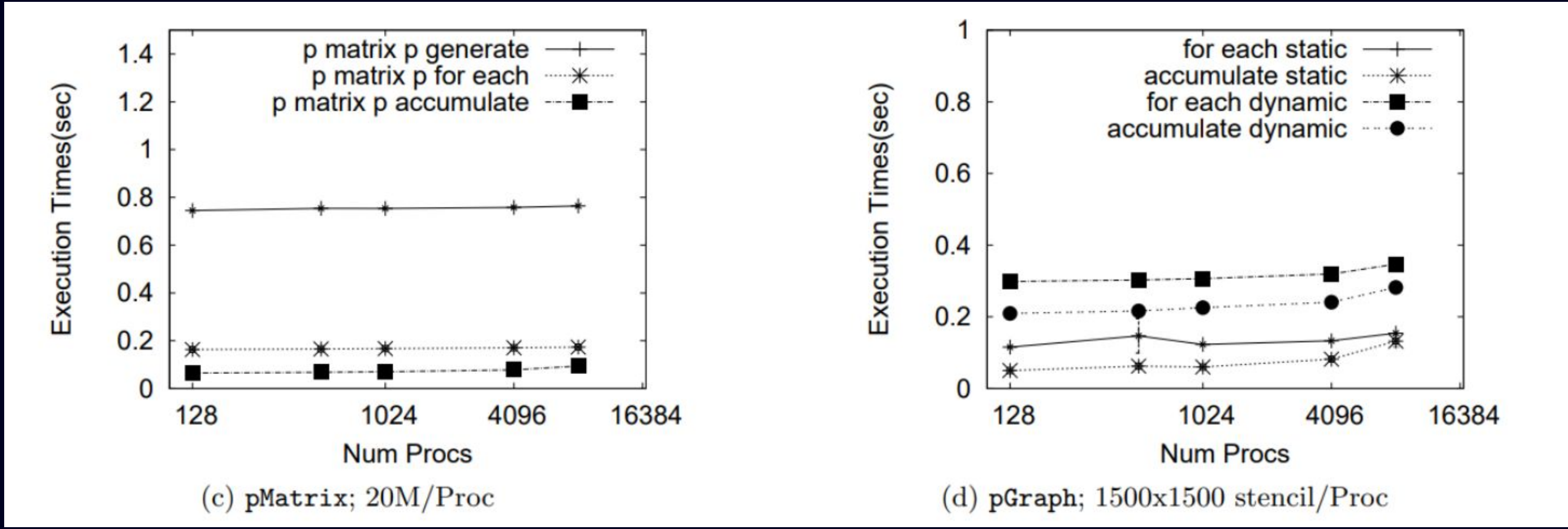


image source: Rauchwerger et al. [1]

Performance Evaluation

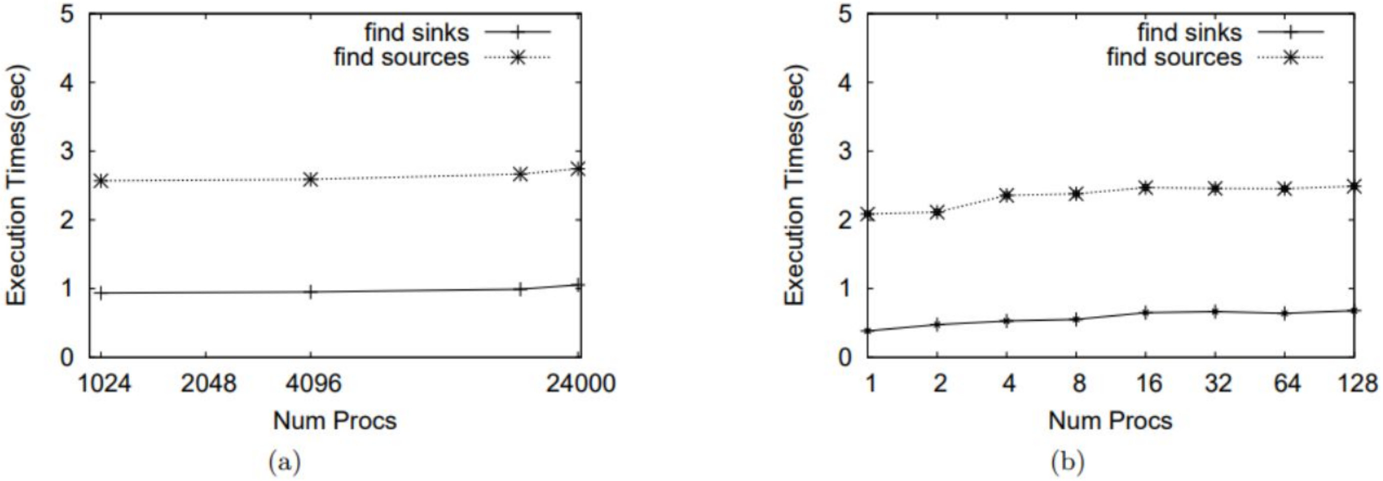


Figure 5: Execution times for pGraph algorithms find_sinks and find_sources on (a) CRAY and (b) P5-CLUSTER.

image source: Rauchwerger et al. [1]

Performance Evaluation

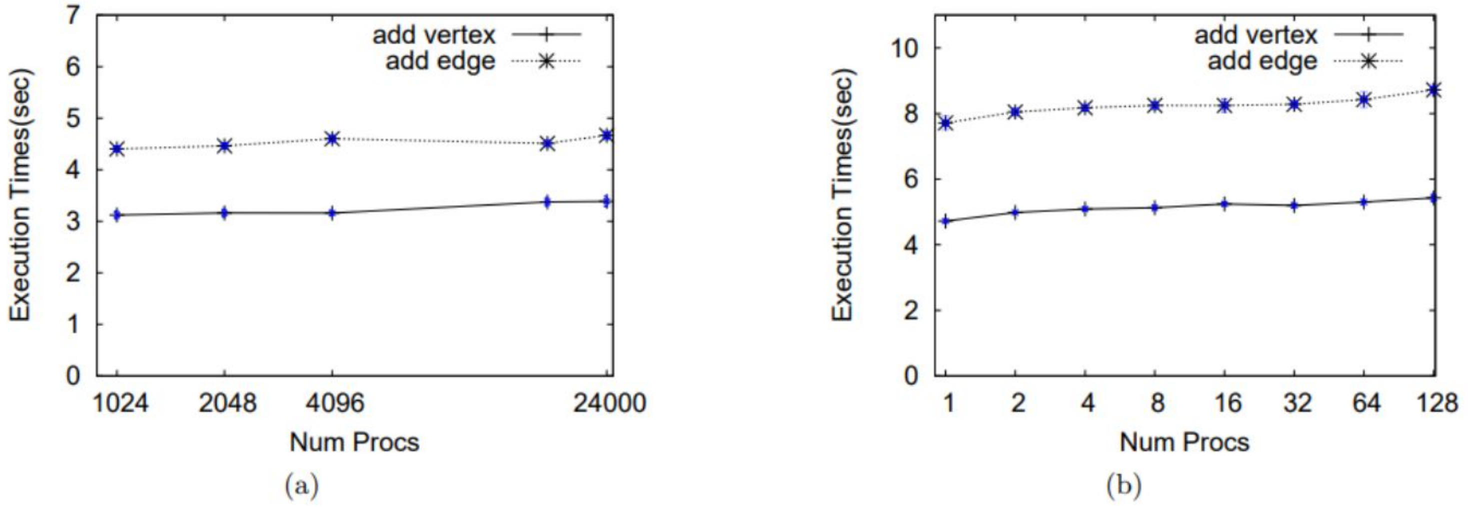


Figure 6: Execution times for pGraph methods add_vertex and add_edge on (a) CRAY and (b) P5-CLUSTER.

image source: Rauchwerger et al. [1]

Performance Evaluation

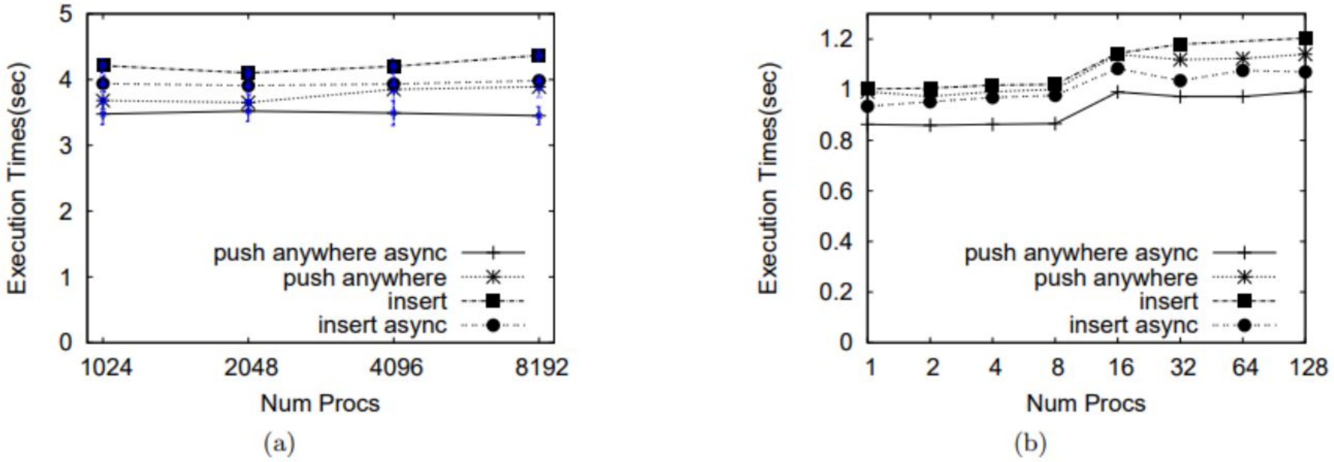
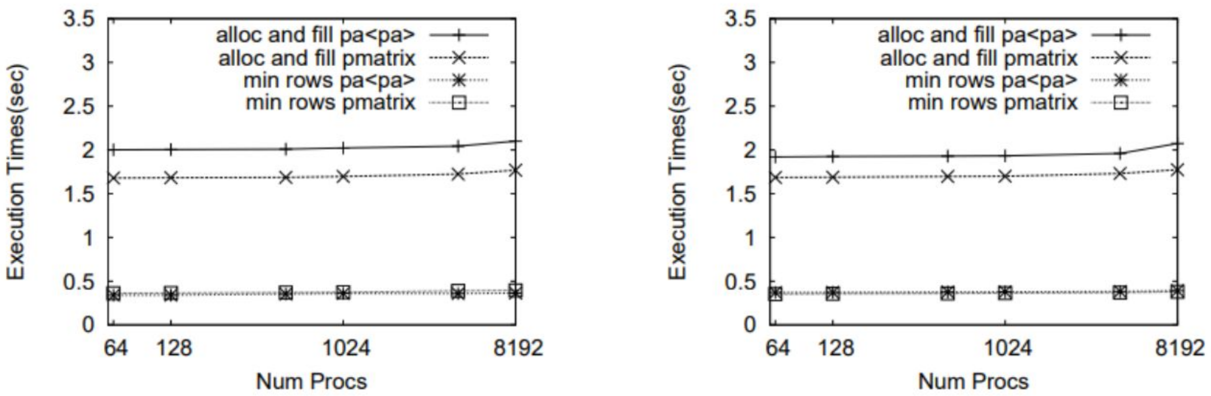


Figure 7: Weak scaling study for pList methods on (a) CRAY using 25M method invocations per processor and (b) P5-CLUSTER using 5M method invocations per processor.

image source: Rauchwerger et al. [1]

Performance Evaluation



$P \times 100M$ Matrix
(a)

$100 \cdot P \times 1M$ Matrix
(b)

Figure 8: Comparison of parray<parray<> > **(pa<pa>) and** pMatrix **on computing the minimum value for each row of a matrix. Weak scaling experiment with (a) $P \times 100M$ and (b) $100 \cdot P \times 1M$ elements.** parray<parray<> > **takes longer to initialize while the algorithm executions are similar.**

image source: Rauchwerger et al. [1]

Conclusion

- ISO Standard
- Ease of programming
- Scalable up to tens of thousands of processors
- Portable to different architectures

Three Five Divisor

Find the sum of all natural numbers from 1 to n that is divisible by 3 or 5

Three Five Divisor

```
ulong_type num = boost::lexical_cast<ulong_type> (argv[1]);

// Array container of unsigned integers.
stapl::array<ulong_type> b(num);

// Creates view over container.
stapl::array_view<stapl::array<ulong_type>> vw(b);

// Fills the container with values from 1 to n.
stapl::iota(vw, 1);

// Numbers which are not divisible by 3 or 5 are set to 0.
stapl::replace_if(vw, three_five_divisor(), 0);

// Adds the total of all elements in container.
ulong_type total = stapl::accumulate(vw, (ulong_type)0);

// Prints the total sum.
stapl::do_once ([&] {
    std::cout << "The total is: " << total << std::endl;
});
//code source: GitLab [5]
```

Three Five Divisor

```
ulong_type num = boost::lexical_cast<ulong_type> (argv[1]);

// Creates a non-storage view in counting order from 1 to num.
auto vw = stapl::counting_view<ulong_type>(num, 1);

// Maps function to all numbers and sums up after.
ulong_type total = stapl::map_reduce(
    three_five_divisor(), stapl::plus<ulong_type>(), vw
);

// Prints the total sum.
stapl::do_once ([&] {
    std::cout << "The total is: " << total << std::endl;
});
//code source: GitLab [5]
```

References

[1] Antal Buss, Harshvardhan, Ioannis Papadopoulos, Olga Pearce, Timmie Smith, Gabriel Tanase, Nathan Thomas, Xiabing Xu, Mauro Bianco, Nancy M. Amato, and Lawrence Rauchwerger. 2010. STAPL: standard template adaptive parallel library. In Proceedings of the 3rd Annual Haifa Experimental Systems Conference (SYSTOR '10). Association for Computing Machinery, New York, NY, USA, Article 14, 1–10. https://doi.org/10.1145/1815695.1815713.

[2] Ioannis Papadopoulos, Nathan Thomas, Adam Fidel, Nancy M. Amato, and Lawrence Rauchwerger. 2015. STAPL-RTS: An Application Driven Runtime System. In Proceedings of the 29th ACM on International Conference on Supercomputing (ICS '15). Association for Computing Machinery, New York, NY, USA, 425–434. https://doi.org/10.1145/2751205.2751233

[3] Gabriel Tanase, Antal Buss, Adam Fidel, Harshvardhan, Ioannis Papadopoulos, Olga Pearce, Timmie Smith, Nathan Thomas, Xiabing Xu, Nedal Mourad, Jeremy Vu, Mauro Bianco, Nancy M. Amato, and Lawrence Rauchwerger. 2011. The STAPL parallel container framework. SIGPLAN Not. 46, 8 (August 2011), 235–246. https://doi.org/10.1145/2038037.1941586

[4] Antal Buss, Adam Fidel, et al. "The STAPL pView." InWkshp. on Lang. and Comp. for Par. Comp.(LCPC) (2010).

[5] https://gitlab.com/parasol-lab/stapl

[6] An, P. et al. (2003). STAPL: An Adaptive, Generic Parallel C++ Library. In: Dietz, H.G. (eds) Languages and Compilers for Parallel Computing. LCPC 2001. Lecture Notes in Computer Science, vol 2624. Springer, Berlin, Heidelberg. https://doi.org/10.1007/3-540-35767-X_13

Q & A