ARMI: A Communication Infrastructure for STAPL

Nathan Thomas
Department of Computer Science
Texas A&M University

ASCI LabFest 2003
ARMI: Adaptive Remote Method Invocation

- abstraction of shared-memory and message passing communication layer
- programmer expresses fine-grain parallelism that ARMI adaptively coarsens
- support for blocking, non-blocking, point-to-point and group communication
The STAPL Programming Environment

```
+--------------------------+
| User Code               |
|                         |
| pAlgorithms             |
| pRange                 |
| pContainers             |
|                         |
+--------------------------+
```

```
+--------------------------+
| ARM1                    |
|                         |
| Pthreads                |
| OpenMP                  |
| MPI                     |
| Native                  |
+--------------------------+
```
Overview

- Comparison of Communication Models
  - RMI can be as easy/natural as shared memory and as efficient as message passing
- ARMI Programming Interface
- ARMI Runtime Environment
- ARMI in TAXI
# Common Communication Models

<table>
<thead>
<tr>
<th>Model</th>
<th>Address Space</th>
<th>Communication</th>
<th>Synchronization</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message Passing</td>
<td>private</td>
<td>matching send/receive</td>
<td>semantics of send/receive</td>
<td>MPI-1.1</td>
</tr>
<tr>
<td>One-Sided</td>
<td>private with shared sections</td>
<td>put/get</td>
<td>fences and locks</td>
<td>MPI-2, SHMEM, ARMCI</td>
</tr>
<tr>
<td>Shared Memory</td>
<td>shared</td>
<td>load/store</td>
<td>locks and semaphores</td>
<td>Pthreads, OpenMP</td>
</tr>
<tr>
<td>Remote Method Invocation</td>
<td>local &amp; remote objects</td>
<td>RMI request</td>
<td>semantics of RMI request</td>
<td>Java RMI, Charm++, Nexus</td>
</tr>
</tbody>
</table>
Disadvantages of the Models

- Shared-Memory
  - lack of explicit data distribution mechanisms
  - unsupported on most large machines

- Message Passing
  - harder to program
    - matching sends and receives
  - If coarse grain parallelization “style” used
    - may increase critical path (BSP copy-in,copy-out)
    - may fail to fully exploit clusters of SMP’s
ARMI Goals & Contributions

Higher level of abstraction enables programmer to concentrate on algorithmic requirements and STAPL RTS selects implementation

- **communication library natural to OO programs**
  - RMI supports data hiding, encapsulation, etc

- **portable efficiency**
  - Programmer expresses maximum (fine-grained) parallelism and STAPL adaptively coarsens communication (and parallelism) to fit underlying architecture.
  - multi-protocol (e.g., MPI, OpenMP)

- **flexible primitives enabling expressive programming**
  - blocking and non-blocking communication primitives
  - Point to point, one-sided, and group communication
Case Study: Sample Sort

- Common parallel algorithm for sorting based on distributing data into buckets

- Algorithm:
  - sample a set of splitters
  - send elements to appropriate bucket based on splitters (e.g., elements less than splitter 0 are sent to bucket 0)
  - sort each bucket
Sample Sort: Shared-Memory

... std::vector< std::vector<int> > buckets( p );
std::vector< lock > locks( p );
...
...fork threads...
...
for( int i=0; i<size; i++ ) {
    int dest = //...appropriate bucket...
    locks[dest].lock();
    buckets[dest].push_back( input[i] );
    locks[dest].unlock();
}
barrier();
...
Sample Sort: Message Passing

```cpp
std::vector< int > bucket;
std::vector< std::vector<int> > outBuckets(p);
...
for( int i=0; i<size; i++ ) {
    int dest = //...appropriate bucket...
    outBuckets[dest].push_back( input[i] );
}
for( int i=0; i<p; ++i )
    if( i != my_id )
        Send( outBuckets[i], i, ... );
for( int i=0; i<p; ++i )
    if( i != my_id )
       Recv( bucket, ... );
...
```
Sample Sort: STAPL

... stapl::pvector< vector<int> > buckets(p);
...
for( int i=0; i<size; i++ ) {
    int dest = //...appropriate bucket...
    stapl::armi_async(dest,pv_handle,push_back,input[i]);
}
stapl::armi_fence();
...

abstracts whether implementation:
• atomically processes each element, as in shared memory
• buffers all elements and sends at once, as in message passing,
• or, some combo of above, as determined for each platform
Overview

- Comparison of Communication Models
- ARMI Programming Interface
  - blocking & non-blocking communication and synchronization primitives
  - built-in support for collective & group operations
  - Argument packing
- ARMI Runtime Environment
- ARMI in TAXI
Communication Primitives

- **Statement** – tell a thread something (non-blocking)
  
  ```
  template<class Class, class Rtn, class Arg1...>
  void armi_async( int destThread, rmiHandle handle, Rtn (*method)(Arg1...), Arg1 a1... )
  ```

- **Question** – ask a thread for something (blocking)
  
  ```
  template<class Class, class Rtn, class Arg1...>
  Rtn armi_sync( int destThread, rmiHandle handle,
                 Rtn (*method)(Arg1...), Arg1 a1... )
  ```
Communication Primitives

- **Question** – ask a thread for something (non-blocking)

```cpp
template<class Class, class Rtn, class Arg1...>
void armi_sync(int destThread, rmiHandle handle,
               Rtn (*method)(Arg1...), Arg1 al..., 
               OpaqueHandle<Rtn> * rtnHandle);
```

*Nonblocking* – function returns without answer, program can poll with `rtnHandle.ready()` and then access RMI’s return value with `rtnHandle.value()`.

- **Collective Operations**

```cpp
template<class Class, class Arg>
void armi_broadcast(Arg *inout, rmiHandle, funcPtr, root)
```

```cpp
template<class Class, class Arg>
void armi_reduce(Arg *in, Arg *out, rmiHandle, funcPtr, root)
```

Parameters always passed by value!
Synchronization Primitives

- **armi_fence** - tree based barrier, implements distributed termination algorithm to ensure that all outstanding RMI requests have been sent, received, and serviced.

- **armi_wait** – blocks until at least one (possibly more) RMI requests is received and serviced.

- **armi_flush** – empties local send buffer, pushing outstanding RMI requests to remote destinations.
Argument Packing

- User guided, semi-automatic packing
  - User defines type and layout of class data members
  - ARMI uses definition to automatically recursively packs
  - Leverages implicit template instantiation of C++
  - Completely automatable with compiler support

```c
class objectA {
    double a[10];
    objectB b;
    void define_type(typer& t){
        t.local( a, 10 );
        t.local( b );
    }
}
```

```c
class objectB {
    int size;
    int* array;
    void define_type(typer& t){
        t.local( size );
        t.dynamic( array, size );
    }
}
```

```
header  a  b.size  b.array
```
Argument Packing

- User defined packing
  - Useful for partial class packing / complex packing situations
  - Can coexist in programs with semi-automatic packing
  - Implemented using selective template specialization in C++

```
template<> struct typer_traits<MyClass> {
    static packType type = PACKED;
    static int packed_size(const MyClass *p) { ... }
    static void pack(MyClass *p, char *buffer) { ... }
    static void unpack(char *buffer, MyClass *p) { ... }
}
```
Overview

- The STAPL Programming Environment
- Comparison of Communication Models
- ARMI Programming Interface
- ARMI Runtime Environment
- ARMI in TAXI
Current Implementation Protocols

- Shared-Memory (OpenMP/Pthreads)
  - shared request queues

- Message Passing (MPI-1.1)
  - sends/receives

- Mixed-Mode
  - combination of MPI with threads
  - flat view of parallelism (for now)
    - take advantage of shared-memory
ARMI Request Handling

Since ARMI doesn’t require matching operations must detect & process incoming requests

- **Version 1: Common ARMI/Computation Threads**
  - trade-off local computation with incoming ARMI\(s\)
  - polling & interrupts used to process ARMI requests

- **Version 2: Dedicated Communication Threads**
  - communication thread handles sending and receiving of ARMI\(s\) for one or more computation threads
    - maintains separate send & receive buffers for each computation thread it services
  - computation thread owning data services request
Overview

- Comparison of Communication Models
- ARMI Programming Interface
- ARMI Runtime Environment
- ARMI in TAXI
ARMI in TAXI

- TAXI developers should see STAPL, not ARM
  – 10K+ lines of code in STAPL (and shrinking)
  – 1 armi_async, 3 armi_reduce calls, 15 armi_fences.
  – All other ARMI requests encapsulated within STAPL

- STAPL encapsulates ARMI, providing implicit communication and synchronization
  – pContainers provide shared object view, determining remote vs. local accesses.
  – pAlgorithms employ efficient parallel implementations allowing user to focus on the problem.
  – pRanges enable properly synchronized parallel execution that enforces user supplied dependencies.
ARMI in the pGrid

TAXI_work_function(pRange p) {
...  
    pGrid.setEdge(vertex, edge,
              BaseEdge::set_psiface,
              element, angleset, flux);
...  
}
ARMI in the pGrid

```cpp
pGrid::setEdge(vertex, edge, func, args) {
    ...
    If (edge is local)
        update edge
    else
        lookup owner
        async_rmi(owner, handle, set_edge,
                  func, args);
    ...
}
```
ARMI in STAPL Executor

- `p_for_all(pRange, work_function, scheduler = default)`
- Round robin over sub pRanges.
- pRange uses a pDDG to enforce dependencies.
- Mark edges to denote a pRange done.
Conclusions

- STAPL/ARMI raises the level of abstraction, hierarchically
  - At user level, communication & synchronization is implicit in pContainer and pAlgorithm use (don’t need to see ARMI)
  - At developer level, specify necessary communication and synchronization, but not their implementation
  - At STAPL RTS level, ARMI primitives ported to each machine enable automatic composition of multi-protocol implementations

- STAPL/ARMI enables portable efficiency
  - let programmer concentrate on algorithmic essentials
  - leave implementation to STAPL RTS