AM++: A Generalized Active Message Framework

Jeremiah Willcock, Torsten Hoefler, Nicholas Edmonds, and Andrew Lumsdaine
Large-Scale Computing

- Not just for PDEs anymore
- Many new, important HPC applications are data-driven ("informatics applications")
  - Social network analysis
  - Bioinformatics
Data-Driven Applications

- Different from “traditional” applications
  - Communication highly data-dependent
  - Little memory locality
  - Impractical to load balance
  - Many small messages to random nodes

- Computational ecosystem is a bad match for informatics applications
  - Hardware
  - Software
  - Programming paradigms
  - Problem solving approaches
Two-Sided (BSP) Breadth-First Search

while any rank’s queue is not empty:
  for i in ranks: out_queue[i] ← empty
  for vertex v in in_queue[*]:
    if color(v) is white:
      color(v) ← black
    for vertex w in neighbors(v):
      append w to out_queue[owner(w)]
  for i in ranks: start receiving in_queue[i] from rank i
  for j in ranks: start sending out_queue[j] to rank j
  synchronize and finish communications
Two-Sided (BSP) Breadth-First Search

- Rank 0
- Rank 1
- Rank 2
- Rank 3

Get neighbors

Redistribute queues

Combine received queues
Messaging Models

- **Two-sided**
  - MPI
  - Explicit sends and receives

- **One-sided**
  - MPI-2 one-sided, ARMCI, PGAS languages
  - Remote put and get operations
  - Limited set of atomic updates into remote memory

- **Active messages**
  - GASNet, DCMF, LAPI, Charm++, X10, etc.
  - Explicit sends, implicit receives
  - User-defined handler called on receiver for each message
Active Messages

- Created by von Eicken et al, for Split-C (1992)
- Messages sent explicitly
- Receivers register handlers but not involved with individual messages
- Messages often asynchronous for higher throughput
Active Message Breadth-First Search

**handler** `vertex_handler(vertex v)`:  
  
  ```python
  if color(v) is white:
    color(v) ← black
    append v to new_queue
  ```

  while any rank's queue is not empty:
    ```
    new_queue ← empty
    ```

    begin active message epoch
    for vertex v in queue:
      for vertex w in neighbors(v):
        ```
        tell owner(w) to run vertex_handler(w)
        ```
    end active message epoch
    queue ← new_queue
Active Message Breadth-First Search

- Get neighbors
- Send vertex messages
- Check color maps
- Insert into queues

Active message handler
Low-Level vs. High-Level AM Systems

- Active messaging systems (loosely) on a spectrum of features vs. performance
  - Low-level systems typically have restrictions on message handler behavior, explicit buffer management, etc.
  - High-level systems often provide dynamic load balancing, service discovery, authentication/security, etc.
The AM++ Framework

- AM++ provides a “middle ground” between low- and high-level systems
  - Gets performance from low-level systems
  - Gets programmability from high-level systems
- High-level features can be built on top of AM++
Key Characteristics

- For use by applications
- AM handlers can send messages
- Mix of generative (template) and object-oriented approaches
  - Object-orientation for flexibility and type erasure
  - Templates for optimal performance
- Flexible/application-specific message coalescing
- Messages sent to processes, not objects
Example

```cpp
mpi_transport trans(MPI_COMM_WORLD);

basic_coalesced_message_type<my_message_data, my_handler, mpi_transport>
    msg_type(trans, 256);

msg_type.set_handler(my_handler());

scoped_termination_detection_level_request<mpi_transport> td_req(trans, 0);
{
    scoped_epoch<mpi_transport> epoch(trans);
    if (trans.rank() == 0)
        msg_type.send(my_message_data(1.5), 2);
}
```

- **Create Message Transport** (Not restricted to MPI)
- **Coalescing layer** *(and underlying message type)*
- **Message Handler**
- **Messages are nested to depth 0**
- **Epoch scope**
AM++ Design
Transport

- Interface to underlying communication layer
  - MPI and GASNet currently
- Designed to send large messages produced by higher-level components
  - Object-oriented techniques allow run-time flexibility (type erasure)
- MPI-style progress model
  - Progress thread optional
  - User must call into AM++
Message Types

- Handler registration for messages within transport
- Type-safe interface to reduce user casts and errors
- Automatic data buffer handling
Termination Detection/Epochs

- AM++ handlers can send messages
  - When have they all been sent and handled?

- *Termination detection* – a standard distributed computing problem

- Some applications send a fixed depth of nested messages

- Time divided into epochs
Message Coalescing

- Standard way to amortize overheads
  - Trade off latency for throughput
- Layered on transport and message type
- Can be specific to application or message type
- Handlers apply to one small message at a time
- Sends are of a single small message
Message Handler Optimizations

- Coalescing uses generative programming and C++ templates for performance on high message rates
- Small-message handler type is known statically
- Simple loop calls handler
- Compiler can optimize using standard techniques
Message Reductions

- Some applications have messages that are
  - Idempotent: duplicate messages can be ignored
  - Reducible: some messages can be combined
- Detect some at sender
  - Cache
AM++ and Threads

- AM++ is thread-safe
- Models for thread use:
  - Run separate handlers in separate threads
  - Split a single message across several threads
- Coalescing buffer sizes affect parallelism in both models
Evaluation: Message Latency

Single-data-rate InfiniBand, GASNet 1.14.0 test section L
Evaluation: Message Bandwidth

![Graph showing bandwidth comparison between GASNet/IBV, GASNet/MPI, and AM++]

- GASNet/IBV
- GASNet/MPI
- AM++

Message Size vs. Bandwidth in MB/s

Single-data-rate InfiniBand, GASNet 1.14.0, test section L
Breadth-First Search: Strong Scaling

Single-data-rate InfiniBand, dual-socket dual-core, $2^{27}$ vertices, degree 4
Breadth-First Search: Weak Scaling

Single-data-rate InfiniBand, dual-socket dual-core, $2^{25}$ vertices/node, degree 4
Delta-Stepping: Strong Scaling

Single-data-rate InfiniBand, dual-socket dual-core, $2^{27}$ vertices, degree 4
Delta-Stepping: Weak Scaling

Single-data-rate InfiniBand, dual-socket dual-core, $2^{24}$ vertices/node, degree 4
Conclusion

- Generative programming techniques used to design a flexible active messaging framework, AM++
  - “Middle ground” between previous low-level and high-level systems
- Features can be composed on that framework
- Performance comparable to other systems