Noncollective Communicator Creation in MPI

James Dinan,[1] Sriram Krishnamoorthy,[2] Pavan Balaji,[1] Jeff R. Hammond,[1]
Manojkumar Krishnan,[2] Vinod Tipparaju,[3] and Abhinav Vishnu[2]

[1] Argonne National Laboratory, Argonne, Illinois
{dinan, balaji}@mcs.anl.gov, jhammond@alcf.anl.gov
[2] Pacific Northwest National Laboratory, Richland, Washington
{sriram, manoj, abhinav.vishnu}@pnl.gov
[3] Oak Ridge National Laboratory, Oak Ridge, Tennessee
tipparajuv@ornl.gov

Abstract. MPI communicators abstract communication operations across application modules, facilitating seamless composition of different libraries. In addition, communicators provide the ability to form groups of processes and establish multiple levels of parallelism. Traditionally, communicators have been collectively created in the context of the parent communicator. The recent thrust toward systems at petascale and beyond has brought forth new application use cases, including fault tolerance and load balancing, that highlight the ability to construct an MPI communicator in the context of its new process group as a key capability. However, it has long been believed that MPI is not capable of allowing the user to form a new communicator in this way. We present a new algorithm that allows the user to create such flexible process groups using only the functionality given in the current MPI standard. We explore performance implications of this technique and demonstrate its utility for load balancing in the context of a Markov chain Monte Carlo computation. In comparison with a traditional collective approach, noncollective communicator creation enables a 30% improvement in execution time through asynchronous load balancing.

1 Introduction

MPI communicators [6] provide communication contexts that differentiate both point-to-point and collective operations. This functionality enables the programmer to isolate communication between application modules by effectively sandboxing communication in different communicators. This has enabled the development of large applications composed of independently developed modules and libraries. In addition to this primary function, communicators also provide the ability to form groups of MPI processes and perform communication, especially collective communication, within these groups. Such process groups enable the programmer to express multiple levels of parallelism within MPI applications, a capability that has been shown to be increasingly important as computing system size increases.

At the MPI implementation level, the key ingredient in a communicator is a context id. All processes participating in a communication operation identify the communicator using its context id, often an integer. The context id essentially serves as another tag, in addition to any user-provided communication tag, in matching communication

operations. As such, consensus on the context id is required in order to correctly match communication operations.

MPI supports collective creation of communicators, where all processes in the parent communicator participate in the creation of the child communicator. However, the recent push towards petascale and beyond has brought forth new application architecture idioms and programming model use cases that highlight the need for noncollective creation of communicators. For example, in applications where a small subset of processes dynamically cooperate to make progress on a work component, this subset of processes might want to create a communicator without synchronizing with the remaining processes in the system. Similarly, when a process fails, recreating the communicator should be possible without involving the failed process. However, current MPI communicator creation operations such as MPI_Comm_dup, MPI_Comm_split, and MPI_Comm_create do not allow for such flexibility.

This collective mode of creation is so widely taught and practiced that noncollective creation of communicators was considered impossible within the MPI standard. In this paper, we present a new communicator creation algorithm that constructs a communicator collectively only on the group of processes that will be members in the new communicator. This algorithm is portable and uses only functionality provided by the current MPI standard. In short, our algorithm works around the MPI API's limitation by hierarchically constructing and merging intercommunicators into intracommunicators.

We present key use cases from a variety of domains that motivate the need for communicator creation that is not collective on a parent communicator. In addition, we evaluate the overhead of this implementation as compared with the traditional collective creation directly supported in the MPI API. We evaluate the benefits of this approach to asynchronous dynamic load balancing through a Markov chain Monte Carlo benchmark kernel. Compared with a traditional collective approach to load balancing, noncollective communicator formation enables a 30% improvement in execution time.

This paper is organized as follows. In Section 2 we present the current state of MPI communicators and motivate the need for noncollective communication creation. In Section 3 we present our noncollective communicator creation algorithm. In Section 4 we present an empirical evaluation of the overhead and performance impact of noncollective communicator creation. Section 5 contains a discussion of how this functionality can be incorporated into the MPI standard to improve performance. We summarize our conclusions in Section 6.

2 Need for Noncollective Communicator Creation

The processes cooperating in a subcomputation of a program are said to form a process group. In MPI, such groups can be conveniently specified using MPI_Group objects. These objects, created using local operations, specify the participation and ordering of processes in a group. While MPI groups allow querying for membership, they are not sufficient for communication operations. Such operations require the creation of an MPI communicator, which backs the group information with one or more context ids.

The widely used interfaces for MPI communicator creation are MPI_Comm_create, MPI_Comm_dup, and MPI_Comm_split. MPI_Comm_dup

and MPI_Comm_split result in valid communicator handles on all processes in the parent group and hence are naturally collective on all member processes in the parent communicator. MPI_Comm_create, on the other hand, takes an MPI_Group object and creates a communicator on the subset of processes specified by the group. While the outcome is useful only for the processes participating in the subcommunicator, it is specified to be collective on the parent communicator. This has resulted in the common belief that MPI communicator creation requires full cooperation of all processes in the parent communicator. In the remainder of this section, we present several case studies where a communicator creation operation that is not collective over the parent communicator is required to enable a certain capability (e.g., collective communication after one or more process failures) or is helpful to improve performance.

2.1 Fault Tolerance

Several solutions have been proposed to provide fault tolerance for MPI programs. All approaches must address the reconstruction of a communicator that can be used for continued program execution. Proposed approaches include the use of explicit intercommunicators [4] and an MPI extension to introduce dynamic communicators that support grow and shrink operations [3]. The MPI standard leaves the behavior of an MPI implementation following process or network failures undefined, and several implementations allow for specific communication operations to proceed in such cases. For example, if a process has failed, point-to-point communication between remaining processes is not affected; all communication with a failed process would return an error.

Supporting collective operations after a failure has occurred is more challenging, as all communicators that contain a failed process can no longer be used. A collective operation on such a communicator can return an error. Furthermore, since all operations to create new communicators are collective, the application cannot create a new communicator that excludes the failed process, thus making collective operations unusable after a process failure has occurred.

With the algorithm we present in this work, a new communicator can be rebuilt by the application after a failure without introducing the complexity associated with intercommunicators or an extension to the MPI standard. Our approach relies on the observation that MPI_COMM_SELF is well defined on all live processes, irrespective of the state of any other communicators.

2.2 Global Arrays

Global Arrays [9] is a global address space programming model that provides a global view of multidimensional, shared arrays distributed across the memory of multiple processes. Much of GA's functionality is implemented on top of the remote memory operations provided by the Aggregate Remote Memory Copy Interface (ARMCI) [7]. Global Arrays and ARMCI were designed to be fully interoperable with MPI and employ MPI for process management, message passing, and collective operations.

Support for process groups in GA was initially built using MPI communicators. Subsequent application use cases motivated GA to support process groups that are collectively constructed only on the processes that are members of the new group. The

implementation of these alternative process groups was not backed by an MPI communicator. The lack of an MPI communicator for each process group necessitated alternative pathways for functionality in the implementation that did not rely on communicators, primarily in supporting two-sided and collective communication. This design was based on the widely held assumption that MPI cannot support the needed mode of communicator creation. While efficient and practical, this broke the interoperability between ARMCI and MPI. GA has henceforth supported both functionalities, letting the user trade MPI interoperability for increased flexibility. The work presented in this paper resolves this dichotomy.

2.3 Dynamic Load Balancing and Multilevel Parallelism

Several applications have stressed the need for flexible management of process groups. Flexible process groups have been used in mixed quantum-mechanical and molecular mechanical calculations (QM/MM) [5] that couple classical force calculations for long-range interactions with short-range quantum mechanical corrections. The work per task performing a quantum mechanical calculation can vary widely and can only be approximately estimated a priori, making static load balancing difficult. One approach [8] employed a dynamic load balancing scheme in which the each QM task specified the number of processes that form a group to execute that task. Idle processes are identified and batched into a group to execute the next available task. This approach required idle processes to form a group while other processes are actively executing other tasks.

Dynamical nucleation theory Monte Carlo (DNTMC) [10,11] simulations are used for determining molecular nucleation rate constants and chemical properties. One of the main components of these algorithms involves many parallel Markov chain walkers to accelerate the exploration of the potential energy surface of interest. The walkers, each of which is executed in parallel on a subgroup, are all periodically synchronized to collect statistics and restart information, determine convergence, and steer for the simulation. One of the major concerns of this model was the load imbalance that can occur between the individual Markov chains. The reason behind this imbalance is the variable time for individual energy evaluations, which depends on the overall molecular cluster configuration and method being used for the evaluation. An alternative method currently under development allows a group that has completed its assigned work to help another group. The two groups merge to form a larger group and accelerate the lagging Markov chain calculation. This approach requires localized creation of groups with participation from only processes contributing to the particular work of interest.

Nonequilibrium umbrella sampling (NEUS) [2] is a technique for obtaining transition rates for rare events. Its computational profile is similar to DNTMC, although load imbalance can emerge from many different sources, as the walkers evaluate multistep dynamic trajectories rather than an energy evaluation. Because of the scalability of the underlying molecular dynamics simulations and the possibility of large variation in the execution time of each trajectory (the termination criteria depend greatly on the physics), NEUS can and should dynamically adjust the number of nodes assigned to each task.

3 Noncollective Formation of MPI Communicators

As discussed in Section 2, the routines provided by MPI for communicator creation (e.g., MPI_Comm_create) are collective over an existing parent communicator. In this section, we define a new group-collective communicator creation model where communicator creation is collective over only the processes that will be members in the resulting communicator. In addition, this algorithm does not require a parent communicator that is valid for collective communication. This is useful when a parent communicator (e.g., MPI_COMM_WORLD) has become invalid for collective communication because of a failure, when all processes in the parent communicator cannot be recruited to participate in communicator creation, and for performance when the output communicator is much smaller than the parent communicator.

The group-collective communicator creation algorithm is given in Algorithm 1. This algorithm accepts as input the MPI group corresponding to the new communicator, an existing communicator that contains all ranks in $group$, and a tag that can be safely used by this operation for communication on $comm$. The algorithm is collective only on processes that are members of $group$, and $group$ must be identical on all ranks. If desired, a check for $grp\_rank =$ MPI_UNDEFINED can be used to filter out callers that are not in $group$, returning MPI_COMM_NULL on these processes. As output, a new communicator is produced where the ranks are ordered according to $group$'s ordering. The algorithm performs $\log |group|$ intercommunicator creation and merge steps to form the final intracommunicator.

The first step in this algorithm is to translate $group$'s ranks, $\{0..|group| - 1\}$, to the corresponding ranks in $comm$. In most MPI implementations, this step requires $O(|group| \cdot |comm|)$ steps except when translating to MPI_COMM_WORLD, whose translation table is cached, yielding a complexity of $O(|group|)$.

The output communicator, $comm'$, is initially assigned MPI_COMM_SELF. This communicator is then recursively merged between pairs of adjacent groups until a single communicator remains. If the current group identity is even, the group attempts to create an intercommunicator with the group to its right. This operation requires a tag that MPI can use internally to create the intercommunicator. The tag argument to the group-collective communicator creation algorithm is particularly important when multiple threads invoke this routine concurrently; the user must supply tags such that each operation can be uniquely identified. If no right neighbor group exists (i.e., $size$ is not a power of two), the group skips this round and will participate as a right neighbor in a future round. If an intercommunicator is created, it is then merged into an intracommunicator and stored in $comm'$. A high/low argument to MPI_Intercomm_merge is used to ensure that the rank ordering given in $pids$ is preserved.

4 Experimental Evaluation

We have evaluated the cost of our group-collective communicator creation method relative to the cost of the parent-collective MPI_Comm_create routine. In addition, we present a Markov chain Monte Carlo benchmark kernel to explore the performance implications of group-collective communicator creation to load balancing. Experiments

Algorithm 1. Group-collective communicator creation algorithm.

INPUT: $group$, $comm$, tag
OUTPUT: $comm'$
REQUIRE: $group$ is ordered by desired rank in $comm'$ and is identical on all callers
LET: $grp\_pids[0..|group|-1] = \mathbb{N}$ and $pids[\,]$ be arrays of length $|group|$

MPI_Comm_rank($comm$, &$rank$)
MPI_Group_rank($group$, &grp_rank), MPI_Group_size($group$, &grp_size)
MPI_Comm_dup(MPI_COMM_SELF, &$comm'$)

MPI_Comm_group($comm$, &$parent\_grp$)
MPI_Group_translate_ranks($group$, grp_size, grp_pids, $parent\_grp$, $pids$)
MPI_Group_free(&$parent\_grp$)

for ($merge\_sz \leftarrow 1$; $merge\_sz < grp\_size$; $merge\_sz \leftarrow merge\_sz \cdot 2$) **do**
  $gid \leftarrow grp\_rank / merge\_sz$, $comm\_old \leftarrow comm'$
  if $gid \mod 2 = 0$ **then**
    if $((gid+1) \cdot merge\_sz < grp\_size$ **then**
      MPI_Intercomm_create($comm'$, 0, $comm$, $pids[(gid+1) \cdot merge\_sz]$, tag, &ic)
      MPI_Intercomm_merge(ic, 0 /* LOW */, &$comm'$)
    end if
  else
    MPI_Intercomm_create($comm'$, 0, $comm$, $pids[(gid-1) \cdot merge\_sz]$, tag, &ic)
    MPI_Intercomm_merge(ic, 1 /* HIGH */, &$comm'$)
  end if
  if $comm' \neq comm\_old$ **then**
    MPI_Comm_free(&ic)
    MPI_Comm_free(&$comm\_old$)
  end if
end for

were conducted on a Blue Gene/P system using IBM MPI, which is a derivative of MPICH2. A node in this system contains a 4-core 850 MHz PowerPC 450 processor with 2 GB of memory. Racks consist of 1024 nodes and the total number of racks is 40, yielding 163,840 total processing cores. Because of a bug in the MPI implementation's intercommunicator creation routine, we have been forced to limit our experimentation to two racks, or 8,192 cores.

4.1 Group Creation Cost

In Figure 1 we present the costs of group- and parent-collective communicator creation over a range of output group sizes. All experiments in this figure were run on 8,192 cores. In the case of MPI_Comm_create, collective communication was performed across all ranks in the parent group (MPI_COMM_WORLD for this experiment) regardless of the output group size. This explains the flat cost of MPI_Comm_create relative to the output group size.

In comparison, the group-collective communicator creation must perform $\log |group|$ collective communication steps; the size of the groups involved in this col-

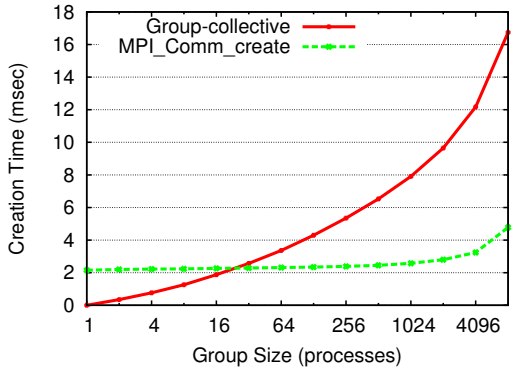

Fig. 1. Communicator creation cost for group-collective versus MPI_Comm_create.

lective communication increases exponentially at each step because of the recursive merging nature of the algorithm. For small groups, we see that this approach is significantly faster than MPI_Comm_create. The cost increases well beyond the cost of MPI_Comm_create; however, as we demonstrate in the next section, this cost can be amortized by potential benefits to the application.

4.2 MCMC Load-Balancing Example

Markov chain Monte Carlo (MCMC) simulations are typically composed of walkers that explore a state space with sequential state transitions. The Monte Carlo transition from one state to the next is tested to determine whether the state is valid; if it is not, it is rejected, and another transition attempt is made. In addition, the amount of computation involved in calculating acceptance can vary across states with respect to the input data. Because of these factors, load balancing MCMC applications is extremely challenging. Often, the work performed by a walker can be parallelized and executed on a group of processes. In our current work with the DNTMC application [11], we have developed a load-balancing solution that reassigns idle processes to active walker groups in order to accelerate that walker.

For this work, we have developed a benchmark kernel that is representative of such MCMC simulations. This benchmark creates a set of initial walker groups of size G and assigns each group a workload. The workload is composed of S work items, corresponding to S state transitions in the Markov chain; processing of each item requires $T/group\_size$ milliseconds; for simplicity, all state transitions are accepted. When a group finishes processing its S work items, it merges with the group to its right. Likewise, groups must periodically check for incoming merge requests; when one arrives, the old group is freed, and a new group is created. We have implemented this algorithm using both group- and parent-collective communicator creation. In the group-collective case, point-to-point merge requests are sent and result in a merge operation that involves only the merging processes. In the parent-collective case, all processes must perform pe-

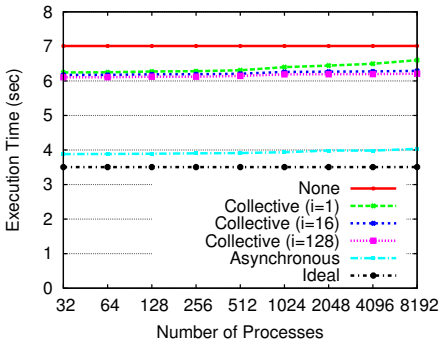

Fig. 2. Markov chain Monte Carlo benchmark weak scaling up to 8192 cores with none, asynchronous, and collective load balancing.

Ld. Bal.	i	Avg.	St. Dev.	Min	Max
None		0.00	0.00	0	0
Async.		14.38	3.54	5	26
Collect.	1	5.38	2.12	2	8
Collect.	2	5.38	2.12	2	8
Collect.	4	5.38	2.12	2	8
Collect.	8	5.38	2.12	2	8
Collect.	16	5.38	2.12	2	8
Collect.	32	5.38	2.12	2	8
Collect.	64	3.75	1.20	2	5
Collect.	128	2.62	0.48	2	3

Table 1. Average number of regrouping operations performed per process for the experiment in Figure 2 on 8,192 cores.

riodic collective exchange of load information followed by regrouping. This collective load balancing is performed every i work units.

In Figure 2 we present data for a weak scaling experiment with the MCMC benchmark kernel. In this experiment G was four processes, T was 100 ms, and S was $10 \cdot R \mod 32$, where R is the group leader's rank. This resulted in a cyclic work distribution of $0, 40, 80, 120, 160, 200, 240, 280, 0, \ldots$. In the baseline case, regrouping is disabled, and the execution time is bounded by the time required to process the longest Markov chain: $S \cdot T/G$ or $280 \cdot 100ms/4 = 7sec$. The ideal execution time is also shown; this is the calculated execution time with perfect load balancing. Because we have chosen a cyclic, triangular workload, the ideal time is half of the baseline execution time.

Collective load balancing with load balancing intervals of $i = 1, 16, 128$ steps are shown and result in a roughly 15% improvement in execution time compared with no load balancing. Asynchronous group-collective load balancing yields over a 40% improvement in execution time compared with the baseline and over a 30% improvement compared with collective load balancing. The gap between ideal and asynchronous load balancing is due to the interval at which load balancing is performed. Polling for load balancing requests is performed once after each step in the Markov chain. The time between polling operations is the step execution time, $T/group\_size$. For the cyclic work distribution with period $P = 8$, this results in an overhead of up to $(P-1) \cdot T/group\_size$ for each group.

Table 1 shows the number of regroupings that occurred for each load-balancing configuration on 8,192 cores. We can see from this data that the collective scheme results in a regular load-balancing pattern. In contrast, the asynchronous scheme takes advantage of more fine-grained load-balancing opportunities, leading to a significantly higher average number of regroupings over all processes.

5 Discussion

Intercommunication creation and merge steps perform an all-reduce operation which requires $O(\log p)$ communication steps. In the group-collective communicator creation algorithm, intercommunicator creation and merge steps are repeated $\log p$ times, yielding a time complexity of $O(\log^2 p)$. In comparison, the standard MPI communicator creation routine performs a single all-reduce step and has time complexity $O(\log p)$. The additional $\log p$ cost associated with group-collective communicator creation can be eliminated by extending MPI to provide a direct method for group-collective communicator formation.

5.1 Group-Collective Communicator Creation

The simplest method by which MPI can provide more efficient support for group-collective communicator creation is to include a group-collective communicator creation routine in the MPI standard. This would allow MPI implementors to provide a direct method for backing the provided group with a context ID, for example via a point-to-point all-reduce. Such a routine would take the form:

```
int MPIX_Group_comm_create(MPI_Comm in, MPI_Group grp, int tag, MPI_Comm *out)
```

In this routine, the input intracommunicator and tag are used to create the output intracommunicator. A communicator and tag are necessary to provide MPI with a safe conduit for noncollective communication; this is similar to the mechanism used by MPI's intercommunicator creation routines. The tag plays an important role in ensuring safety of this routine in the presence of threads. Creation of the new communicator is collective over members of the input group, and the input group must be a subset of the input communicator's group. We have included an implementation of this routine using the portable algorithm presented in this paper as an extension in version 1.4 of the MPICH2[1] MPI distribution. We are working toward an integrated implementation that uses MPICH2's internal API to eliminate the overheads identified in this algorithm.

5.2 Generalized Multicommunicators

An alternative to group-collective communicator creation would be to accomplish communicator creation with a single multicommunicator creation and merging step, eliminating a factor of $\log p$ from the creation cost. We present the concept of a multicommunicator as generalization of the current MPI communicator. In the current standard, an MPI intracommunicator is defined to contain a single MPI group. An intercommunicator is defined to contain two nonoverlapping MPI groups. A multicommunicator would be capable of containing an arbitrary number of nonoverlapping groups.

Multiple groups within a single communicator present a significant programmability challenge and significant difficulty in mapping multicommunicators to existing MPI routines. For the purpose of incorporating these generalized communicators with existing MPI functionality, the multicommunicator can be flattened into an intercommunicator. This flattening would merge all nonlocal groups into a single remote group

and produce a new intercommunicator. Thus, group-collective communicator formation could be achieved in three steps: multicommunicator creation, flattening into an intercommunicator, and merging of the intercommunicator into an intracommunicator.

6 Conclusion

We have presented an algorithm for MPI communicator creation that is collective over the output group and utilizes only functionality in the current MPI standard. This type of group-collective communicator creation is a key capability for fault tolerance, multilevel parallelism, and load balancing. We have measured the overhead of our technique and demonstrated its effectiveness on a Markov chain Monte Carlo benchmark kernel. Compared with a traditional collective approach, group-collective communicator creation yields a 30% improvement in execution time to the MCMC benchmark through improved load balance.

References

1. MPICH2 Project Website (June 2011), `http://www.mcs.anl.gov/research/projects/mpich2/`
2. Dickson, A., Maienschein-Cline, M., Tovo-Dwyer, A., Hammond, J.R., Dinner, A.R.: Flow-dependent unfolding and refolding of an RNA by nonequilibrium umbrella sampling. ArXiv e-prints (1104.5180), cond–mat.stat–mech (Apr 2011)
3. Graham, R.L., Keller, R.: Dynamic communicators in MPI. In: Proc. 16th European PVM/MPI Users' Group Meeting on Recent Advances in PVM and MPI. pp. 116–123. Springer-Verlag, Berlin, Heidelberg (2009)
4. Gropp, W.D., Lusk, E.: Fault tolerance in MPI programs. International Journal of High Performance Computer Applications 18(3), 363–372 (2004)
5. Kamiya, M., Hirata, S., Valiev, M.: Fast electron correlation methods for molecular clusters without basis set superposition errors. The Journal of Chemical Physics 128(7), 74103 (2008)
6. MPI Forum: MPI: A Message-Passing Interface Standard. Version 2.2 (September 4th 2009)
7. Nieplocha, J., Carpenter, B.: ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems. In: Parallel and Distributed Processing, Lecture Notes in Computer Science, vol. 1586, pp. 533–546. Springer Berlin / Heidelberg (1999), 10.1007/BFb0097937
8. Nieplocha, J., Krishamoorthy, S., Valiev, M., Krishnan, M., Palmer, B., Sadayappan, P.: Integrated data and task management for scientific applications. In: Computational Science ICCS 2008, Lecture Notes in Computer Science, vol. 5101, pp. 20–31. Springer Berlin / Heidelberg (2008)
9. Nieplocha, J., Palmer, B., Tipparaju, V., Krishnan, M., Trease, H., Aprà, E.: Advances, applications and performance of the global arrays shared memory programming toolkit. Int. J. High Perform. Comput. Appl. 20(2), 203–231 (2006)
10. Schenter, G.K., Kathmann, S.M., Garrett, B.C.: Dynamical nucleation theory: A new molecular approach to vapor-liquid nucleation. Physical Review Letters 82(17), 3484 (1999)
11. Windus, T.L., Kathmann, S.M., Crosby, L.D.: High performance computations using dynamical nucleation theory. Journal of Physics: Conference Series 125(1), 012017 (2008)