Scalable Algorithms for Constructing Balanced Spanning Trees on System-ranked Process Groups

Akhil Langer, Ramprasad Venkataraman, **Laxmikant V. Kale**

Parallel Programming Laboratory
University of Illinois

September 25 2012

Pitch

- Not all messaging needs fully-capable communicators
- Its worthwhile to consider cheaper constructs

- We propose:

Unranked or System-ranked Process Groups

User cannot choose member ranks

Cheap and Scalable Creation Mechanisms

- Shrink-and-Balance
- Rank-and-Hash

- ~ 100X faster than MPI_Comm_split on 32K cores of IBM BG/P

Is process group creation/management scalable?

- Memory capacity is growing slower than available concurrency

Runtime systems have to adopt resource-conserving mechanisms

- Typical Process Group Implementations
  - Each member can id everyone else
  - Storage: $O(n)$ (on each member process)
  - Time for creation: $O(n \log n)$
- Applications can create many such groups simultaneously

Is process group creation/management scalable?

- Memory capacity is growing slower than available concurrency

Runtime systems have to adopt resource-conserving mechanisms

- Typical Process Group Implementations
  - Each member can id everyone else
  - Storage: $O(n)$ (on each member process)
  - Time for creation: $O(n \log n)$
- Applications can create many such groups simultaneously

How can we use less than $O(n)$ memory?

Distributed enrollment
Distributed storage

Achieving distributed enrollment

Central specification of membership

MPI_Group_incl(
MPI_Group group,
int n,
int ***ranks**, ← **not scalable**
MPI_Group *newgroup)

Distributed enrollment

MPI_Comm_split(
MPI_Comm comm,
int color,
int key,
MPI_Comm *newcomm)

Achieving distributed storage

- Distributed Tables

EuroMPI 2010

A Scalable MPI Comm split Algorithm for Exascale Computing.
Sack, P., Gropp, W.
In: Recent Advances in the Message Passing Interface. pp. 110. EuroMPI10 (2010)

Achieving distributed storage

- Distributed Tables

EuroMPI 2010

A Scalable MPI Comm split Algorithm for Exascale Computing.
Sack, P., Gropp, W.
In: Recent Advances in the Message Passing Interface. pp. 110. EuroMPI10 (2010)

- Process Chains

EuroMPI 2011

Exascale algorithms for generalized MPI comm split.
Moody, A., Ahn, D., Supinski, B.
In: Recent Advances in the Message Passing Interface. pp. 9-18. EuroMPI11 (2011)

Achieving distributed storage

- Distributed Tables

EuroMPI 2010

A Scalable MPI Comm split Algorithm for Exascale Computing.
Sack, P., Gropp, W.
In: Recent Advances in the Message Passing Interface. pp. 110. EuroMPI10 (2010)

- Process Chains

EuroMPI 2011

Exascale algorithms for generalized MPI comm split.
Moody, A., Ahn, D., Supinski, B.
In: Recent Advances in the Message Passing Interface. pp. 9-18. EuroMPI11 (2011)

- Unranked / System-Ranked Process Groups

EuroMPI 2012

Scalable Algorithms for Constructing Balanced Spanning Trees on System-Ranked Process Groups Langer, A., Venkataraman, R., Kale L.
In: Recent Advances in the Message Passing Interface. pp. 9-18. EuroMPI12 (2012)

System-Ranked Process Group

- User cannot specify or influence ranks of members

MPI_Comm_split(
MPI_Comm comm,
int color,
~~**int key,**~~
MPI_Comm *newcomm)

- Ranks are assigned by runtime system
- Hence, any mapping of application logic / data to ranks has to be handled manually after creation

Are user-supplied ranks needed all the time?

barrier, broadcast, reduce, allreduce

- Input / output not dependent on ranks
- Assume commutative operators

- Sizeable fraction of collective communication in applications involve these operations [1], [2], [3]
- Several algorithms can be expressed with just these collectives

[1] NERSC6 Workload Analysis and Benchmark Selection Process.
Antypas, K., Shalf, J., Wasserman, H.
Tech. Rep. LBNL-1014E, Lawrence Berkeley National Lab(2008)

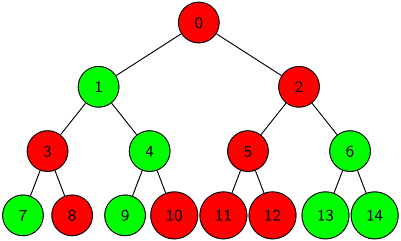
[2] Automatic MPI Counter Profiling.
Rabenseifner, R.
In: 42nd CUG Conference(2000)

[3] Parallel Scaling Characteristics of Selected NERSC User Project Codes
Skinner, D., Verdier, F., Anand, H., Carter, J., Durst, M., Gerber, R.
Tech. Rep. LBNL/PUB-904, Lawrence Berkeley National Lab (2005)

Problem Statement

Represent process groups using spanning trees

- Low memory footprint (distributed storage)
- Recursive, splitting of original tree (distributed enrollment)
- Immediate availability of efficient synchronization / housekeeping
- Can use spanning tree for the target collectives too



To support system-ranked groups

Starting from a parent tree, construct balanced spanning tree over enrolled members only

The Reference Centralized Algorithm

The Reference Centralized Algorithm

- *Upward pass*: gatherv
  Members of new group contribute their process ids
- *Downward pass*
  pick immediate children and split the remaining list
- $O(m + \log n)$ time and $O(m)$ memory [4]

[4] m is size of the new subtree

The Shrink-and-Balance Algorithm

Algorithm: Shrink-and-Balance

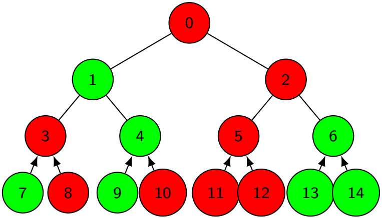
Upward Pass

- Use enrollment data to shrink original spanning tree by excluding non-participating processes
- *"fill" holes* with member processes
  - leaf process
  - immediate child process
- leaf process - send $min(d_{i,k}, subtree\_size(v)) = O(\log n)$ candidate fillers to the parent
- $O(\log n)$ space and $O(\log^2 n)$ time

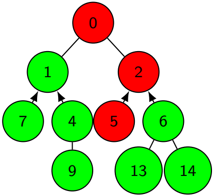
[4] $d_{i,k}$ is depth of a rank i process in a balanced spanning tree of branching factor k
$d_{i,k} = \lfloor \log_k(i(k-1)+1) \rfloor$

Algorithm: Shrink-and-Balance

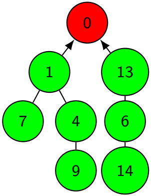
Upward pass



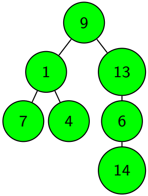
(a)



(b)



(c)



(d)

Algorithm: Shrink-and-Balance

Downward Pass

- Upward pass yields participants-only spanning tree that need not be balanced
- Balance tree while minimizing vertex migrations
  - compute ideal height of a perfectly balanced spanning tree
  - target height yields max size of subtrees[5]
  - based on current size, mark subtrees as vertex suppliers and consumers, respectively
  - request supplier for vertex if child is missing (takes $O(\log n)$ time)
  - "matchmaking" step to assign suppliers to one or more consumers
  - vertex concludes its role by calling balancing step on its children
- $O(\log^2 n)$ time, as a child could be missing at each level

[5] $max\_size = \frac{k^h - 1}{k - 1}$

The Rank-and-Hash Algorithm

Algorithm: Rank-and-Hash

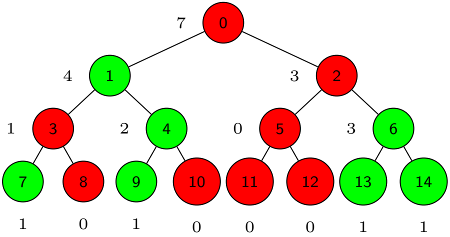
Upward Pass

- Reduction
- Store size of each subtree

Algorithm: Rank-and-Hash

Upward Pass

- Reduction
- Store size of each subtree



(a) Subtree sizes after the upward pass

Algorithm: Rank-and-Hash

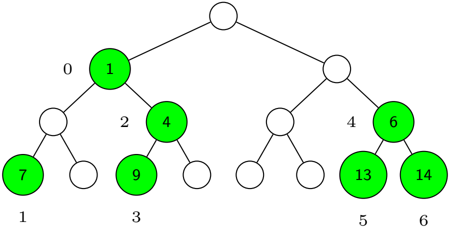
Downward Pass

- Size of tree determines available ranks $[0, m)$
- Range split amongst subtrees based on their sizes
- Splitting continues down the original spanning tree until all available ranks divided
- Non-participating processes not assigned any ranks

Algorithm: Rank-and-Hash

Downward Pass

- Size of tree determines available ranks $[0, m)$
- Range split amongst subtrees based on their sizes
- Splitting continues down the original spanning tree until all available ranks divided
- Non-participating processes not assigned any ranks



(c) Ranks after the downward pass

Algorithm: Rank-and-Hash

Identifying Tree Neighbors

- Process ids of parent and children discovered through *intermediary* processes
- *id* of intermediary process (H_i), for rank i computed via a hash function

Algorithm: Rank-and-Hash

Identifying Tree Neighbors

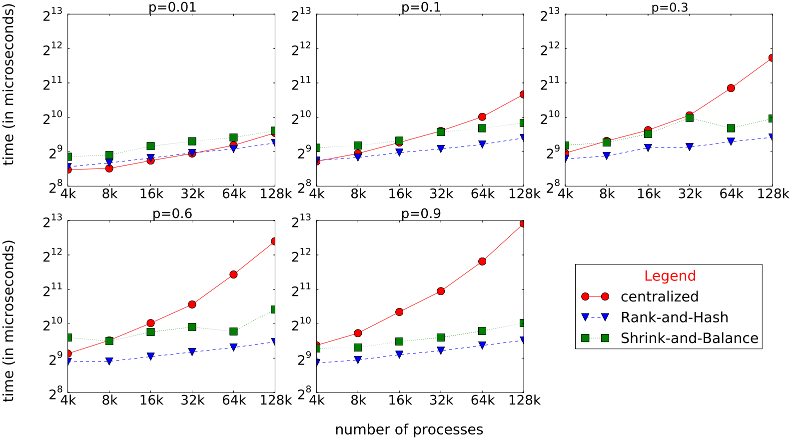
- Process ids of parent and children discovered through *intermediary* processes
- *id* of intermediary process (H_i), for rank i computed via a hash function
- Each rank i, sends its id to H_i and H_p (where, p is rank of its parent)
- Receive msgs from H_i and H_p with ids of children and parent, respectively

Experimental Setup

- Time measured between broadcast on original spanning tree and reduction on the newly constructed tree
- Sample from a uniform distribution $u(0, 1)$ and use participation probability p to determine participation of a process in the group.
- Repeatable seeds to ensure identical groups across runs
- Algorithms implemented in Charm++
- Runs on BG/P "Intrepid"

Results

Performance Comparison on up to $128k$ cores of BG/P



Results

Performance Comparison on up to $128k$ cores of BG/P

- Distributed schemes outperform the centralized scheme at modest process counts (except for very small p)
- Shrink-and-Balance slower than Rank-and-Hash
  - longer critical path

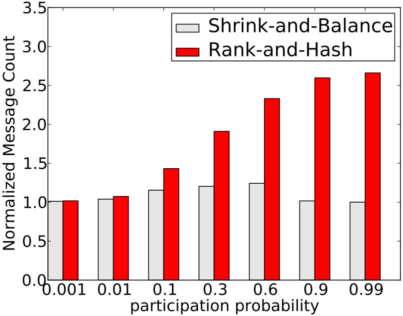
Results

Performance Comparison on up to $128k$ cores of BG/P

- Distributed schemes outperform the centralized scheme at modest process counts (except for very small p)
- Shrink-and-Balance slower than Rank-and-Hash
  - longer critical path
- Both schemes attain the goal of reduced memory footprint!

Results

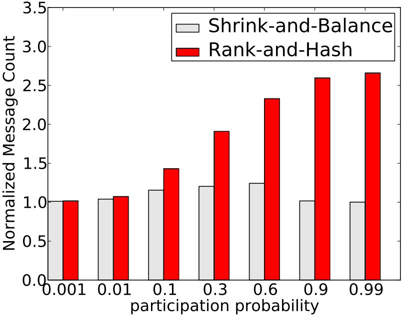
Normalized message counts w.r.t. the centralized scheme on $128k$ cores of BG/P



- Shrink-and-Balance has far fewer messages than Rank-and-Hash
- at $p = 0.6$, number of messages sent by Centralized, Shrink-and-Balance and Rank-and-Hash were 2.1, 2.6 and 4.9×10^5, respectively

Results

Normalized message counts w.r.t. the centralized scheme on $128k$ cores of BG/P



- Shrink-and-Balance has far fewer messages than Rank-and-Hash
- at $p = 0.6$, number of messages sent by Centralized, Shrink-and-Balance and Rank-and-Hash were 2.1, 2.6 and 4.9×10^5, respectively

- Shrink-and-Balance may perform better when
  - multiple groups are being formed simultaneously
  - group formation occurs simultaneous with other communication in the application

Results

Comparison with MPI_Comm_split on $32k$ cores of BG/P

- MPI_Comm_split comparison with multi-color Rank-and-Hash

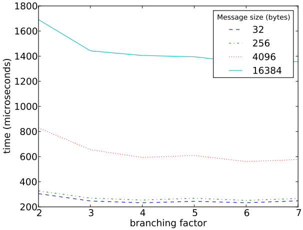
Group Creation Time (in milliseconds)

# splits	MPI-Comm-split	Rank-and-Hash
1	134.968	0.708
2	106.573	0.713
4	96.989	0.760
8	93.536	0.785

Related Work

- Moody et al[6] proposed generalized MPI_Comm_split
  - process groups as chain - $O(1)$ space and $O(\log n)$ time
  - requires $O(n)$ messaging to exchange process ids during collective call
  - does collective communication using binary spanning trees
- Several differences
  - lesser dependencies on remote information for progress of collective hence, more prominent for one-sided transfer calls supported by some network messaging APIs
  - construct spanning trees of arbitrary branching factors

Broadcast time on $32k$ core of BG/P



[6] Moody, A., Ahn, D., de Supinski, B.: Exascale algorithms for generalized MPI comm split. In: Recent Advances in the Message Passing Interface. pp. 9 - 18. EuroMPI11 (2011)

Summary

Space and time complexities for different group creation schemes

	MPI(typical)	Centralized	Shrink-&-Balance	Rank-&-Hash
Space	$O(n)$	$O(m)$	$O(\log n)$	$O(1)$
Time	$O(n + m \log m)$	$O(m + \log n)$	$O(\log^2 n)$	$O(\log n)$
Msg Count	$n \log n$	$n + m$	$\Omega(n + m)$	$n + 4m + \frac{m}{k}$
Max Msg Size	$O(n)$	$O(m)$	$O(\log n)$	$O(1)$

Summary

- System assigned ranks eliminate sorting of user-supplied keys
- Spanning-Tree based groups
  - Balanced
  - k-ary
  - Low memory usage
  - Outperforms traditional creation mechanisms

- Evaluate performance in the presence of other computation and communication akin to real application execution scenarios
- Account for network-topology by executing these algorithms hierarchically