Table of Contents

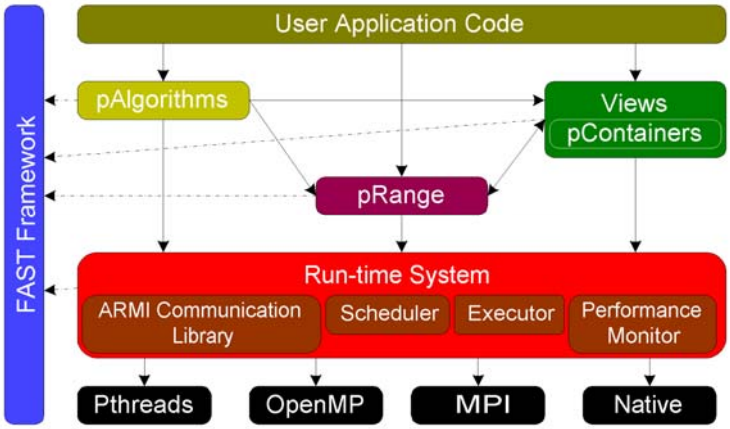
The STAPL Model

- Standard Adaptive Parallel Library
- Developed by Lawrence Rauchwerger, Nancy Amato, Bjarne Stroustrup and several grad students at Texas A&M
- Library similar and compatible with to STL
- Strong library development support
- Places parallelism burden primarily on library developers
- Commercial simple variant : Intel TBB

Adapted From: http://parasol.tamu.edu/stapl/

Standard Template Adaptive Parallel Library

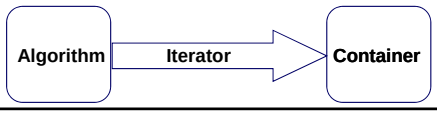
A library of parallel, generic constructs based on the C++ Standard Template Library (STL).



Standard Template Library (STL)

Generic programming components using C++ templates.

- **Containers - collection of other objects.**
  - vector, list, deque, set, multiset, map, multi_map, hash_map.
  - Templated by data type. vector<int> v(50);
- **Algorithms - manipulate the data stored in containers.**
  - manipulate the data stored in containers.
  - count(), reverse(), sort(), accumulate(), for_each(), reverse().
- **Iterators - Decouple algorithms from containers.**
  - Provide generic ***element access*** to data in containers.
  - can define custom ***traversal*** of container (e.g., every other element)
  - count(vector.begin(), vector.end(), 18);



Execution Model

- Two models: User and Library Developer
- Single threaded – User
- Multithreaded – Developer
- Shared memory – User
- PGAS – Developer
- Data & task parallelism
- Implicit communications: User
- Explicit communications: Developer

Execution Model

- Memory Consistency:
  - Sequential for user
  - Relaxed for developer (Object level)
  - Will be selectable
- Atomic methods for containers
- Synchronizations: Implicit & Explicit

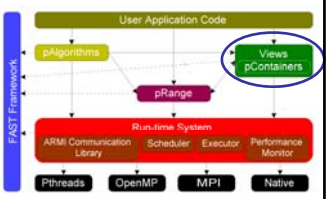
STAPL Components

- Components for Program Development
  - pContainers, Views, pRange, pAlgorithms
- Run-time System
  - Adaptive Remote Method Invocation (ARMI)
  - Multithreaded RTS
  - Framework for Algorithm Selection and Tuning (FAST)

pContainers

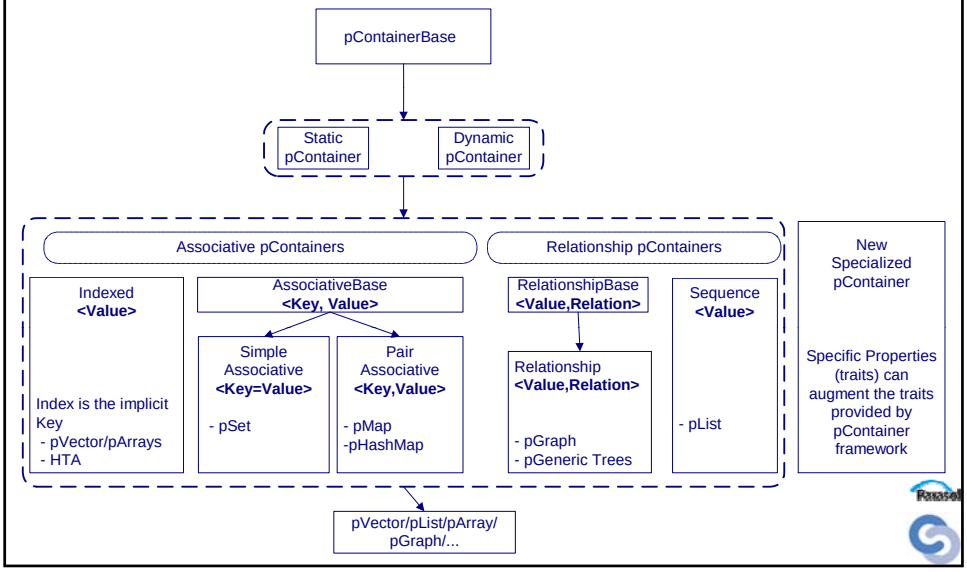
Generic, distributed data structures with parallel methods.

- Ease of Use
  - Shared object view
  - Generic access mechanism through Views
  - Handles data distribution and remote data access internally
  - Interface equivalent with sequential counterpart
- Efficiency
  - OO design to optimize specific containers
  - Template parameters allow further customization
- Extendability
  - New pContainters extend Base classes
- Composability
  - pContainers of pContainers



Currently Implemented
pArray, pVector, pGraph, pMap, pHashMap, pSet, pList

pContainer Taxonomy

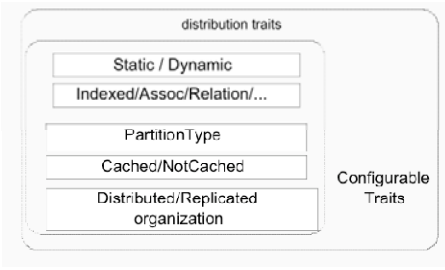


pContainer Customization

Optional user customization through pContainer **Traits**.

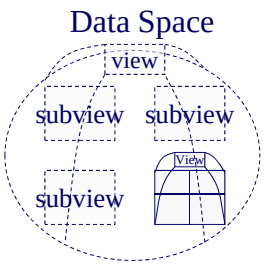
- Enable/Disable Performance Monitoring.
- Select Partition Strategies.
- Enable/Disable Thread Safety.
- Select Consistency Models

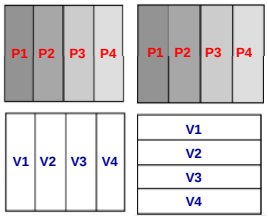
```
class p_array_traits {
 Indexed, Assoc/Key=Index,
 Static,IndexedView<Static,...,
   Random>,
 DistributionManagerTraits,
 -u-Monitoring,
 -u-Relaxed
}
```



View

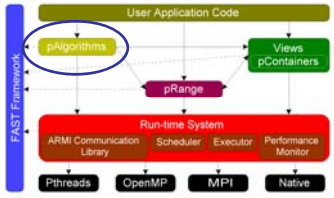
- STAPL equivalent of STL iterator, extended to allow for efficient parallelism.
- Focus on processing value range, instead of single item.
- Generic *access* mechanism to pContainer.
- Custom *traversal* of pContainer elements.
- Hierarchically defined to allow control of locality and granularity of communication/computation.





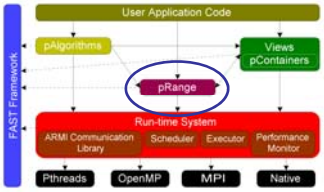
Gray -> the pContainer physical partition.
Transparent -> logical views of the data.

pAlgorithms



- pAlgorithms in STAPL
  - Parallel counterparts of STL algorithms provided in STAPL.
  - Common parallel algorithms.
    - Prefix sums
    - List ranking
  - pContainer specific algorithms.
    - Strongly Connected Components (pGraph)
    - Euler Tour (pGraph)
    - Matrix multiplication (pMatrix)
  - Often, multiple implementations exist that are adaptively used by the library.

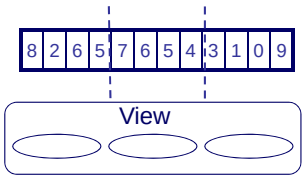
pRange



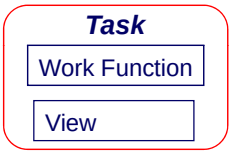
- pRange is a parallel task graph.
- Unifies work and data parallelism.
- Recursively defined as a tree of *subranges.*

pRange -- Task Graphs in STAPL

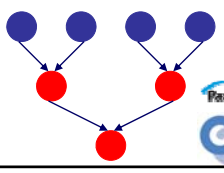
- Data to be processed by pAlgorithm
  - View of input data
  - View of partial result storage



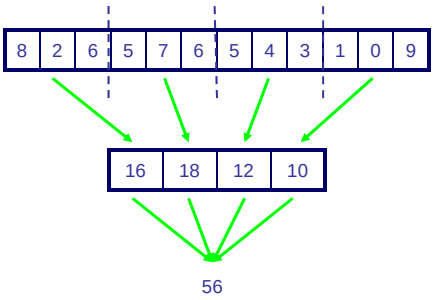
- Work Function
  - Sequential operation
  - Method to combine partial results
- Task
  - Work function
  - Data to process



- Task dependencies
  - Expressed in Task Dependence Graph (TDG)
  - TDG queried to find tasks ready for execution

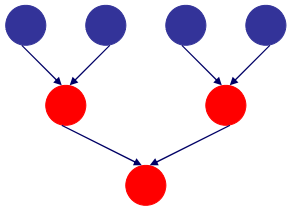


Task graph of pAlgorithm



A task is a work function and the set of data to process.

- = Find sum of elements
- = Combine partial results

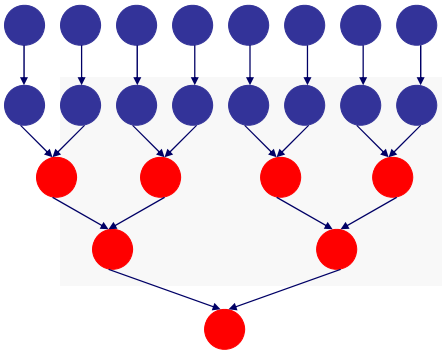


Tasks aren't independent.

Dependencies specify execution order of tasks.

Composing Task Graphs

- Increases amount of concurrent work available
- Forms a MIMD computation
- Dependencies between tasks specified during composition



- = Add 7 to each element
- = Find sum of elements
- = Combine partial results

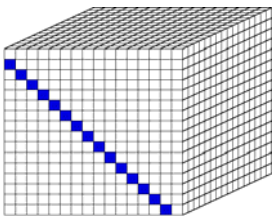
Dependencies only needed if tasks process the same data

Simple Dependence Specification

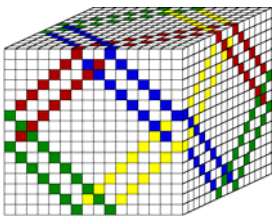
- Goal: Developer concisely expresses dependencies
  - Enumeration of dependencies is unmanageable
- Common patterns will be supported in pRange
  - Sequential – sources depend on sinks
  - Independent – no new dependencies needed in composed graph
  - Pipelined – dependencies follow a regular pattern

Discrete Ordinates Particle Transport Computation

- Important application for DOE
  - E.g., Sweep3D (3D Discrete Ordinates Neutron Transport) and UMT2K (Unstructured Mesh Transport)
- Large, on-going DOE project at TAMU to develop application in STAPL (TAXI)

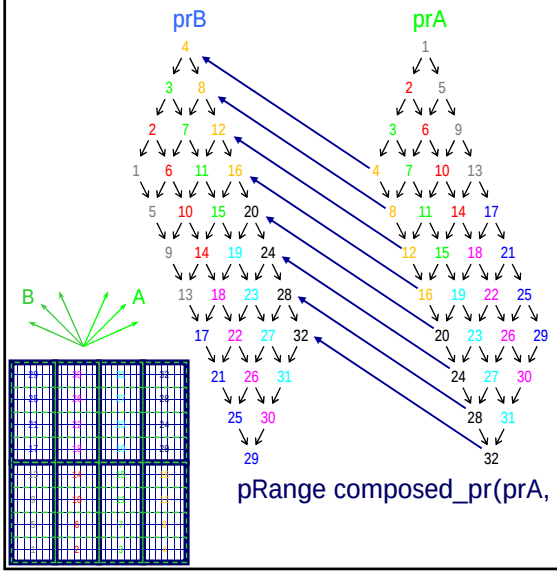


One sweep



Eight simultaneous sweeps

Pipeline Pattern Example



- pRanges are sweeps in particle transport application
- Reflective materials on problem boundary create dependencies
- Pipeline pattern will allow easy composition

```
pRange composed_pr(prA, prB, Pipeline(4,32,4));
```

pRange Summary

- Binds the work of an algorithm to the data
- **Simplifies** programming task graphs
  - Methods to create tasks
  - Common dependence pattern specifications
  - Compact specification of task dependencies
  - Manages task refinement
  - Simple specification of task graph composition
- Supports multiple programming models
  - Data-parallelism
  - Task-parallelism

STAPL Example - p_count

Implementation

```
template<typename View, typename Predicate>
class p_count_wf {
  //constructor - init member m_pred

  plus<result_type> combine_function(void)
  { return plus<result_type>(); }

  template<typename ViewSet>
  size_t operator()(ViewSet& vs)
  {
    return count_if(vs.sv0().begin(),
           vs.sv0().end(), m_pred);
  }
};

template<typename View, typename Predicate>
p_count_if(View& view, Predicate pred) {
  typedef p_count_wf<View,Predicate>   wf_t;
  wf_t wf(pred);
  return pRange<View, wf_t>(view, wf).execute();
}
```

Example Usage

```
stapl_main() {
  p_vector<int>             vals;
  p_vector<int>::view_type  view
    = vals.create_view();

  ... //initialize

  int ret = p_count(view,
   less_than(5));
}
```

STAPL Example - p_dot_product

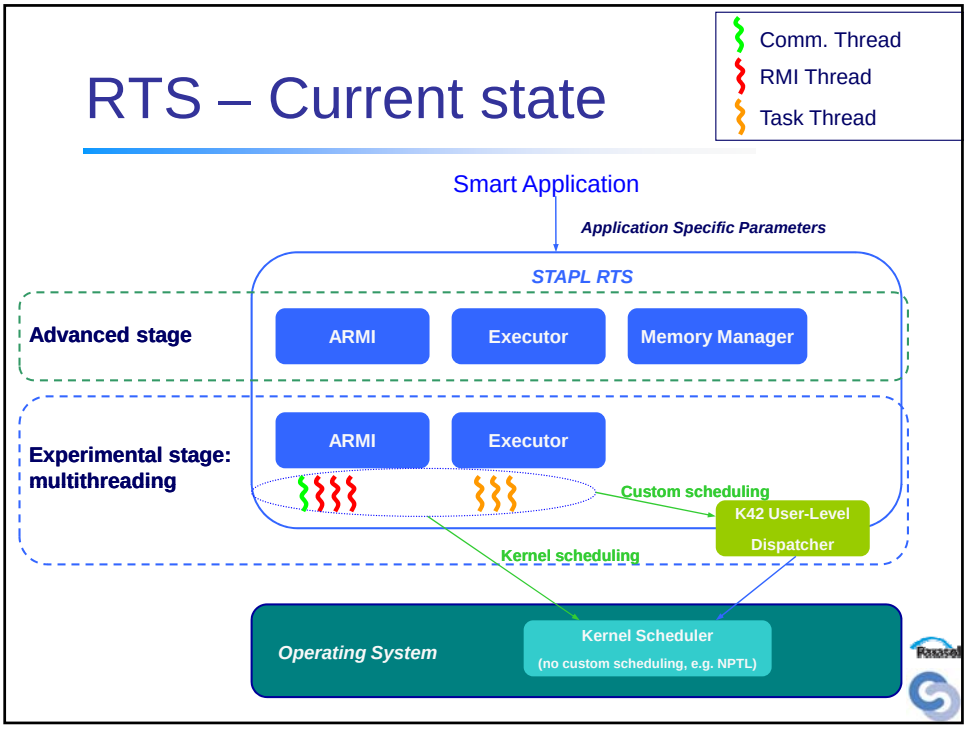
Implementation

```
template<typename View>
class p_dot_product_wf {
  plus<result_type> get_combine_function(void)
  { return plus<result_type>(); }

  template<typename ViewSet>
  result_type operator()(ViewSet& vs)
  {
    result_type result = 0;
    ViewSet::view0::iterator i = vs.sv0().begin();
    ViewSet::view1::iterator j = vs.sv1().begin();
    for(; i!=vs.sv0.end(); ++i, ++j) {
     result += *i * *j;
  }
};

template<typename View1, typename View2>
p_dot_product(View1& vw1, View2& vw2) {
  typedef p_dot_product_wf<View1, View2>   wf_t;
  wf_t wf;
  return pRange<View1, View2, wf_t>(vw1, vw2, wf).execute();
}
```

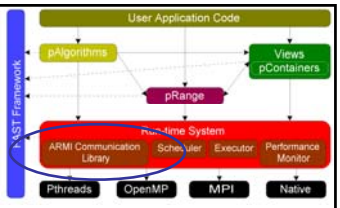
Example Usage

```
stapl_main() {
  p_vector<int>             vals;
  p_vector<int>::view_type  view1
    = vals.create_view();

  p_vector<int>             more_vals;
  p_vector<int>::view_type  view2
    = more_vals.create_view();

  ... //initialize

  int ret = p_dot_product(view1, view_2);
}
```

RTS – Current state



ARMI – Current State



ARMI: Adaptive Remote Method Invocation

- Abstraction of shared-memory and message passing communication layer (MPI, pThreads, OpenMP, mixed, Converse).
- Programmer expresses fine-grain parallelism that ARMI adaptively coarsens to balance latency versus overhead.
- Support for sync, async, point-to-point and group communication.
- Automated (de)serialization of C++ classes.

ARMI can be as easy/natural as shared memory and as efficient as message passing.

ARMI Communication Primitives

Point to Point Communication

armi_async - non-blocking: doesn't wait for request arrival or completion.

armi_sync - blocking and non-blocking versions.

Collective Operations

armi_broadcast, armi_reduce, etc.

can adaptively set groups for communication.

Synchronization

armi_fence, armi_barrier - fence implements distributed termination algorithm to ensure that all requests sent, received, and serviced.

armi_wait - blocks until at least at least one request is received and serviced.

armi_flush - empties local send buffer, pushing outstanding to remote destinations.

RTS – Multithreading (ongoing work)

In ARMI

- Specialized communication thread dedicated the emission and reception of messages
  - Reduces latency, in particular on SYNC requests
- Specialized threads for the processing of RMIs
  - Uncovers additional parallelism (RMIs from different sources can be executed concurrently)
  - Provides a suitable framework for future work on relaxing the consistency model and on the speculative execution of RMIs

In the Executor

- Specialized threads for the execution of tasks
  - Concurrently execute ready tasks from the DDG (when all dependencies are satisfied)

RTS Consistency Models

Processor Consistency (default)
- Accesses from a processor on another's memory are sequential
- Requires in-order processing of RMIs
  - Limited parallelism

Object Consistency
- Accesses to different objects can happen out of order
- Uncovers fine-grained parallelism
  - Accesses to different objects are concurrent
  - Potential gain in scalability
- Can be made default for specific computational phases

Mixed Consistency
- Use Object Consistency on select objects
  - Selection of objects fit for this model can be:
    - Elective – the application can specify that an object's state does not depend on others' states.
    - Detected – if it is possible to assert the absence of such dependencies
- Use Processor Consistency on the rest

RTS Executor

Customized task scheduling
- Executor maintains a ready queue (all tasks for which dependencies are satisfied in the DDG)
- Order tasks from the ready queue based on a scheduling policy (e.g. round robin, static block or interleaved block scheduling, dynamic scheduling …)
- The RTS decides the policy, but the user can also specify it himself
- Policies can differ for every pRange

Customized load balancing
- Implement load balancing strategies (e.g. work stealing)
- Allow the user to choose the strategy
- K42 : generate a customized work migration manager

RTS Synchronization

- Efficient implementation of synchronization primitives is crucial
  - One of the main performance bottlenecks in parallel computing
  - Common scalability limitation

Fence
  - Efficient implementation using a novel Distributed Termination Detection algorithm

Global Distributed Locks
  - Symmetrical implementation to avoid contention
  - Support for logically recursive locks (required by the compositional SmartApps framework)

Group-based synchronization
  - Allows efficient usage of ad-hoc computation groups
  - Semantic equivalent of the global primitives
  - Scalability requirement for large-scale systems

Productivity

- Implicit parallelism
- Implicit synchronizations/communications
- Composable (closed under composition)
- Reusable (library)
- Tunable by experts (library not language)
- Compiles with any C++ compiler (GCC)
- Optionally exposes machine info
- Shared Memory view for user
- High level of abstraction – Generic Programming

Performance

- Latency reduction: Locales , data distribution
- Latency Hiding: RMI, multithreading, Asynch Communications
- Optionally exposes machine info
- Manually tunable for experts
- Adaptivity to input and machine (machine learning)

Portability

- Library – no need for special compiler
- RTS needs to be ported – not much else
- High level of abstraction

References

Cilk

http://supertech.csail.mit.edu/cilk/

http://supertech.csail.mit.edu/cilk/manual-5.4.6.pdf

Dag-Consistent Distributed Shared Memory,
Blumofe, Frigo, Joerg, Leiserson, and Randall, In 10th International Parallel Processing Symposium (IPPS '96), April 15-19, 1996, Honolulu, Hawaii, pp. 132-141.

TBB

http://www.intel.com/cd/software/products/asmo-na/eng/294797.htm

TBB Reference Manual – provided with package

References

- HPF
  - HPFF Homepage - http://hpff.rice.edu/
  - High Performance Fortran: history, overview and current developments. H Richardson, Tech. Rep. TMC-261, Thinking Machines Corporation, April 1996.
- Chapel
  - http://chapel.cs.washington.edu/
  - Chapel Draft Language Specification.
    http://chapel.cs.washington.edu/spec-0.702.pdf
  - An Introduction to Chapel: Cray's High-Productivity Language.
    http://chapel.cs.washington.edu/ChapelForAHPCRC.pdf

References

- Fortress
  - http://research.sun.com/projects/plrg
  - Fortress Language Specification. http://research.sun.com/projects/plrg/fortress.pdf
  - Parallel Programming and Parallel Abstractions in Fortress. Guy Steele. http://irbseminars.intel-research.net/GuySteele.pdf
- Stapl
  - http://parasol.tamu.edu/groups/rwergergroup/research/stapl
  - A Framework for Adaptive Algorithm Selection in STAPL, Thomas, Tanase, Tkachyshyn, Perdue, Amato, Rauchwerger, In *Proc. ACM SIGPLAN Symp. Prin. Prac. Par. Prog. (PPOPP)*, pp. 277-288, Chicago, Illinois, Jun 2005.

Table of Contents

UPC

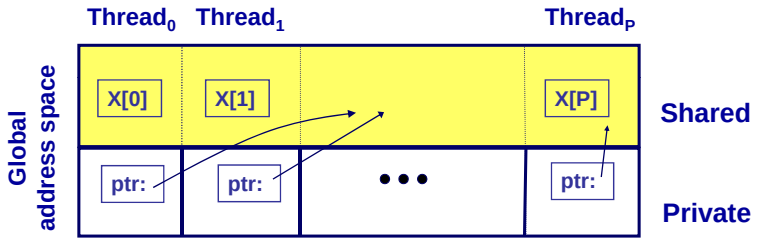
- Unified Parallel C
- An explicit parallel extension of ISO C
- A partitioned shared memory parallel programming language
- Similar to the C language philosophy
  - Programmers are clever

Adapted from http://www.upc.mtu.edu/SC05-tutorial

Execution Model

- UPC is SPMD
  - Number of threads specified at compile-time or run-time;
  - Available as program variable **`THREADS`**
  - **`MYTHREAD`** specifies thread index (**`0..THREADS-1`**)
- There are two compilation modes
  - Static Threads mode:
    - THREADS is specified at compile time by the user
    - THREADS as a compile-time constant
  - Dynamic threads mode:
    - Compiled code may be run with varying numbers of threads

UPC is PGAS



- The languages share the global address space abstraction
  - Programmer sees a single address space
  - Memory is logically partitioned by processors
  - There are only two types of references: local and remote
  - One-sided communication

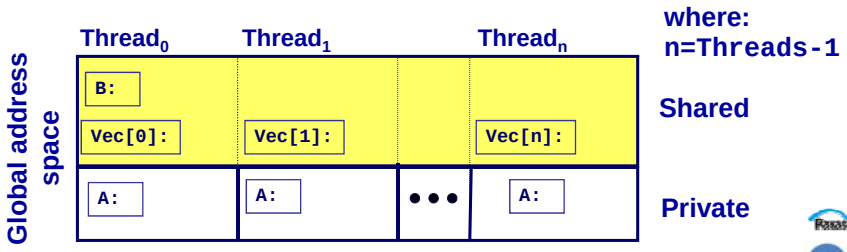
Hello World

- Any legal C program is also a legal UPC program
- UPC with P threads will run P copies of the program.
- Multiple threads view

```
#include <upc.h>  /* UPC extensions */
#include <stdio.h>

main() {
  printf("Thread %d of %d: hello UPC
  world\n", \
         MYTHREAD, THREADS);
}
```

Private vs. Shared Variables

- Private scalars (`int A`)
- Shared scalars (`shared int B`)
- Shared arrays (`shared int Vec[TREADS]`)
- Shared Scalars are always in threads 0 space
- A variable local to a thread is said to be **affine** to that thread



Data Distribution in UPC

- Default is cyclic distribution
  - `shared int V1[N]`
  - Element `i` affine to thread `i%THREADS`
- Blocked distribution can be specified
  - `shared [K] int V2[N]`
  - Element `i` affine to thread `(N/K)%THREADS`
- Indefinite ()
  - `shared [0] int V4[4]`
  - all elements in one thread
- Multi dimensional are linearized according to C layout and then previous rules applied

Work Distribution in UPC

- UPC adds a special type of loop

```
upc_forall(init; test; loop; affinity)
   statement;
```

- Affinity does not impact correctness but only performance
- Affinity decides which iterations to run on each thread. It may have one of two types:
  - Integer: **`affinity%THREADS`** is **`MYTHREAD`**
  - E.g., **`upc_forall(i=0; i<N; i++; i)`**
  - Pointer: **`upc_threadof(affinity)`** is **`MYTHREAD`**
  - E.g., **`upc_forall(i=0; i<N; i++; &vec[i])`**

UPC Matrix Multiply

```
#define N 4
#define P 4
#define M 4
// Row-wise blocking:
shared [N*P/THREADS] int a[N][P], c[N][M];
// Column-wise blocking:
shared[M/THREADS] int b[P][M];

void main (void) {
  int i, j , l; // private variables

  upc_forall(i = 0 ; i<N ; i++; &c[i][0])
    for (j=0 ; j<M ;j++) {
      c[i][j] = 0;
      for (l= 0 ; l<P ; l++)
        c[i][j] += a[i][l]*b[l][j];
    }
}
```

0	0	0	0
1	1	1	1
2	2	2	2
3	3	3	3

0	1	2	3
0	1	2	3
0	1	2	3
0	1	2	3

Replicating **b** among processors would improve performance

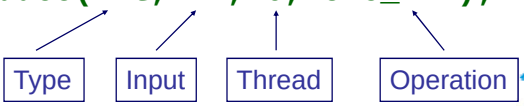
Synchronization and Locking

- Synchronization
  - Barrier: block until all other threads arrive
    - `upc_barrier`
  - Split-phase barriers
    - `upc_notify` this thread is ready for barrier
    - `upc_wait` wait for others to be ready
- Locks: `upc_lock_t`
  - Use to enclose critical regions
    - `void upc_lock(upc_lock_t *l)`
    - `void upc_unlock(upc_lock_t *l)`
  - Lock must be allocated before use

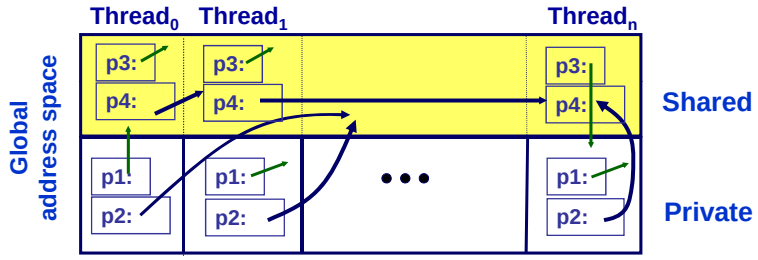
Collectives

- Must be called by all the threads with same parameters
- Two types of collectives
  - Data movement: scatter, gather, broadcast,…
  - Computation: reduce, prefix, …
- When completed the threads are synchronized
- E.g.,

```
res=bupc_allv_reduce(int, in, 0, UPC_ADD);
```



UPC Pointers




```
int *p1;         /* private pointer to local memory */
shared int *p2;  /* private pointer to shared space */
int *shared p3; /* shared pointer to local memory */
shared int  *shared p4; /* shared pointer to
                           shared space */
```

- Pointers-to-shared are more costly to dereference
- The use of shared pointers to local memory are discouraged

Memory Consistency

- UPC has two types of accesses:
  - Strict: Will always appear in order
  - Relaxed: May appear out of order to other threads
- There are several ways of designating the type, commonly:
  - Use the include file:

    ```
    #include <upc_relaxed.h>
    ```
  - All accesses in the program unit relaxed by default
  - Use strict on variables that are used as synchronization (`strict shared int flag;`)

    ```
    data = …              while (!flag) { };
    flag = 1;             … = data;   // use the data
    ```

Additional Features

- Latency management: two levels of proximity exposed to the user
- Portability: UPC compilers are available for many different architectures
- Productivity: UPC is a low-level language, the main objective is performance