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

Akhil Langer, Ramprasad Venkataraman and Laxmikant Kale
Department of Computer Science
University of Illinois at Urbana-Champaign
{alanger, ramv, kale}@illinois.edu

***Abstract*—Parallel programs often express operations on a subset *(process group)* of all the participating processes or ranks. Subcommunicators in MPI are an example of such process groups. Often, these process groups are used only for simple collective communication (broadcast, reduction, allreduce) over the members of the process group.**

Current algorithms to create process groups tend to be centralized schemes which store or manipulate data structures of size proportional to the number of members in the process group. In extreme scale parallel architectures, these algorithms will consume a prohibitive amount of memory to manage the process group and also suffer from scalability limits.

In this paper, we contend that MPI subcommunicators pack more features than are necessary for a sizeable fraction of the use cases. We make a case for *system-ranked process groups*, intended primarily for simple collective operations. As opposed to MPI communicators, member ranks in system-ranked process groups are assigned by the runtime system.

This paper presents distributed algorithms for the creation of spanning trees for rudimentary collective communication over system-ranked process groups that are obtained by splitting an original parent process group. Our schemes use only a small constant amount of memory per node and also perform better than a reference centralized scheme even at modest process counts. We also demonstrate performance results up to $131,072$ cores of BlueGene/P.

Our algorithms can apply to the creation of MPI subcommunicators as well as to equivalent entities in other parallel programming paradigms.

***Keywords*-sub-communicators, spanning trees, distributed algorithms, exascale**

I. Introduction

Process Groups are subsets of processes (ranks) in a parallel program that participate in some specific portion of the parallel execution and are addressable as a unified entity. Such process groups greatly facilitate the expression of parallel algorithms and the development of modular parallel software components. Apart from these productivity benefits, they allow the underlying runtime software stack to provide encapsulated, high-performance routines for communication within these process groups. Most programming models provide entities equivalent to process groups (communicators in MPI) and mechanisms to create, store and manage these entities.

Parallel runtime software typically manages the numerous process groups created by user code via schemes whose storage requirements and computational complexity increase in proportion to the number of members in the process group (or worse). On current and future extreme-scale architectures, these approaches will consume a prohibitive amount of memory and hit scalability limits. This is compounded by the fact that parallel programs typically create and use many such process groups.

It is quite important that runtime implementations provide inexpensive process groups. Trends in high performance system architecture point to a slower growth in the available memory than the increase in the number of threads of execution [1]. As memory continues to become a more valuable resource during parallel program execution, it is imperative that runtime software adopt leaner, resource-conserving algorithms and book-keeping mechanisms to manage process groups.

The work presented in this paper is motivated by these realizations. To retain relevance to multiple parallel programming paradigms, we do not consider the design of specific data structures or book-keeping mechanisms required for managing the full life cycle of process groups in a parallel program. Instead we focus only on mechanisms for the creation of process groups. We preface our work by making a case for system-ranked process groups with a relaxed feature set that can be realized by simply constructing spanning trees over the process group (section II). We then explore distributed algorithms for the creation of communication trees spanning new process groups obtained by enrolling a subset of members from a parent process group. To ensure support for nested (or recursive) partitioning of a parent process group into smaller process groups, we assume that initial communication for spawning a new process group will occur over the spanning tree of the parent. We base our algorithms on the assumption that memory is a constrained resource, and impose limits on their transient and final consumptions. We ensure that memory requirements are bounded and do not increase with the size of the process group.

Our efforts have resulted in distributed tree construction algorithms that only consume $O(k)$ memory per process and take $O(\log_k N))$ time (where k is the branching factor of the spanning tree, and N is the number of members in the parent process group). We present two new schemes for the distributed construction of spanning trees: a Shrink-and-Balance algorithm in section V, and a Shrink-and-Hash algorithm in section VI. We also discuss some variations in the algorithms and analyze their relative merits. We discuss and implement a reference "centralized" spanning tree con-

struction program that exhibits the aforesaid $O(N)$ memory and time consumption behavior (section IV). To corroborate our analysis with actual measurements, we implement our algorithms and compare their performance with the reference implementations (section VII). Our implementations scale well to large supercomputers that exist today and the performance of the new algorithms is competitive with the centralized scheme.

II. SYSTEM-RANKED PROCESS GROUPS

In this section we propose that system-ranked process groups will satisfy a sizeable portion of the use-cases for process groups in a parallel program. Our convictions arise partly from studying several tera/petascale parallel applications implemented atop a parallel programming framework that provides system-ranked process groups as a core feature.

A. Usage Contexts

We commence by discussing scenarios in which the intended usage of a process group does not depend on a specific assignment of ranks to processes within the process group.

1) Simple Collective Communication: Parallel programs predominantly use process groups to perform collective communication amongst members. From the list of supported collective operations, *barrier, broadcast, reduce and allreduce* can satisfy a large fraction of collective communication requirements in parallel algorithms. A sizeable fraction of collective communication calls in applications may involve these operations. The results of these collective operations are independent of the ranks from which the individual data contributions arise from. Below we enumerate some examples of parallel algorithms where collective communication needs within process groups can be met by the calls listed earlier.

- Dense LU Factorization: Dense LU is a popular compute benchmark that used on almost all major supercomputers. Implementations typically adopt a parallel, block decomposition and use process groups to broadcast factorized blocks across processed owning a row of blocks. They also use process groups to group processes owning columns of blocks. These column process groups participate in what is effectively an allreduce operation where the pivot row is identified via a reduction and then broadcast to all the processes in the section. All operations described here do not depend on any specific assignment of ranks to the processed participating in any of the sections. Recent work has demonstrated high performance implementations in modern parallel programming paradigms that provide support for system-ranked process groups [2], [3].
- Plane-wave based ab-initio quantum chemistry: OpenAtom is a massively parallel quantum chemistry application that divides the computation into several phases. A description of the structure of the application can be found in [4] and demonstrates the extensive use of multiple process groups, predominantly for performing broadcasts and reductions.
- Master-Worker Algorithms A master-worker expression of several parallel algorithms offers a large space of programs and benchmarks that primarily use broadcasts and reductions during their execution. Many of these have use for process groups in efficiently expressing parallel logic. Some examples include:
  - Map-Reduce
  - Histogram sorting
  - Divide-and-Conquer
  - Monte Carlo computations

2) Parallel Libraries / Modules: Parallel libraries, and modular components in a parallel program are often given their own execution and communication contexts by spawning their execution within process groups. Such use is a compelling case for letting the runtime assign ranks within the process group.

B. Benefits

MPI is the de facto standard in parallel programming. In MPI, process groups are always created by having the client program supply a key that determines its rank in the process group. However, as discussed above, such user-assigned ranks within a process group are not always necessary. MPI communicators, by always requiring user-assigned ranks, bind themselves into supporting communication infrastructure that is bulkier than needed. Removing this requirement can lead to more scalable resource management by the runtime software.

III. TERMINOLOGY

Table I lists the definition of various symbols used in the paper.

TABLE I: List of Symbols

n	Total number of processes
m	Number of processes that participate in the new process group i.e. size of the new process group
k	Branching factor of the spanning tree
my_id	process identifier, goes from 0 to $n-1$ for n processes
my_rank	rank of the process in a process group (my_rank is same as my_id when all the processes are present in the process group)
$d_{i,k}$	Depth (generation number) of a process with rank $i-1$ in a balanced spanning tree of branching factor k
p	fraction of members of the original process group that participate in the new process group
Original spanning tree	Balanced spanning tree (branching factor k) of all the processes
Hole	A process in the original spanning tree that is not participating in the new process group

Given its rank (my_rank) and the size of the process group, a process can compute the number of children it has and the ranks of its parent and children in a balanced spanning tree of branching factor k using the formulas given in Equation 1, 2

and 3 respectively.

$$\text{\#children} = \begin{cases} k, & k*(my\_rank+1) < n \\ max(0, n-1-k*my\_rank), & otherwise \end{cases} \quad (1)$$

$$\text{parent\_rank} = \begin{cases} 0, & my\_rank = 0 \\ \lfloor \frac{my\_rank-1}{k} \rfloor, & otherwise \end{cases} \quad (2)$$

$$\text{children rank} = k*my\_rank + i + 1 \text{ for } i = 0 \text{ to \#children} \quad (3)$$

The height of a balanced spanning tree of size n and branching factor k is given by equation (4).

$$d_{n,k} = \lceil \log_k(n(k-1)+1) \rceil \quad (4)$$

IV. The Reference Centralized Algorithm

The centralized algorithm takes place in the following two steps:

A. Upward Pass

Each process participates in a *gather* operation that collects enrollment information. Only the participating processes contribute their ids into the gather. The result of the operation is an array of process ids that are members of the new process group. This gather takes place over the original spanning tree of size n and branching factor k. Since there are $d_{n,k}$ levels in the spanning tree, the latency term is given by $d_{n,k}\alpha$. If there is no network contention, the bandwidth term is given by the summation $\sum_{i=1}^{i=d_{n,k}} k^i\beta$. Assuming that a fraction p of the members of the original process group participate in the new process group, the bandwidth cost reduces by a factor of p, hence the total time complexity of this phase is:

$$T_1 = d_{n,k}\alpha + pnk\beta$$

B. Downward Pass

At the completion of the enrollment (upward) phase, the root of the parent process group possesses the $id's$ of all the participating processes. It selects one of the participating processes as the root of the new process group and hands the membership roster over to it. The new root process selects k process as its children, splits the remaining process list into k parts and sends one part to each child. The process is repeated on each of the child processes until no more processes are left. The cost of this step is:

$$T_2 = d_{m,k}\alpha + mk\beta$$

Since $pn \cong m$, the total cost of the centralized scheme is given by:

$$T = T_1 + T_2 \quad (5)$$

$$T = (d_{n,k} + d_{m,k})\alpha + 2pnk\beta = O(n) \quad (6)$$

And the maximum memory at the root = $O(n)$

V. The Shrink-and-Balance Algorithm

A. Upward Pass

The Centralized scheme collects enrollment information, but does not act on this information until it reaches the root of the parent spanning tree. Because we'd like to avoid gathering $O(N)$ enrollment data, we begin our search for an improved algorithm by using enrollment information earlier, during the initial upward phase. The Shrink-and-Balance scheme is based on the idea of immediately using enrollment information to shrink the original spanning tree by excluding processes which will not be members in the new process group. Excluding non-participating processes during the upward enrollment phase will result in *holes* at the vertices of the original tree where processes choose to drop out of the new process group. In order to maintain a contiguous tree structure, these holes need to be "filled" with processes that are members in the new process group. We achieve this in one of two ways:

- Child-as-filler
  Any process that will not participate in the new process group selects a participating child to replace itself in the new spanning tree. The nominated child, which may have other children of its own, now "fills" the hole left by the parent. Additionally, it assumes parenthood of its immediate siblings that are participating in the new process group. If a vertex and none of its chilren are participating, a no-participation message is sent to its parent vertex. Figure 1 demonstrates the step-by-step progress of the upward pass of this scheme.
  The repurcussions of promoting an immediate child to fill a hole is the increase in the number of children that the filler process is now burdened with. We observe in our experiments that this is quite likely to result in "bushy" vertices at the end of the upward phase, where some vertices have many more than k children.
- Leaf-as-filler In this scheme, a hole is filled with one of the leaf vertices of the subtree rooted at the hole. During the upward pass, a list of candidate filler processes comprised only of leaves in the new spanning tree is generated and transmitted up the original spanning tree. A process with id i sends a maximum of $d_{i+1,k}$ candidate fillers to its parent. If the parent (with id j) is not participating in the new process group it picks a *victim* process from the set of candidate fillers that it has received from its children. A maximum of $d_{j+1,k}$ fillers from the remaining set is sent to its parent. Figure 2 demonstrates the step-by-step progress of the upward pass.
  The Leaf-as-filler scheme increases the size of the messages during the upward phase compared to the Child-as-filler scheme. However, it can avoid the bushy trees that may result from the Child-as-filler approach. The size of the messages is tunable based on the number of candidate fillers that are transmitted upward. However, each vertex in the spanning tree must transmit a number that is sufficient to fill a string of consecutive holes all the way to the root of the original spanning tree. The scheme

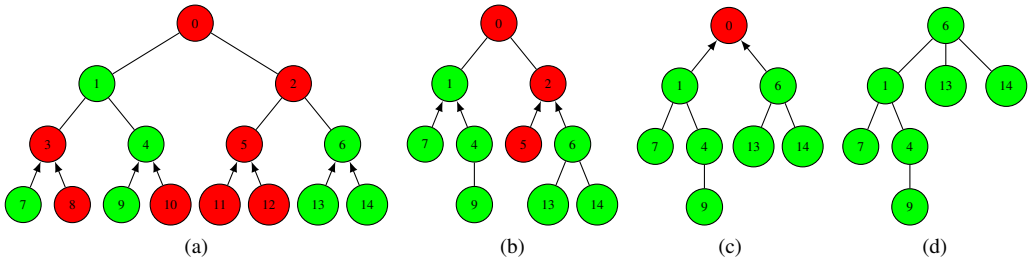


Fig. 1: Level-by-level demonstration of the upward pass in the Child-as-filler scheme

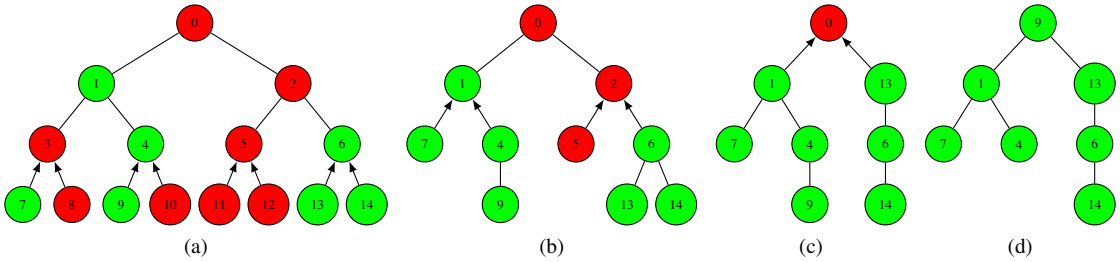


Fig. 2: Level-by-level demonstration of the upward pass in the Leaf-as-filler scheme

can also be used to transmit extra fillers, that are passed down sibling subtrees for filling holes elsewhere in the tree. Such extra fillers, may move some surplus vertices out of supplier subtrees during the upward pass itself, thereby reducing the effort required during the downward pass to balance the tree.

In the downward pass when using a Leaf-as-filler scheme a list of *victim* process ids is also sent alongwith the rank information. A vertex updates the sizes of its children based on the list of *victim* ids received from its parent.

With either strategy, the upward pass is commenced by the leaves in the original spanning tree, and is initiated at any vertex upon receipt of messages from all children of that vertex in the original tree. The process of filling holes applied in this bottom-up manner yields subtrees of only participating processes at each step of the upward phase. This property is used to count and transmit the size of the newly constructed participating sub-tree upward. This results in constant message sizes and memory consumption during the upward phase unlike the gather performed in the reference Centralized scheme.

In order to speed up the critical path (the upward communication), messages transmitted up the spanning tree can be sent directly by the non-participating process at a hole on behalf of the filler process. This will overlap upward progress with the messages that perform the actual "filling" (i.e., a relocation message to the filler, and to all processes that are now part of newly formed parent-child relationships.

B. Downward Pass

Once a participants-only spanning tree is realized during the upward pass, further communication to complete all pending steps in the algorithm takes place over this tree, and the non-participating processes are no longer involved. Although, the upward pass yields a contiguous spanning tree, there are no guarantees on its quality. Hence, in order to obtain a balanced tree with the desired branching factor, the algorithm continues into a downward pass. The algorithm is listed in listing V-A

In the downward pass, the Shrink-and-Balance algorithm attempts to balance the tree while keeping the number of vertex migrations to a minimum. At the end of the upward pass, the root of the tree is aware of its size. This is used to compute the ideal height h of a perfectly balanced spanning tree of that size (Equation 4). This target height is used to compute the maximum number of vertices that can fit within each of the subtrees of the root (Equation 7). If the size of any of the subtrees is greater than its maximum capacity, it indicates that some vertices must move out of these subtrees in order to limit its height to $h-1$. These subtrees are then marked as a vertex *suppliers* with the number of excess vertices marked as the *availability* of that supplier. After identifying the vertex suppliers, they are assigned to child vertices (*consumers*) that have sufficient leftover capacity ($size < maximumcapacity$) to accommodate all the excess vertices in that supplier. Suppliers with availability greater than the remaining capacity of any of consumers, are split amongst them. Suppliers assigned to a single consumer, are sent directly to that vertex. However, those that are amongst multiple consumers are sent a vertex request message on behalf of each each consumer. The supplier then identifies child suppliers from its subtree that can be assigned directly to the assigned consumers.

$$maxChildSize = \frac{k^{h-1}-1}{k-1} \tag{7}$$

Algorithm 1 Downward Pass in Shrink-and-Balance algorithm

Input: vertex supplier list($\mathcal{S}$), list of fillers used by ancestors($\mathcal{F}$), finalSize
update children sizes based on the fillers used by ancestors
for filler f in $\mathcal{F}$ **do**
  $\{size(c) \leftarrow size(c) - 1 | f \in subtree(c)\}$
end for
myCurrentSize $\leftarrow \sum size(c_i) + \sum nAvailable(\mathcal{S}_i)$
calculate new height
$h \leftarrow \log_k(size * (k-1) + 1)$
calculate maximum capacity of children
maxChildSize $\leftarrow \frac{k^{h-1}-1}{k-1}$
currentChildCnt $\leftarrow 0$
for each child with size > 0 **do**
  currentChildCnt $\leftarrow$ *currentChildCnt* $+1$
  extraNodes $\leftarrow max(0,$ *childSize* $-$ *maxChildSize*$)$
  if *extraNodes*> 0 **then**
    add child to the vertexSupplier list $\mathcal{S}$
  end if
end for
vertices4Export $\leftarrow max(0,$ *myCurrentSize* $-$ *finalSize*$)$
finalChildrenCnt $\leftarrow min($*finalSize*$, k)$
calculate number of missing children
childVtcsRequired $\leftarrow$ finalChildrenCnt $-$ currentChildCnt
request *childVtcsRequired* vertices from suppliers in $\mathcal{S}$
select *vertices4Export* vertices from $\mathcal{S}$ and add to $\mathcal{E}$
# remaining vertices in $\mathcal{S}$ are assigned to the children
for every child **do**
  remainingCapacity$(c_i) \leftarrow$ *maxChildSize*$-$*childSize*
end for
# sort suppliers in decreasing order of available vertices
$sort(\mathcal{S})$
for supplier $s \in \mathcal{S}$ **do**
  if $\exists c \in \mathcal{C}:$ *remainingCapacity(c)* $> nAvailable(s)$ **then**
    assign s to c
    update *remainingCapacity(c)*
    $\mathcal{S}$.remove(s)
  end if
end for
for supplier $s \in \mathcal{S}$ **do**
  distribute/split s amongst children in $\mathcal{C}$
  update remainingCapacity of children
end for
for $c \in \mathcal{C}$ **do**
  if $c.id >= 0$ **then**
    call downwardpass on c
  end if
end for
for suppliers which were split amongst children
send vertex request to them and they will directly
send the suppliers to the children.

if some child id was not known at that time,
the suppliers will be sent to the parent
and parent will forward the suppliers once the child id
is known

Once the root vertex completes this "matchmaking" step of assigning consumer(s) to each supplier, it completes it participation in the downward pass. Each of the supplying children, can now proceed in a similar fashion, where they are aware of target sizes, and simply match the assigned consumer(s) with the appropriately sized subtrees. Once a subtree is identified that can be completely consumed by an assigned consumer, the root of the subtree is made a child of the consumer. Consumers wait to receive the appropriate number of vertices, and then launch a balance process within their own subtrees. At the end of such a downward pass, all subtrees with excess vertices move them to other trees which can absorb the supply without growing beyond target height limits. This results in a tree that is as shallow as possible (and hence as balanced as need be). It should be noted, that the algorithm ventures into such vertex relocations only if it will result in an overall shorter spanning tree and does not unnecessarily strive for perfect balance.

This approach limits the total memory footprint of the creation process to a very small number. Our experiments bear this out, although a theoretical upper bound on the space complexity of this approach is yet to be established.

VI. The Shrink-and-Hash Algorithm

The Shrink-and-Hash scheme works by assigning ranks to the participating processes in the downward pass and then enabling the discovery of the process ids corresponding to a rank in the new process group via a hash function.

A. Upward Pass

The upward pass for the hashing scheme proceeds in a manner identical to the upward phase of the Shrink-and-Balance scheme. Each process contributes the size of its subtree. Let us denote a process's participation decision by *in* $\in \{0, 1\}$, where $in = 0$ means the process is not participating and $in = 1$ means that the process is participating in the new process group.

$$size(u) = in + \sum_{j=0}^{j=\#children} size(c_j) \tag{8}$$

where *size(u)* is the size of subtree rooted at child u. At the end of this upward pass, each node knows the size of subtree rooted at itself as well as the size of the subtrees rooted at each of its children.

B. Downward Pass

The downward pass in this scheme consists of assigning ranks to each participating process in the new process group. The starting point for this process is knowledge of the sizes of each subtree which is available at the root. These sizes are used to assign ranges of ranks to each child process and progressively distribute the ranks down the tree.

Let the children of a node be denoted by $c_1, c_2, ..., c_j$ and their corresponding sizes be $s_1, s_2, ..., s_j$. The root node assigns rank 0 to itself and sends rank $1, 1 + s_1, 1 + s_1 +$

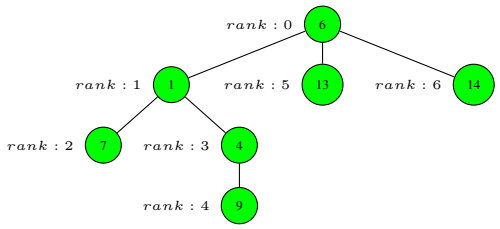


Fig. 3: Element ranks after the downward pass in the Hashing Scheme

$s_2, ..., 1 + \sum_{i=1}^{j-1} s_i$ to $c_1, c_2, c_3, ...c_j$ respectively. The same process is repeated in the subtrees rooted at the children with starting ranks as the ones sent by the parent. Note that messages are not sent to children with no participating node in their subtrees.

Figure 4a and 4b shows the subtree size and rank information after the upward and the downward pass respectively. It should be noted that, like the Shrink-and-Balance scheme, the structure of the tree at the end of the downward pass depends on the hole filling scheme used in the upward pass. For example, figure 3 shows the ranks of participating nodes at the end of the downward pass when using the Child-as-filler scheme for the upward pass.

1) No-filler: Because the Shrink-and-Hash scheme simply doles out ranks to the participating processes in the downward pass, we realized that the presence of non-participating processes can be accommodated by not assigning any ranks to them. Generating ranks in the downward phase does not necessarily require a "shrunk" tree. This No-filler scheme is a third variant of the upward pass strategies and is illustrated in figure 4.

C. Identifying Tree Neighbors

Given the rank of a process and the total tree size, the process can determine the rank of its parent and children using the formulae given in Equation 1, 2 and 3. However, the process ids associated with the parent or child ranks will be unknown for a newly shrunk tree. This algorithm enables the discovery of the process ids corresponding to parent or child ranks in the new subtree by having each rank talk to an intermediary process. The process id of this intermediary can be discovered through a predetermined hash function that will yield a process id H_i, given the rank i in a section. Hence, parent and child ranks in the new process group can exchange process ids via the intermediary process that is identified via the hash function.

Lets denote the intermediary process id of the parent's rank to be H_p and that of the children ranks to be H_{c_i} for $i = 0$ to #children. The process then sends a message containing (my_rank, my_id) to H_p and messages containing (my_rank, my_id, #children) to H_{c_i} for $i = 0$ to #children. All intermediary processes coordinate the exchange of the appropriate process ids. Thus each participating process sends a total of $1+$ *#children* messages (cost: $\alpha + (k+1)\beta$) and receives 2 messages(cost: $\alpha + 2\beta$): one containing the id's of its children and the other one containing the parent id. Since an intermediary process can be the same as one of the participating processes, an additional overhead of receiving and sending a total of $2k+2$ messages (cost: $(2k+2)\beta$) can be incurred, leading to a total cost of:

$$T = 2\alpha + (3k+5)\beta = O(1)$$

VII. Analysis and Results

A. Experimental Setup

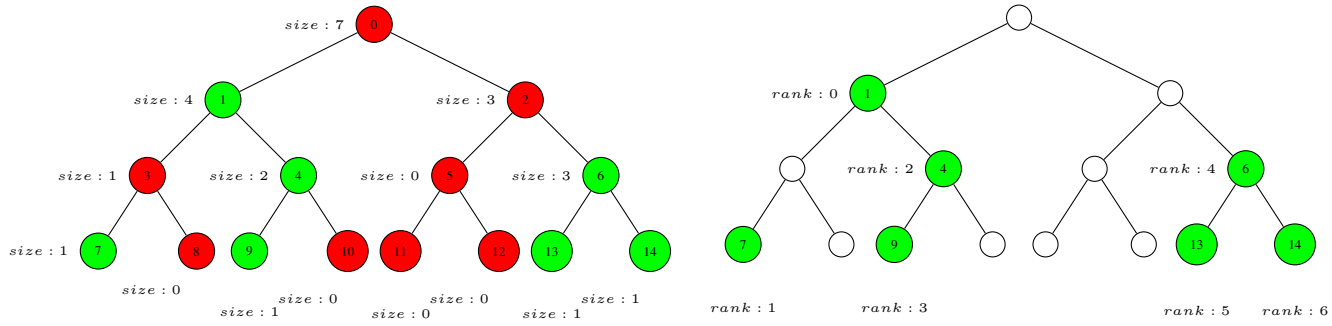
To begin the construction of the new process group a broadcast is issued by process 0 to indicate the beginning of the construction operation. At the end of the construction, a reduction over the newly constructed spanning tree takes place to indicate completion of the operation. We time the construction operation from the beginning of the broadcast to the end of the reduction. Our experiments are designed to measure the time taken to create process groups of varying sizes from a parent process group that spans all the processes in the system. We control the size of the new process group by specifying a participation probability. All processes sample from a random number distribution to determine their participation in the new process group. We ensure identical process groups across multiple runs (to compare the different schemes) by ensuring the same seed to our random number generator. The runs were performed on "Intrepid", an IBM BG/P supercomputer at Argonne National Laboratory. They were performed in the VN mode to maximize the process counts in the experiments. We observed that the variation in execution time across multiple runs was negligible as compared to the program execution time. Hence, we report program timings only from a single run for a given set of input parameters. These algorithms were implemented using the Charm++ parallel programming framework. We report our results for spanning trees with branching factor k but similar results were obtained for other branching factors.

In subsection VII-B we discuss and analyze the space and time complexity of the three filler-identifying schemes. In subsection VII-C, we analyze the number of messages exchanged in different algorithms for constructing the balanced spanning trees. Finally in subsection VII-D, the performance of the proposed distributed algorithms are compared with the reference Centralized scheme.

B. Filler Identifying Schemes

1) No-filler scheme: Both the upward pass and the downward pass take place over the original spanning tree and hence the total cost comprises of a broadcast and reduction, totaling to:

$$T = 2 * d_{n,k}(\alpha + k\beta) = O(d_{n,k}) = O(\log_k n)$$



(a) Subtree size information at each node after the upward pass

(b) Rank information after the downward pass

Fig. 4: The No-filler scheme

2) Child-as-filler scheme: Upon every replacement of a hole with a child, the branching factor of a node increases by a maximum of $k-1$ vertices. In the worst case, the replacement can take place at every vertex on the path to the root and the same vertex can be elected as the *victim* at every level. The branching factor of the tree at the root vertex can thus become $k + (d_{n,k} - 1)(k-1)$. Therefore, the space complexity of this approach is $O(\log_k n)$. Each hole removal requires 1 relocation message sent to the *victim* vertex. The relocation message updates the *victim* with information about its new parent and children. The relocation message sizes are of the order of $O(k)$. Since the relocation messages are sent and processed in parallel with the upward pass, they do not contribute to the time complexity of this scheme. Figure 5 demonstrates this with the help of an execution trace of the processors obtained using Projections [5] (a performance analysis tool). Each process is represented by a horizontal line with the colored blocks on the line representing the execution time spans of methods on that process. Methods are invoked upon receipt of an incoming message.

The downward pass, in the worst case, occurs over a tree of branching factor $O(\log_k n)$. The total time complexity of this scheme is

$$T = d_{n,k}(\alpha + k\beta) + d_{n,k}(\alpha + \log_k n\beta) = O((log_k n)^2)$$

3) Leaf-as-filler scheme: The upward pass messages contain the filler ids and the downward pass messages contain the victim ids. Hence the message sizes are of the order of $O(log_k n)$. Akin to the Child-as-filler scheme, the relocation messages are processed in parallel to the upward pass and hence do not contribute to the time complexity. Hence, the total time complexity of this scheme is:

$$T = 2 * d_{n,k}(\alpha + k\beta) = O(\log_k n)$$

C. Message Counts

The centralized scheme requires one upward and one downward pass. Total number of messages for the centralized scheme:

$$M_c = (n-1) + (m-1)$$

In the Shrink-and-Hash scheme, besides the messages sent during the upward and the downward pass for doing the enrollment and rank dissemination respectively, messages are also exchanged for obtaining process id's corresponding to given ranks (we call these as the *hashing* messages). For a complete k-ary tree with l leaf nodes, there are $t = \frac{l-1}{k-1}$ internal nodes. Total number of messages generated by process group member nodes informing the hashed processes about id's associated with the ranks is $2 * m$. Total number of messages generated by the hashed processes informing the process group members about their parent and children id's = $t + m$, where t messages are the child id messages sent only to internal nodes and m are the parent id messages sent to all but one node. Hence, total number of hashing messages $= 3 * m + t = 3m + \frac{m-1}{k}$.

Our experiments also show that the total number of messages in the upward and downward pass in the No-filler, Leaf-as-filler and Child-as-filler schemes are the same (within ± 1). This is because the number of relocation messages sent during the upward pass in the Child-as-filler and Leaf-as-filler schemes are the same as the number of messages sent ot the holes during the downward pass in the No-filler scheme.

Finally, for the Shrink-and-Balance scheme the number of messages sent for balancing the tree depends on the actual tree structure after shrinking. We have not yet established any theoretical upper bound on the number of messages exchanged but our experiments show that the number of messages in this scheme are similar in number to the messages sent in the reference centralized scheme. Figure 6 compares the messages sent in the Shrink-and-Balance scheme and the Shrink-and-Hash scheme with the centralized scheme as the reference scheme. Number of messages in the Shrink-and-Hash and the Shrink-and-Balance scheme are 125% and 8% of the Centralized scheme at $p = 0.3$, respectively. Table 2 gives in-detail comparison of the message counts for the Shrink-and-Balance and the Shrink-and-Hash scheme with the Centralized scheme.

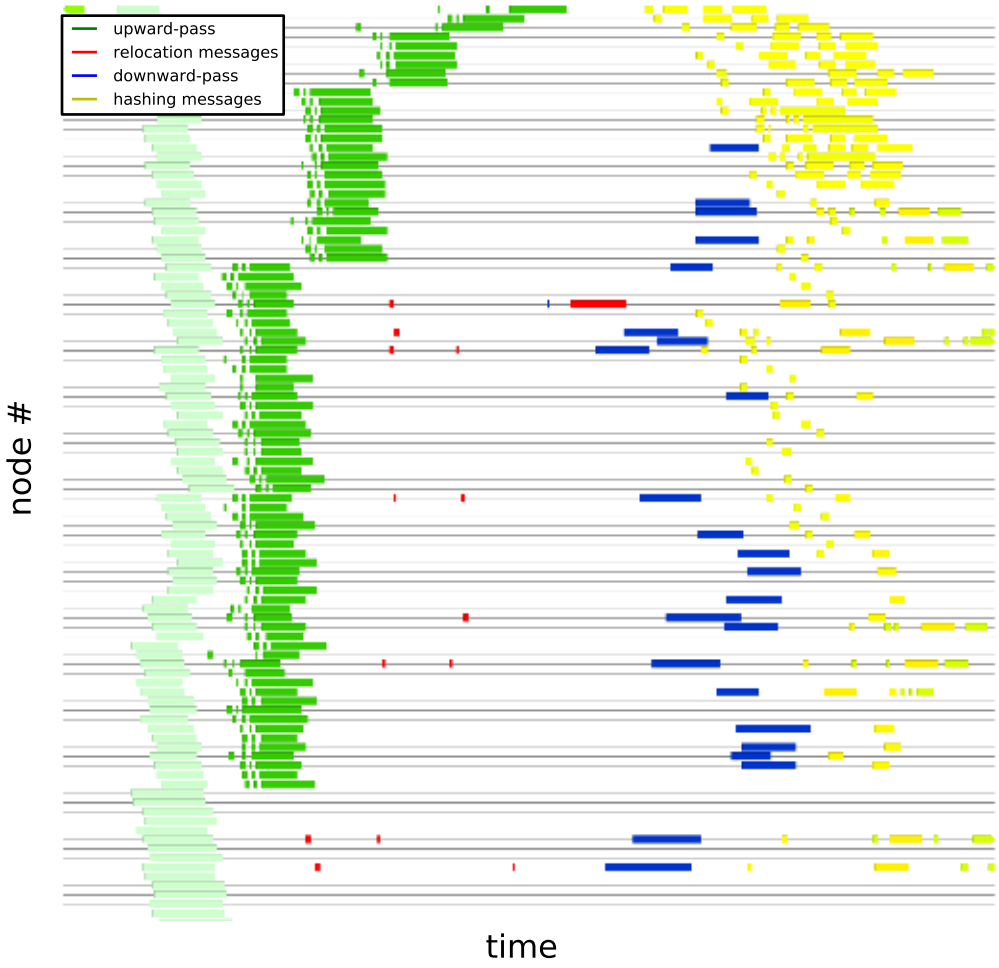


Fig. 5: Projection traces showing the processing of relocation messages in parallel to the upward pass

TABLE II: Comparing the tree construction time (in microseconds) and total number of messages sent during the construction in the centralized, Shrink-and-Balance and the Shrink-and-Hash schemes at different scales and subgroup sizes

fraction		2k		4k		8k		16k		32k		64k		128k	
		time	Msgs	*time*	Msgs	*time*	Msgs	*time*	Msgs	*time*	Msgs	*time*	Msgs	*time*	Msgs
0.001	*cent*	334	2050	359	4100	399	8203	464	16409	534	32825	613	65687	674	131363
	spass	378	2065	458	4124	616	8267	715	16550	847	33123	981	66430	1190	132846
	hash	354	2072	389	4139	436	8299	474	16620	541	33301	582	66905	705	133697
0.01	*cent*	388	2074	472	4144	506	8285	549	16555	655	33149	727	66312	859	132583
	spass	508	2172	596	4307	718	8611	824	17161	1013	34478	1121	69014	1246	137853
	hash	402	2252	459	4456	471	8890	520	17675	635	35615	609	71326	688	142346
0.1	*cent*	492	2263	541	4512	611	8991	723	17999	870	36056	1214	72213	1852	144401
	spass	639	2595	708	5163	835	10294	981	20657	1137	41518	1224	83342	1432	166660
	hash	442	3236	482	6437	506	12715	566	25561	607	51414	690	103397	743	206716
0.3	*cent*	577	2671	616	5313	797	10610	962	21238	1403	42589	2089	85326	3666	170567
	spass	677	3209	761	6348	914	12665	1052	25446	1507	52311	1347	102684	1614	205129
	hash	466	5114	515	10103	542	20093	594	40342	639	81207	680	163142	746	325774
0.6	*cent*	571	3284	673	6545	822	13120	1195	26209	1704	52479	2939	105015	5337	209854
	spass	735	3640	992	7581	1023	14525	1374	32384	1517	58917	1546	113569		
	hash	497	7649	503	15215	547	30584	602	61055	653	122353	746	244977	804	489115
0.9	*cent*	622	3914	716	7820	970	15611	1401	31183	2119	62366	3903	124750	7823	249206
	spass	786	3967	905	7926	1116	15853	1330	31696	1614	63427	1785	126830		
	hash	488	10187	535	20341	584	40584	609	81027	678	162088	720	324209	783	647166
0.99	*cent*	625	4073	776	8158	1029	16314	1514	32636	2320	65247	4098	130513	8337	260982
	spass	800	4078	921	8170	1184	16342	1387	32685	1684	65341	1946	130686		
	hash	495	10829	534	21711	577	43417	621	86860	698	173606	721	347277	802	694391

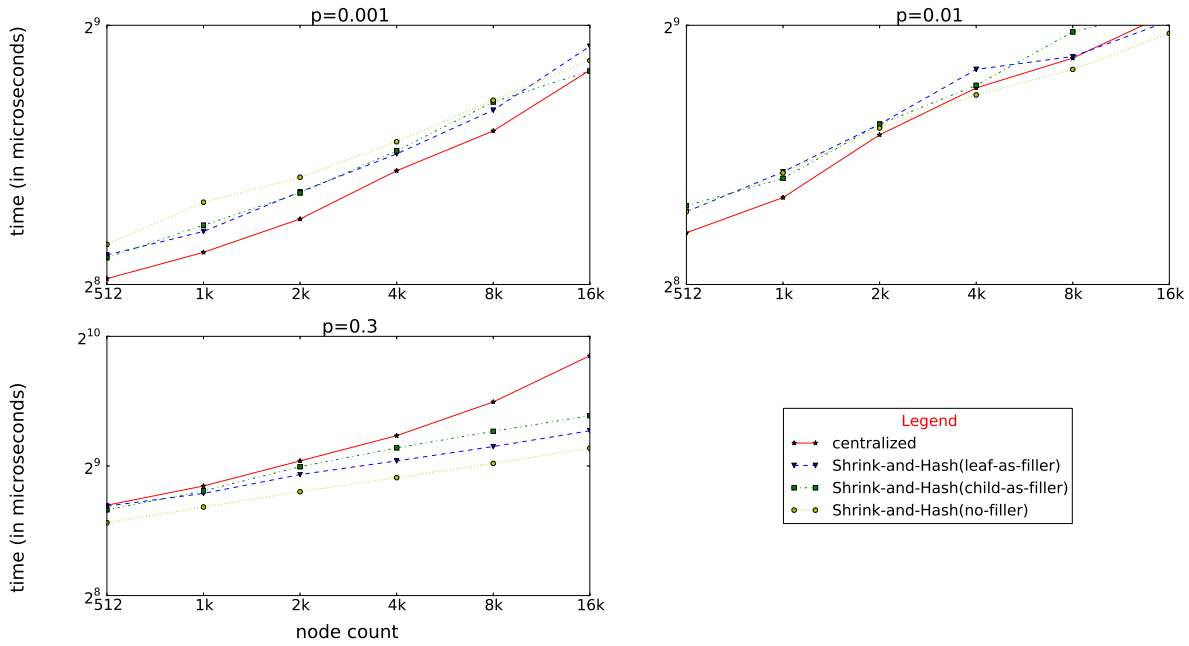


Fig. 7: Scaling results with Shrink-and-Hash scheme for different process group sizes for spanning trees with branching factor 3

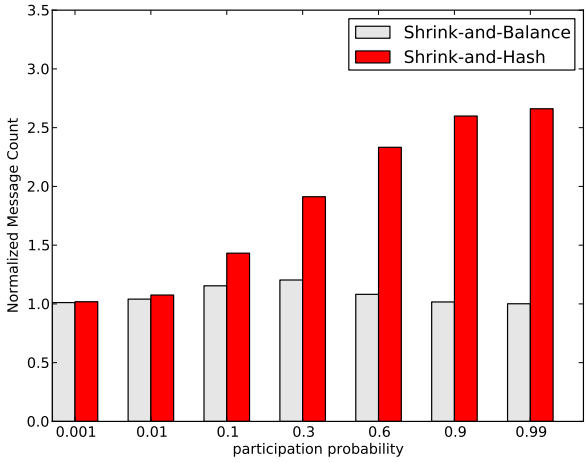


Fig. 6: Message count comparison between Shrink-and-Hash and Shrink-and-Balance scheme relative to the Centralized scheme for different process group sizes with $64k$ as the size of the initial process group. Centralized scheme is the reference scheme here(1.0)

D. Total time

The results show that the distributed schemes outperform the centralized scheme in most of the cases even at very modest number of process counts. In this subsection we first analyze the performance of the Shrink-and-Hash scheme with the three different filler-identifying schemes and then we compare the performance of the Centralized, Shrink-and-Balance and the Shrink-and-Hash schemes.

Figure 7 compares performance of the Shrink-and-Hash scheme implemented with the three filler-identifying schemes, namely the No-filler, Child-as-filler and Leaf-as-filler scheme. Except at very small participation probabilities (e.g. $p = 0.001$ in Figure 7), the No-filler scheme consistently performs better than the other two distributed schemes. The slow down in the Leaf-as-filler scheme is attributed to the extra processing associated with the selection and packing of fillers and the larger message sizes ($O(logn)$). The Child-as-filler scheme performs worse than the other two schemes at $p = 0.01$ and $p = 0.3$. This is because of its large branching factor($O(k \log_k n)$) during the downward pass.

In Table 2 we compare the execution time for the Shrink-and-Balance scheme (with Leaf-as-filler scheme used for the upward pass) and the Shrink-and-Hash scheme with the Centralized scheme. Figure 8 summarizes the results by showing the performance of the three schemes at $64k$ processes and different process group sizes. Shrink-and-Hash scheme outperforms the Centralized scheme in all cases while the Shrink-and-Balance scheme starts to outperform the Centralized scheme for $p = 0.1$ and greater.

The Shrink-and-Balance scheme, despite offering much lower message counts performs worse than the Shrink-and-Hash scheme. This can be explained by shorter critical path to completion offered by the Shrink-and-Hash scheme. There are only $O(k)$ messages on the critical path after the end of

the downward pass (which is of length $O(\log n)$). In contrast, the Shrink-and-Balance scheme contends with a longer critical path in the downward pass.

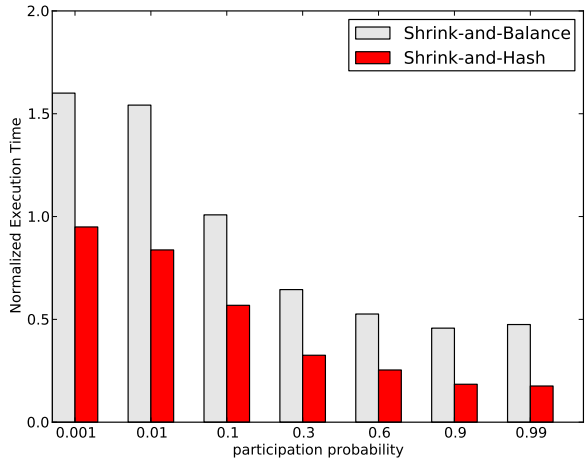


Fig. 8: Execution time comparison between Shrink-and-Hash and Shrink-and-Balance scheme relative to the Centralized scheme(1.0)

VIII. Related Work

There has been significant amount of emphasis and subsequent work on the need of exascale algorithms for communication in message driven paradigms. Balaji et.al. in [6], [7] discuss the memory overheads in communicator creation in MPI. The per communicator memory usage increases with system size which significantly affects the number of subcommunicators that can be created as the system size increases. e.g. in BG/P for $128K$ processes, the number of new communicators that can be created drops to as low as 264 with default MPI settings.

Moody et. al. in [8] present use cases for a generalized MPI_Comm_split in which the reordering function is disabled and processes within the new subgroup are ordered according to their rank in the initial group. They present a linked list based distributed representation of the subgroups that use $O(1)$ memory and can run scans and associative reductions in $O(\log N)$ time. They mention that tree-based collectives can also be implemented using linked list representation but they do not give any details. Further, their approach does not support general point-to-point communication.

Paul Sack et.al. propose in [9], a distributed algorithm for subcommunicator construction in MPI_Comm_split that uses $O(n/p)$ memory where p is the number of sorting processes used for sorting. Subcommunicator construction times are order of magnitude larger than our spanning tree construction algorithms.

IX. Conclusions and Future Work

Existing schemes for subcommunicator creation in MPI are not suitable for exascale systems. We presented motivations for supporting system-ranked process groups. We developed two different algorithms, Shrink-and-Balance and Shrink-and-Hash, for constructing balanced spanning trees over a process group. The Shrink-and-Hash scheme takes only $O(1)$ space complexity and $O(logN)$ time. We presented the performance of these algorithms on up to $128K$ cores of BG/P (80% of the total system size). Our results show that these algorithms perform better than the Centralized scheme even at modest process counts and significantly outperform the Centralized scheme at large scale. These algorithms are yield process groups that are well-suited for collective operations like broadcasts, reductions, and allreduce. They can also be adapted for point-to-point communication.

There are several immediate extensions to the work we describe here. We intend comparing the performance of the Shrink-and-Balance and Shrink-and-Hash schemes in the presence of other communication and computation akin to real application execution scenarios. We also plan to evaluate the performance of a multi-color enrollment process that more closely mimics the MPI's comm_split operation. We believe these experiments will throw more light on the relative merits of the two algorithms discussed here, and possibly lead the way to further improvements. We also plan to evaluate solutions for accounting for network-topology while constructing the spanning trees.

References

[1] E. Peter Kogge, "Exascale computing study: Technology challenges in achieving exascale systems," Tech. Rep., 2009.

[2] J. Lifflander, P. Miller, R. Venkataraman, A. Arya, T. Jones, and L. Kale, "Exploring partial synchrony in an asynchronous environment using dense LU," Parallel Programming Laboratory, Tech. Rep. 11-34, August 2011.

[3] ——, "Mapping dense lu factorization on multicore supercomputer nodes," in *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2012*, May 2012.

[4] E. Bohm, A. Bhatele, L. V. Kale, M. E. Tuckerman, S. Kumar, J. A. Gunnels, and G. J. Martyna, "Fine Grained Parallelization of the Car-Parrinello ab initio MD Method on Blue Gene/L," *IBM Journal of Research and Development: Applications of Massively Parallel Systems*, vol. 52, no. 1/2, pp. 159–174, 2008.

[5] L. V. Kale, G. Zheng, C. W. Lee, and S. Kumar, "Scaling applications to massively parallel machines using projections performance analysis tool," in *Future Generation Computer Systems Special Issue on: Large-Scale System Performance Modeling and Analysis*, vol. 22, no. 3, February 2006, pp. 347–358.

[6] P. Balaji, A. Chan, R. Thakur, W. Gropp, and E. Lusk, "Toward message passing for a million processes: Characterizing mpi on a massive scale blue gene/p," *Computer Science-Research and Development*, vol. 24, no. 1, pp. 11–19, 2009.

[7] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, S. Kumar, E. Lusk, R. Thakur, and J. Träff, "Mpi on a million processors," *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pp. 20–30, 2009.

[8] A. Moody, D. Ahn, and B. de Supinski, "Exascale algorithms for generalized mpi_comm_split," *Recent Advances in the Message Passing Interface*, pp. 9–18, 2011.

[9] P. Sack and W. Gropp, "A scalable mpi_comm_split algorithm for exascale computing," *Recent Advances in the Message Passing Interface*, pp. 1–10, 2010.