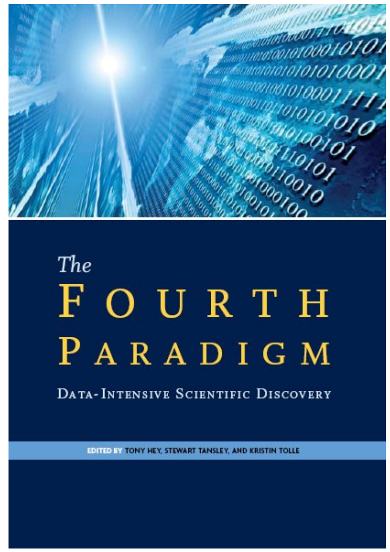
AM++: A Generalized Active Message Framework

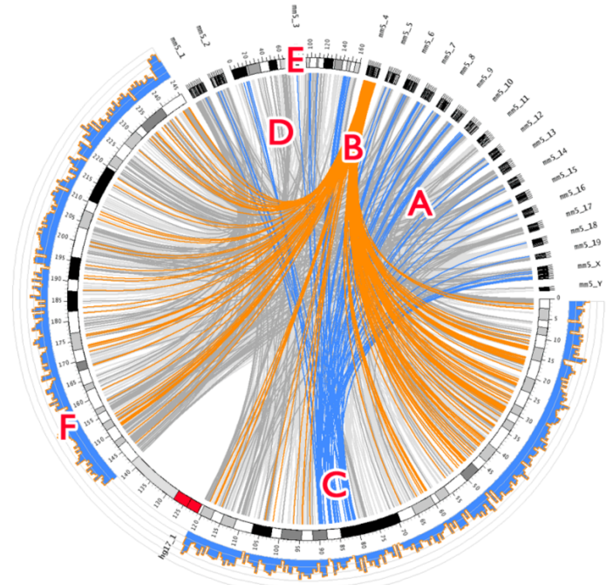
Andrew Lumsdaine
Indiana University

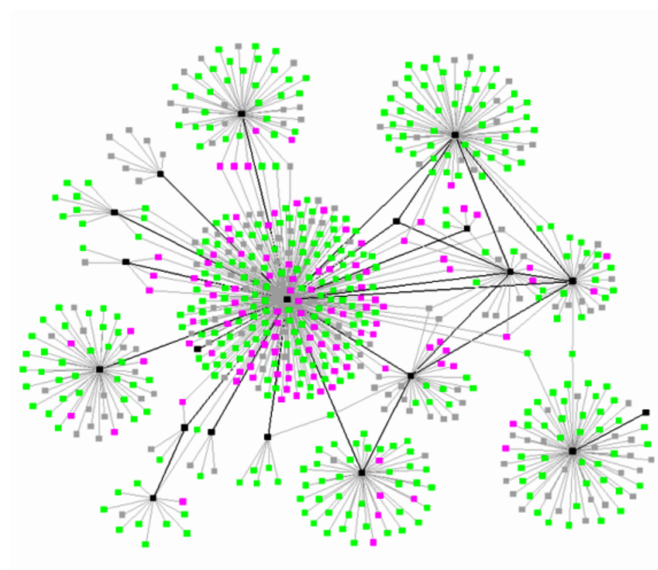
INDIANA UNIVERSITY
PERVASIVE TECHNOLOGY INSTITUTE

Large-Scale Computing

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







INDIANA UNIVERSITY
PERVASIVE TECHNOLOGY INSTITUTE

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?

```
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());
}
```

Compute Bound

```
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];
```

Bandwidth Bound

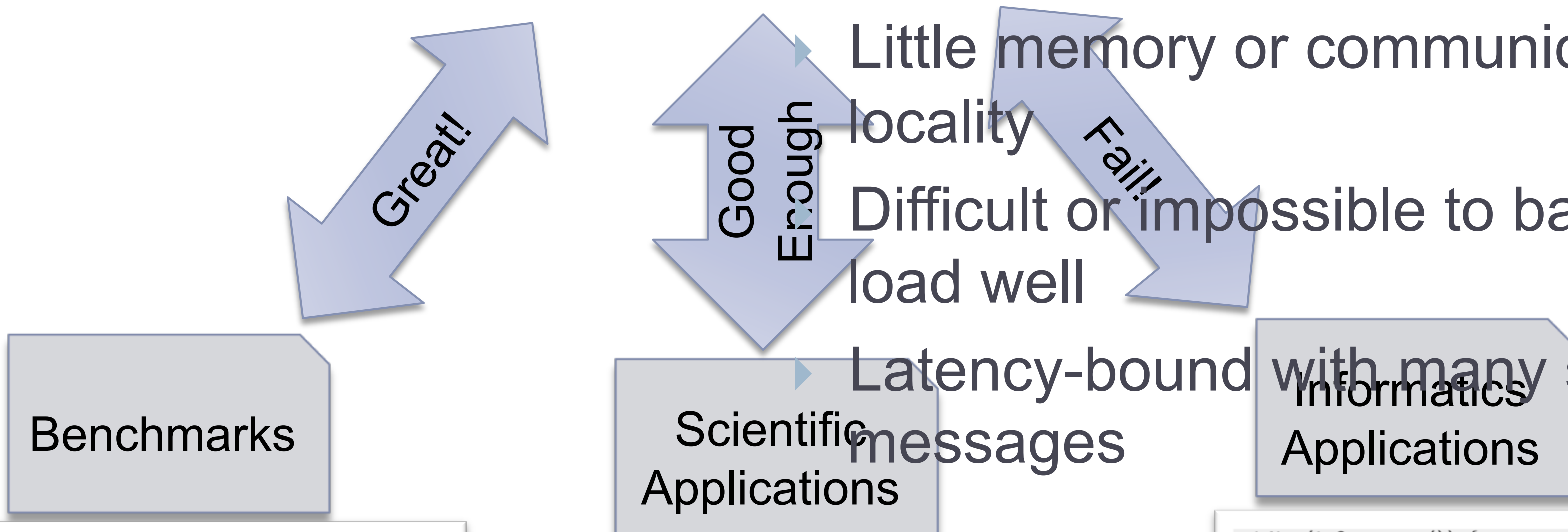
```
for (int i = 0; i < M; ++i)
  for (int j = row[i]; j < row[i+1]; ++j)
    Y[i] += A[j] * X[col[j]];
```

Latency Bound


```
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




```
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];
```

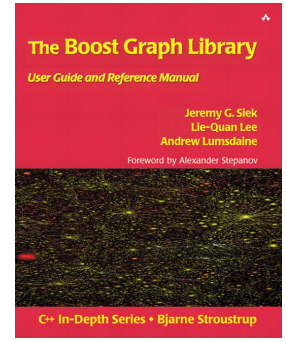
```
for (int i = 0; i < M; ++i)
  for (int j = row[i]; j < row[i+1]; ++j)
    Y[i] += A[j] * X[col[j]];
```

```
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

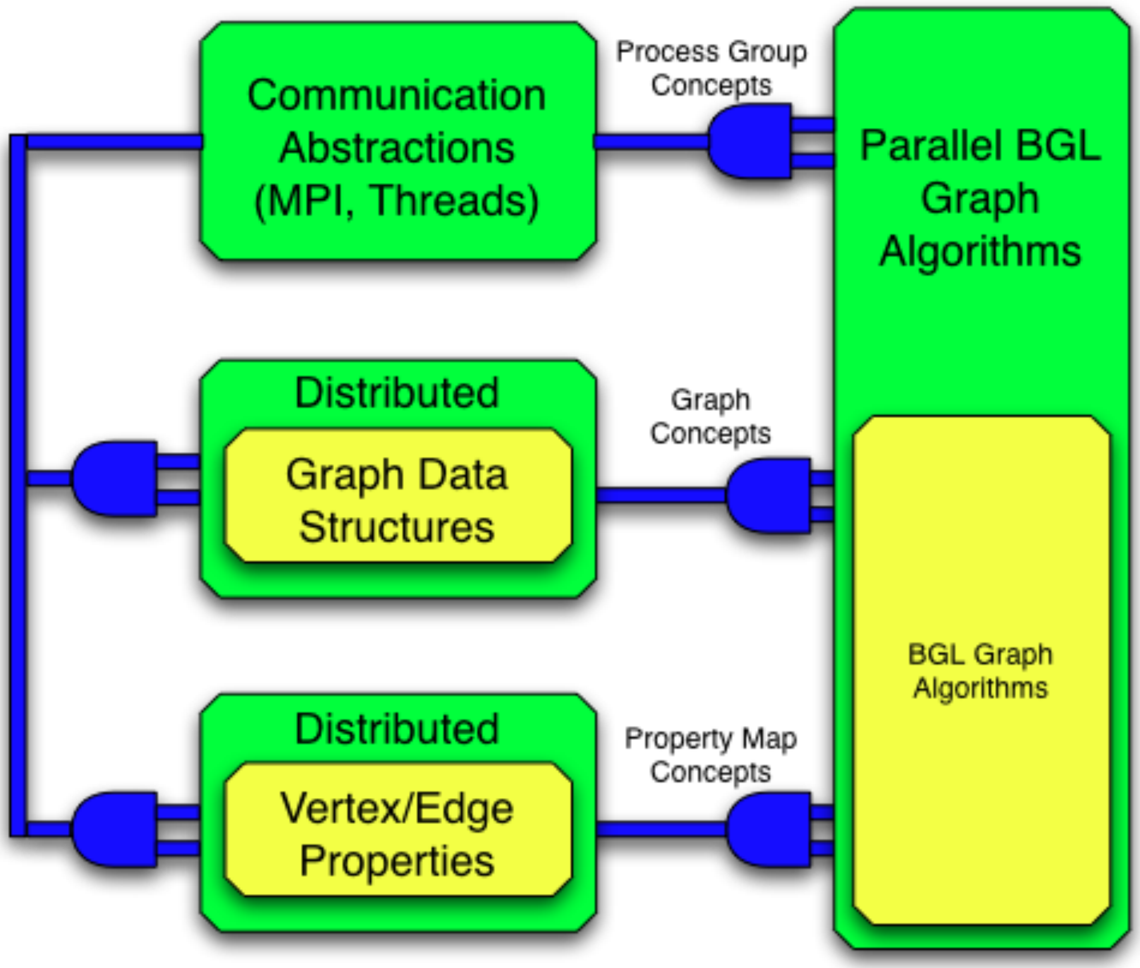


- ***Goal***: To build a generic library of efficient, scalable, distributed-memory parallel graph algorithms.
- ***Approach***: Apply advanced software paradigm (Generic Programming) to categorize and describe the domain of parallel graph algorithms. Separate concerns. Reuse sequential BGL software 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

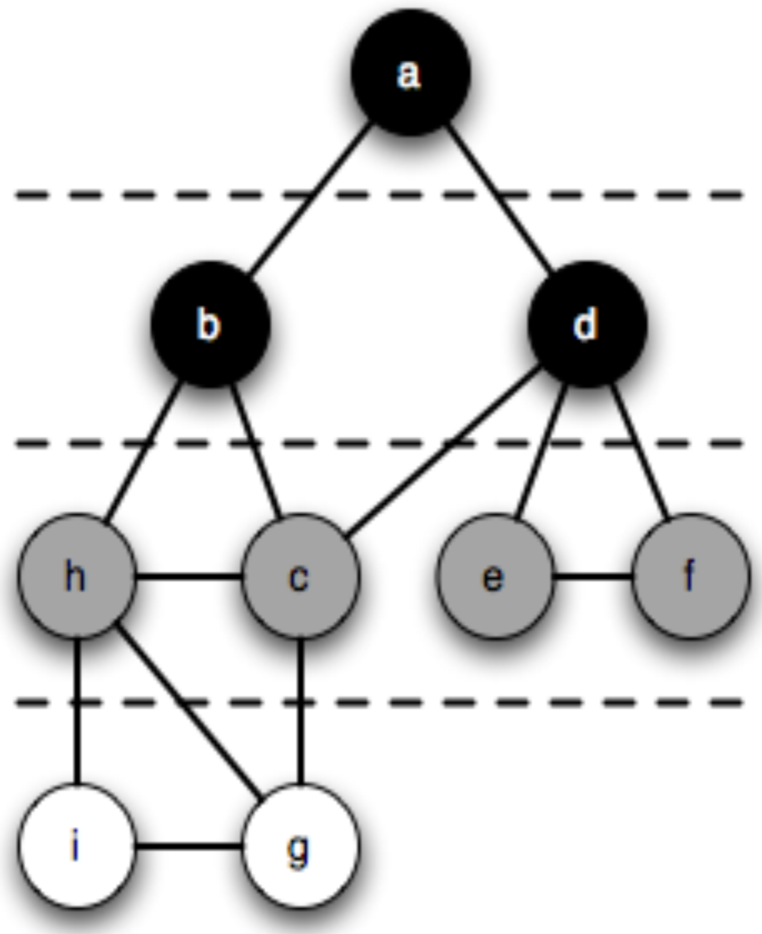
- Generic interface from the Boost Graph Library

```
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

```
put(color, s, Color::gray());
Q.push(s);
while (! Q.empty()) {
  Vertex u = Q.top(); Q.pop();
  for (e in out_edges(u, g)) {
    Vertex v = target(e, g);
    ColorValue v_color = get(color, v);
    if (v_color == Color::white()) {
      put(color, v, Color::gray());
      Q.push(v);
    }
  }
  put(color, u, Color::black());
}
```



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*):

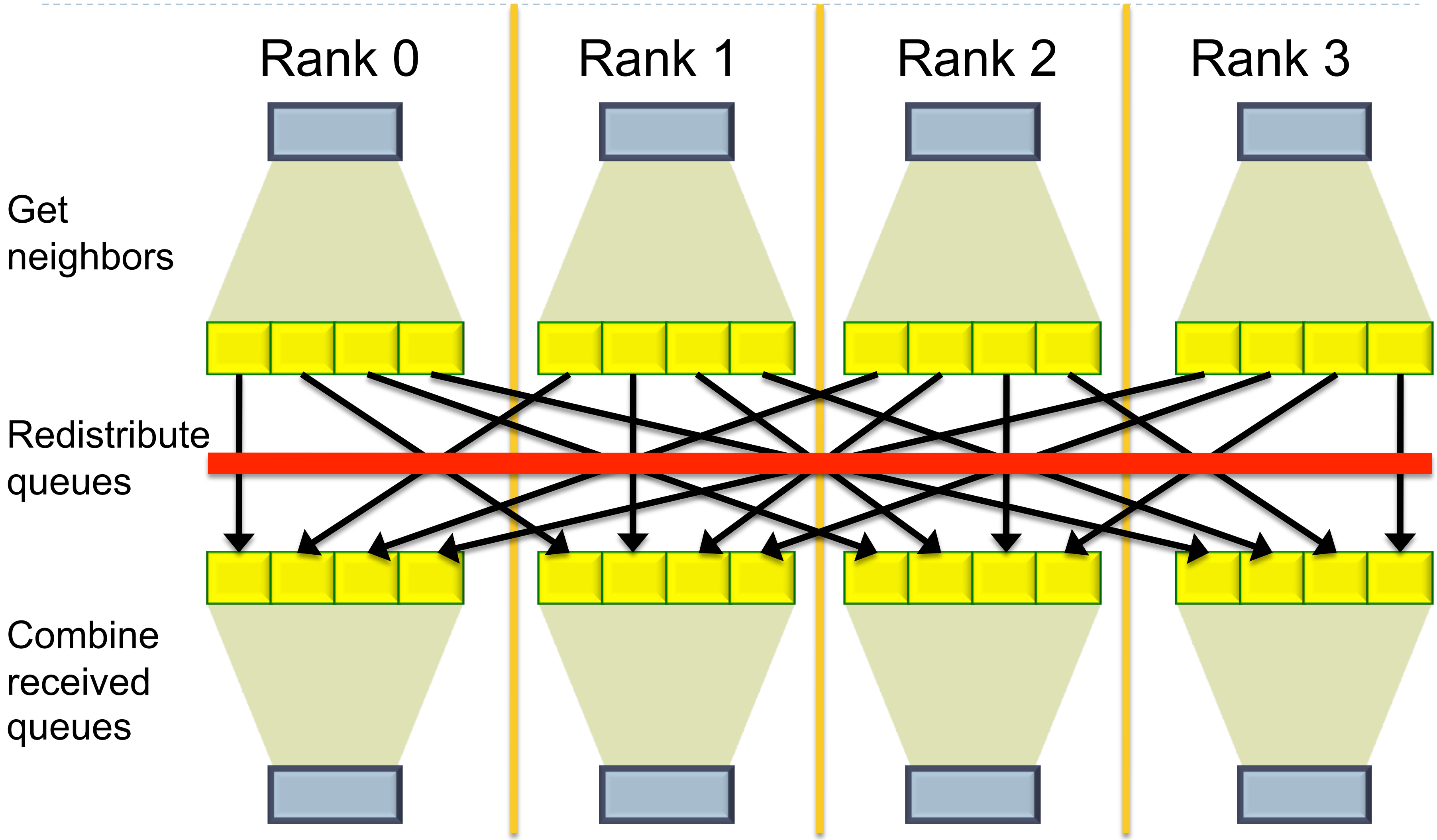
        append *w* **to** *out_queue*[owner(*w*)]

  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



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 expressivenes
  - Utilize underlying hardware efficiently

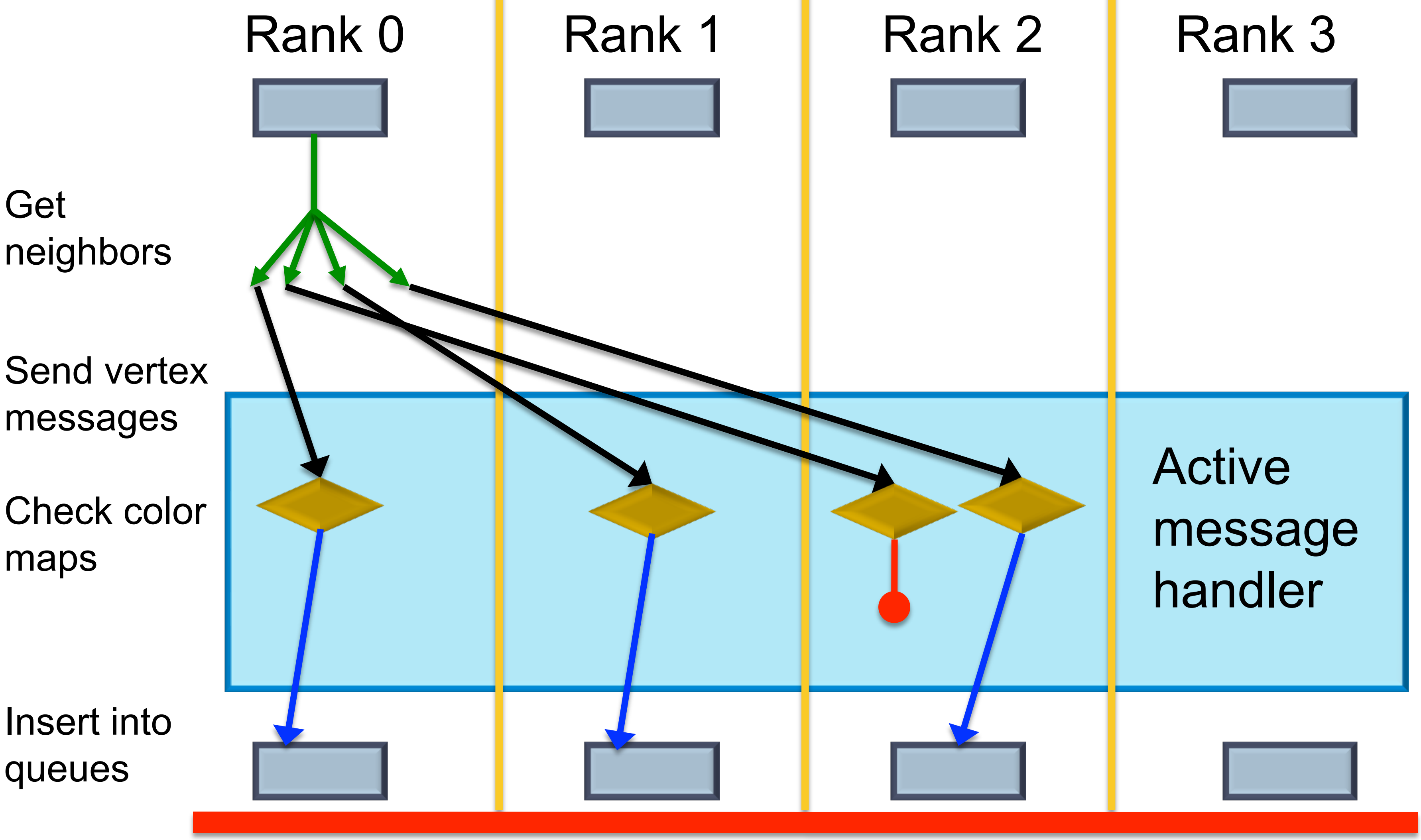
Messaging Models

- 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

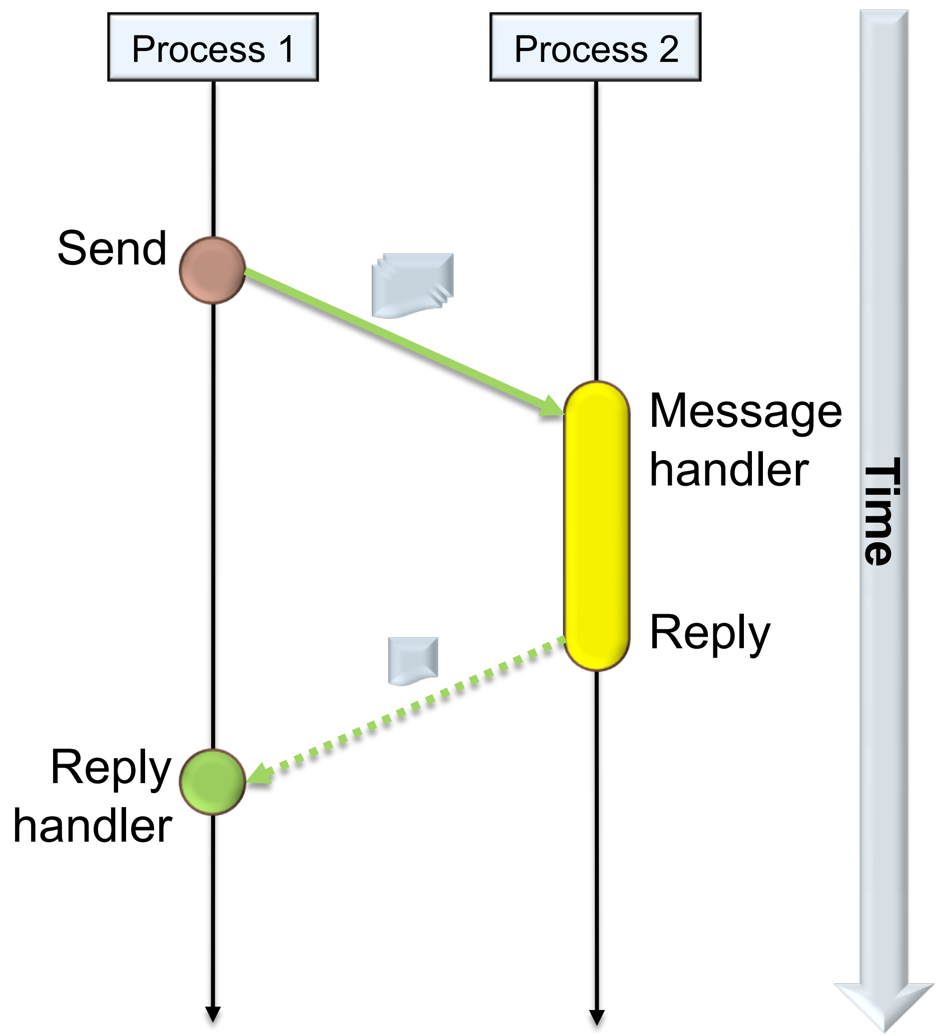
```
handler vertex_handler(vertex v):
  if color(v) is white:
    color(v) ← black
    append v to new_queue

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



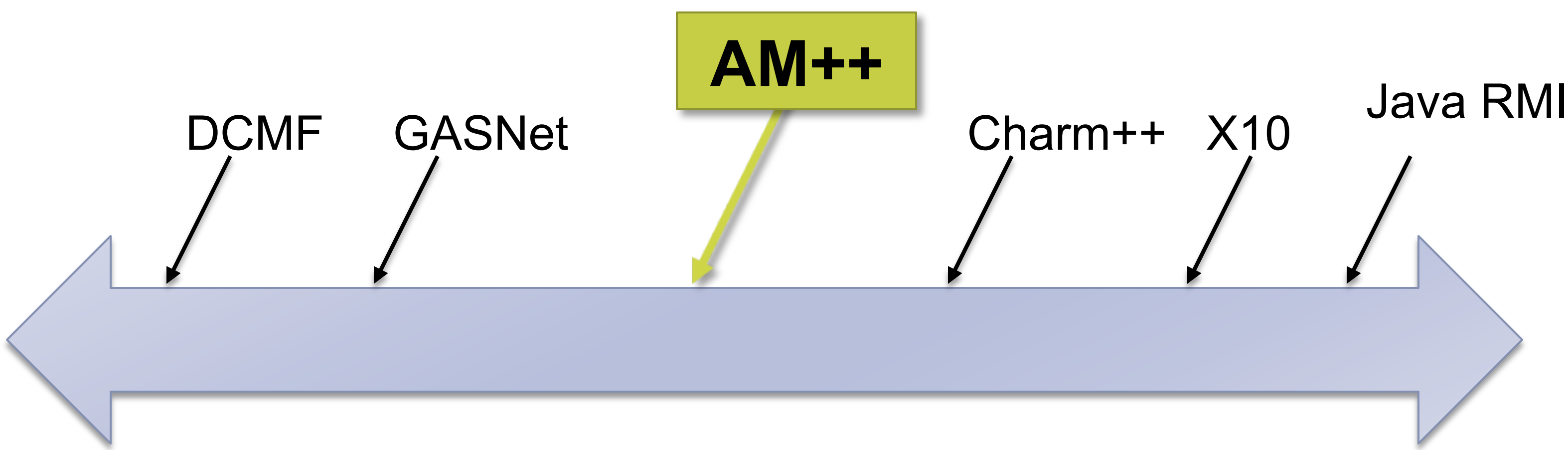
Active Messages

- 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



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)

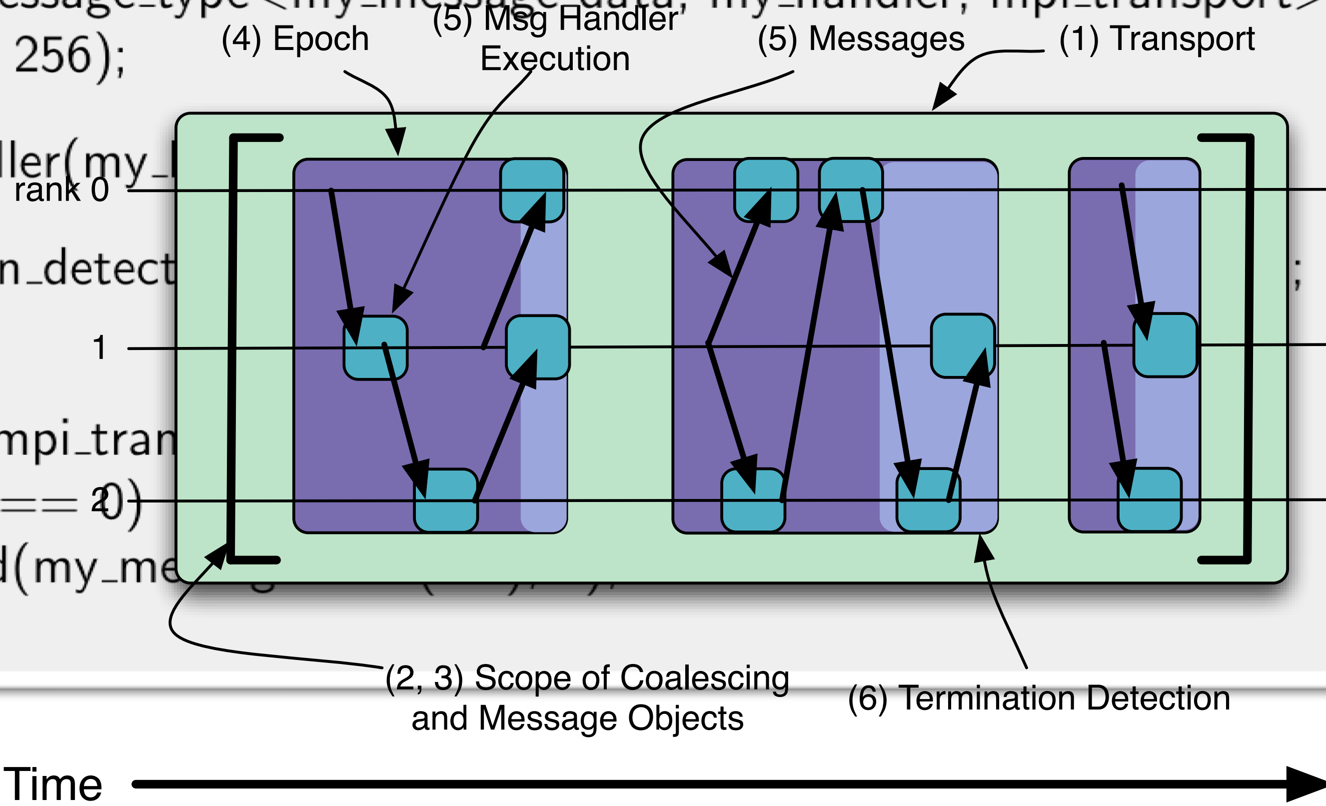
Message Handler

Messages are nested to depth 0

Epoch scope

Transport Lifetime

```
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_l

scoped_termination_detect                                                  ;

{
   scoped_epoch<mpi_tran
   if (trans.rank() == 0)
      msg_type.send(my_me
}
```

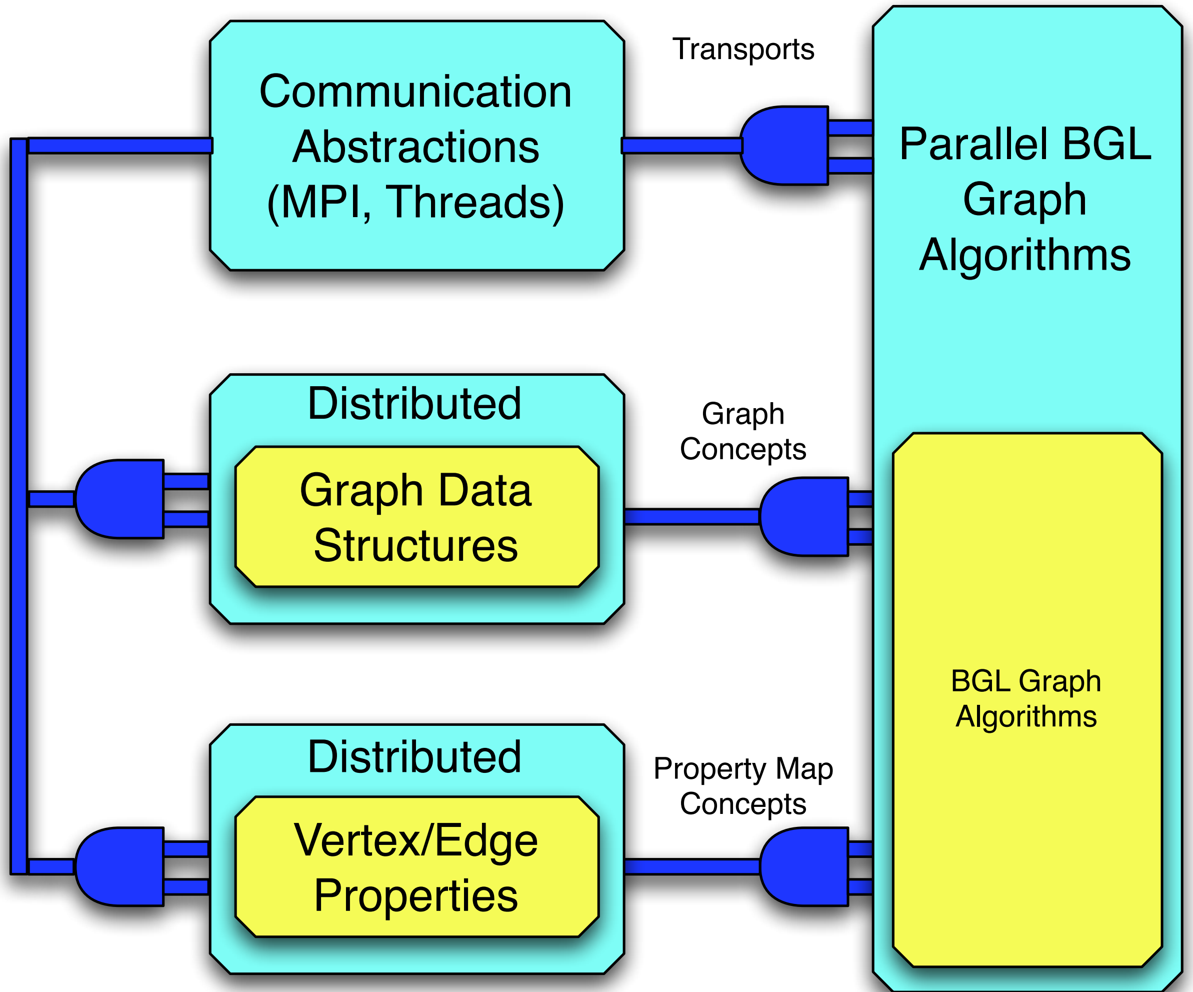


INDIANA UNIVERSITY
PERVASIVE TECHNOLOGY INSTITUTE

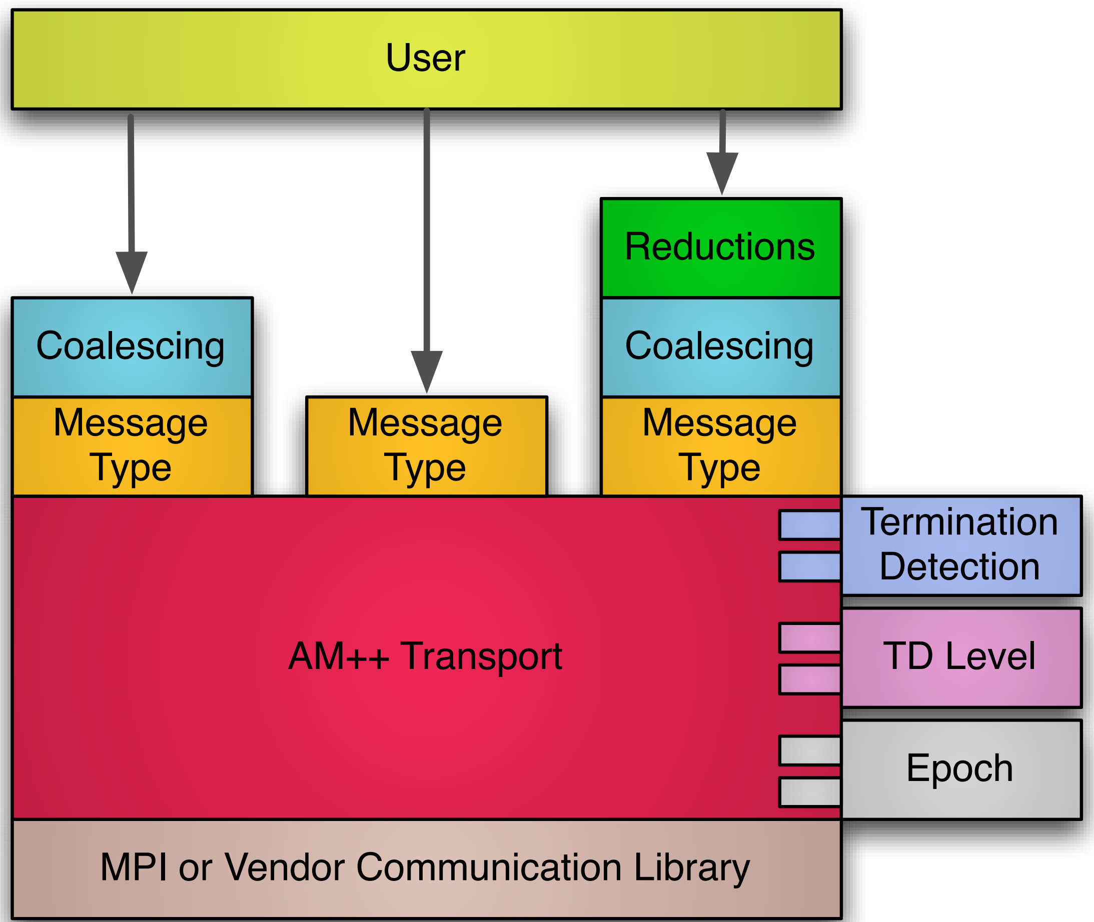
Resource Allocation Is Initialization

- 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

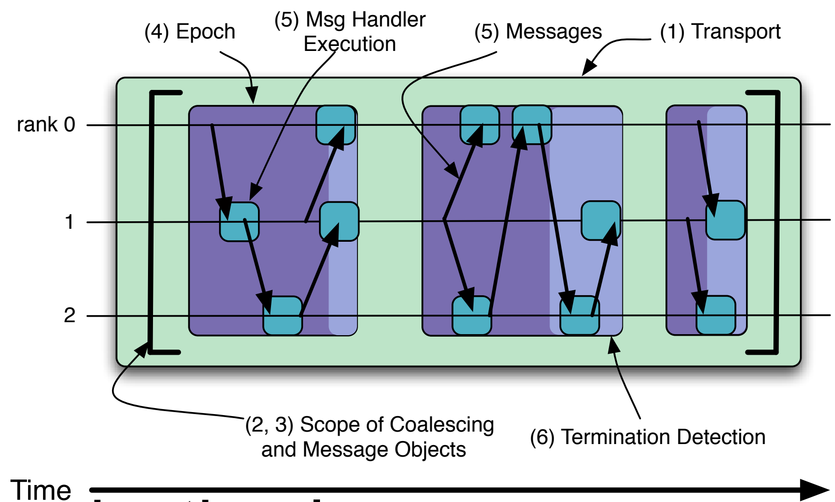


AM++ Design


User
Reductions
Coalescing
Coalescing
Message Type
Message Type
Message Type
Termination Detection
AM++ Transport
TD Level
Epoch
MPI or Vendor Communication Library

Transport

```
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);
}
```

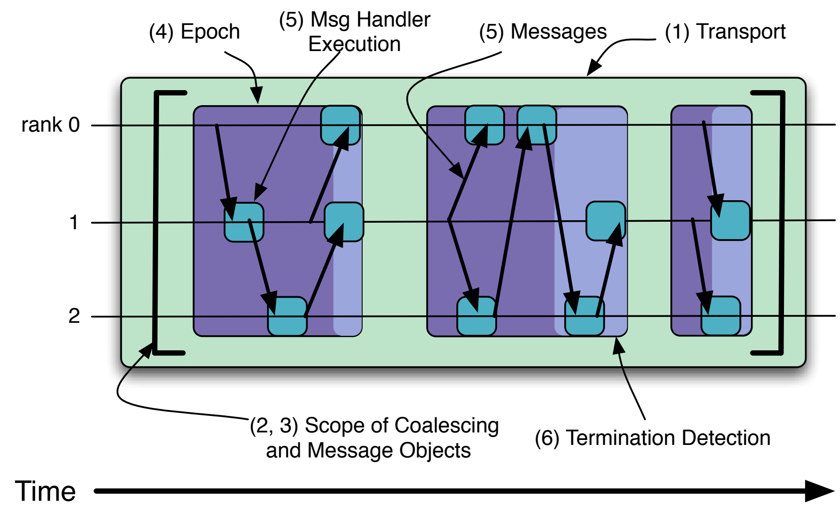


- Interface to underlying communication layer
  - MPI and GASNet currently
- Designed to send large messages produced by higher-level components
  - Object-oriented techniques allow run-time flexibility

INDIANA UNIVERSITY
PERVASIVE TECHNOLOGY INSTITUTE

Message Types

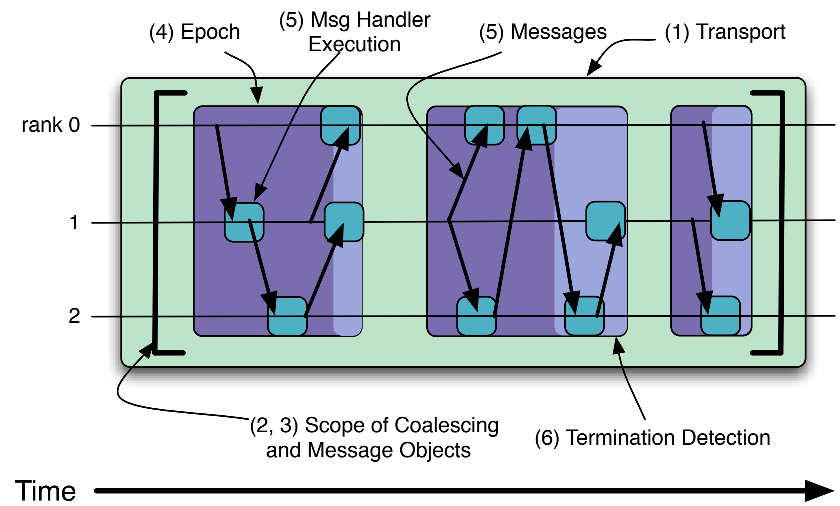
```
mpi_transport trans(MPI_COMM_WORLD);

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);
}
```



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

Termination Detection/Epochs

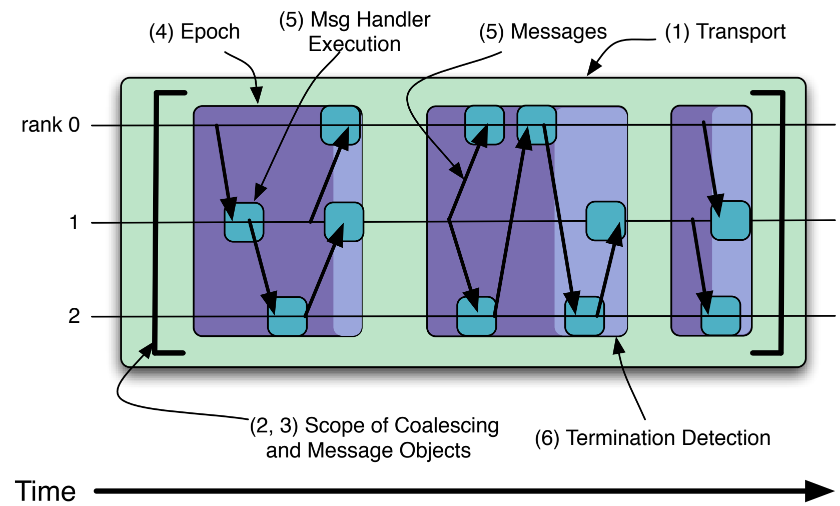
```
mpi_transport trans(MPI_COMM_WORLD);

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++ 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

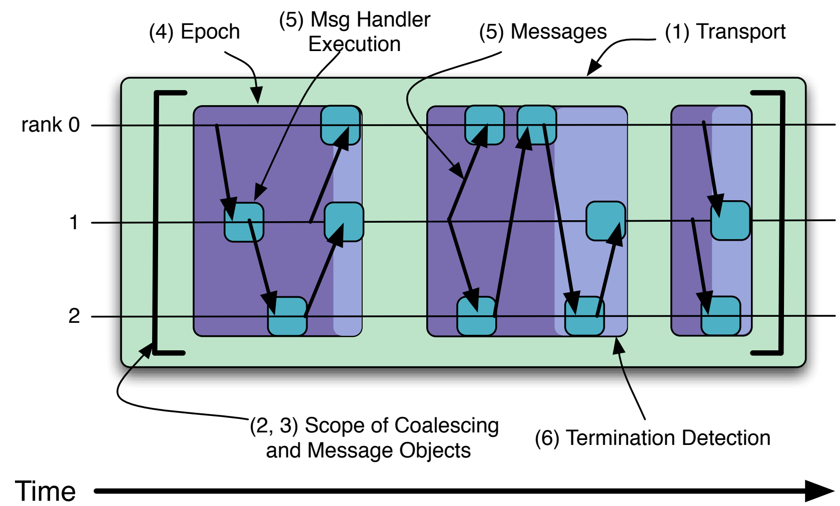
```
mpi_transport trans(MPI_COMM_WORLD);

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);
}
```



- 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

Message Handler Optimizations

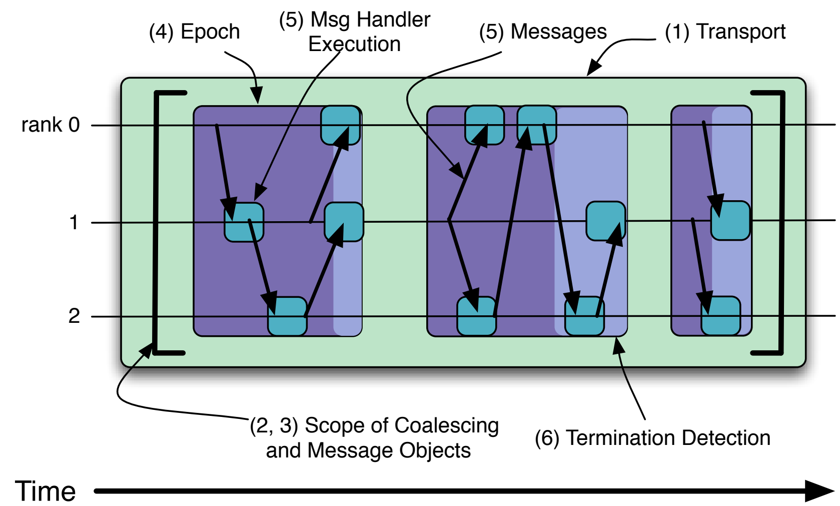
```
mpi_transport trans(MPI_COMM_WORLD);

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

Message Reductions

```
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);
}
```

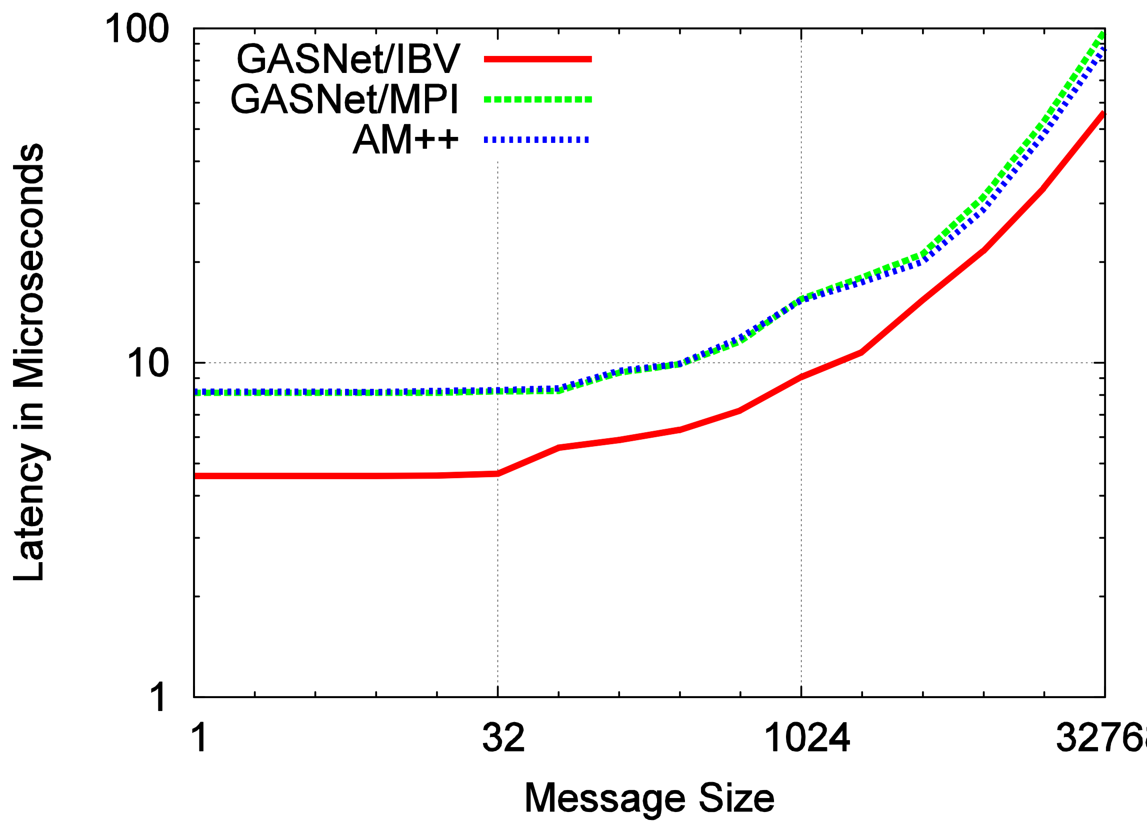


- Some applications have messages that are
  - Idempotent: duplicate messages can be ignored
  - Reducible: some messages can be combined
- Catch some of these sender-side

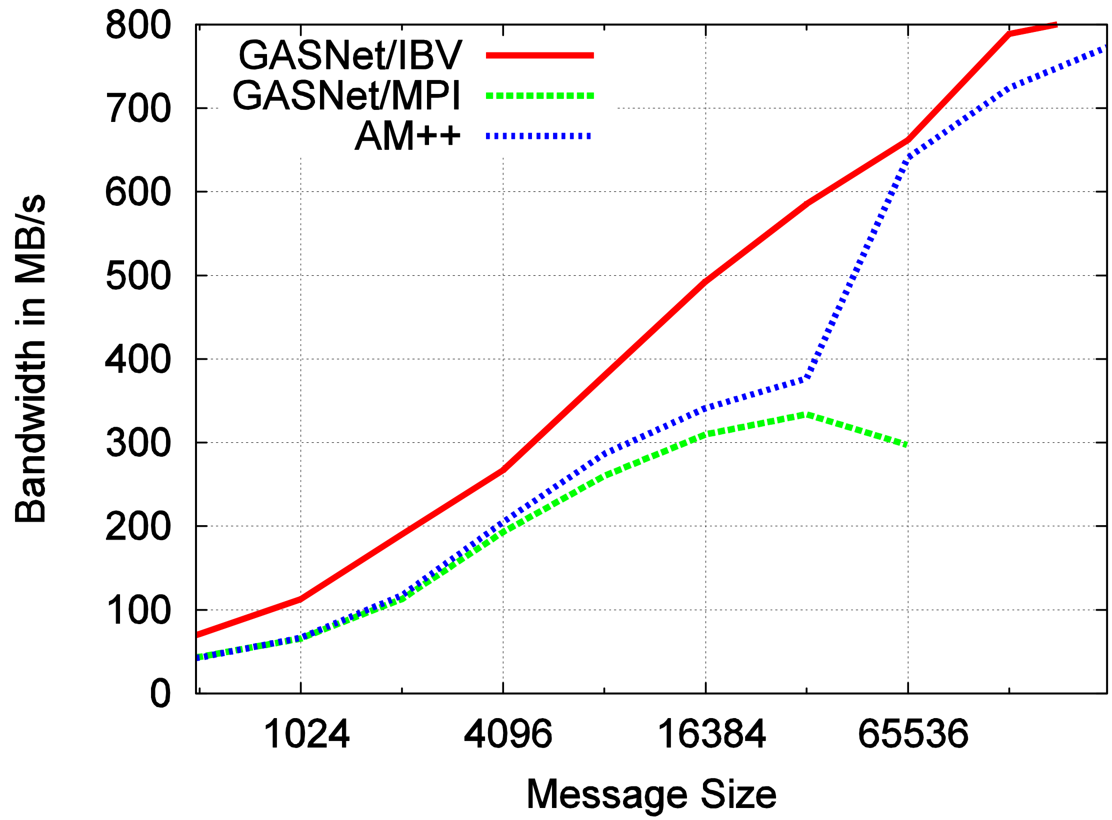
AM++ and Threads

- 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

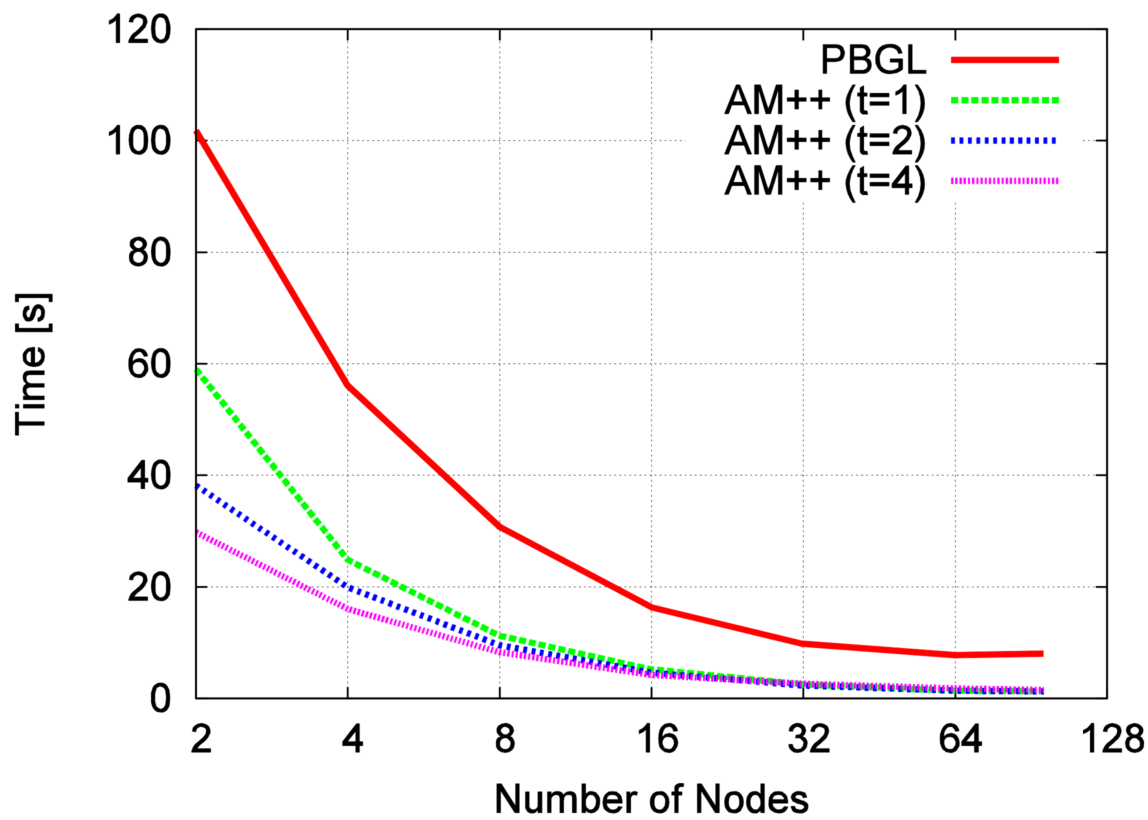
Evaluation: Message Latency



Evaluation: Message Bandwidth

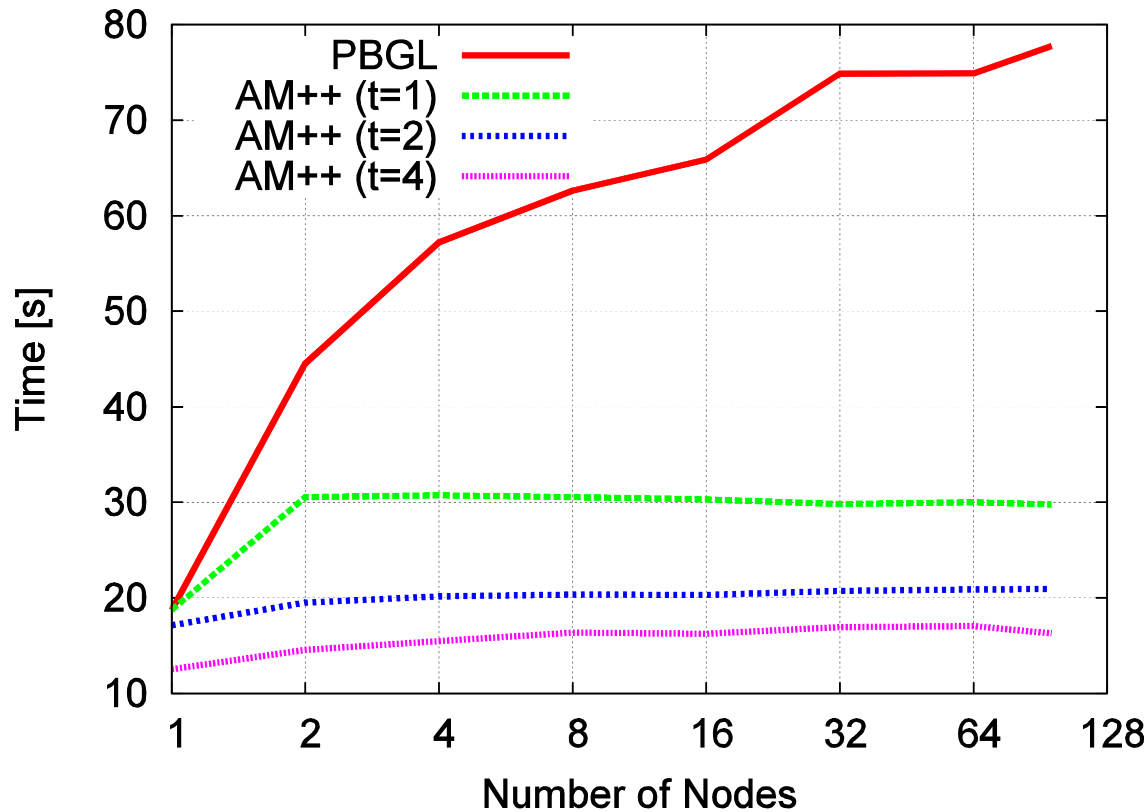


Breadth-First Search: Strong Scaling

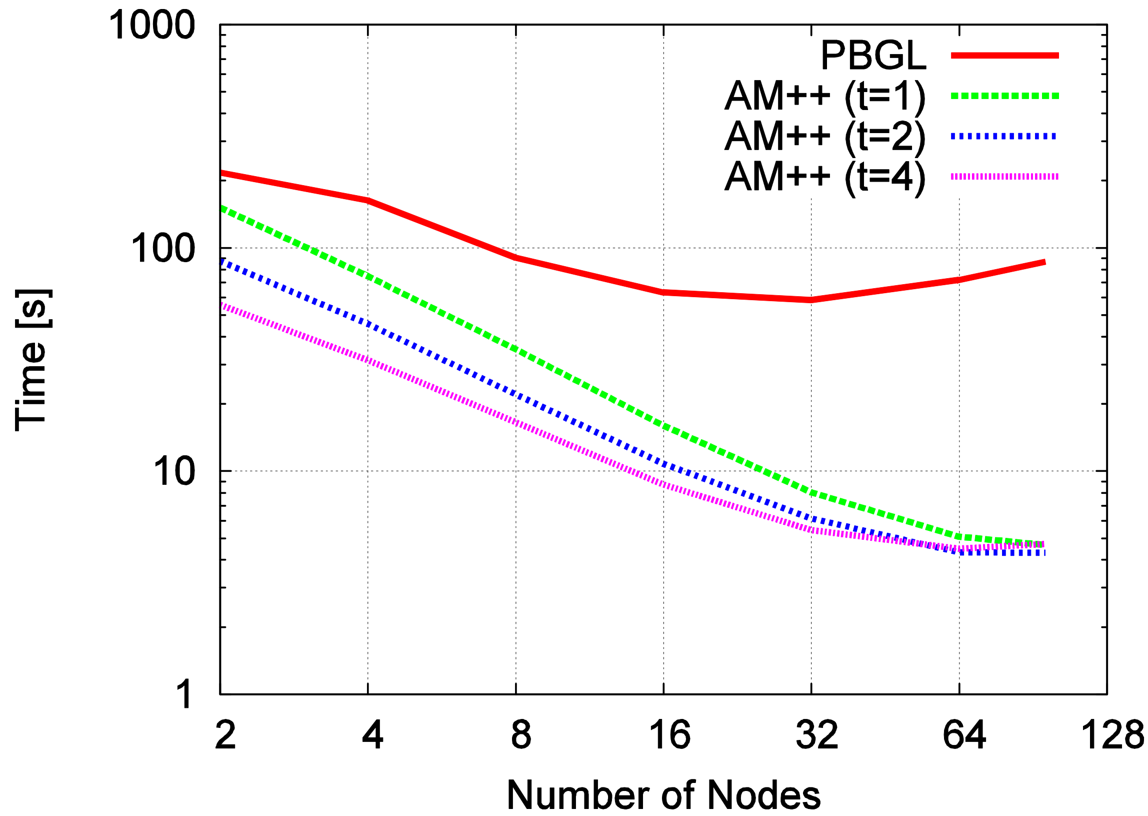


ER graph: 2^27 vertices, 2^29 edges

Breadth-First Search: Weak Scaling

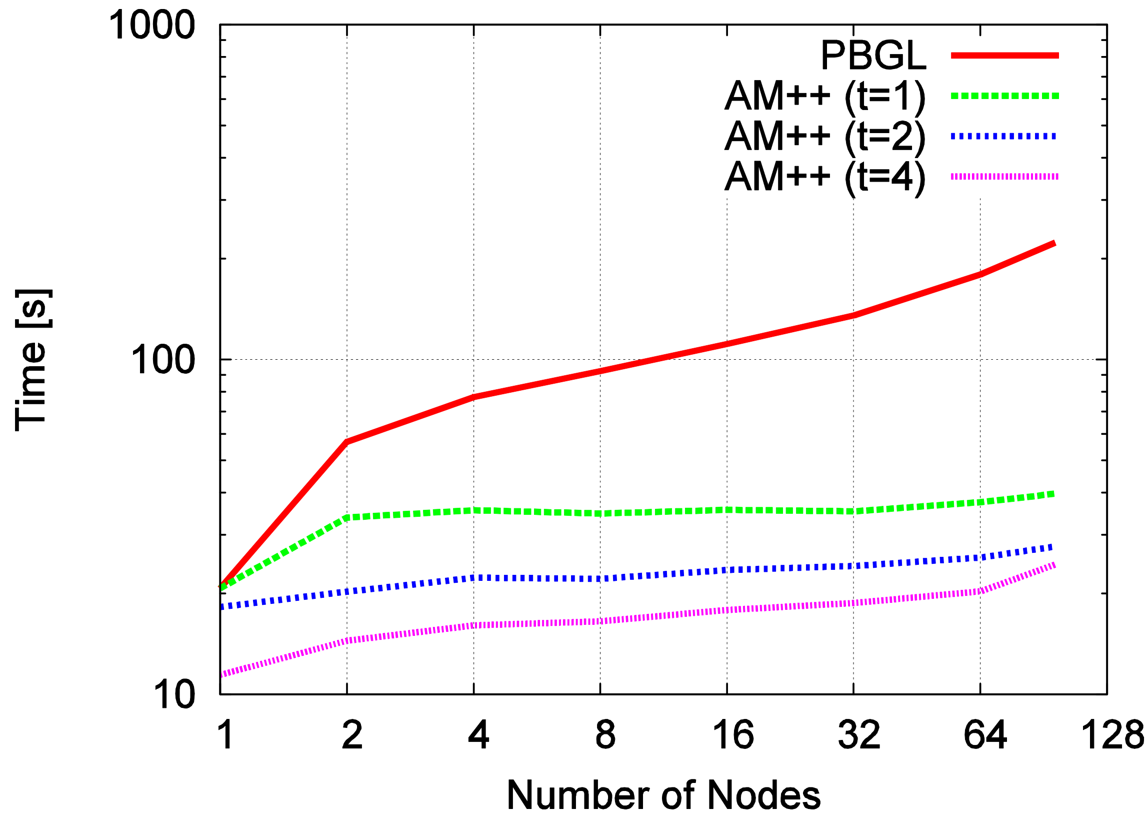


Delta-Stepping: Strong Scaling



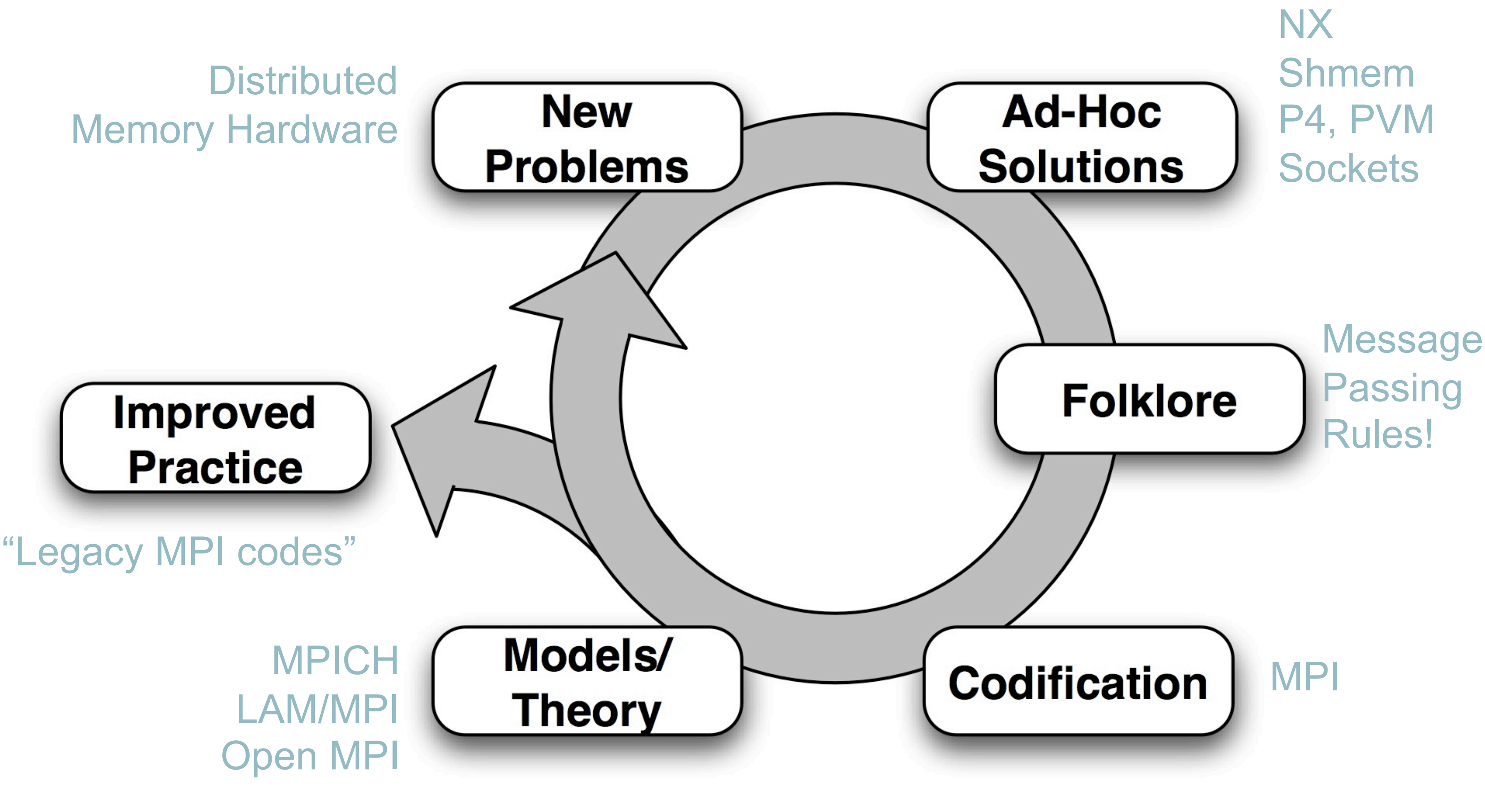
INDIANA UNIVERSITY
PERVASIVE TECHNOLOGY INSTITUTE

Delta-Stepping: Weak Scaling



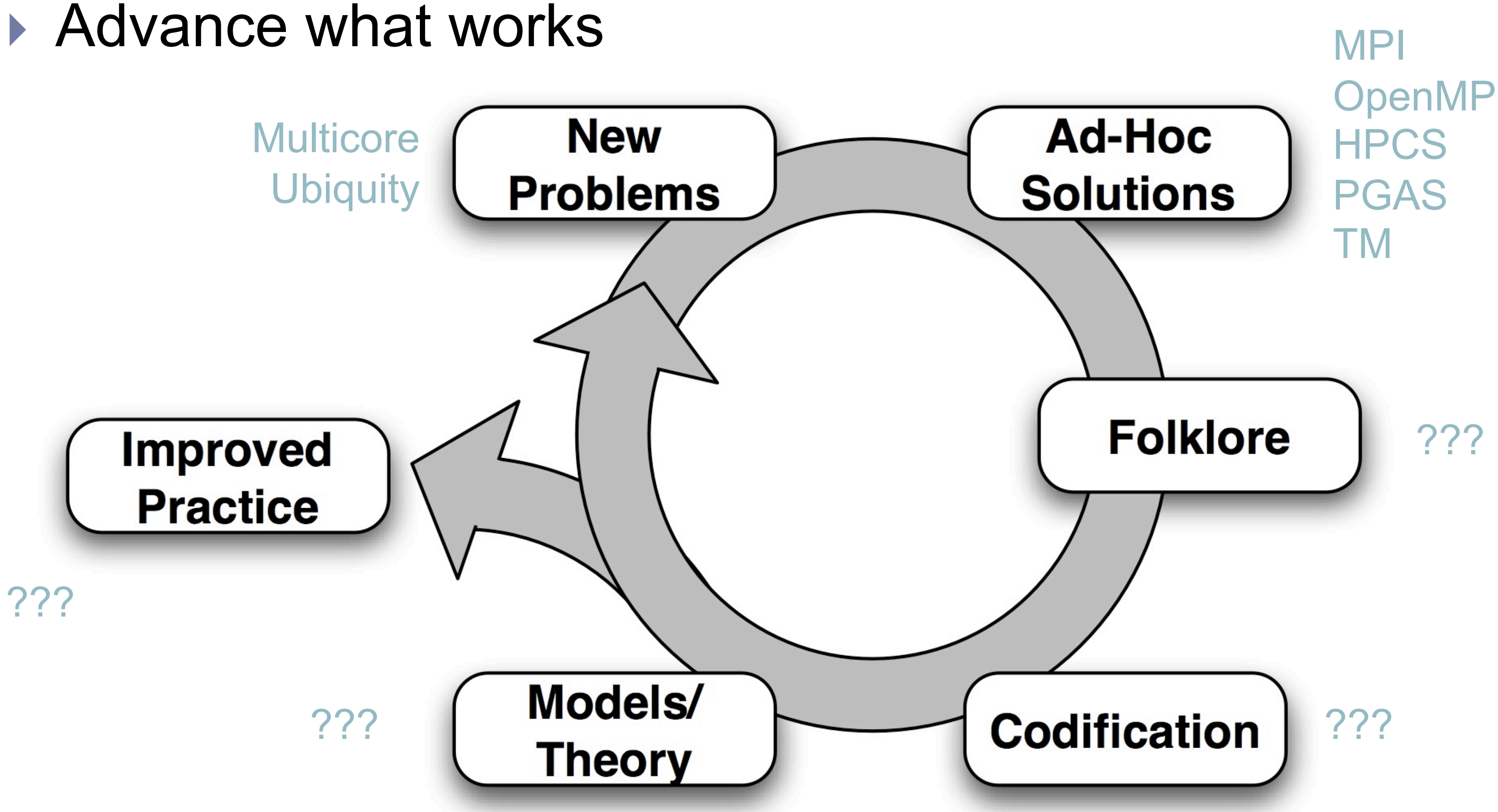
INDIANA UNIVERSITY
PERVASIVE TECHNOLOGY INSTITUTE

Why MPI Worked



Multicore Ubiquity

- Advance what works



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