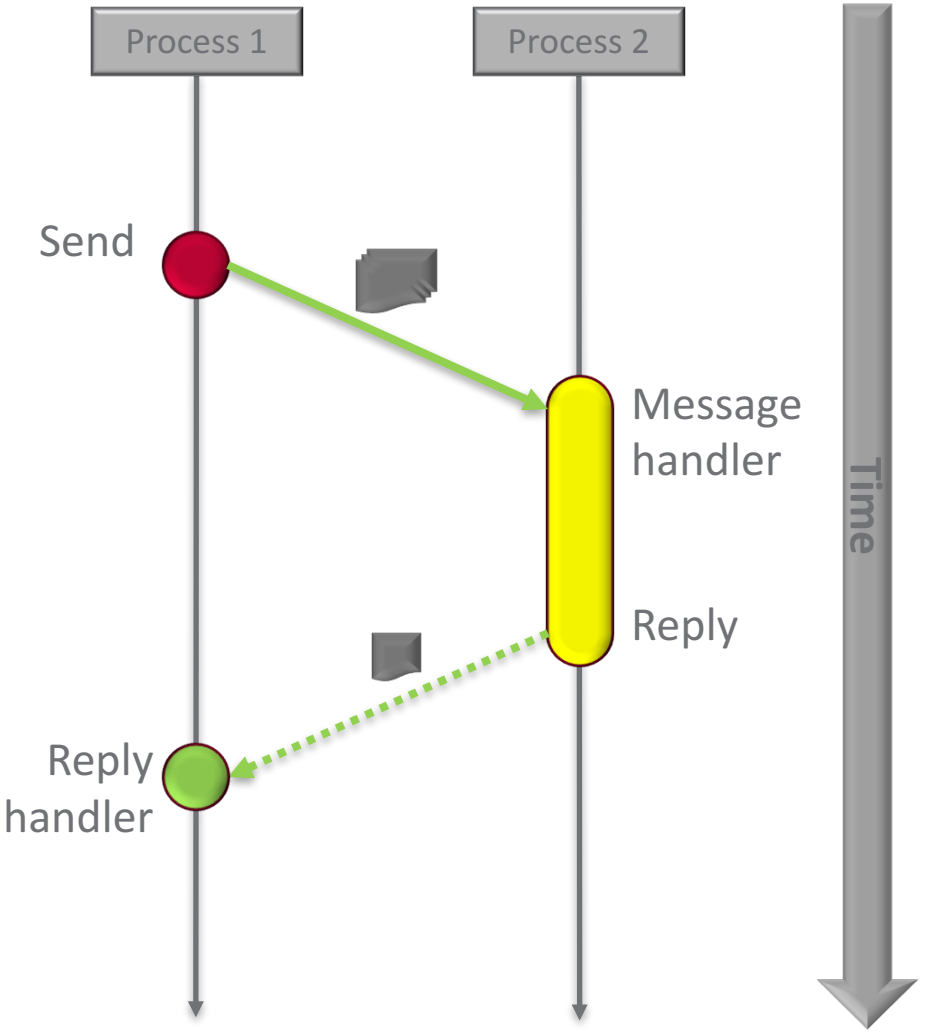
Active Messages

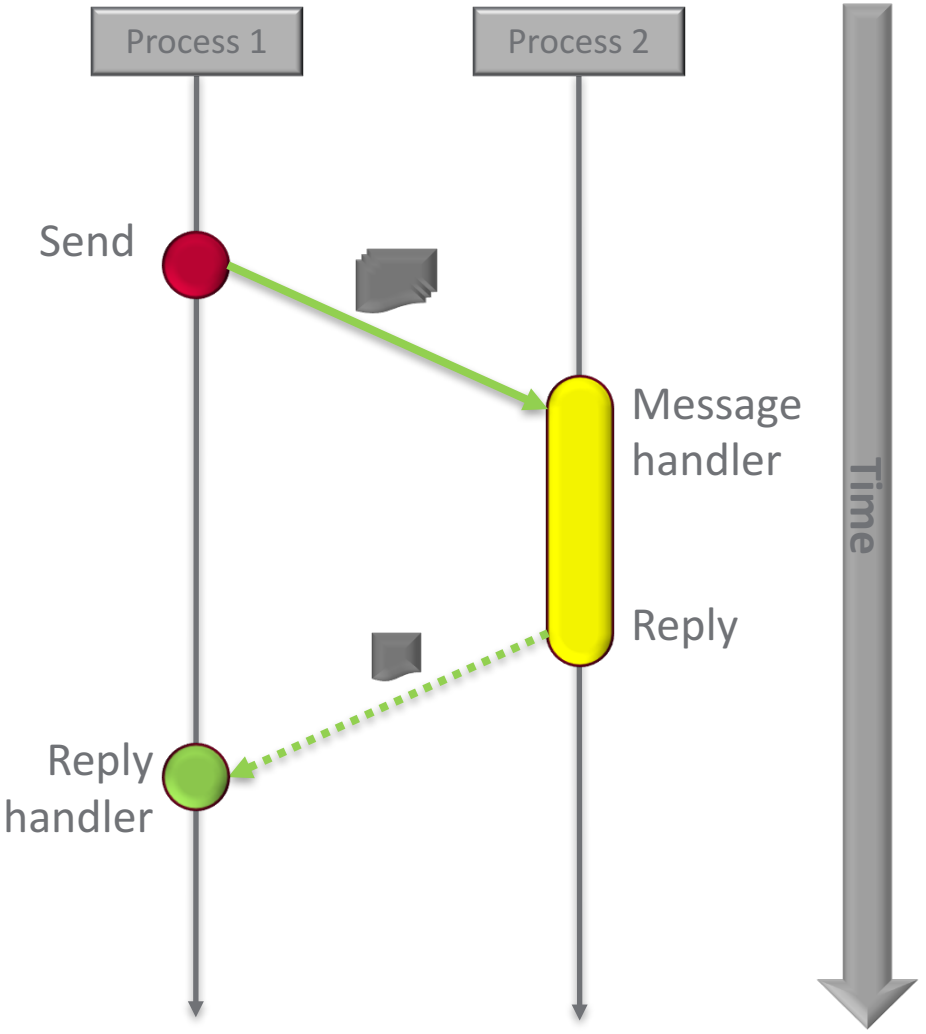
Active Messages

- 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



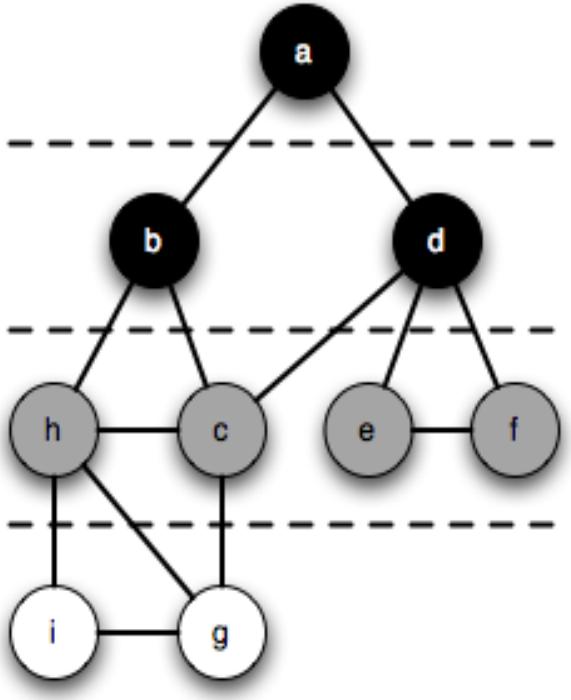
Active Messages

- Highly asynchronous
  - Communication implicit on receiver
  - Less waiting for all nodes to reach same location
  - Messages sent in background while work continues
- Allows compound actions with one message
  - Generalization of PGAS remote atomic operations
  - Fixed set of remote operations is not efficient
    - Complicated updates cannot be done with fetch-and-X or compare-and-swap



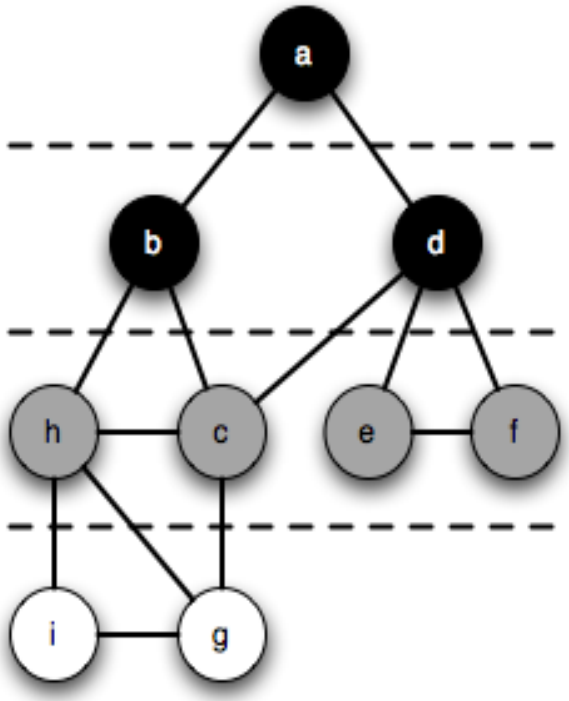
Find the Sequential Trap

ENQUEUE(Q, s)
while ($Q \neq \emptyset$)
  $u \leftarrow$ DEQUEUE(Q)
  for (each $v \in Adj[u]$)
    if ($color[v] =$ WHITE)
      $color[v] \leftarrow$ GRAY
      ENQUEUE(Q, v)
    else $color[u] \leftarrow$ BLACK



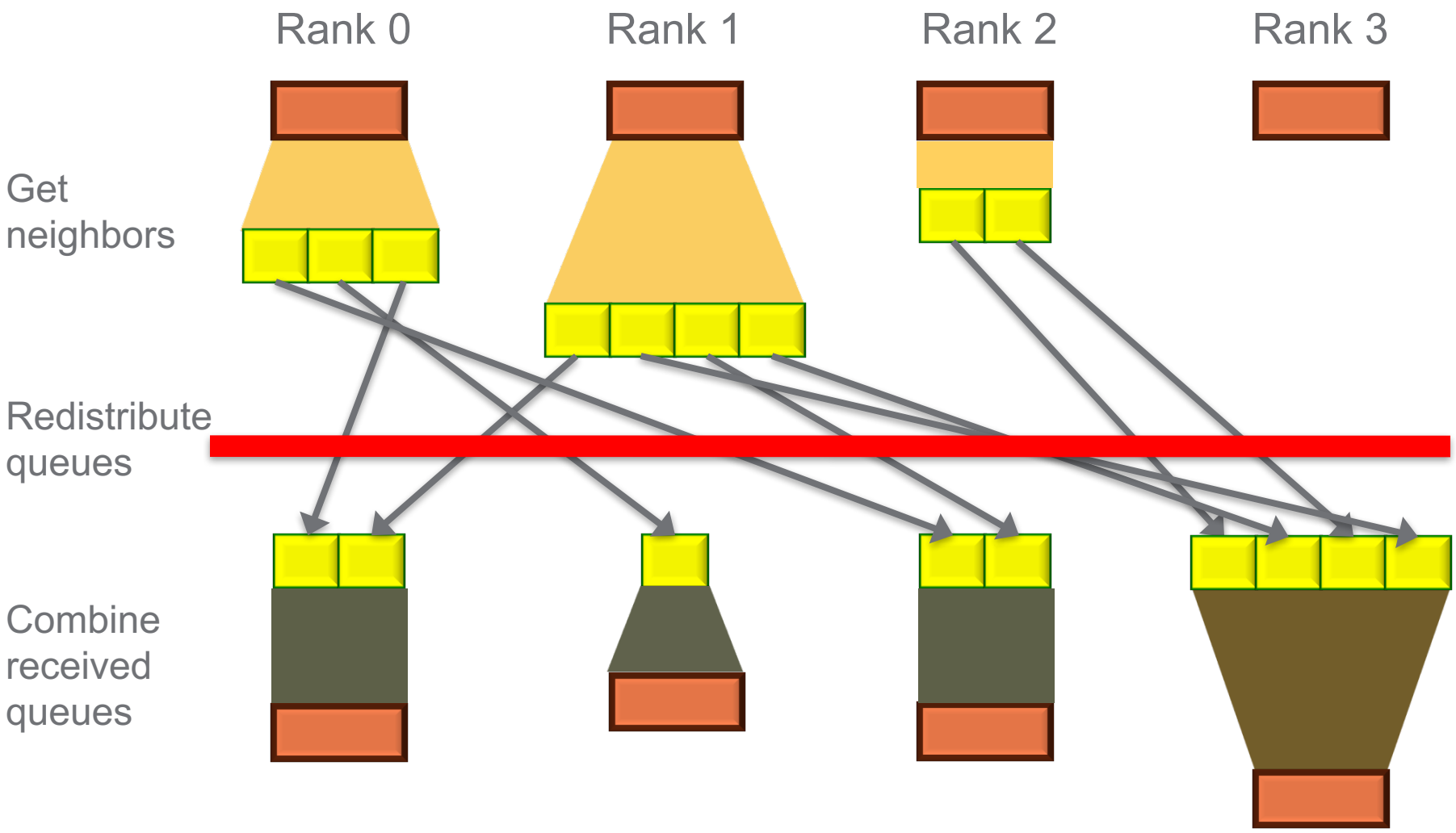
Find the Synchronization Trap

ENQUEUE(Q, s)
while ($Q \neq \emptyset$)
  $u \leftarrow$ DEQUEUE(Q)
  for (each $v \in Adj[u]$)
    if ($color[v] =$ WHITE)
      $color[v] \leftarrow$ GRAY
      ENQUEUE(Q, v)
    else $color[u] \leftarrow$ BLACK



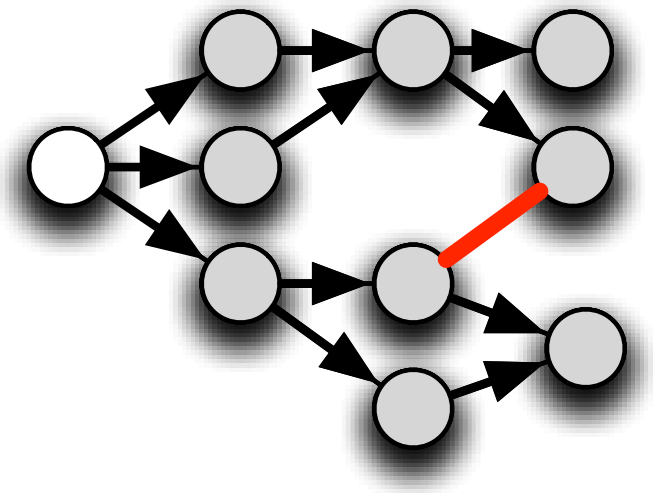
for *i* **in** *ranks*: **start receiving** *in_queue*[*i*] **from rank** *i*
for *j* **in** *ranks*: **start sending** *out_queue*[*j*] **to rank** *j*
synchronize and finish communications

BSP Breadth-First Search



Graph algorithms: Commonalities

- Very fine-grained task graph
- Some “inner loop” data parallelism
- Lots of small memory accesses and messages → latency bound
  - How do we deal with latency? → Asynchrony



discovered at runtime

Graph algorithms: A range of execution patterns

Pacific Northwest NATIONAL LABORATORY
Proudly Operated by Battelle Since 1965

Task parallel
Fine grained
Irregular / non-local
Asynchronous
Latency bound
Dynamic

Active Messages

Data parallel
Coarse grained
Regular / local
Synchronous
Bandwidth bound
Static

SPMD & BSP

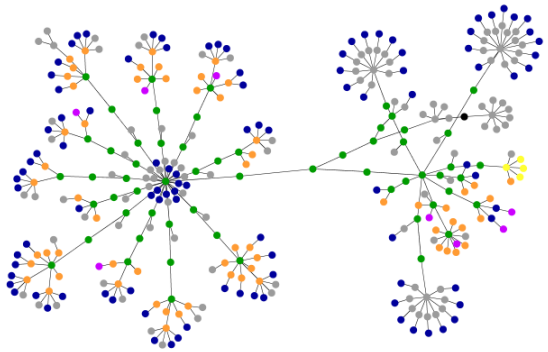
Graph Traversal

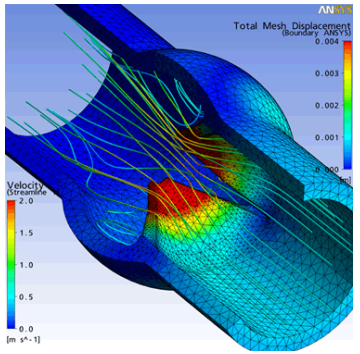
Sparse Linear Algebra

Adaptive Mesh Refinement

Subgraph Isomorphism

HPL PDE Solvers

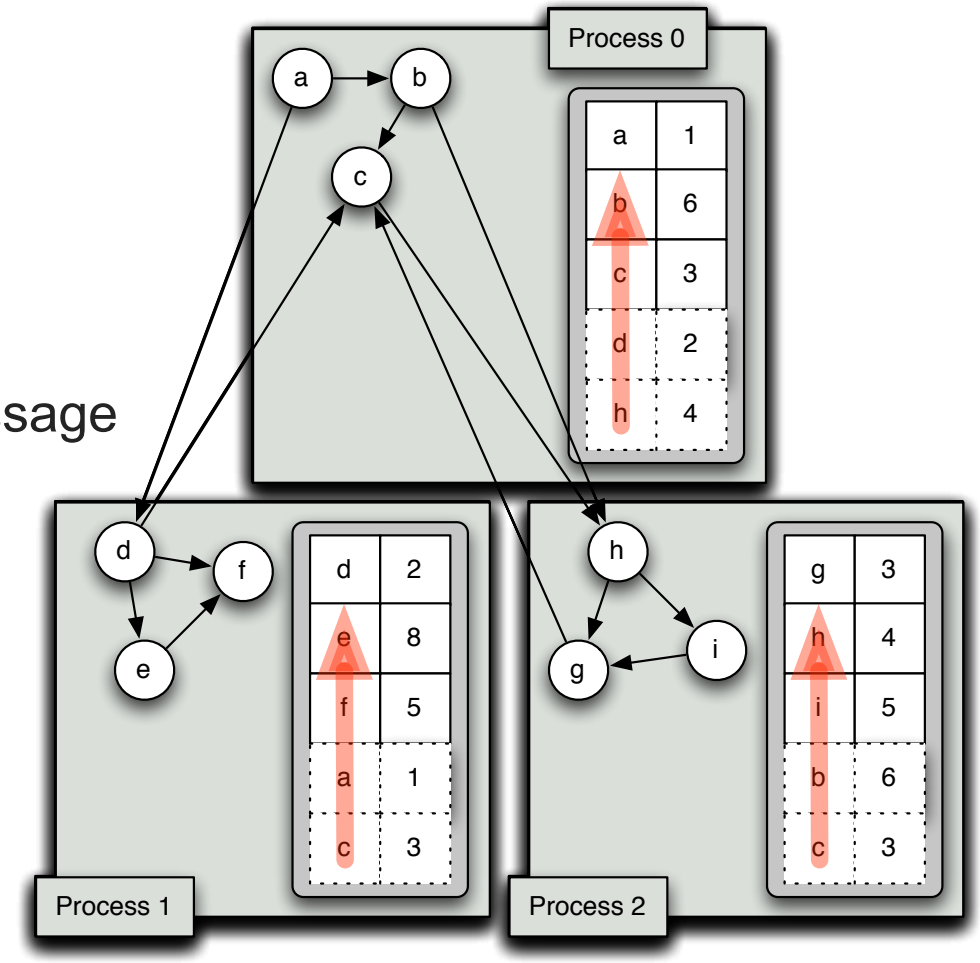




BSP Algorithms: Moving Data to Control Flow

- Perform remote data access
- Barrier
- Use received data
- Barrier
- Full network RTT on every message
- Data reuse unlikely

- Memory Utilization
- Synchronization
- Latency
- Split data and control flow



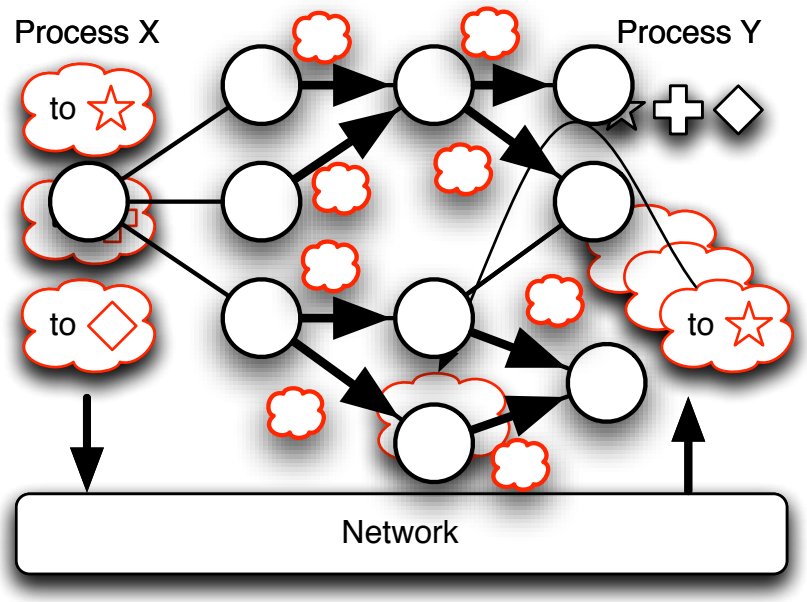
Active Messages: Controlling Resource Utilization

- Traditional scientific computing applications
  - Neighborhood communication
  - Large ratio of local to remote data
  - Spatial and temporal locality → data reuse
  - Satisfy computational dependencies by moving data to control flow (e.g., ghost cells)
- Graph applications
  - Poor partitions → everything is a boundary value
  - Minimal data reuse
  - Memory requirements
    - Difficult to compute a priori
    - Asymptotically large
    - Average case is often much more reasonable

Why Active Messages: Expressiveness

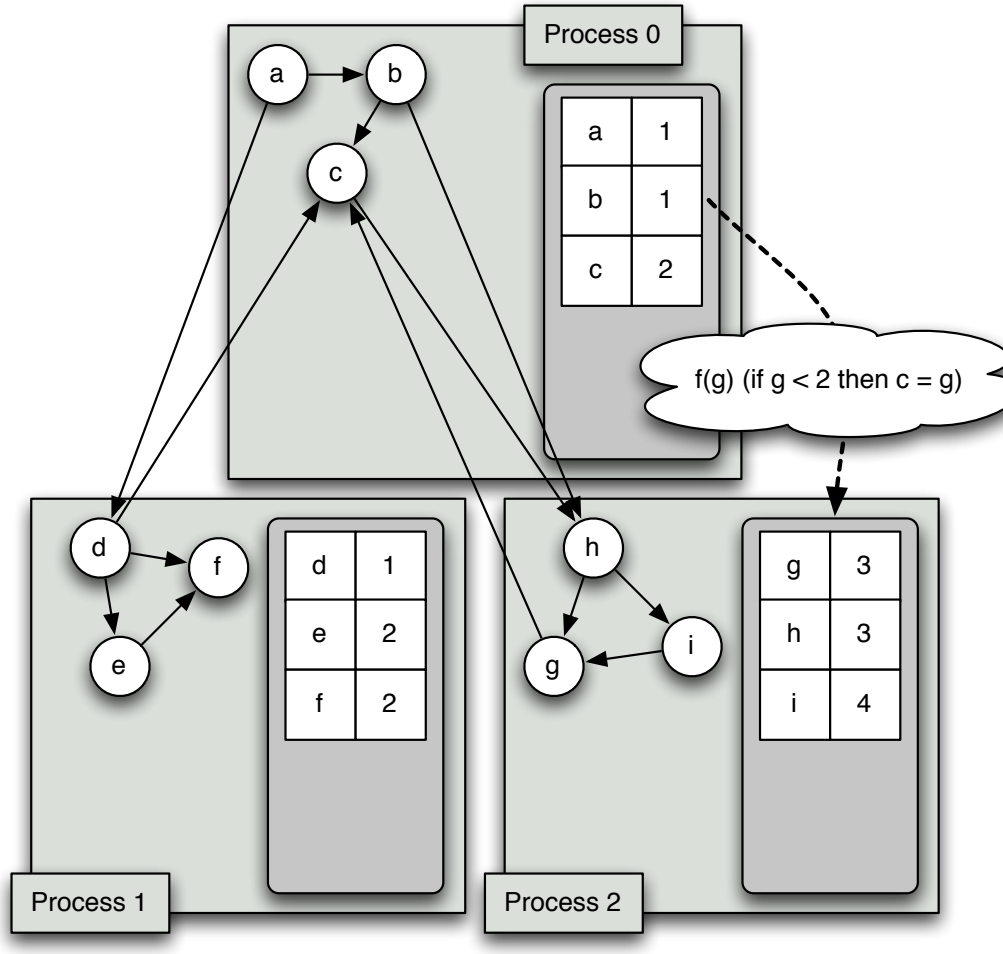
- Capable of capturing both data parallelism and task parallelism
- Preserve full computational dependency information until runtime
  - No artificial dependencies
  - No static coarsening
- Unified data and control flow
- Richer semantics than RDMA
- Uniformity of access

- Some loss of S/W modularity and conceptual integrity



Graph Algorithms: Moving Control Glow to Data

- Asynchronous, no sender-side state (fire and forget)
- ½ RTT in some cases
- Asynchrony hides latency
- No additional (static) storage for non-local data
  - Messages need to be buffered

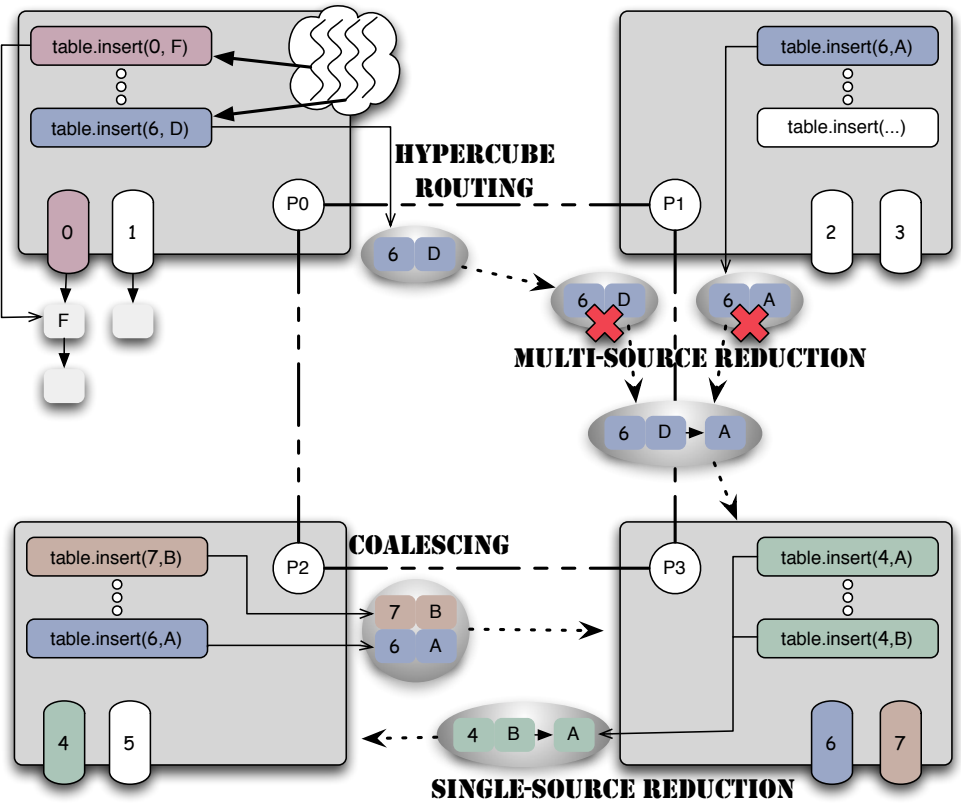


Why active messages: Efficiency

- Fine-grained messages expose maximal parallelism and asynchrony
  - Likely excessive overhead in naïve implementation
- Separates optimization from algorithm expression
  - Leverage knowledge of input graph at runtime
- Control resource utilization at runtime
- Reduce global synchronization vs. two-sided messaging

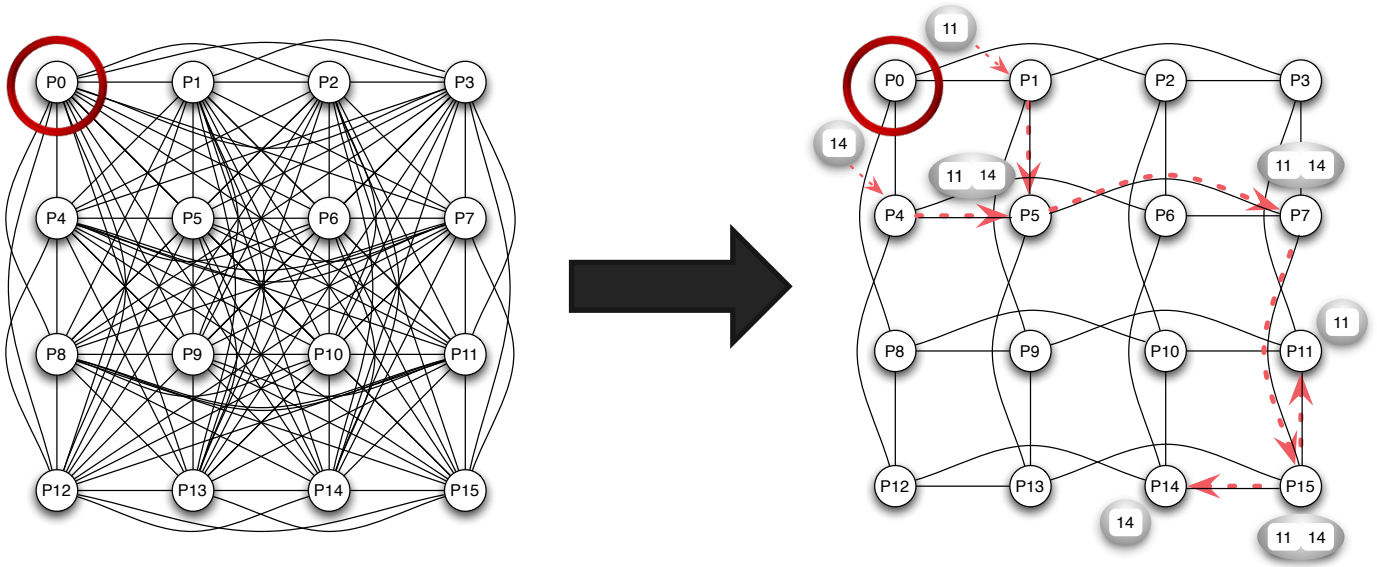
Runtime Optimization

- Message coalescing
  - Amortize overhead
  - Avoid network injection rate limits
- Message reductions
  - Cache properties of non-local objects
  - Eliminate redundant computation
  - Distributed computation into network
- Active routing
  - Exploit physical topology
  - Reduce memory consumed by communication buffers



Results: Software Routing

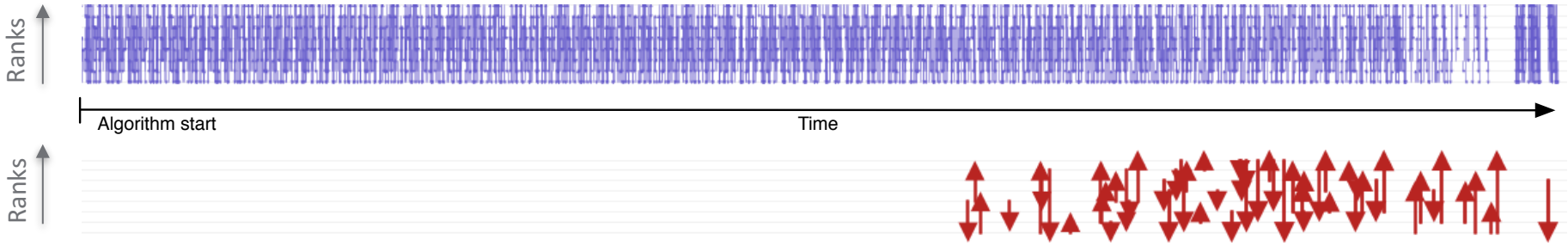
- Coalescing buffers limit scalability
  - Communications typically all-to-all
- Impose a limited topology with fewer neighbors
- Better scalability, higher latency, higher bandwidth



Multi-source coalescing

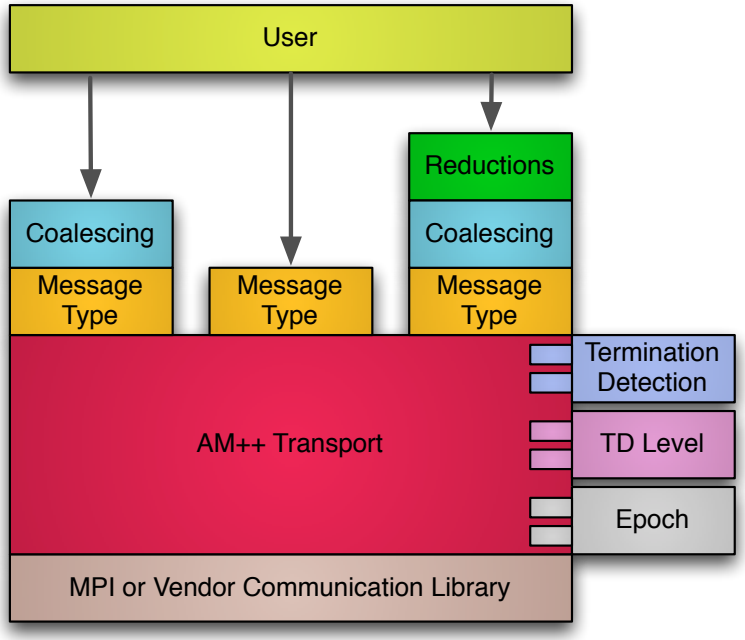
Results: BSP vs. Active Messages

- How often do we: communicate? synchronize?
- Can we act on the communicated data as soon as it is received?
- False choice → AM specification can become BSP at runtime
  - The AM and BSP results presented here were generated using the same algorithm specification and runtime
  - BSP basically amounts to turning off asynchronous message execution and (largely) the overlap of communication and computation



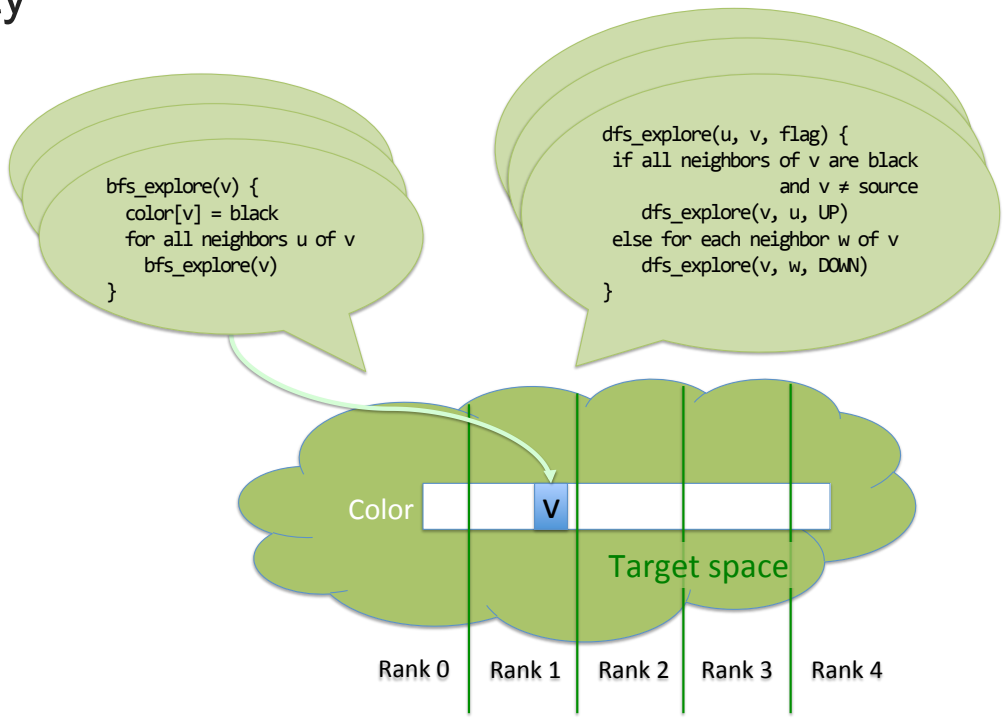
Active Pebbles / AM++

- Programming and execution model
  - Programming model allows natural expression of algorithms using active messages
  - Execution model makes it run efficiently
- For fine-grained algorithms with many tiny messages
  - No artificial coarsening needed
  - Adds latency to obtain higher throughput
- Realized in AM++ library



Programming Model

- Program with natural granularity
- Transparent addressing
  - Targets can be arbitrary
  - User-defined data distributions
    - Static and dynamic
  - Self-sends do not need to be detected manually
- Epoch model
  - Enforces message delivery
  - Handlers can be recursive

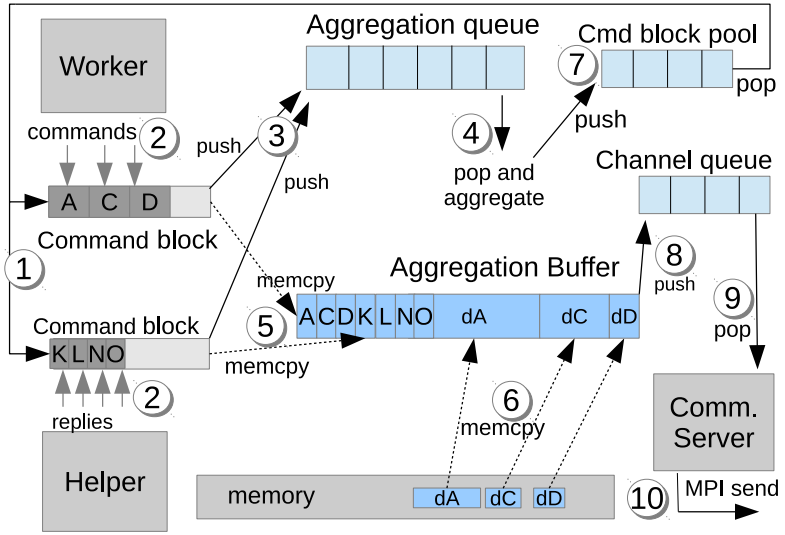


GMT – Global Memory and Threading

- Partitioned Global Address Space (PGAS) data model
- Lightweight software multithreading to hide latency of remote operations
- Asynchronous user level task parallelism
  - Loop level parallelism (parFor)
  - Support for active messages
    - Execute “on data” (using the PGAS)
    - Execute “on node” (currently defined through MPI ranks)
- Two-level message aggregation

Message aggregation

- Two-level aggregation
  - Queues are per destination node
  - Command blocks
    - "local" to a core
  - Aggregation queue
    - Common to a node
  - Aggregation buffers
    - Buffers where data are effectively copied before the MPI send operation



Q&A

- What standard interfaces can you layer on top of? MPI, PGAS, UCX, …
  - Both GMT and AM++ are layered on MPI
  - In theory, active messages could be implemented with PGAS, put with completion, …
- Granularity requirements
  - "Natural" level at application level, coalescing etc. in the runtime
- Need for helper threads at source and target, and whether those can be explicitly managed
  - GMT has helper threads, AM++ does not
- How should work related to active messages be layered, i.e., what should be done about the HiHAT User Layer, what done by implementation below User Layer (e.g. code that needs to do setup, make trade-offs) but above Common Layer, and what part should be below Common Layer (e.g. code to map to specific transports)?

Conclusions

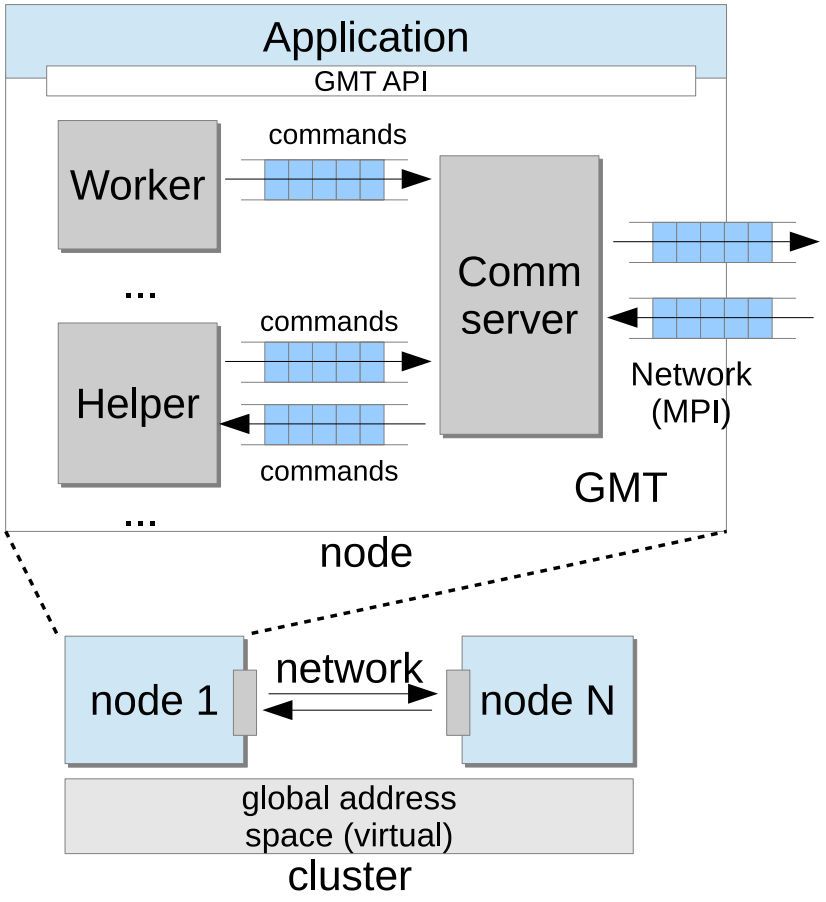
- Active Messages
  - are lightweight
  - move control flow to data
  - are asynchronous
  - allow fine granularity
  - allow a wide range of optimizations at runtime

Discussion

- User interface layered above HiHAT
- App can use MPI or SHMEM to set up its own receive code in some other rank/PE
- MPI or SHMEM could be used directly, outside of HiHAT, for inter-rank/PE comms, and we can consider switching to using HiHAT for comms if/when it adds cross-rank support. We're trying to avoid "competition" over resources between HiHAT and other comms mechanisms, so are starting with a HiHAT within rank and MPI/whatever between ranks approach.
- MPI/SHMEM/whatever comms do want to be integrated into HiHAT's dependence system, so that actions that get enabled upon the arrival of data can be async'ly enqueued and can execute when ready, vs. having to block/poll until data arrives before enqueuing.
- Any special requirements
  - Receiver thread wants async user-level wakeup signaling vs. block/sleep/OS thread wakeup or poll
  - Ordering requirements, like fence or quiet
  - Working data arrival notification into HiHAT's dependence management system

GMT architecture

- Three classes of pthreads (pinned to cores)
  - Worker
    - Executes application code through lightweight tasks
  - Helper
    - PGAS and communication management
  - Communication Server
    - MPI communication



Put/Get With Completion: Even Lighter Active Messages

Results: Runtime message reductions