STAPL: An Adaptive, Generic, Parallel C++ Library

Tao Huang, Alin Jula, Jack Perdue, Tagarathi Nageswar Rao, Timmie Smith, Yuriy Solodkyy, Gabriel Tanase, Nathan Thomas,
Anna Tikhonova, Olga Tkachyshyn, Nancy M. Amato, Lawrence Rauchwerger
stapl-support@tamu.edu
Parasol Lab, Department of Computer Science, Texas A&M University, http://parasol.tamu.edu/

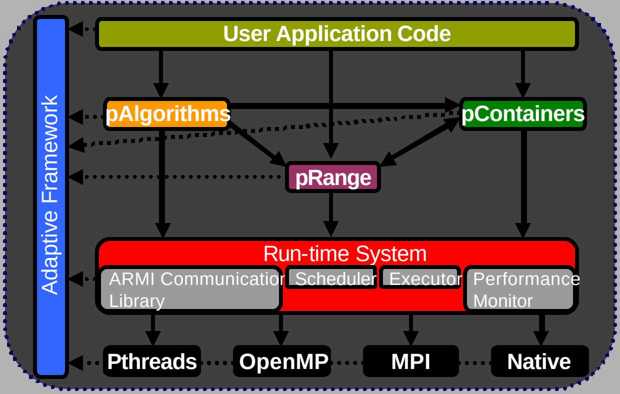
STAPL: Standard Template Adaptive Parallel Library

The Standard Template Adaptive Parallel Library (**STAPL**) is a framework for parallel C++ code. Its core is a library of ISO Standard C++ components with interfaces similar to the (sequential) ISO C++ standard library (**STL**).

The goals of STAPL are:

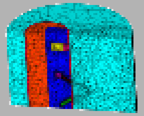
- **Ease of use**
  Shared Object Model provides consistent programming interface, regardless of a actual system memory configuration (shared or distributed).
- **Efficiency**
  Application building blocks are based on C++ STL constructs that are extended and automatically tuned for parallel execution.
- **Portability**
  ARMI runtime system hides machine specific details and provides an efficient, uniform communication interface.

STAPL Overview

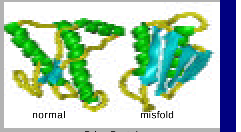


Applications using STAPL

- **Particle Transport Computation**
  - Efficient Massively Parallel Implementation of Discrete Ordinates Particle Transport Calculation.

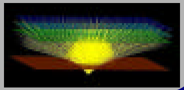

Oil well logging simulation

- **Motion Planning**
  - Probabilistic Roadmap Methods for motion planning with application to protein folding, intelligent CAD, animation, robotics, etc.



Prion Protein

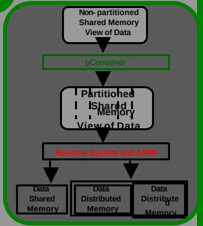
- **Seismic Ray Tracing**
  - Simulation of propagation of seismic rays in earth's crust.

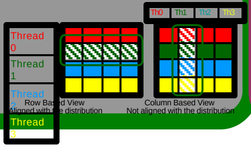


pContainer

Distributed data structure with parallel methods.

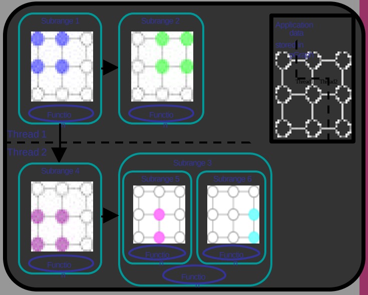
- Provides a shared-memory view of distributed data.
- Deploys an Efficient Design
  - Base classes implement basic functionality.
  - New pContainers can be derived from Base classes with extended and optimized functionality.
- Easy to use defaults provided for basic users; advanced users have the flexibility to specialize and/or optimize methods.
- Supports multiple logical views of the data.
  - For example, a pMatrix can be accessed using a row based view or a column based view.
  - Views can be used to specify pContainer (re)distribution.
  - Common views provided (e.g. row, column, blocked, block cyclic for pMatrix); users can build specialized views.
- pVector, pList, pHashMap, pGraph, pMatrix provided.





pRange

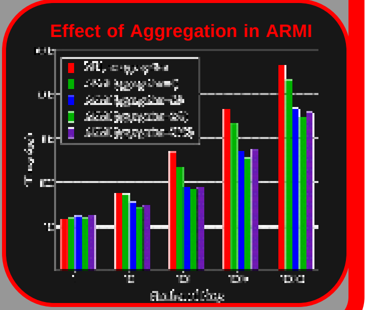
- Provides a shared view of a distributed work space
  - Subranges of the pRange are executable tasks
  - A task consists of a function object and a description of the data to which it is applied
- Supports parallel execution
  - Clean expression of computation as parallel task graph
  - Stores Data Dependence Graphs used in processing subranges



pRange defined on a pGraph across two threads.

ARMI Communication Library

- Provides high performance, RMI style communication between threads in program
  - async_rmi, sync_rmi, standard collective operations (i.e., broadcast and reduce).
- Transparent execution using various lower level protocols such as MPI and Pthreads – also, mixed mode operation.
- Controllable tuning – message aggregation.

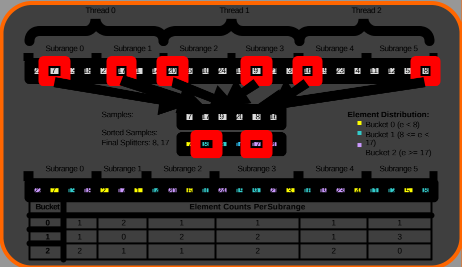


pAlgorithms

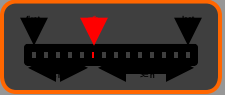
- **pAlgorithms** are parallel equivalents of algorithms.
- **pAlgorithms** are sets of parallel task objects which provide basic functionality, bound with the pContainer by pRange.
- STAPL provides parallel STL equivalents (copy, find, sort, etc.), as well as graph and matrix algorithms.

Example algorithm: Nth Element (Selection Problem)

The Nth Element algorithm partially orders a range of elements: it arranges elements such that the element located in the nth position is the same as it would be if the entire range of elements had been sorted. Additionally, none of the elements in the range [nth, last) is less than any of the elements in the range [first, nth). There is no guarantee regarding the relative order within the sub-ranges [first, nth) and [nth, last).



Bucket	Element Counts Per Subrange					
0	1	2	1	1	1	1
1	1	0	2	2	1	3
2	2	1	1	2	2	0



p_nth_element(pRange &pr, pContainer &pcont, Iterator nth) {

- Select a sample of s elements.
- Select m evenly spaced elements, called *splitters*.
- Sort the splitters and select k *final splitters*.
- Splitters determine the ranges of virtual “buckets”.
- Total the number of elements in each “bucket”.
- Traverse totals to find bucket B containing the nth element.
- Recursively call p_nth_element(B.pRange(), B, nth).

}

Example code (main):

```
typedef stapl::pArray<int> pcontainerType;
typedef pcontainerType::PRange prangeType;

void stapl_main(int argc, char **argv) {
   // Parallel container to be partially sorted:
   pcontainerType pcont(nElements);
   // Fill the container with values ...
   // Declare a pRange on your parallel
container:
   prangeType pr(pcont);

   //parallel function call
   p_nth_element(pr, pcont, nth);

   // synchronization barrier
   stapl::rmi_fence();
}
```

Example (distribute elements into virtual “buckets”):

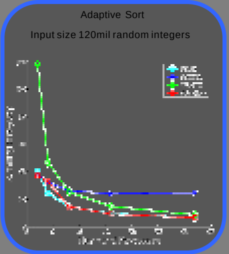
```
template< typename Boundary, class pContainer >
class distribute_elements_wf : public work_function_base<Boundary>
{
  pContainer *splitters; nSplitters = splitters->size();
  vector<int> bucket_counts(nSplitters);
  distribute_elements_wf(pContainer &sp) : splitters(&sp) {}

  void operator() (Boundary &subrange_data ) {
  typename Boundary::iterator_type first1 = subrange_data.begin ();
  while (first1 != subrange_data.end()) {
   int dest ;
   pContainer::value_type val = *first1;
   if (nSplitters > 1) { //If at least two splitters
   pContainer::value_type *d = std::upper_bound(&splitters[0], &splitters[nSplitters], val );
    dest = (int)(d -(&splitters[0]));
   } else {
    if ( nSplitters == 2){ //one splitter
     if(val < splitters[0])   dest = 0; }
     else   dest = 1;
    } else { dest = 0;  //No splitter, send to self
   }}
   bucket_counts[dest]++; ++first1;  // Increment counter for the appropriate bucket
}}};
```

Adaptive Algorithm Selection Framework

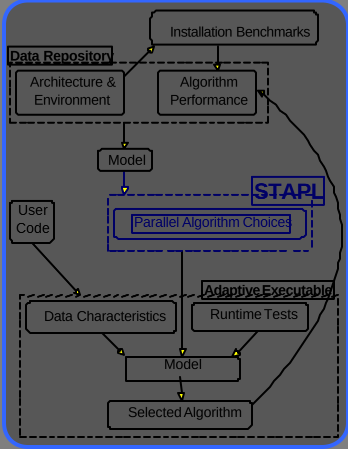
STAPL has a ***library of multiple functionally equivalent solutions*** for many important problems.

While they produce the same end result, their performance differs depending on:

- System architecture
  - Number of processing elements
  - Memory hierarchy
- Input characteristics
  - Data type
  - Size of input
  - Others (i.e. presortedness for sort)



Our framework automatically chooses an implementation that ***maximizes performance***.



- Performance Database
  - Handle various algorithms/problems with different profiling needs.
- Model Generation / Installation Benchmarking
  - Occurs once per platform, during STAPL installation
  - Choose parameters that may affect performance (i.e., input size, algo specific, etc.)
  - Run a sample of experiments, insert timings into performance database
  - Create a model to predict the “winning” algorithm in each case
- Runtime Algorithm Selection
  - Gather parameters
  - Query model
  - Execute the chosen algorithm

References

- **“A Framework for Adaptive Algorithm Selection in STAPL,”** N. Thomas, G. Tanase, O. Tkachyshyn, J. Perdue, N. Amato, L. Rauchwerger. Symposium on Principles and Practice of Parallel Programming (PPOPP), June 2005.
- **“Parallel Protein Folding with STAPL,”** S. Thomas, G. Tanase, L. Dale, J. Moreira, L. Rauchwerger, N. Amato. Journal of Concurrency and Computation: Practice and Experience, 2005.
- **“ARMI: An Adaptive, Platform Independent Communication Library,”** S. Saunders, L. Rauchwerger. Symposium on Principles and Practice of Parallel Programming (PPOPP), June 2003.
- **“STAPL: An Adaptive, Generic Parallel C++ Library,”** P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. Amato and L. Rauchwerger. Workshop on Languages and Compilers for Parallel Computing (LCPC), Aug 2001.
- **“SmartApps: An Application Centric Approach to High Performance Computing,”** L. Rauchwerger, N. Amato, J. Torrellas. Workshop on Languages and Compilers for Parallel Computing (LCPC), Aug 2000.