AM++: A Generalized Active Message Framework

Andrew Lumsdaine
Indiana University
Large-Scale Computing

- Not just for PDEs anymore
- Computational ecosystem is a bad match for informatics applications
  - Hardware
  - Software
  - Programming paradigms
  - Problem solving approaches
This talk

- About lessons learned in developing two generations of a distributed memory graph algorithms library
- Problem characteristics
- PBGL Classic and lessons learned
- AM++ overview
- Performance results
- Conclusions
Supercomputers, what are they good for?

Compute Bound

Bandwidth Bound

Latency Bound

\begin{verbatim}
while (! Q.empty()) {
    Vertex u = Q.top(); Q.pop();
    for (v in neighbors(u))
        if (color[v] == Color::white()) {
            color[v] = Color::gray();
            Q.push(v);
        }
    color[u] = Color::black();
}
\end{verbatim}

for (int i = 0; i < M; ++i)
    for (int j = 0; j < N; ++j)
        C[i][j] += A[i][k] * B[k][j];

for (int i = 0; i < M; ++i)
    for (int j = row[i]; j < row[i+1]; ++j)

while (! Q.empty()) {
    Vertex u = Q.top(); Q.pop();
    for (v in neighbors(u))
        if (color[v] == Color::white()) {
            color[v] = Color::gray();
            Q.push(v);
        }
    color[u] = Color::black();
}
Informatics Apps: Data Driven

- Data access is data dependent
- Communication is data dependent
- Execution flow is data dependent
- Little memory or communication locality
- Difficult or impossible to balance load well
- Latency-bound with many small messages

Scientific Applications

```c
for (int i = 0; i < M; ++i)
    for (int j = 0; j < N; ++j)
        for (int k = 0; k < K; ++k)
            C[i][j] += A[i][k] * B[k][j];
```

Benchmarks

```c
for (int i = 0; i < M; ++i)
    for (int j = row[i]; j < row[i+1]; ++j)
```

while (!Q.empty()) {
    Vertex u = Q.top(); Q.pop();
    for (v in neighbors(u)) {
        if (color[v] == Color::white()) {
            color[v] = Color::gray();
            Q.push(v);
        }
    }
    color[u] = Color::black();
}
Data-Driven Applications

- Many new, important HPC applications are data-driven ("informatics applications")
  - Social network analysis
  - Bioinformatics

- Different from "traditional" applications
  - Communication is highly data-dependent
  - Little memory or communication locality
  - Difficult or impossible to balance load well
  - Latency-bound with many small messages

- Current models do not fit these applications well
The Parallel Boost Graph Library

- **Goal**: To build a generic library of efficient, scalable, distributed-memory parallel graph algorithms.

- **Approach**: Apply advanced software paradigm (Generic Programming) to categorize and describe the domain of parallel graph algorithms. Separate concerns. Reuse sequential BGL software base.

- **Result**: Parallel BGL. Saved years of effort.
BGL: Algorithms (partial list)

- Searches (breadth-first, depth-first, A*)
- Single-source shortest paths (Dijkstra, Bellman-Ford, DAG)
- All-pairs shortest paths (Johnson, Floyd-Warshall)
- Minimum spanning tree (Kruskal, Prim)
- Components (connected, strongly connected, biconnected)
- Maximum cardinality matching
- Max-flow (Edmonds-Karp, push-relabel)
- Sparse matrix ordering (Cuthill-McKee, King, Sloan, minimum degree)
- Layout (Kamada-Kawai, Fruchterman-Reingold, Gursoy-Atun)
- Betweenness centrality
- PageRank
- Isomorphism
- Vertex coloring
- Transitive closure
- Dominator tree
Parallel BGL Architecture
Algorithms in the Parallel BGL (partial)

- Breadth-first search*
- Eager Dijkstra’s single-source shortest paths*
- Crauser et al. single-source shortest paths*
- Depth-first search
- Minimum spanning tree (Boruvka*, Dehne & Götz‡)
- Connected components‡
- Strongly connected components†
- Biconnected components
- PageRank*
- Graph coloring
- Fruchterman-Reingold layout*
- Max-flow†

* Algorithms that have been lifted from a sequential implementation
† Algorithms built on top of parallel BFS
‡ Algorithms built on top of their sequential counterparts
“Implementing” Parallel BFS

Generic interface from the Boost Graph Library

```
template<class IncidenceGraph, class Queue, class BFSVisitor, class ColorMap>
void breadth_first_search(const IncidenceGraph& g,
vertex_descriptor s, Queue& Q,
BFSVisitor vis, ColorMap color);
```

- Effect parallelism by using appropriate types:
  - Distributed graph
  - Distributed queue
  - Distributed property map

- Our sequential implementation is also parallel!
Breadth-First Search

\[
\text{put(color, s, Color::gray());} \\
\text{Q.push(s);} \\
\text{while (! Q.empty()) {} } \\
\text{Vertex u = Q.top(); Q.pop();} \\
\text{for (e in out_edges(u, g)) {} } \\
\text{Vertex v = target(e, g);} \\
\text{ColorValue v_color = get(color, v);} \\
\text{if (v_color == Color::white()) { } } \\
\text{put(color, v, Color::gray());} \\
\text{Q.push(v);} \\
\text{}} \\
\text{put(color, u, Color::black());} \\
\]
Two-Sided (BSP) Breadth-First Search

while any rank’s queue is not empty:
  for $i$ in ranks: $\text{out\_queue}[i] \leftarrow$ empty
  for vertex $v$ in $\text{in\_queue}[\ast]$:
    if color($v$) is white:
      color($v$) $\leftarrow$ black
      for vertex $w$ in neighbors($v$):
        append $w$ to $\text{out\_queue}[\text{owner}(w)]$

for $i$ in ranks: start receiving $\text{in\_queue}[i]$ from rank $i$
for $j$ in ranks: start sending $\text{out\_queue}[j]$ 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
PBGL: Lessons learned

- When MPI is your hammer
- All of your problems look like a thumb

- How you express your algorithm impacts performance
  - PBGL needs a data-driven approach
  - Data-driven expressiveness
  - Utilize underlying hardware efficiently
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
Data-Driven Breadth-First Search

**handler vertex_handler(vertex v):**

```
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

Rank 0
Rank 1
Rank 2
Rank 3

Active message handler
Active Messages

- Created by von Eicken et al, for Split-C (1992)
- Messages sent explicitly
- Receivers register handlers but are not involved with individual messages
- Messages typically asynchronous for higher throughput
The AM++ Framework

- AM++ provides a “middle ground” between low- and high-level systems
  - Gives up some performance for programmability
  - Give up some high-level features (such as built-in object load balancing) for performance and simplicity
- Missing features can be built on top of AM++
- Low level performance can be specialized

Diagram:

- AM++
- DCMF
- GASNet
- Charm++
- X10
- Java RMI
Important Characteristics

- Intended for use by applications
- AM handlers can send messages
- Mix of generative (template) and object-oriented approaches
  - OO for flexibility when small performance loss is OK
  - Templates when optimal performance is essential
- Flexible/application-specific message coalescing
  - Including sender-side message reductions
- Messages sent to processes, not objects
Example

```
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
Transport Lifetime

```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{
    scoped_epoch<mpi_transport>{
        if (trans.rank() == 0)
            msg_type.send(my_message data);
    }
}
```

- (1) Transport
- (2) Scope of Coalescing and Message Objects
- (3) Epoch
- (4) Message Handler Execution
- (5) Messages
- (6) Termination Detection
Resource Allocation Is Initialization

- Want to ensure cleanup of various kinds of “scoped” regions
  - Registrations of handlers
  - Epochs
  - Message nesting depths

- Resource Allocation Is Initialization (RAII) is a standard C++ technique for this
  - Object represents registration, epoch, etc.
  - Destructor ends corresponding region

- Exception-safe and convenient for users
Parallel BGL Architecture

- Communication Abstractions (MPI, Threads)
- Distributed Graph Data Structures
- Distributed Vertex/Edge Properties
- Transports
- Graph Concepts
- Property Map Concepts
- Parallel BGL Graph Algorithms
- BGL Graph Algorithms
AM++ Design

- Coalescing Message Type
- Message Type
- Reductions Coalescing Message Type

AM++ Transport

MPI or Vendor Communication Library

Termination Detection
TD Level
Epoch
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
Message Types

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

```c++
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);
}
```
Termination Detection/Epochs

- AM++ handlers can send messages
  - When have they all been sent and handled?
- Some applications send a fixed depth of nested messages
- Time divided into epochs (consistency model)
Message Coalescing

- Standard way to amortize overheads
- Layered on top of AM++ transport and message type
- Allows handlers that apply to one small message at a time
- Sends can be of a single small message

```c
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);
}```
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

```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<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);
```
Message Reductions

- Some applications have messages that are
  - Idempotent: duplicate messages can be ignored
  - Reducible: some messages can be combined
- Catch some of these sender-side

```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);
}
AM++ and Threads

- AM++ is thread-safe
  - MPI transport, coalescing, reductions
- Locking can be disabled for single-threaded use
- Can run separate handlers in separate threads
  - Each coalesced message processed in a single thread
- Or split a single message across several threads
  - Using OpenMP, etc. in the handler-call loop
- Coalescing buffer sizes affect parallelism in both models
  - But in different ways
Evaluation: Message Latency

![Graph showing latency vs message size for GASNet/IBV, GASNet/MPI, and AM++](image-url)
Evaluation: Message Bandwidth

![Graph showing message bandwidth evaluation for GASNet/IBV, GASNet/MPI, and AM++.]
Breadth-First Search: Strong Scaling

ER graph: $2^{27}$ vertices, $2^{29}$ edges
Breadth-First Search: Weak Scaling

![Graph showing time vs number of nodes for different algorithms.](image-url)
Delta-Stepping: Strong Scaling

![Graph showing time vs. number of nodes for different algorithms: PBGL, AM++ (t=1), AM++ (t=2), AM++ (t=4).]
Why MPI Worked

Distributed Memory Hardware

New Problems

Ad-Hoc Solutions

Folklore

Codification

Improved Practice

“Legacy MPI codes”

MPICH, LAM/MPI, Open MPI

NX Shmem P4, PVM Sockets

Message Passing Rules!
Multicore Ubiquity

Advance what works

- New Problems
- Models/Theory
- Ad-Hoc Solutions
- Folklore
- Codification

- Improved Practice

MPI
OpenMP
HPCS
PGAS
TM

???

INDIANA UNIVERSITY
PERSASIVE TECHNOLOGY INSTITUTE
Conclusion

- Data driven problems need data-driven messaging
- Generative programming techniques can be used to design a flexible active messaging framework, AM++
  - Intended for application programs/libraries
  - A “middle ground” between previous low-level and high-level systems
- Features can be composed on that framework
  - Application-specific message coalescing
  - Message reductions/duplicate removal
- Performance comparable to other systems and better than previous Parallel BGL