Scalability of Communicators and Groups in MPI

Humaira Kamal
Dept. of Computer Science
University of British Columbia
Vancouver, BC, Canada
kamal@cs.ubc.ca

Seyed M. Mirtaheri
Dept. of Computer Science
University of British Columbia
Vancouver, BC, Canada
mirtaher@cs.ubc.ca

Alan Wagner
Dept. of Computer Science
University of British Columbia
Vancouver, BC, Canada
wagner@cs.ubc.ca

ABSTRACT

As the number of cores inside compute clusters continues to grow, the scalability of MPI (Message Passing Interface) is important to ensure that programs can continue to execute on an ever-increasing number of cores. One important scalability issue for MPI is the current implementation of communicators and groups. Communicators and groups are an integral part of MPI and play an essential role in the design and use of libraries. It is challenging to create an MPI implementation to support communicators and groups to scale to the hundreds of thousands of processes that are possible in today's clusters. In this paper we present the design and evaluation of techniques to support the scalability of communicators and groups in MPI.

We have designed and implemented a fine-grain version of MPI, called FG-MPI based on MPICH2, that allows thousands of full-fledged MPI processes inside an operating system process. Using FG-MPI we can create hundreds and thousands of MPI processes, which allowed us to implement and evaluate solutions to the scalability issues associated with communicators. We describe techniques to allow for sharing of group information inside processes, which required a re-definition of the context-id and the design of scalable operations to create the communicators. A set plus permutation framework is introduced for storing group information that makes use of a variety of different representations. We also propose a set, instead of map, representation for MPI group objects that takes advantage of our framework. Performance results are given for the execution of a MPI benchmark program with upwards of 100,000 processes with communicators created for various groups of different sizes and types.

1. INTRODUCTION

As the size of clusters grows, there is increased interest in the scalability of MPI (Message Passing Interface) applications. There are now several machines on the TOP500 list [21] with more than 200,000 cores. As MPI is the dominant model used for parallel programming in high performance computing [6], MPI applications remain one of the main targets of large cluster machines and there is the need to scale them to execute with hundreds and thousands or even millions of MPI processes. Many of the scalability issues associated with MPI were highlighted in a recent paper by Balaji *et al* [4] entitled "MPI on a million processors". The focus of this paper is to address scalability issues of MPI middleware related to groups and communicators.

MPI groups and communicators are among the more challenging scalability issues associated with the MPI middleware. Information about ranks inside a group or communicator is normally stored in list form using an array [10]. The space requirement using this approach is thus linear with respect to the number of MPI processes and unsuitable for large scale systems. For one million processes the standard array for ranks to store a single group may require up to 4 Mbytes. The memory consumed by groups is especially critical on machines like the BlueGene machine with nodes having only 512 Mbytes per processor, but in general since the use of many groups and communicators in MPI programs is common it can easily become an issue on any large machine.

Starting in 2008, we began implementing a version of MPI called Fine-grain MPI (FG-MPI) [16]. FG-MPI uses non-preemptive cooperating fibers (i.e., coroutines) to support thousands of MPI processes inside an operating system (OS) process. These processes are all full-fledged MPI processes, each with their own MPI rank, that communicate using MPI middleware. FG-MPI is a fine-grain version of MPICH2 [3] and extends the process-based model of MPICH2 to multiple MPI processes inside an OS process while maintaining MPICH2's support for intra-node and inter-node communication. By decoupling MPI processes from that of an OS process, it is possible to vary the number of MPI processes to match the problem rather than the machine. It is also possible, with sufficient memory, to scale up the number of processes to arbitrarily large numbers of MPI processes. The scalability of FG-MPI makes it possible to use the system to investigate scalability issues associated with MPI and MPI middleware. Using FG-MPI we were able to explore different designs and experimentally test and evaluate the scalability of these solutions inside an MPI implementation without the need for a large cluster.

This paper makes the following contributions with regards to the scalability of groups and communications inside MPI. We investigate techniques for sharing the group map inside communicators. Although our use of this technique was inside a process, the same approach can be extended to sharing group information amongst processes inside a machine. The technique we developed required a re-definition of context-ids and a new scalable algorithm for creating communicators. We introduce a framework for the use of various compact representations of the group map inside the communicator. A key element of this framework is the decomposition of the group map into a set and permutation. The representations investigated include compression, implicit representations based on Binary Decision Diagrams (BDDs), and succinct data structures. We experimentally investigate the time/space trade-offs of these representations for the type of groups structures that arise in MPI programs. Using a benchmark program that we designed we investigate the limitations with respect to the number of communicators, the size of their group, and the overhead for messaging in using the proposed framework on an MPI world varying from thousands to over 100,000 processes. To support scalability of MPI groups we propose to define a group as a set rather than a mapping. Not only does this provide an intuitive view of groups but it can lead to much smaller representations for groups and takes advantage of the set and permutation framework we introduce. We describe the small changes to MPI needed to support this view of groups. These changes allow the use of sets while maintaining the ability to map MPI processes as needed.

In Section 2 we give an overview of FG-MPI. In Section 3 we discuss the space requirements for communicators and groups. Section 4 describes the changes necessary in FG-MPI to support the sharing of the group information for MPI communicators among fine-grain MPI processes. Section 5 introduces our set and permutation framework used to minimize the amount of memory needed to store the group mapping. In Section 6 we compare the representations used by the framework in terms of space and time and discuss strategies with different space/time trade-offs that emerge from our experiments. Section 7 describes the definition of a group as a set rather than a mapping. In Section 8 we evaluate the system using an MPI benchmark and present the space and time results of creating different sizes and numbers of communicators and messaging among its group members. Conclusions appear in Section 9.

2. FINE-GRAIN MPI

FG-MPI is a fine-grain version of MPICH2 to support multiple MPI processes inside each OS process [16]. As shown in Figure 1, MPI processes may reside in the same OS process or spread across several OS processes on the same machine or distributed across different machines. To avoid any ambiguity with the term *"process"*, we refer to an MPI process as a *"proclet"* and reserve the term *process* for an operating system's process. Proclets inside a process non-preemptively yield to a scheduler that schedules the next runnable proclet for execution. The proclet scheduler is derived from Capriccio [22], a user-level thread library, based on a very efficient coroutine library [20] with very low context switching overhead.

Proclets are full-fledged MPI processes each with its own unique rank in the MPI environment and communicates using the standard MPI message passing routines. If a proclet blocks on a communication operation during a call to the MPI middleware, it yields to another proclet through the scheduler. Within an OS process, the proclets share middleware communication structures such as the unexpected message queue, the posted receive queue as well as the communication channel endpoints for sending/receiving messages to other proclets outside of its own address space. As a result, a communication call by one proclet will progress messages for other proclets sharing its address space, which ensures we perform as much communication as possible for every call to the middleware. Proclets use the Nemesis [8] communication subsystem for sending/receiving messages to proclets in other address spaces on the same machine or different machines. Nemesis uses efficient low-latency, lock-free shared memory queues for communication between processes on the same machine, and provides multi-network support for communication between machines.

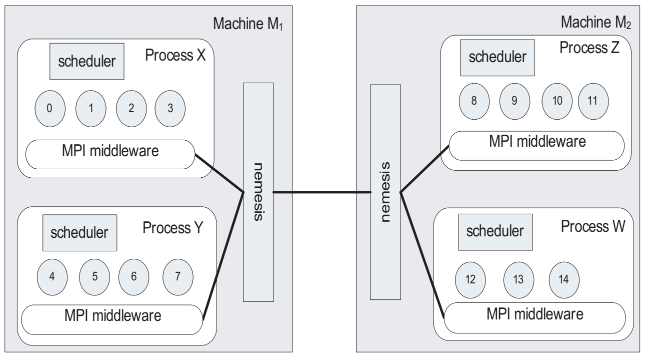

Figure 1: An example of an FG-MPI system with two machines, two processes per machine, and 15 proclets. Proclets use the Nemesis communication subsystem for communication between processes on the same machine or different machines.

FG-MPI extends the hierarchical communication structure of MPI to another level where now in addition to communication between proclets on different machines and communication between proclets in different processes on the same machine, there is also communication between proclets inside the same process (see Figure 1). We call proclets inside the same process "collocated proclets". The hierarchical structure of an MPI execution can be characterized in terms of C, the maximum number of proclets (coroutines) per process in the system, P, the maximum number of processes per machine, and M, the number of machines. The notation $[C, P, M]$ is used to classify different sizes of systems for the purpose of discussing their scalability. Notice that C and P are defined as the maximum proclets per process and process per machine to give upper bounds on time and space with respect to the total number of proclets. In general the number of proclets per process or processes per machine can vary as shown in Figure 1. Even though all processes do not have 4 proclets, according to the definition it is a $[4, 2, 2]$ system. The standard "one MPI process per process" model corresponds to a $[1, P, M]$ system.

This notation allows us to discuss scalability in terms of the communication hierarchy and different trade-offs in space and communication time. Given $N = C \times P \times M$ proclets in a $[C, P, M]$ system, one goal was to reduce the non-scalable $\mathcal{O}(N^2)$, space and communication time to $\mathcal{O}(CP^2M^2)$ Although this may not be directly applicable to a $[1, P, M]$ system, similar techniques used to reduce C^2 to C can likewise be used for P. In our case C denotes coroutines inside processes, but more generally the notation may be useful for representing systems with threads or multiple connection endpoints [11].

3. COMMUNICATORS AND GROUPS

Communicators and groups are an essential part of MPI and exist to support the development of higher level libraries [12]. A communicator is an opaque object and every communication routine in MPI takes a communicator as a parameter. Communicators divide the communication into disjoint communication contexts such that a message sent in one context can only be received by a routine with a communicator in the same context. Every proclet inside a communicator is assigned a rank from 0 to the group size minus one. In particular, since all new communicators are derived from the pre-defined communicator `MPI_COMM_WORLD`, the group of all proclets, every proclet in a system with N proclets is assigned a "world" rank from 0 to $N-1$.

In order to route messages to their destination a communicator needs to know the location of the destination proclet. In a $[1, P, M]$ system the endpoint of a communication is a process and, as done in MPICH2, one can store the information about the destination as an array of pointers mapping group rank to a communication endpoint object. In FG-MPI, since the communication endpoints are no longer processes but proclets inside processes, we use a proclet's world rank to uniquely identify the destination of a message. When a message is sent in FG-MPI we add the world rank of the destination proclet and the Nemesis endpoint associated with the process to the message. Using the endpoint object Nemesis routes the message to the correct process and the middleware inside the process uses the world rank as part of the MPI matching criteria to deliver the message to the correct receive buffer. Most implementations of MPI store the mapping from group rank to destination proclet as an array for every destination, thus a group of size $\mathcal{O}(N)$ requires $\mathcal{O}(N^2)$ space in total. Given that in general MPI programs may contain many groups of various sizes the amount of space consumed by group maps becomes prohibitive.

The space consumed by group maps becomes all the more challenging as one attempts to scale up the number of MPI processes to one million. For example, consider a $[1000, 100, 10]$ system with one million proclets. Assuming that we can optimally encode each destination using $\lceil \lg(10^6) \rceil = 20$ bits, a new communicator which re-maps `MPI_COMM_WORLD` requires 2.5 MBytes of storage per proclet, 2.5 GBytes per process and 250 GBytes per machine and 2.5 TBytes in total! The techniques in this paper reduce the memory requirements for this example to 2.5 GBytes in total, and possibly significantly less depending on the mapping.

It is reasonable to expect that the size of systems will continue to grow and a $[1, 8, 200000]$ system will soon exist, if it does not already. This was a motivation for the "million process MPI" paper [4] and systems with more than 100,000 cores already exist [21]. At this point, and even on many smaller systems, the memory saving techniques will be essential in order to use MPI and to take advantage of the benefits of communicators. Although we do not have access to a system of this scale, we are able to implement and evaluate these techniques inside FG-MPI for $[C, P, M]$ systems of this scale.

The first step to reduce the space requirements for communicators is to share the group map among collocated proclets. In a $[C, P, M]$ system with $N = C \times P \times M$ proclets, this allows us to reduce the space requirements from $\mathcal{O}(N^2)$ to $\mathcal{O}(N \times P \times M)$. In Section 4, we discuss the issues and optimizations involved in sharing of data structures among proclets. The second strategy, which benefits all $[C, P, M]$ systems, is to reduce the N factor. In Section 5 memory reduction techniques using different storage representations are discussed.

4. SHARING THE GROUP MAP

The two main routines for creating intracommunicators[1] in MPI are `MPI_Comm_split` and `MPI_Comm_create`. Communicators are created with respect to an outer communicator, where the new communicator or communicators is a subset of the proclets of the outer communicator. These routines are collective operations over the proclets in the outer communicator. The routines return with a local handle to the new communicator, which contains several fields (a) a pointer to the proclet group map, and (b) a context identifier. The group map identifies the members of the group associated with the communicator. The context identifier (context-id) is a fixed sized field inside the communicator that must match on all communications using the communicator. As long as the context-id of a communicator for a proclet is unique, the proclet cannot mistakenly receive a message sent to it using a different communicator. In this way libraries and the MPI middleware can use context-ids as a scoping mechanism.

To enable sharing the group map structure among the collocated proclets belonging to the new communicator, a hash table is used to coordinate the allocation of memory for the structure. Consider Figure 2 showing two processes X and

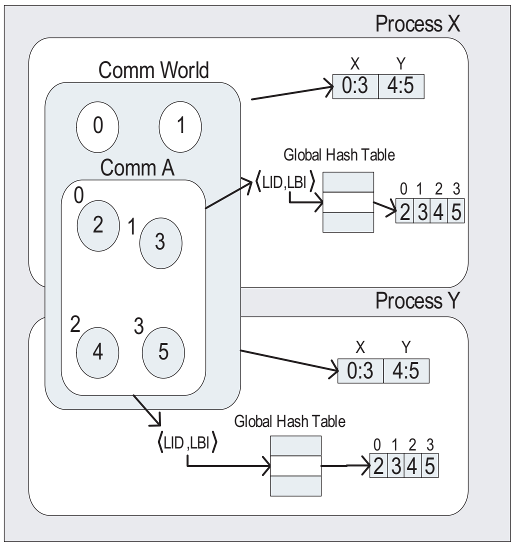

Figure 2: The sharing of the proclet group inside communicator A with 6 proclets inside two processes X and Y of one machine M.

Y inside one machine where proclets $\{2, 3, 4, 5\}$ are creating communicator A. One of the proclets inside each process creates the group map structure and stores a pointer to the structure inside the global hash table. Subsequently, the other collocated proclets belonging to the same communicator use this hash table to lookup the group map pointer

[1]At present we consider only intra and not inter communicators.

and cache the pointer inside its own communicator structure during the communicator creation operation. For sending messages, a proclet accesses the group map to find the world rank of the destination to put in the message envelope and then finds the process endpoint in the world rank table to route the message to the appropriate process.

An important precondition to the use of the hash table is that all collocated proclets belonging to the same new communicator need to have the same key to lookup the shared group map pointer in the global hash table. Furthermore, these keys need to be different for different communicators. Given that a context-id has to be created as part of the communicator and the properties of a context-id is similar to that of a key, we chose to generate a context-id that can serve both purposes.

During a communicator create routine, the context-id for a new communicator A is constructed from a combination of a leader identifier (LID) and leader bit index (LBI) denoted by $\langle LID, LBI\rangle$ where

- LID is the world rank of a representative chosen from the group underlying A, and
- LBI is an integer, uniquely chosen for every LID.

For LBI, every proclet maintains a bitmap of size 2^k where a one in location i of the bitmap signifies the use of that position, the bit is zero otherwise. For a given LID, LBI is the index of a free position in the bitmap which is then set to one. The context-id is constructed by concatenating the L bits needed to represent a world rank and k bits needed to represent LBI. In our implementation we set $k = 6$ and $L = 21$, which allows for a world size of 2^{21} (millions of proclets) where each proclet can be chosen leader for as many as 64 communicators[2]. Context-id $\langle LID, LBI\rangle$ is globally unique since LID and LBI are equal only when they belong to the same communicator. Since $\langle LID, LBI\rangle$ is globally unique it is also process unique and therefore can be used as a key for sharing the group mapping for the communicator.

The pseudo-code for `MPI_Comm_split` is given in Procedure 1. The routine uses a leader-based approach where one proclet (the root) in the outer communicator gathers and then distributes the necessary information to all the other proclets. In line 4 the color and key information is gathered to the root. We also gather the bitmap information into B, which is used later in Line 10. The gathered data, together with rank of the proclets stored in S, is sorted by color, key and rank. Proclets belonging to the same color class belong to the same new communicator. In lines 4 to 16, the root computes the context-id. Although any bitmap with an unset bit can be chosen, we balance the choice of leader by finding the one that has been chosen the least number of times. We use the CID value to also define a local leader for each group. The root uses the world rank array to determine when proclets are collocated and for each group, and for each process it choses a local leader. Finally, in lines 17 to 25, we distribute the context-id and group information to all the proclets. All proclets receive the context-id, but since the group information is shared, we distribute the group map only to the local leaders as identified in the context-id information. The remaining proclets yield and are not rescheduled until after the local leader has created the group map. For `comm` of size N, it takes $\mathcal{O}(N \lg N)$ time to sort the data at the root, $\mathcal{O}(N)$ communications, and $\mathcal{O}(N)$ space.

[2]The reason for the choice of 21 and 6 bits is that MPICH2 uses 5 bits in the context-id for other purposes [2], and 21 bits and 6 bits allows for millions of proclets with a reasonable sized number of leader bits, while still fitting into a 32 bit word.

Procedure 1 `MPI_Comm_split(comm,color,key,ncomm)`

```
Let: Let $N$ be the size of comm and rank 0 the root.
 1: if root then
 2:    // Arrays to store Colors, Keys and Bitmaps
 3:    Allocate $C[N]$, $K[N]$, and $B[N]$
 4: MPI_Gather($[C, K, B], root$)
 5: if root then
 6:    Allocate $S[N]$ // initially S[i]=i
 7:    Sort $S, C, K, B$ with respect to $C$, $K$ and rank.
 8:    for each color class $C_i$ do
 9:       Find $k$ in $B$ with the fewest number of bits set
10:       $LBI \leftarrow$ first unset bit in $B[k]$
11:       Allocate $CID[N]$ // info for context-id
12:       for each $j$ in the color class $C_i$ do
13:          if $S[k]$ is the first collocated proclet for group
             then
14:             $CID[j] \leftarrow \{S[k], LBI, LEADER = yes\}$
15:          else
16:             $CID[j] \leftarrow \{S[k], LBI, LEADER = no\}$
17: MPI_Scatter({$CID$,group-rank,group-size,local-
    leader},$root$)
18: Create $\langle LID, LBI\rangle$ from CID values
19: if root then
20:    for each color class $C_i$ do
21:       MPI_Send(group membership info to local leaders)

22: else if local leader of a group for a process then
23:    MPI_Recv(group membership info from root)
24: else
25:    yield to wait for local leader
26: Use $\langle LID, LBI\rangle$ to create or get pointer to group map
```

The pseudo-code for `MPI_Comm_create` is given in Procedure 2. The main difference between the routines is that in `MPI_Comm_create` each proclet knows the group information. As well, since MPI 2.2, the group parameter to `MPI_Comm_create` can differ which can result in multiple

Procedure 2 `MPI_Comm_create(comm,group,ncomm)`

```
Require: Let $G$ be group map from parameter group and
    $N$ be the size of comm.
 1: if I am rank 0 of $G$ then
 2:    Allocate $B[|G|]$ where $B[0] \leftarrow$ bitmap of rank 0
 3:    for each $i = 1$ to $|G| - 1$ do
 4:       MPI_Recv(B[i], from rank $i$ in $G$)
 5: else
 6:    MPI_Send(bitmap, to rank 0 in $G$)
 7: if I am rank 0 of $G$ then
 8:    Find $k$ in $B$ with the fewest number of bits set
 9:    $LBI \leftarrow$ first unset bit in $B[k]$, $LID \leftarrow k$
10:    for each $i = 1$ to $|G| - 1$ do
11:       MPI_Send({$\langle LID, LBI\rangle$, group rank}, rank $i$ in $G$)
12: else
13:    MPI_Recv({$\langle LID, LBI\rangle$, group rank}, rank 0 in $G$)
14: Use $\langle LID, LBI\rangle$ to create or get pointer to group map.
```

communicators. Because proclets already have the group information, the only information missing is the context-id. In lines 1 to 6 rank 0 of the group gathers the bitmap information from the rest of the group members. Rank 0 computes $\langle LID, LBI \rangle$, and in lines 7 to 13 sends the context-id information back to the group members. Because there can be more than one group, we cannot use collectives, and instead use explicit sends and receives to the group elements. For multiple groups, all of the groups can be actively sending and receiving messages at the same time since the groups are non-overlapping. In line 14 the proclet with smallest group rank uses it to create the group map and the remaining proclets will simply cache a pointer to their group map.

4.1 Discussion

The context-id as currently used in MPICH2 cannot be used as a key to share the group map for FG-MPI. In MPICH2 the context-id is directly generated from a bitmap of size 2^k that is allocated for each proclet at start-up with all bits initially set to one[3]. A new context-id is generated by performing a bitwise AND in `MPI_Allreduce`, where all proclets in the outer communicator of the operation obtain the result of the bitwise AND operation where a one bit in the i^{th} position is an available context-id. As long as all proclets agree on which bit to choose, the lowest set bit [13], then they will all agree on the same k-bit value to use to generate the context-id. Notice, that in the case of `MPI_Comm_split` every new communicator receives the same context-id. This is sufficient to ensure "safe" communication because the underlying groups for the new communicators are all disjoint, thus the context-id is guaranteed to be unique for all the communicators of which it is a member. But, because all of the new communicators have the same context-id, it cannot be used as a key to share the group map for each separate communicator. For FG-MPI we require that the context-id for communicators with members inside the same process be unique.

The definition of the context-id also affects the number of the communicators that can be created. In the case of MPICH2, a proclet can be a member of at most 2^k communicators, independent of the size of the world. As well, the limit on the number of context-ids can only be increased by increasing k, which essentially doubles the size of the bitmap and the amount of data for `MPI_Allreduce`. The context-id we have defined can be created as long as there is *one* proclet in the new communicator that has not been a leader 2^k times. Thus, the opportunity to create new context-ids grows with the size of the group which in turn depends on the size of the world. The difference is, in our case, the number of context-ids scales with the size of `MPI_COMM_WORLD` and k does not need to scale.

The other major difference in communicator creation is the use of leader-based communication where one proclet (the root) in the outer communicator gathers and then distributes the necessary information to all the other proclets. This avoids the "ALL" type collectives, like `Allgather`, which in the worst case consumes $\mathcal{O}(N^2)$ for an outer communicator G of size N. An advantage of MPICH2's context-id is that it can be generated with a single `MPI_Allreduce` operation, and in the case of `MPI_Comm_create` no further communication is needed. However, in the case of `MPI_Comm_split`, this is possible only after an earlier `MPI_Allgather` operation which temporarily requires $\mathcal{O}(N^2)$ space. This is a good example of the trade-offs between communication time and space where originally a $[C, P, M]$ system uses $\mathcal{O}(C^2P^2M^2)$ communications and $\mathcal{O}(C^2P^2M^2)$ space now takes $\mathcal{O}(CPM)$ communication and $\mathcal{O}(CP^2M^2)$ space. The same techniques can be used to extend group sharing to processes, but it would incur some additional shared memory synchronization overheads.

[3]MPICH2 uses k=11, allowing the context ID to fit into 16 bits. The size of the bitmap is 2 Kbits.

The use of leaders is a key part of the design of FG-MPI. Not only does it avoid the scalability problems for creating communicators, leaders played an important role in defining location-aware collectives that can take advantage of the hierarchical communication structure in a $[C, P, M]$ system. In the next section we discuss different strategies for compact representations of the map structure that were used to reduce the overall memory consumption.

5. MAPPING MEMORY REDUCTION

The second technique to reduce the memory needed to store communicators is to use an efficient and compact representation to store the proclet group map. Although shown in Figure 2 as an array, we extended MPICH2 to use the framework shown in Figure 3 to store the proclet group map. The storage scheme depends on the properties of the map.

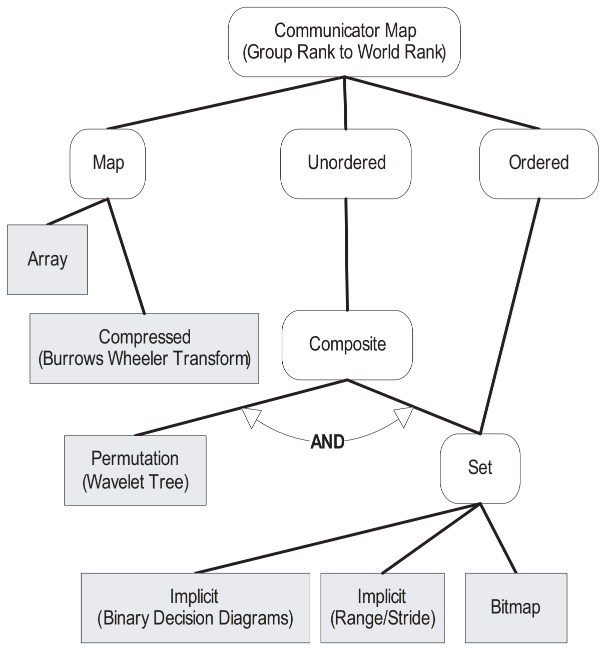

Figure 3: Set and Permutation Framework: an AND-OR diagram of the representations used for storing the mapping from group rank to world rank. Rectangles with rounded edges are properties and the boxes are the data structures. The branches are OR unless specifically marked as AND.

The idea is to provide an open framework with a portfolio of representations from which to implement strategies with respect to different space and time trade-offs. The framework in Figure 3 distinguishes between simple map representations and between ordered and unordered groups. We separate out map representations because, as described in Section 6, arrays give the best time and using compression is often best with respect to space. However there are more interesting trade-offs when one distinguishes between groups using a default ordering and those groups for which the user has defined a particular non-default ordering. For proclet i, let $gid_A(i)$ denote the group rank of i with respect to communicator A, and let $wid(i)$ be the world rank of i. For communicator A, The default ordering, or *ordered map*, is the map where for all i and j in A

$$gid_A(i) < gid_A(j) \texttt{ if and only if } wid(i) < wid(j).$$

Formally in the literature, the default map is called an *ordered set* (i.e., dictionary) and maps not satisfying the previous property are called *unordered* [5]. In MPI this is the default map in the sense that for many applications it suffices to specify the group and it is not necessary to order the mapping in one specific way. When the mapping is an ordered set then given $gid(i)$, we need to return the $gid(i)^{\text{th}}$ smallest $wid(i)$ in the set. On ordered sets, this operation is called $SELECT$ and its inverse is called $RANK$.

Further details about the representations in Figure 3 are given in following sections: range/stride representation in Section 5.1, Binary Decision Diagrams in Section 5.2, bitmaps in Section 5.3, Burrows-Wheeler Transform in Section 5.4, and Wavelet Tree with Runs in Section 5.5. Later in Section 6 we describe strategies build on top of the framework that attempt to minimize space, minimize time, and one that judiciously trades off time and space. Both the framework and strategies are designed to allow the framework to be extended with more representations and new strategies.

5.1 Range and Stride Representations

One simple type of group consists of a simple range or stride. A simple range and stride can be represented as the following function

$$wid(i) = gid_A(i) \times S_1 + S_2$$

mapping $gid_A(i)$, ranging from 0 to group size minus one, to its $wid(i)$ value. In the formula S_1 is the stride and S_2 is the offset from zero. Storing a communicator using a range representation is ideal since it only requires constant space and constant time to perform `SELECT`.

A Range/Stride representation can arise from the use of `MPI_Comm_create` where the group parameter is the result of `MPI_Group_range_incl` and `MPI_Group_range_excl` group management operations. In general, these operations or sequences of them result in a collection of ranges and not one as required for our range representation.

5.2 Binary Decision Diagrams

For many explicit representations the space requirements can be large even when we are able to store them close to their information theoretic lower bound. Implicit representations on the other hand can take a advantage of specific patterns in the data to obtain very compact representations. Sets can be represented as a boolean function through a Binary Decision Diagram (*BDD*) [7]. BDDs are boolean function representations that are widely used in practice for circuit design and scale to millions of nodes. We extracted a subset of the BDD routines from the BuDDy package by Jørn Lind-Nielsen [1] and integrated the data structures and operations into our framework.

We construct our group as a boolean function that returns true (1) if the input is a member of the group and false (0) otherwise. The input to create this function is the set of binary representations of $wid(i)$ for all the members in the group, where each bit is a binary variable. Since in our case the group map is static, once the BDD is created we extract the paths that lead to true and discard the graph structure. Paths can be represented as a sequence of nodes where each node is either 0, 1, or X (don't care), thus requiring 2 bits per node. Paths are of length $\lceil \lg(N) \rceil$ and therefore p paths require at most $2p \lceil \lg(N) \rceil$ bits.

The algorithm for $SELECT$ finds the $gid_A(i)^{th}$ smallest fully specified path from the ordered collection of paths extracted from the BDD. The algorithm proceeds iteratively from the first node in the set of all paths down to the last node (i.e. the $(\lceil \lg(N) \rceil - 1)^{th}$ node). On each step, it uses the current rank (initially set to $gid_A(i)$) to determine whether or not the $wid(i)$ is in a path with bit 0 or 1 (don't cares are split half and half). Depending on the value of $gid_A(i)$, we discard the paths with a 0 or those with 1, appropriately adjust the value of $gid_A(i)$, and repeat the operation for the next node on the remaining paths [4]. The algorithm returns the $gid_A(i)^{th}$ $wid(i)$ in at most $\mathcal{O}(p \times \lg N)$ steps. The number of paths left at each step of the algorithm depends on the size of the BDD as well as on the number of X's encountered, which can result in costly $SELECT$ times. We avoid this situation by limiting the working memory size for the BDD and the total count of the number of paths, and creating the BDD only when the limits have not been exceeded.

Figure 4 shows a range-of-ranges example for using a BDD to represent a 2D sub-mesh of a 3D mesh with over one million nodes. The BDD shown in Figure 4-(b) is all that is required to store the sub-mesh. We have assumed a standard ordering where $X[j]$ denotes bit j in the binary representation of the $wid(i)$. The sub-mesh is defined by a range

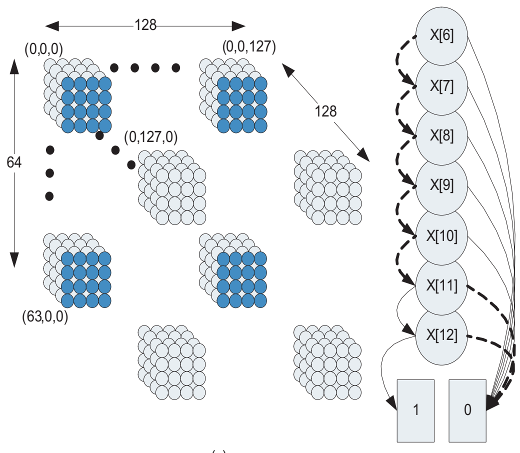

Figure 4: (a) A 2^{20} node 3D mesh with a 2^{13} 2D sub-mesh defined by varying dimensions X and Y, for a fixed value of Z. (b) The BDD representation of the sub-mesh.

or ranges and as Figure 4-(b) shows the resulting BDD is small. Only 5 bytes are needed to store the paths for this sub-mesh of size 2^{13} in a mesh of size 2^{20}.

[4] Due to space limitations the full algorithm has been omitted.

5.3 Bitmap Representation

The bitmap representation uses a bit array to store the ordered group map, where we set the $wid(i)^{\text{th}}$ bit to one whenever $wid(i)$ is in the group. The bit-array requires the size of `MPI_COMM_WORLD` bits, independent of the size of the group. One simple heuristic is to store the bit-array starting from the first bit set up to the last bit set. Thus $wid(largest) - wid(smallest) - 1$ bits are needed to store the array.

A disadvantage of using bitmaps is that $SELECT$ is a linear time operation. To alleviate this problem we store the cumulative total number of set bits in blocks of 1,000 items in an auxiliary structure. A binary search is performed on the auxiliary structure to determine the block containing the $gid(i)^{\text{th}}$ set bit. Within a block, a linear search is used to find the bit, when possible, we use the `popcnt` assembly instruction to count the number of set bits in a 32 or 64 bit register[5]. The auxiliary structure adds only $\lceil(\texttt{first} - \texttt{last} - 1)/1,000\rceil$ integers and and its size is small in comparison to the bitmap. Although there are asymptotically faster structures, for set sizes of one million the simple above-mentioned scheme performs well and uses small additional space. There is a rich literature on compressing bitmaps and there are a variety of bitmap representations that could be used to further improve the performance [19]. The framework can be easily extended to include other encoding schemes.

5.4 Burrows-Wheeler Transformation

One alternative for reducing the amount of storage for cases such as large unordered maps is to compress the group array of $wid(i)$s. Our framework uses functions from the *bzip2* library that uses *Burrows-Wheeler transform* (BWT) [9] and *Huffman coding* [18]. Even though originally BWT was designed to compress text, certain bit patterns in the numerical data such as ranges and sequences increase the chance of recurring characters after which BWT based compression techniques perform reasonably well. The block compression nature of BWT lets us perform $SELECT$ efficiently. Blocks of 1,000 $wid(i)$s are compressed and $SELECT$ is implemented by simply uncompressing the $\lceil gid_A(i)/1,000\rceil^{th}$ block. The $SELECT$ operation takes the same amount of time irrespective of group size, world size and the location of the item, however, the operation is relatively slow because the entire block needs to be decompressed. As an optimization we cache the last decompressed block, which greatly reduces the time for selecting the same $wid_A(i)$ or when iterating over all members of a group.

Further enhancements to this compression technique is possible using recent improvements in the computation of BWT by Okanohara and Sadakane [14], notably an approach with linear time complexity [17].

5.5 Wavelet Tree Representation

To store unordered group maps we take the set representation and add a permutation(π) of the default ordering of the set to the required ordering of the $gid_A(i)$. The amount of storage needed for the permutation depends on the group size not on the world size. We use a succinct data structure called *Wavelet Tree on Runs* [5] that compresses the permutation and provides fast computation of $\pi(i)$ and $\pi^{-1}(i)$.

There are a wide range of algorithms available to encode permutations. Jérémy and Gonzalo [5] have studied several techniques to compress a permutation and suggested three solutions: *Wavelet Tree on Runs (WTR)*, *Stricter Runs* and *Shuffled Sequences*. WTR which is based on the reverse of the *merge sort* algorithm, takes advantage of ordered subsequences in π performs $\pi(i)$ and $\pi^{-1}(i)$ in $\Omega(1 + \log \rho)$ time using $n\lceil \lg \rho\rceil(1 + o(1)) + \Omega(\lg(n))$ bits, where ρ represents number of *runs* in the permutation [5]. A *run* is a largest monotonically increasing subsection of the permutation. The algorithm performs a Hu-Tucker algorithm [15] on runs and encodes the result in a binary trie.

WTR is optimized to compress permutations with less number of runs. This property is particularly interesting since there are few MPI group manipulation functions that increase the number of runs. Since groups are derived from `MPI_COMM_WORLD`, which is monotonically increasing, and most of the group management functions keep the ordering of the originating groups, permutations with small number of runs are more probable. While the *Stricter Runs* and *Shuffled Sequences* techniques support more sophisticated structures they are not as probable as WTR for a typical MPI application and their potential memory overhead render them less useful than WTR for our application. Before creating the WTR representation we first calculate the number of runs and only use WTR when the number of runs is less than a preset limit to avoid an explosion in the working memory.

6. IMPLEMENTATION AND EVALUATION

The framework is implemented as an independent module that hides all the internals of the different representations. The API includes routines to create and destroy the group map along with $SELECT$ operation. We assume that the array group representation is used as initial input to the module. There is also a routine for configuring different strategies that implement various time/space trade-offs. The three strategies we have defined attempt to optimize space, time, and a hybrid one that is a balance between the two. Each OS process could choose a strategy based on the characteristics of the machine and the same group map could use different strategies in different processes.

In Section 6.1 and 6.2 we give the results for experiments comparing the time and space requirements for various ordered and unordered group maps and discuss the strategies that emerge from the experiments. The experiments were conducted on a wide variety of different groups including those used to exercise a particular representation as well as the type of groups that arise from the group management operations in MPI. For illustrative purposes we focus on groups derived from a MPI world of size 2^{20}, which exceeds one million proclets. All experiments were conducted on an Intel® Core™ i7 2.67 GHz workstation with 6 Gbytes of memory using a standalone implementation of the framework module in FG-MPI. From our experiments we have selected ones that best illustrate the rationale for having the different representations and their influence on our choice of hybrid strategy.

6.1 Ordered Groups

Table 1 compares the space and time requirements for several different ordered groups. For each group and represen-

[5]The `popcnt` instruction is part of Intel's Streaming SIMD extensions version SSE4.2 and SSE4a implemented in the i7 and it is part of AMD's Advanced Bit Manipulation instructions in the "Barcelona" processor.

Group : Size	Set Type	Total (KB)	*bpg* (bits)	S-All (msecs)	S-One (μs)
1: 999,999	`MPI_COMM_WORLD`, one rank randomly removed				
	ARRAY	3906	32	8.58	0.009
T1	BWT	857	7	154.2	139.790
T2 ,S, ST	BDD	0.55	0.004	293.2	0.289
	BM	122	1	488.4	0.540
	BMp	–	–	100.8	0.169
2: 500,000	10,0000 ordered ranges of size 50				
	ARRAY	1953	32	4.29	0.007
T1	BWT	526	9	82.2	150.562
S	BDD	110	1.8	1588.0	45.829
T2, ST	BM	122	2	173.9	0.410
	BMp	–	–	51.0	0.167
3: 749,999	{all odd ranks} $\cup$ {0 . . . 499, 999}				
	ARRAY	2930	32	6.43	0.009
T1	BWT	597	6.5	109.0	131.647
S, T2, ST	BDD	0.41	0.004	181.9	0.243
	BM	122	1.3	320.4	0.491
	BMp	–	–	76.9	0.164
4: 463,040	Upper half with prime numbers removed				
	ARRAY	1809	32	3.98	0.009
T1	BWT	534	9.5	80.408	158.331
	BDD	871	15	148.4	337.964
S, T2, ST	BM	61	1.08	247.0	0.580
	BMp	–	–	46.0	0.160
5: 10,000	4-D sub-grid of 6-D grid, [*,10,*,10,*,*]				
	ARRAY	39	32	0.10	0.010
	BWT	14	12	2.5	149.405
S, ST	BDD	4	3	10.2	1.032
T1-2	BM	111	91	1.8	0.229
	BMp	–	–	1.0	0.150
6: 10,000	4-D sub-grid of 6-D grid, [10,10,*,*,*,*]				
S, ST	R/S	0.13	0.10	0.16	0.016
	ARRAY	39	32	0.10	0.010
T1	BWT	13	10	1.4	126.963
T2	BDD	0.30	0.24	1.5	0.156
	BM	1.3	1.1	4.7	0.487
	BMp	–	–	0.9	0.113
7: 10,000	$wid(i) = 2 \times gid_A(i) + 600000$ where $x <> 8849$				
	ARRAY	39	32	0.09	0.009
	BWT	9.43	7.72	2.03	138.906
S, T1-2, ST	BDD	0.38	0.31	1.83	0.190
	BM	5.00	4.10	1.84	0.219
	BMp	–	–	0.99	0.130
8: 1,500	Randomly choose 1,500 items from [0,999999]				
ST	ARRAY	5.99	32	0.01	0.009
T1-2, S	BWT	5.78	31.55	0.03	0.015
	BDD	12	65	14.5	759.165
	BM	122	667	0.20	0.178
	BMp	–	–	0.1988	0.1656

Table 1: Space and time costs for different ordered groups using BDD, BWT, Bitmaps (BM), Bitmaps with `popcnt` (BMp), and Range/Stride (R/S). Annotations on the left show row with best Space (S), best SELECT time for All items (T1) and best SELECT time for One item (T2) for {BWT, BDD, BM} and choice for the hybrid strategy (ST).

tation we report space in terms of the total amount of space and, to allow comparison between different group sizes, the average number of bits per group member (*bpg*). For time, we compare the different representations with respect to two types of access patterns. First, the total time required to iterate though the entire group map, Select-All (S-All), which is the overhead for a collective operation, assuming a simple implementation on top of the point-to-point send/receive. Second, the select time for one member, Select-One (S-One), which is the overhead for a send operation. The S-One time is computed as the average over a 1,000 randomly selected group destinations. In reporting the time for bitmaps we also give the time on architectures supporting the `popcnt` operation. The annotations (see Table 1) show the best representation for space, time for S-All and S-One among the non-array representations and the representation chosen by the hybrid strategy.

As Table 1 shows, for time, an array is always the fastest. A simple range, because of cache hits, can potentially do better but even with 10,000 elements the array access is faster. A good trade-off when `popcnt` is available is BMp. It significantly speeds up BM and, except for R/S or when the array is smaller (e.g., Group 8), does well in terms of space.

For space, the best representation varies and a *bpg* value or estimate can be calculated and used to choose the best representation. The strategy chooses between (a) *bgp* value returned by BDD create, if successful, (b) *bpg* for a group A of size M, calculated by

$$bpg = (\max\{wid(i)\} - \min\{wid(i)\} - 1)/M \qquad (1)$$

where $\max\{wid(i)\}$ and $\min\{wid(i)\}$ are the largest and smallest $wid(i)$, respectively, in group A, and (c) *bpg* estimate for BWT calculated as lgN, for a world of size N. As Table 1 shows the one giving the best space varies between BDD $\{1, 2, 3, 5, 7\}$, BM $\{4\}$, R/S $\{6\}$, and BWT $\{8\}$. Although BDD appears to do well it depends on the specific patterns and it is easy to construct groups such as Group 4 and 8 where it does poorly. Notice that BM does well for dense groups, Group 4, but not for sparse groups such as Group 8 where our heuristic fails to reduce the *bpg*. BDDs also give the most variation for S-All and S-One times. S-One time tends to increase with increased BDD size (higher *bpg*) while S-All time increases with the number of X's in the path structure.

The hybrid strategy (ST) chooses BDD $\{1, 3, 5, 7\}$, BM $\{2, 4\}$, R/S $\{6\}$ and array $\{8\}$. This is similar to the space optimized strategy, except we avoid BWT and choose BM and an array over a BDD in some cases. Since the select time for BDD is so dependent on the path structure, a BDD is only chosen when *bpg* is small (<5) and significantly smaller than *bpg* for BM. In the case of groups 4 and 8, BDD create would fail (we removed the limits for these tests).

6.2 Unordered Groups

Table 2 compares the space and time requirements for several different unordered groups. As for ordered groups, array always gives the best performance for time, but for space the best representation varies. The amount of space required for WTR depends directly on the number of runs and as shown by Group 2, the worst case, WTR fails when the number of runs exceeds the predefined limit. The best representation is found by calculating *bpg* for each representation and choosing the one with the smallest *bpg*, which succeeds.

The ST strategy uses the same criteria for choosing the appropriate set representation. The only major change is that because WTR adds both space and time, an array is chosen more often and as shown in Group 4, when compression is sufficiently good, BWT is chosen. Although the use of WTR

Group : Size	Set Type	Total (KB)	*bpg* (bits)	S-All (msecs)	S-One (μs)
1: 1,000,000	World, 1 item removed (50 pairwise-shuffling)				
	ARRAY	3906	32	8.58	0.009
T1	BWT	864	7.08	153.5	142.080
S, T2, ST	WTR+BDD	812	6.65	1142.9	1.534
	WTR+BM	934	7.65	1489.1	1.988
	WTR+BMp	–	–	1089.5	1.586
2: 1,000,000	`MPI_COMM_WORLD` randomly permuted				
ST	ARRAY	3906	32	8.58	0.009
S,T1-2,	BWT	3346	27	320.5	306.926
	WTR	∞	–	–	–
3: 500,000	10,000 blocks randomly permuted				
	ARRAY	1953	32	4.29	0.009
T1	BWT	431	7.07	76.5	139.132
S,T2,ST	WTR+R/S	304	4.98	350.6	0.967
	WTR+BDD	304	4.98	424.4	1.128
	WTR+BM	365	5.98	593.0	1.529
	WTR+BMp	–	–	399.8	1.159
4: 10,000	Monotonically decreasing group				
	ARRAY	39	32	0.09	0.009
S, ST	BWT	12	10	1.4	125.473
T2	WTR+R/S	556	456	13.2	1.681
T1	WTR+BDD	556	456	016.0	1.994
	WTR+BM	557	457	17.9	2.220
	WTR+BMp	–	–	14.0	1.785

Table 2: Space and time costs for different unordered groups, BWT or a composite structure with a set (R/S, BDD, BM, BMp) and permutation (WTR). Annotated as in Table 1.

can save considerable amount of space, the combination of permutation and set can lead to considerable variation in communication costs. One interesting technique we have not evaluated is the persistent communication mechanism in MPI to allow the user to amortize select time over several communications to the same destination. Similarly, because the ALL and send times are part of message start-up costs, the impact of select is less for large messages.

7. GROUP MANAGEMENT OPERATIONS

MPI also provides an opaque structure for groups with routines to create groups and operations to add and remove members. The sole purpose of groups is as input to `MPI_Comm_create`, which is one way to create a new communicator. Proclets can extract the underlying group from a communicator by `MPI_Comm_group` and then use a sequence of group operations to construct a new group, which can be used to create a communicator with `MPI_Comm_create`. One could use colors and keys with `MPI_Comm_split`, but `MPI_Comm_create` has the advantage that since the group is defined locally the group information does not need to be distributed (see Procedure 2 in Section 4).

Three scalability issues related to groups are as follows. First, since the group operators do not modify their input but return a new object each time, without careful management of the group objects, a sequence of operations can result in several large objects. Second, there are operations such as `MPI_Group_incl` and `MPI_Group_excl` with an array as a parameter where the array could conceivable grow large [4]. As a consequence, although groups operations are all local, typically the same operations are performed in all the intended group members, which results in a system-wide spike in the use of memory that remains until the shared communicator is created and the group object is freed.

A third issue related to the succinct representation of groups is that the order of the parameters to the operations can easily result in unordered lists from two ordered lists. For example, given group $A = [0, 1, 2]$ and $B = [1, 2, 3]$, then `MPI_Group_union(A,B,&C)` results in $C = [0, 1, 2, 3]$, while `MPI_Group_union(B,A,&C)` results in $C = [1, 2, 3, 0]$. Theoretically, unordered collections (maps) require more storage than ordered ones (sets). These issues relate to the MPI standard and not its implementation, and group management is on the list of items to be by reviewed by the MPI Forum for MPI 3.0.

Building on our set and permutation framework to scale communicators, we propose making the following change to the definition of a group to improve its scalability. The suggested change is to modify the definition of a group from a mapping to a simple set. In the remainder we first discuss consequences of such a change including the modifications needed to the MPI standard. Second, we discuss the benefits of this change as it relates to our work on communicators.

7.1 Groups as Sets

As a set, it is no longer possible to define a re-mapping of the ranks of a group. However, for those cases where a re-mapping is needed it can be accomplished by the following operation, once the group has become a communicator,

```
MPI_Comm_split(comm, 0, newrank, *newcomm).
```

This routine produces a new communicator that maps the current rank of proclet in `comm` to `newrank`. This operation can be efficiently supported by the framework since, for the non-map representations, we already have the set and only need to add the permutation (WTR). We can even avoid copying the set by keeping reference counts so that both `comm` and `newcomm` can share the set. This optimization to `MPI_Comm_split` could be supported by adding a `MPI_PERMUTE` constant to the MPI standard for use as the color parameter to signal it is a permutation of the existing set.

Existing code that does the re-mapping with group operations needs to be modified and, because `MPI_Comm_group` now returns a set rather than map, code which depends on returning a map needs to be modified. A final consequence to this change is that for unordered groups `MPI_Comm_translate_ranks` cannot be used to translate the rank of a proclet in one group to that of another. This could be supported by adding `MPI_Comm_translate_ranks` and `MPI_Comm_compare`, that are similar to their group counterparts except that they operate on communicators rather than group objects. Like their group counterparts they are local operations. The use of `MPI_IDENT` for `MPI_Group_compare` would need to be deprecated since all groups use the default ordering. Routines `MPI_Group_translate_ranks` and `MPI_Group_compare` are still useful for obtaining ranks and comparing the underlying sets.

In terms of a composite representation, the translation of $gid_A(i)$ in communicator A is

$$gid_B(i) = \pi_B^{-1}(RANK_B(SELECT_A(\pi_A(gid_A(i)))))$$

in communicator B, since $SELECT_A(\pi_A(gid_A(i))) = SELECT_B(\pi_B(gid_B(i)))$ where $RANK$ and $SELECT$ are inverses. For a world of size N this operation requires $\mathcal{O}(\log N)$

queries for $RANK$ and almost constant time for π^{-1} in the WTR representation. Although the $\log N$ queries makes translation relatively costly, the operation is local, not required very often, if necessary could be cached and finally bulk translations can save time by using iterators.

7.2 Implementation of Groups

Our original motivation for investigating BDDs was because of the group management routines in MPI. The BDD package BuDDY has efficient implementations for the union, intersection and difference of two groups which can result in compact representations for many sets. BBD operations have the advantage that their size is proportional to the input sizes for simple boolean operations. As well, one nice feature of the BuDDy package is that the underlying BDD operations share the underlying BDD graph structure and thus may avoid the problem of returning new objects for each group operation. It may even be possible to extend this copying to other proclets and maintain one combined BDD per process.

Chaarawi and Gabriel [10] have investigated more general range representations to reduce the storage needed for groups and communicators. The technique underlying their approach was to store a group operation in terms of its difference from its input. Thus the resulting group can be efficiently stored as a sequence of updates to the membership. The BuDDy package has the same ability and maintains a common graph to store a set boolean expressions, which achieves the same goal as [10] in compactly storing the result of a sequence of operations. Unlike [10], we do not couple the group representation to that of the communicator, but rather at communicator creation time extract the path information from the BDD to obtain a more compressed static structure. The approach in [10] was tailored to ranges and strides whereas unfortunately, as shown in Table 1, the size of BDD's depends on the values of S_1 and S_2, which can grow very large for some strides.

8. EXPERIMENTS WITH FG-MPI

In this section we investigate the overhead of creating different sizes and numbers of communicators and messaging among their group members. Our test setup consists of two 4-core workstations connected by a local area network. One of the machines is an Intel® Core™ i7 workstation running at 2.67 GHz and the other is Intel Xeon E5530 at 2.4 GHz. Both machines have 6 GB of memory and run Ubuntu version 9.04 (Linux kernel 2.6.28-15-Generic) with hyper-threading enabled.

8.1 Communicator Creation and Messaging

We have designed an MPI benchmark to evaluate the creation of different numbers and sizes of communicators and the messaging time among the group members using a variety of parameters such as the size of the groups, the group map representation, the size of the outer communicator for communicator creation and the distribution of proclets to processes.

We designed a number of experiment sets to evaluate the above parameters both in terms of space and time. The results for these experiments are shown in Figure 5. In these experiments the `MPI_WORLD_COMM` size is 110,000[6] proclets which are evenly distributed over 16 OS processes, with 6875 proclets in each process and 8 processes on each of two machines. Each of the data points reported in the figure is the mean value of at least three independent runs.

Each class of experiments given in Sections 8.1.1, 8.1.2 and 8.1.3 investigates one property of the system, while keeping the other parameters constant. The distribution of proclets to processes is kept similar within each experiment class, but differs across different experiment sets. Thus, it is not possible to directly compare the results between different experiment sets.

The saving on space using our map framework versus the array is also reported for each experiment set. This denotes the total amount of space saved by using our framework, for 8 communicators in a system of 16 OS processes across two machines. Sharing of group map among collocated proclets is enabled for both our framework and the array map, therefore, that effect is factored out. As Figure 5 shows, our framework does better than arrays in terms of space for all cases.

Apart from the saving on space, we report three different times in Figure 5: The time to create a communicator of the indicated size, the point to point communication time of 10,000 messages communicated among randomly chosen sender and receiver pairs inside a group, and the time to perform ten collective gather operations by rank 0 in the group. The time for the point to point communication time of 10,000 random messages includes the time for synchronization at the end of messaging.

8.1.1 *Experiment set A*

In this set three different group sizes were evaluated while keeping the group map representation, the size of outer communicator and the distribution of proclets to processes the same. Experiment set A1 creates a group of size 10,000, A2 creates a group of size 50,000 and A3 creates a group of size 100,000. The group in all three cases is a monotonically decreasing stride. Based on this group, the framework selects BWT as the underlying map. Set A shows the ability to create different sized communicators from small to large without a corresponding overhead in terms of space and time. If we look at the communicator creation times, the results for A1, A2 and A3 show the scalability of `MPI_Comm_split` routine with respect to the size of the group. Since in `MPI_Comm_split`, the group map is only sent to the leaders, we avoid an expensive communication operation to all group members and that number is dependent on the number of leaders and does not increase with the size of the group. In all our experiment sets, the group members are evenly distributed over `MPI_WORLD_COMM`, which was done to avoid concentration of group proclets inside any OS process or machine. In this case the number of leaders is equal to the number of OS processes. The time for creation of A1, A2 and A3 thus shows little variation because it is largely dependent on the size of the outer communicator.

The time to perform collective gather operations also scales well as we increase the group size from 10,000 to 100,000 taking 1.9 seconds for A1, 2.54 seconds for A2 and 3.14 seconds for A3 using BWT map.

The framework using BWT is designed for large group sizes and as we increase the group size, the saving on space

[6]We have encountered a problem in the receive path of our system for world sizes greater than 110,000. Thus we only report on world sizes of up to 110,000.

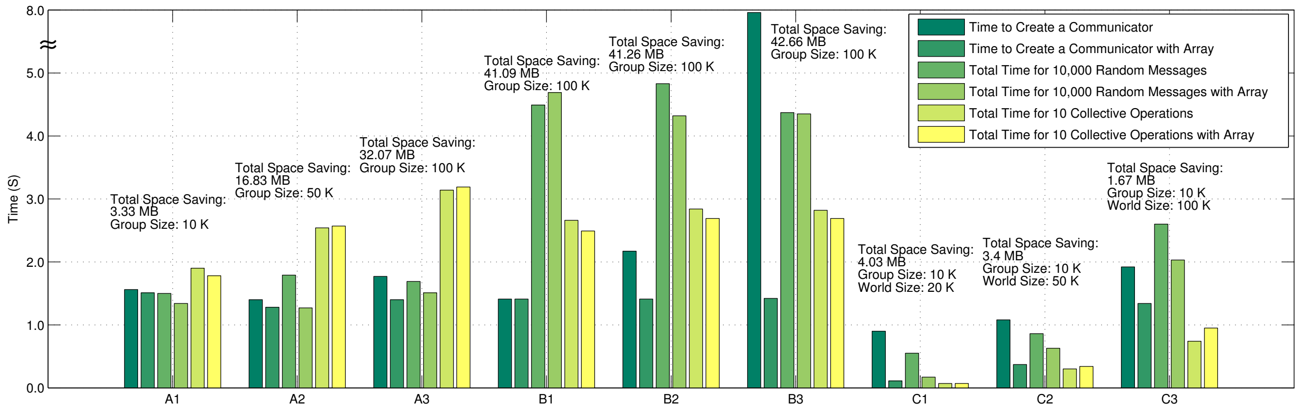

Figure 5: Space and time comparison of communicator creation and messaging for different experiment sets.

also becomes more efficient. This is apparent in the space numbers reported in Figure 5.

For random point-to-point communication, BWT shows additional lookup overhead compared to arrays, but the times are still competitive. Use of cache in BWT implementation reduces the average lookup time, but in case of a cache miss the overhead to decompress a block adds to the lookup time.

8.1.2 *Experiment set B*

In this set three different map representations were evaluated while keeping the remaining parameters the same. Experiment B1 uses an unordered range with 2 runs, B2 uses a stride with one item randomly removed and B3 is a range of ranges. The group size in all cases is 100,000. Based on these group maps, the framework selects WTR+stride for B1, bitmap for B2 and BDD for B3. As the results show, the characteristics of the representations are quite different and it points to the challenge in selecting the best trade-off for different groups.

The communicator creation time using BDD is quite large compared to all other representations. This was expected since finding the smallest BDD graph is an NP-hard problem and the BuDDy implementation could be improved. The experiment shows substantial saving of space compared to the corresponding array map representation for the given group size of 100,000 for all the representations. BDD provides the highest savings (42.66 MB) in terms of space. BDD is expected to do even better in terms of space as the size of group grows large.

Although the array shows the best results for point-to-point messaging due to its constant lookup time in nearly all the experiments, we see that the messaging time for B1 using WTR+stride does better than the array map. This is because the amount of space required to store a WTR+stride is small and fits in the system's cache, which is not the case for the large array. B2, on the other hand, has a larger overhead in terms of lookup time, but it wins on the space compared to arrays.

8.1.3 *Experiment set C*

In this set the size of the outer communicator was varied while keeping the remaining parameters same. Experiment set C1 uses an outer communicator (`MPI_COMM_WORLD`) of size 20,000, C2 of size 50,000 and C3 of size 100,000. The group map in every case is a union of a range and a stride and the size of the group being created in each case is 10,000. For this group, the framework selects bitmap for storing the group map. As expected, the results show that the size of the outer communicator has a direct effect on the communicator creation time. For all the times reported, the results show good scalability from world sizes of 20,000 to 100,000 and the effect of increasing the total number of proclets in our system.

Bitmap, as expected, has a higher lookup and communicator creation time compared to arrays. We see that C1 shows a larger saving on space compared to C3, The reason for this is that as we increase the `MPI_COMM_WORLD` size, the bitmap becomes sparser and hence less space efficient.

8.2 Large Number of Communicators

Using our benchmark program, we also tested creation of a large number of communicators. Our system comprised of two machines, with 16 OS processes evenly distributed on the two machines and 6250 proclets in each process. Our empirical results showed that the system scaled well and the maximum number of communicators that could be created were limited only by MPICH2's configuration parameters. For efficiency reasons, there are parameters in MPICH2 to control the allocation of a pool of communicators. These parameters were not changed for the following experiment which creates a large number of large communicators.

We ran a test where `MPI_Comm_split` was used to split `MPI_COMM_WORLD` of size 100,000 into 100 communicators of size 1000 each. This operation was performed repeatedly in a loop and we successfully created 4,100 context-ids inside a process before the system exhausted the communicator memory pool.

9. CONCLUSIONS

We have implemented and evaluated scalable techniques for groups and communicators using FG-MPI, a fine-grain version of MPI that scales to over one hundred thousand MPI processes. We explored two different techniques for reducing the storage requirements for communicators inside MPI. The first idea was the sharing of the group map among collocated proclets. Although not directly applicable to "one process per processor" type MPI systems the technique and the algorithms developed can be applied to share the group map among processes on the same machine. Sharing the group map led to an introduction of a new definition of context-id as well as location-aware algorithms for `MPI_Comm_split` and `MPI_Comm_create`. A major part of the design of FG-MPI was also the de-coupling of MPI processes from OS-processes that led to maps from group identifiers to their world identifiers, which made it possible to implement the framework introduced in second part of the paper.

The framework we introduced for concise representations of the group map was based on the observation that a mapping can be decomposed into a set and permutation. This decomposition allows us to use a compact set representation for the usual case where a specific mapping is not required and only adds a permutation when needed. The use of sets allowed us to consider implicit representation such as BDDs that we believe are well-suited for the types of group operations used to construct communicators. We incorporated operations from the widely-used BuDDy BDD package, and further optimized the final representation by only storing the resulting paths. In addition for unordered maps we incorporated Wavelet Trees on Runs, a state-of-the-art succinct data structure, well-suited when there are monotonically increasing runs in the map, which is likely the case in MPI. As a final catch all, we added BWT (i.e., bzip2) compression so that in memory constrained environments some compression is possible when there is no discernible pattern. We investigated time and space trade-offs between the representations to design strategies on top of the framework.

Building on our set and permutation approach we proposed changes to the definition of a group in MPI. It is a relatively minor change but allows groups to benefit from set compression techniques that can be stored in less space than general maps. We also proposed the use of BDDs because of their support for binary operations along with their ability to store multiple results in a single graph, thus sharing the graph structure inside a proclet.

Acknowledgements: We would like to thank Jérémy Barbay and Carlos Bedregal for their kind contribution and sharing of the *Wavelet Tree on Runs* library.

10. REFERENCES

[1] BuDDy - A Binary Decision Diagram Package, `http://vlsicad.eecs.umich.edu/BK/Slots/cache/www.itu.dk/research/buddy/index.html`.

[2] Argonne National Laboratory. Communicators and Context IDs. Available from `http://wiki.mcs.anl.gov/mpich2/index.php/Communicators_and_Context_IDs`.

[3] Argonne National Laboratory. MPICH2: A high performance and portable implementation of MPI standard. Available from `http://www.mcs.anl.gov/research/projects/mpich2/index.php`.

[4] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, S. Kumar, E. L. Lusk, R. Thakur, and J. L. Träff. MPI on a million processors. In *PVM/MPI*, pages 20–30, 2009.

[5] J. Barbay and G. Navarro. Compressed representations of permutations, and applications. In S. Albers and J.-Y. Marion, editors, *26th Intl. Symp. on Theoretical Aspects of Computer Science (STACS)*, pages 111–122, Dagstuhl, Germany, 2009.

[6] V. R. Basili, J. C. Carver, D. Cruzes, L. M. Hochstein, J. K. Hollingsworth, F. Shull, and M. V. Zelkowitz. Understanding the high-performance-computing community: A software engineer's perspective. *IEEE Softw.*, 25(4):29–36, 2008.

[7] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35:677–691, 1986.

[8] D. Buntinas, W. Gropp, and G. Mercier. Design and evaluation of Nemesis, a scalable, low-latency, message-passing communication subsystem. In *Proc. of the Sixth IEEE Intl. Symp. on Cluster Computing and the Grid (CCGRID)*, pages 521–530, Washington, DC, USA, 2006. IEEE Computer Society.

[9] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corp., 1994.

[10] M. Chaarawi and E. Gabriel. Evaluating sparse data storage techniques for mpi groups and communicators. In *ICCS '08: Proceedings of the 8th international conference on Computational Science, Part I*, pages 297–306, Berlin, Heidelberg, 2008. Springer-Verlag.

[11] E. D. Demaine, I. Foster, C. Kesselman, and M. Snir. Generalized communicators in the message passing interface. *IEEE Trans. Parallel Distrib. Syst.*, 12(6):610–616, 2001.

[12] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI (2nd ed.): Portable parallel programming with the message-passing interface.* MIT Press, Cambridge, MA, USA, 1999.

[13] W. D. Gropp and R. Thakur. Issues in developing a thread-safe MPI implementation. In *PVM/MPI*, pages 12–21, 2006.

[14] W.-K. Hon, K. Sadakane, and W.-K. Sung. Breaking a time-and-space barrier in constructing full-text indices. *SIAM J. Comput.*, 38(6):2162–2178, 2009.

[15] T. C. Hu and A. C. Tucker. Optimal computer search trees and variable-length alphabetical codes. *SIAM Journal on Applied Mathematics*, 21(4):514–532, 1971.

[16] H. Kamal and A. Wagner. FG-MPI: Fine-grain MPI for multicore and clusters. In *11th IEEE Intl. Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC) held in conjunction with IPDPS-24*, April 2010.

[17] D. Okanohara and K. Sadakane. A linear-time burrows-wheeler transform using induced sorting. In *SPIRE*, pages 90–101, 2009.

[18] J. Seward. bzip2 and libbzip2, version 1.0.5 a program and library for data compression. Available from `http://www.bzip.org/`.

[19] M. Stabno and R. Wrembel. RLH: Bitmap compression technique based on run-length and huffman encoding. *Info. Sys.*, 34(4-5):400 – 414, 2009.

[20] E. Toernig. Coroutine library. Available from `http://www.goron.de/~froese/coro/coro.html`.

[21] TOP500. Top 500 supercomputing sites. Available from `http://www.top500.org/`.

[22] R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer. Capriccio: scalable threads for internet services. In *SOSP '19*, pages 268–281, New York, NY, USA, 2003. ACM.