Scalable 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. Current implementations of process groups (subcommunicators) have non-scalable (O(group size)) memory footprints and even worse time complexities for setting up communication. We propose *system-ranked process groups,* where member ranks are picked by the runtime system, as a cheaper and faster alternative for a subset of collective operations (barrier, broadcast, reduction, allreduce).
This paper presents two distributed algorithms for balanced, k-ary spanning tree construction over system-ranked process groups obtained by splitting a parent group. Our schemes have much smaller memory footprints and also perform better, even at modest process counts. We demonstrate performance results up to 131,072 cores of BlueGene/P.

Keywords: distributed algorithms, exascale, spanning trees, process groups, sub-communicators

1 Introduction

Process Groups are subsets of processes (ranks) in a parallel program that participate in specific portions of the parallel execution and are addressable as a unified entity. Most parallel programming models provide entities equivalent to process groups (communicators in MPI) and mechanisms to create, store and manage these entities. Several existing parallel runtime implementations require $O(n)$ storage and $O(n \log n)$ computation per process to create and manage a process group with n members [5,6]. They will consume prohibitive amounts of memory and reach scalability limits on current and future extreme-scale architectures. Parallel programs typically compound this problem by creating and using many such groups. Trends in high performance system architecture point to a slower growth in the available memory than in the number of threads of execution [12]. Thus, 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, and focuses on mechanisms for the creation of process groups. In order to remain relevant to multiple parallel programming systems, we do not consider MPI-specific solutions, nor do we bind ourselves to the current standard. We preface

our work by making a case for *system-ranked process groups* with a reduced feature set that can be realized by simply constructing spanning trees over the group (Section 2). We then explore distributed algorithms for the creation of communication trees spanning new groups obtained by enrolling a subset of members from a parent group. To ensure support for nested (or recursive) partitioning of a parent group, 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 its transient and final consumption.

Our efforts have resulted in two distributed tree construction algorithms: a Shrink-and-Balance algorithm (Section 4), and a Rank-and-Hash algorithm (Section 5). They consume just $O(\log n)$, $O(1)$ memory per process and $O(\log^2 n)$, $O(\log n)$ time respectively. To corroborate our analysis with actual measurements, we implement our algorithms and compare their performance (Section 6) with a reference "centralized" implementation (Section 3) that exhibits $O(m)$ space and $O(m + \log n)$ time complexity; and with a comm_split from a vendor-tuned MPI implementation. Our algorithms scale well to large supercomputers and exhibit competitive performance at large process counts.

2 System-Ranked Process Groups

We propose that unranked or system-ranked process groups be supported in parallel programming systems as they will satisfy a portion of use-cases for process groups at a much lower resource cost.

Motivations Our stance germinates from the observation that user-assigned ranks within a process group are not always necessary to express parallel algorithms. This is especially true of a subset of collective operations: *barrier, broadcast, reduce and allreduce*. The results of these collectives are independent of the ranks from which the individual data contributions arise (assuming commutative operations). There is also evidence that a sizable fraction of collective communication in applications involve these operations [1,13,15]. We enumerate a few examples of algorithms and applications that use just these collectives:

Parallel Linear Algebra: Several algorithms for manipulating linear systems of equations use block or compressed representations of matrices. The algorithms are then expressed by collectively addressing processes that own a row or column of matrix elements/blocks. For eg, recent work has demonstrated a high

Terminology

- **n** Number of processes in parent process group
- **m** Number of processes participating in the new process group
- **k** Branching factor (degree) of the spanning tree
- $d_{i,k}$ Depth of a rank i process in a balanced spanning tree of branching factor k
- **f** fraction of members of original process group participating in new group

performance dense LU factorization using only the aforementioned collectives on non-trivially defined groups of processes, in a parallel programming paradigm that supports unranked and system-ranked process groups [10].

Master-Worker Algorithms: A master-worker expression of several parallel algorithms primarily use broadcasts and reductions during their execution. Many of these have use for process groups in efficiently expressing parallel logic. Some examples include: *a*) Map-Reduce *b*) Histogram sorting *c*) some Divide-and-Conquer algorithms, and *d*) Monte Carlo computations

Ab-initio Quantum Chemistry: OpenAtom is a massively parallel quantum chemistry application with several phases of computation in a step. A description of the parallel structure [3] demonstrates the use of multiple process groups just for performing broadcasts, reductions and allreduces among members.

Approach Since the collectives of interest can be expressed as operations over a communication tree spanning the members of the group, we chose a tree-based representation of groups and cast the problem of efficient group construction into the *efficient construction of trees spanning the members of a new group.*

Possible Functionality System-ranked process groups are primarily for supporting the aforementioned collectives. User-specified roots for the collectives can be supported by: *a*) forwarding data from the tree root to the user-specified root *b*) tolerating some imbalance by using any vertex in the tree as a broadcast root *c*) constructing multiple (but a small number of) trees with different roots. Point-to-point messaging can be supported by discovering and caching the ids of the (typically) small number of frequent communication partners. Finally, in keeping with a pay-only-for-use policy, user-supplied ranks can be supported atop system-ranked groups by performing the sort as an additional step.

Benefits Letting the runtime system assign ranks to processes enrolling in a group liberates it from having to sort user-supplied keys to identify ranks. Avoiding this $O(m \log m)$ computation can result in significant speedups of group creation mechanisms. Tailoring the communicators (groups) to subsets of use-cases will permit implementations that are less resource-intensive and faster. It may also permit optimization of the communication operations themselves.

3 The Reference Centralized Algorithm

Group creation in MPI typically performs an allgather followed by a sort. Removing support for user-assigned ranks eliminates the sorting, but still requires an allgather which takes $O(m)$ storage on each process. Our reference "centralized" implementation replaces this allgather with a gatherv-scatter that only has an $O(m + \log n)$ time and $O(m)$ transient memory footprint. We believe this represents a conservative baseline for comparing our distributed algorithms.

The implementation performs an *upward* pass (gatherv) over the original spanning tree in which only members of the new group contribute their process

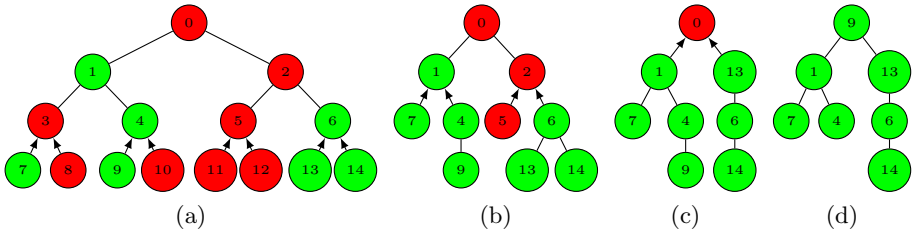


Fig. 1: Tree shrinking in the Shrink-and-Balance scheme using leaves as fillers. Red: non-participating process (hole); Green: participating process.

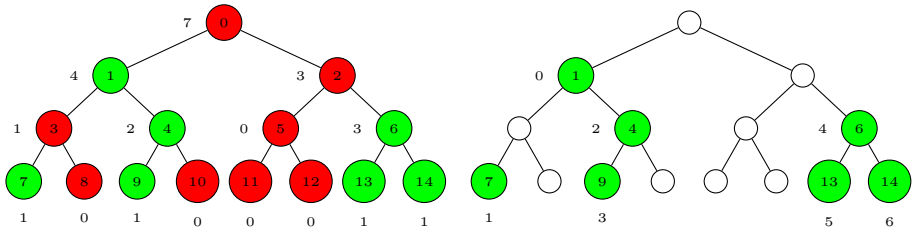
ids. The resulting list is sent to the root process of the new group (picked from the list). This is followed by a *downward* pass (scatter) over the spanning tree under construction. A vertex's immediate children are picked from the list of members and the remaining list partitioned among them to populate their sub-trees.

4 The Shrink-and-Balance Algorithm

Upward Pass The Centralized scheme collects enrollment information, but does not act on it until it reaches the root of the parent spanning tree. Because we'd like to avoid gathering $O(n)$ enrollment data, we base our first algorithm on the idea of using information earlier. The Shrink-and-Balance scheme immediately uses enrollment data during the upward pass to shrink the original spanning tree by excluding non-participating processes. This results in *holes* at the vertices of the original tree where processes choose to drop out of the new group. In order to maintain a contiguous tree structure, these holes are "filled" with processes that are members in the new group. Fillers are either a participating leaf process (Figure 1) or a participating immediate child process. Using leaf vertices as fillers requires a process v with rank i to send $\min(subtree(v), d_{i,k})$ leaves as candidate fillers to its parent; to potentially fill holes at each of its $d_{i,k}$ (1) direct ancestors. Since at most $\log n$ vertices may be sent, the space and time complexity of the upward pass is $O(\log n)$ and $O(\log^2 n)$, respectively. Space constraints prevent a description of using immediate children as fillers.

Downward Pass Although the upward pass yields a contiguous, participants-only spanning tree, there are no guarantees on its quality. To obtain a balanced tree with the desired branching factor, the algorithm continues into a downward pass. All further communication now occurs over the newly constructed tree.

In the downward pass, the scheme balances the tree while minimizing the number of vertex migrations. The size of the new group is used to compute the ideal height h $(d_{m-1,k} + 1)$ of a perfectly balanced tree spanning tree (1). This target height yields the maximum size of each of the subtrees of the root (2). If the size of a subtree is greater than its maximum capacity, some vertices must move out of it in order to limit its height to $h - 1$. All such subtrees (i.e. their



(a) Subtree sizes after the upward pass (b) Ranks after the downward pass

Fig. 2: Assignment of ranks in the Rank-and-Hash scheme

roots) are marked as vertex *suppliers*. Similarly, child vertices whose subtrees are smaller than their maximum permitted sizes, are marked as potential *consumers*.

$$d_{i,k} = \lfloor \log_k(i(k-1)+1) \rfloor \tag{1}$$

$$max\_size = \frac{k^h - 1}{k - 1} \tag{2}$$

Each vertex V (starting with the root), performs a "matchmaking" step, ensuring that each of its supplier subtrees is assigned one or more consumers that can absorb the excess vertices of the supplier within their subtrees. If V has missing children, it requests vertices from supplier(s) for itself. Once these supplier vertices are assigned, each subtree is within its size limits. The vertex V concludes its role by invoking a balancing step on each of its subtrees. The downward pass thus recurses down the tree, ensuring a tree that is as shallow as possible. By checking the current height of a subtree against its maximum permissible height, the algorithm avoids striving unnecessarily for perfect balance.

Our experiments show that the memory footprint of the downward pass is very small, although a theoretical upper bound is yet to be established. For e.g, on $128K$ cores of BG/P, there were at most 13 suppliers for process groups of sizes ranging from 0.1% to 99% of the parent group. Identifying (and moving) an excess vertex may take $O(\log n)$ time in the worst case. Since this can happen at every level of the shrunk tree, the time complexity of the scheme is $O(\log^2 n)$.

5 The Rank-and-Hash Algorithm

The Rank-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 any rank via a hash function.

Upward Pass The upward pass is a simple reduction up the original spanning tree. Each vertex receives participation information from its children, stores this information and passes a reduced count (including its participation decision) up the spanning tree. This leaves each vertex with the size of each of its subtrees.

Downward Pass The size of the tree determines the available ranks $[0, m)$. This range is split among the subtrees based on their sizes. Splitting continues down the original spanning tree until all the available ranks are divided among all the participating processes. Non-participating processes are not assigned any ranks. Figures 2a and 2b show the subtree size and rank information after the upward and the downward passes respectively.

Identifying Tree Neighbors A process of rank i in the new group can compute the ranks of its parent p (3), and children (4) in the tree. However, the process ids of these ranks are still unknown. The discovery of ids is done via *intermediary* processes that mediate an exchange of ids between a parent and its children. The id of an intermediary process (H_i) representing rank i, is computed via a function that hashes rank i to id H_i. Each rank i, sends its id to the intermediaries H_i and H_p, representing the rank i and its parent p. In return, it receives two messages: one each from H_p and H_i with the ids of its parent and children, respectively.

$$p = \begin{cases} 0, & i = 0 \\ \lfloor \frac{i-1}{k} \rfloor, & otherwise \end{cases} \tag{3}$$

$$C_i = \begin{cases} [k*i+1, k*(i+1)] & k*(i+1) < m \\ [k*i+1, m) & k*i+1 < m \leq k*(i+1) \\ \phi & otherwise \end{cases} \tag{4}$$

Memory consumption for each of these phases is small and is independent of group size ($O(1)$). The overall time complexity is $O(\log n)$ each for the upward and downward phases, and $O(1)$ for the hashing phase.

6 Results

Experimental Setup We bracket our algorithms between a broadcast on the original spanning tree and a reduction on the newly constructed tree and time the whole phase. The size of the new group is specified via a participation fraction f. All processes sample from a uniform distribution $u(0, 1)$ and use f to determine their participation in the new group. For fair comparisons, we use repeatable seeds to ensure identical groups across multiple runs. We also apply the same sequential optimizations to all implementations. The runs were performed on "Intrepid", an IBM BlueGene/P supercomputer at Argonne National Laboratory. These algorithms were implemented using the Charm++ [7] parallel programming framework. We report results only for spanning trees with branching factor 3 but similar patterns were observed with other branching factors.

Performance Figure 3 compares the scalability of the two algorithms with the baseline centralized scheme for process groups of different sizes. The results show that except at very low participation fractions (e.g. at $f = 0.01$), the distributed schemes outperform the baseline even at modest process counts. The Shrink-and-Balance scheme is slower than the Rank-and-Hash scheme because of a longer critical path. However, both attain the goal of reduced memory footprint.

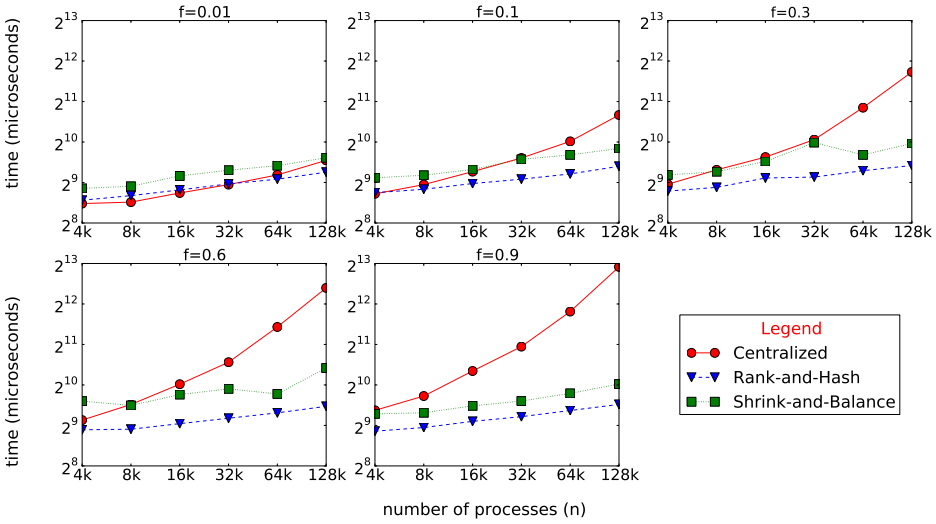


Fig. 3: Scaling behavior of the three algorithms on IBM BG/P at various participation fractions (f) with n ranging from $4,096$ to $131,072K$ processes

Message Counts The total number of messages for a reduce or gather over the original spanning tree is $n-1$; for a broadcast or scatter over the new tree is $m-1$. Hence, the centralized scheme sends $O(n+m)$ messages. The Rank-and-Hash scheme has an additional phase for id exchange. All m vertices send 2 messages to intermediary processes and receive 1 message with parent information. The $\frac{m}{k}$ non-leaf vertices also receive a message with child information. The total message count is hence $n+4m+\frac{m}{k} = O(n+m)$. The Shrink-and-Balance scheme requires additional messages to fill holes during the upward pass, and to identify and move excess vertices during the downward pass. An upper bound on message counts is elusive because it depends on the location of the holes and the quality of the shrunk tree. Our experiments show that counts are far fewer than the Rank-and-Hash scheme. At $128K$ processes and $f = 0.6$ the number of messages sent by the Centralized, Shrink-and-Balance and the Rank-and-Hash scheme were 2.1, 2.6 and 4.9×10^5, respectively. Figure 4 compares message counts in both schemes with the reference. We expect that at extreme scales, when multiple groups are being formed simultaneously, or when group formation occurs concurrently with other communication in the application, the Shrink-and-Balance scheme may have an advantage over the Rank-and-Hash scheme.

Comparison with MPI_Comm_split In the two widely-used open-source MPI implementations: MPICH and OpenMPI, MPI_Comm_split is implemented as an $O(n)$ allgather followed by $O(\frac{n}{c}\log\frac{n}{c})$ sort, where c is the number of colors (assuming splits of equal size). To compare, we implemented a multi-color ver-

sion of the Rank-and-Hash scheme. During the upward-pass member counts of each color are gathered at the root, which takes $O(\min(c \log n, n))$. If the original tree is shrunk into c pieces, dissemination in the downward pass can be accomplished in $O(\log n)$, totaling to a time complexity of $O(\min(c \log n, n))$. As the number of colors approaches n, time complexity of the Rank-and-Hash scheme approaches that of MPI_Comm_split. In Table 1, we compare the performance of MPI_Comm_split with Rank-and-Hash scheme on $32,768$ cores of BG/P.

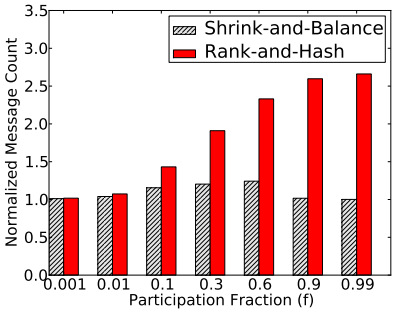


Fig. 4: Message counts, normalized against centralized scheme at same f, for $n = 128K$ on IBM BG/P

splits (c)	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

Table 1: Execution times (ms) of the Rank-and-Hash scheme and MPI_Comm_split from vendor's implementation on $32K$ cores of BG/P

7 Related Work

Balaji et al [2] discuss the memory overheads of subcommunicator storage in MPI and note that memory usage increases with process count, significantly affecting the number of subcommunicators that can be created. They report that, on BlueGene/P, the number of new communicators that can be created at $128K$ processes drops to as low as 264 from 8189 at $1K$ processes. Their findings strengthen the argument for cheap process groups.

Sack et al [14] propose a distributed algorithm for ordered subcommunicator construction that uses $O(n/p)$ memory and $O(p \log n + \log^2 n + \frac{n}{p} \log p)$ time where p is the number of processes used for parallel key sorting. They reduce storage requirements to $O(n/p)$ by using distributed tables for storing the ranks.

Recent work by Moody et al [11] mentions a generalized MPI_Comm_split They propose creating and storing process groups as chains in $O(1)$ memory and $O(\log n)$ construction time. They perform collectives by exchanging appropriate process ids during the operation. Our work exhibits several differences. First, we avoid the extra $O(n)$ messaging required to exchange process ids during every collective call. This also results in lesser dependencies on remote information for the progress of the collective, which should lead to lesser wait times and faster completion of the operation. We believe this benefit will become more prominent for implementations that exploit one-sided data transfer calls provided by some network messaging APIs [4, 9]. Second, our schemes can construct and use communication trees of arbitrary branching factors. Achieving this using chains will

be difficult, and will effectively amount to constructing a spanning tree. This is of practical consequence for collectives on many architectures as binary trees do not always perform as well as other k-ary trees.

Other work [8,16] describes several techniques for the compact representation of MPI groups. They are quite effective in the presence of exploitable patterns in the member ranks. In contrast, we do not design specific data structures or compaction mechanisms, nor do we provide a complete solution for the current MPI standard. We believe several use cases can be met by the altered functionality we propose; and our work explicitly targets relevance beyond MPI.

8 Summary and Future Work

In this paper, we have motivated support for system-ranked process groups and discussed how they are suited to a subset of collectives. We have developed two algorithms, Shrink-and-Balance and Rank-and-Hash, for creating balanced, k-ary tree based process groups while consuming small amounts of memory. We discovered that our algorithms are also faster than a reference implementation even on 128K processes of a terascale supercomputer [1]; and significantly faster than the comm_split implementation of the native MPI library. We summarize our analysis in Table 2.

There are several immediate extensions to the work described here. The Shrink-and-Balance scheme sends fewer messages despite having a longer critical path than the Rank-and-Hash scheme. We intend to evaluate performance in the presence of other communication and computation akin to real application execution scenarios. We also plan further experiments with larger numbers of splits. We believe these experiments will throw more light on the relative merits of the two algorithms discussed here, and possibly lead to further improvements.

Another planned direction is to account for network-topology. The algorithms described here can be executed hierarchically, such that each subtree is restricted to a small neighborhood of the network. This can reduce the number of network links traversed along the tree and improve performance of the targeted collectives. The complexity will be very similar to the current schemes.

This work was supported in part by DOE DE-SC0001845 and NSF ITR-0833188.
[1] *ALCF compute resources were used under DOE contract DE-AC02-06CH11357.*

Table 2: Space and time complexities for different group creation schemes

	MPI(typical)	Centralized	Shrink-and-Balance	Rank-and-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	$O(n)$	$O(m)$	$O(\log n)$	$O(1)$

References

1. Antypas, K., Shalf, J., Wasserman, H.: NERSC6 Workload Analysis and Benchmark Selection Process. Tech. Rep. LBNL-1014E, Lawrence Berkeley National Lab (2008)
2. Balaji, P., Chan, A., Thakur, R., Gropp, W., Lusk, E.: Toward Message Passing for a Million Processes: Characterizing MPI on a Massive Scale Blue Gene/P. Computer Science-Research and Development 24(1), 11–19 (2009)
3. Bohm, E., Bhatele, A., Kale, L.V., Tuckerman, M.E., Kumar, S., Gunnels, J.A., Martyna, G.J.: 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 52(1/2), 159–174 (2008)
4. Cray Inc.: Using the GNI and DMAPP APIs (2010), http://docs.cray.com/books/S-2446-3103/S-2446-3103.pdf
5. Gabriel, E., Fagg, G., Bosilca, G., Angskun, T., Dongarra, J., Squyres, J., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., et al.: Open mpi: Goals, concept, and design of a next generation mpi implementation. In: Recent Advances in Parallel Virtual Machine and Message Passing Interface. pp. 353–377 (2004)
6. Gropp, W., Lusk, E.: Users guide for mpich, a portable implementation of mpi. mathematics and computer science division, argonne national laboratory, 1996. Tech. rep., ANL-96/6
7. Kale, L., Arya, A., Bhatele, A., Gupta, A., Jain, N., Jetley, P., Lifflander, J., Miller, P., Sun, Y., Venkataraman, R., Wesolowski, L., Zheng, G.: Charm++ for productivity and performance: A submission to the 2011 HPC class II challenge. Tech. Rep. 11-49, Parallel Programming Laboratory (November 2011)
8. Kamal, H., Mirtaheri, S.M., Wagner, A.: Scalability of communicators and groups in mpi. In: Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing. pp. 264–275. HPDC '10, ACM (2010)
9. Kumar, S., Mamidala, A., Faraj, D., Smith, B., Blocksome, M., Cernohous, B., Miller, D., Parker, J., Ratterman, J., Heidelberger, P., Chen, D., Steinmacher-Burow, B.: Pami: A parallel active message interface for the bluegene/q supercomputer. In: Proceedings of 26th IEEE International Parallel and Distributed Processing Symposium (IPDPS). Shanghai, China (May 2012)
10. Lifflander, J., Miller, P., Venkataraman, R., Arya, A., Jones, T., Kale, L.: Mapping dense lu factorization on multicore supercomputer nodes. In: Proceedings of IEEE International Parallel and Distributed Processing Symposium 2012 (May 2012)
11. 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. EuroMPI'11 (2011)
12. Peter Kogge, E.: ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems. Tech. rep. (2009)
13. Rabenseifner, R.: Automatic MPI Counter Profiling. In: 42nd CUG Conference (2000)
14. Sack, P., Gropp, W.: A Scalable MPI_Comm_split Algorithm for Exascale Computing. In: Recent Advances in the Message Passing Interface. pp. 1–10. EuroMPI'10 (2010)
15. Skinner, D., Verdier, F., Anand, H., Carter, J., Durst, M., Gerber, R.: Parallel Scaling Characteristics of Selected NERSC User Project Codes. Tech. Rep. LBNL/PUB-904, Lawrence Berkeley National Lab (2005)
16. Träff, J.L.: Compact and efficient implementation of the mpi group operations. In: Recent advances in the message passing interface. pp. 170–178. EuroMPI'10 (2010)