Group Operation Assembly Language - A Flexible Way to Express Collective Communication

Torsten Hoefler[1], Christian Siebert[2], Andrew Lumsdaine[1]

[1] Open Systems Lab
Indiana University
Bloomington IN 47405, USA
{htor,lums}@cs.indiana.edu

[2] NEC Laboratories Europe
NEC Europe Ltd., Germany
Rathausallee 10, 53757 Sankt Augustin
siebert@it.neclab.eu

***Abstract*—The implementation and optimization of collective communication operations is an important field of active research. Such operations directly influence application performance and need to map the communication requirements in an optimal way to steadily changing network architectures. In this work, we define an abstract domain-specific language to express arbitrary group communication operations. We show the universality of this language and how all existing collective operations can be implemented with it. By design, it readily lends itself to blocking and nonblocking execution, as well as to off-loaded execution of complex group communication operations. We also define several offline and online optimizations (compiler transformations and scheduling decisions, respectively) to improve the overall performance of the operation. Performance results show that the overhead to express current collective operations is negligible in comparison to the potential gains in a highly optimized implementation.**

I. Introduction

It is well understood that the abstraction level of collective group operations, such as Multicast [1] or MPI collective operations [2], assists developers of parallel and distributed programs to write portable and efficient code. Common group operations are typically offered by additional communication libraries or as part of the operating system functionality. This separation enables implementers to make them as efficient as possible for particular target architectures. After decades of research, many implementations already use advanced algorithms based on several parameters, like topology/hierarchies (e.g., hypercubes/SMPs), features (e.g., multicast or RDMA), bandwidth/latency, message sizes/distribution, number of participants, arrival patterns, and many more. A prominent example are today's implementations of blocking MPI collective operations (e.g., [3]). However, the static nature of common implementations often considers only few parameters (e.g., runtime behavior is often ignored). In addition, many simplifying assumptions are made to ensure a manageable complexity (like idle interconnect and processors). As target systems grow in their complexity and size, such simple and too restricting schemes are not sufficient anymore. If some of those simplifying assumptions are not guaranteed, the statically scheduled version can easily degenerate (e.g., due to a bad process arrival pattern, trees could degenerate to lists). Thus, we propose to delay important optimization decisions in order to utilize the most and up-to-date information (e.g., including current network load). One necessity is the adaptation of the actual execution order of the individual building blocks that forms the operation. Dynamic scheduling strategies at runtime are a well-understood concept to address these issues, and a promising candidate to gain the desired optimizations. However, dynamic scheduling itself is a separate complex topic [4]. It is possible to implement dynamic scheduling in existing libraries, but it is certainly a non-trivial effort, which would need to be done for every target architecture.

Based on those observations, we propose a scheme to *separate the definition and the execution* of group communication operations. We will specify a complete *domain specific language* that enables implementers to describe currently known group operation algorithms. Based on this language, we will discuss the (optimized) translation into an efficient binary representation. Finally, we also present an *interpreter*, which dynamically schedules and executes the group operations. Because such a more general description and specification of a group operation does not fix as many parameters as a direct implementation, the *interpreter* is quite flexible in the way it executes the operation. For example, it allows refinement of earlier scheduling decisions (e.g., using runtime information), or the utilization of alternative and additional resources. We discuss different levels of the specification and give first optimization examples that can be performed during the different translation steps.

A. Historical Background

Several decades ago, a general shift started from machine-dependent assembly programming to the general use of high-level languages such as Fortran. This level of abstraction started a new era of portable, optimized and highly-productive programming. Given that the first compiler for Fortran (1957) produced optimized code whose performance was comparable to that of hand-coded assembly language, this approach quickly gained acceptance. It can be argued that due to this abstraction, several optimization possibilities vanish. However, because modern compilers are typically tuned by experts that have an excellent knowledge of the target architecture (e.g., the processor manufacturers), the compilers nowadays generate machine-code which is likely even better than self-written assembly code. Those evolutions continue in the shift towards still higher level languages such as C++, Java, or C# today.

Our proposal lifts the low-level network programming to this new abstraction layer. Defining simple communication patterns or even complex group operations in an abstract

way allows a more rapid development whereas increasing portability. Future translation systems could take the burden of applying general optimization techniques from the implementer (similar to transformations like loop unrolling), and additionally enable sophisticated optimizations that might be target specific (like SSE utilization). The independence to execute such a group operation (i.e., detached from the main CPU) even allows the complete offloading of the operations to a dedicated network card. This plays a special role because it enables true asynchronous progression (i.e., packets are forwarded without main CPU involvement) and reduces resource contention as well as the influence of OS jitter.

Without loss of extensibility, we consider simple network primitives as basic building blocks for complex group communication operations. Our proposed language can be used to implement such operations and to compile them into an optimized (possibly target-specific) representation that is eventually executed while being dynamically scheduled at runtime.

B. Universal Group Communication Operations

In order to design a general specification language for group communication algorithms, we characterize possible transformations that can be applied to data in a parallel system. Our definition bases on process-local data items denoting non-overlapping memory regions, identified by d_j^i.

Definition 1. *A* **group** G *is a set of* p *individual processes, where each process can be identified by a unique rank number. The rank is given as an integral value in the interval* $[0, p)$*. Each process* i *in the group possesses a set of data items* D_i*.*

The set $S = \cup_{i=0}^{p-1} D_i$ that consists of the sets of data items of all processes is called (group) state.

Definition 2. *A* **universal group communication** *is described by an initial starting state* S_s *and a final state* S_f*.*

For example, a broadcast's starting state would be a data item d_0^r on a designated root process r and no data items on the other processes, i.e., $D_r = \{d_0^r\}$ and $S_s = \{D_r\}$. The broadcast replicates this data to all other processes of the specified group, i.e., $S_f = \{D_0, D_1, \ldots, D_{p-1} | \forall 0 \leq i < p : D_i = \{d_0^r\}\}$.

Definition 3. *A* **group communication operation** $\mathcal{F}(S_1) \Rightarrow S_2$ *describes a communication function that realizes the state transition from* S_1 *to* S_2*.*

A *group communication operation* $\mathcal{F}(S_1) \Rightarrow S_2$ can itself be constructed from the strictly ordered application of other group communication operations: $\mathcal{F}(S_1) = \mathcal{F}_2(\mathcal{F}_1(S_1))$ with $\mathcal{F}_1(S_1) \Rightarrow S_x$ and $\mathcal{F}_2(S_x) \Rightarrow S_2$. The simplest group communication operation consists of a single primitive function.

Theorem 1. *Any state change* $S_1 \Rightarrow S_2$ *can be expressed with a strictly ordered sequence of primitive operations. The minimally needed* primitive operations *are transmission of data items between a pair of processes, and local transformations of process-local data items.*

Proof: Any data item d can simply be transmitted to all processes and then transformed locally. Local transformations can copy, transform, or delete data items. Thus, any state change can be modeled. ■

We showed that any group operation can be modeled as a ordered sequence of message transmissions and local transformations. We will relax this definition in Section II.

C. Background and Related Work

Well-established communication libraries, such as MPI [2], show that a limited set of group operations is very useful to support application developers in the development of applications that communicate in regular patterns. In addition to that, it is shown by newer developments that it can be very useful to extend group communication operations to irregular communication algorithms, such as parallel sparse matrix computations. A specific proposal for inclusion in the next generation MPI standard are sparse collective operations [5]. Such operations allow the definition of arbitrary independent communication relations among a group of processes. This allows more flexibility than the well-known set of static collective operations and has a good optimization potential.

Another field of research are nonblocking collective operations. Such operations discern start and completion of a collective operation, and therefore allow to execute a predefined group communication operation simultaneously to other computations on the main CPU. Asynchronous progression makes an abstraction of the group operation necessary and typical implementations [6], [7] use communication schedules to represent the communication. However, as far as our experience goes there is currently no efficient way to create and handle the necessary schedules. The existing implementations rely on too simple description and execution schemes that certainly limit flexibility and prevent better optimizations.

Efficient online work scheduling algorithms for a set of given resources has been studied in detail in the context of operating systems. Findings in this area can be applied in order to implement an efficient scheduler for group operations.

Based on Moore's law [8], it is clear that parallelism in computer architectures will continue to emerge and it is easy to conclude that new concepts have to be found to utilize this parallelism efficiently. Our concept to separate complex group communication from the computation enables an additional level of functional parallelism. If we have an abstract and versatile definition of communication operations, then it is obvious that those operations can be executed on different units, e.g., a spare core or a network interface card. New nonblocking interfaces, such as nonblocking collectives or asynchronous collective file I/O provide a mighty tool to the developer that allows full exploitation of the additional resources. Well-known issues like missing asynchronous progression [9] will also be solved implicitly by our approach.

II. A UNIVERSAL DEFINITION FOR GROUP OPERATIONS

Theorem 1 shows that we can define a language for universal group communication operations that is based on two

primitives and a strict ordering among operations. Schemes implemented in LibNBC [6] and IBM's Collective Component Messaging Interface [7] already rely on such a strict ordering. In order to improve the efficiency of the scheduler, we show how this ordering can be relaxed to provide more degrees of freedom at each given scheduling decision.

Theorem 2. *Strict ordering is only necessary relative to each data item.*

Proof: All data items denote non-overlapping memory regions at a specific process. We can introduce dependencies that are required for a semantically correct execution of a group communication operation. For each data item, the order of: transformations, receives, receives and sends, receives and transformations, and transformations and sends has to be preserved to ensure correct execution of the group operation. The relative ordering of transformations and transmissions between different data items is not required for semantical correctness because they identify distinct memory locations. ■

We note that different intermediate states are possible while the operation is running. Those different possible states form the pool where the scheduler can choose from.

An example is given in the following ($a \xrightarrow{c} b$ stands for the transmission of item c from process a to b, and $c = f(g, h)$ represents the local transformation (reduction) of g and h to c).

Example 1. *Given the state transition* $S_s = \{D_0 = \{d_0^0, d_1^0\}, D_1 = \{d_0^1\}, D_2 = \{d_0^2\}\}$ *to* $S_f = \{D_0 = \{d_0^0 \cdot d_0^1, d_1^0 \cdot d_0^2\}, D_1 = \{d_0^1\}, D_2 = \{d_0^2\}\}$*, then* $\mathcal{F}(S_s) \Rightarrow S_f : 1 \xrightarrow{d_0^1} 0 \circ 2 \xrightarrow{d_0^2} 0 \circ f(d_0^0, d_0^1) \circ f(d_1^0, d_0^2)$ *would be a possible group communication operation.*

It is easy to see that transformations can be applied to this operation as long as the order relative to the data items is preserved. The transmissions $1 \xrightarrow{d_0^1} 0$ and $2 \xrightarrow{d_0^2} 0$ can for example be exchanged or even executed simultaneously (which is likely to happen in a communication network). Also the local computations $f(d_0^0, d_0^1)$ and $f(d_1^0, d_0^2)$ can be executed independently. However, the order relative to data items has to be preserved, in this case, data must be received before the transformation can be applied. This simple example shows that many possibilities exist to actually implement group communication operations (typically the number of alternatives grows exponentially with the number of primitives). On a given target architecture, some of them allow faster execution (e.g., in parallel) or achieve a better overall message schedule than others.

In the following section, we propose a formal language to express arbitrary group communication operations, without implicitly limiting the number of possible implementations.

III. The Group Operations Assembly Language

In this section, we specify our *Group Operations Assembly Language* (GOAL) that can be used to describe arbitrary group communication operations. A group communication operation (global view) is split into the corresponding process-individual parts (local view), which is then implemented using GOAL. First, we present a human-readable form of the GOAL, which can be directly generated, e.g., by profiling tools. We present a particular example language that can easily be adjusted to the user's need and which should be extended to support target-specific features. Higher-level languages that for instance directly support collective operations should be layered above such an assembly language - even though this paper will focus on the ground work and only envisions follow-up products.

An assembler is used to translate a GOAL source code into a binary object code (e.g. by creating op-codes and resolving symbolic names). This intermediate representation could also be created directly through a library interface whenever the performance critical assembly step should be avoided.

Next, all primitives in the described group operation will be pre-scheduled taking the attached dependencies into account (e.g., by running a topological sort on the resulting DAG), optimized and transformed into a suitable representation for an efficient execution.

A. The textual GOAL specification

The textual GOAL is designed as a simple domain specific language that can be processed by an $LR(1)$ parser. In order to enable simple evaluation schemes, we require that local transformations are free of side effects and are guaranteed to terminate.

In our language, a buffer is a consecutive area of memory that is simply identified by its starting address and a size. Such a buffer is conceptually similar to a data item, although buffers are allowed to have overlapping memory regions. Processes are simply identified by an integer, and the actual translation to a hardware address is left to the implementation. Local transformations are defined on two buffers and are called tasks. Figure 1 shows the Extended Backus-Naur Form (EBNF) of the GOAL language. Primitives for local transformations (*function*) and data transmission (*send*/*recv*) are available. All these primitive operations can be executed in arbitrary order or in parallel as long as no restricting dependency is specified. Such a dependency is necessary if an action can only be started after another action has been completed. A dependency can be defined between any pair of primitive operations and is specified as `requ A -> B`, which means that `A` requires `B` to be finished before it is allowed to be executed. Dependencies might also become necessary if buffers overlap and have to be defined by the user of GOAL. Additionally to ensuring semantical correctness, dependencies can be used to restrict scheduling decisions (e.g., the order of sends in a binomial tree is crucial for its runtime as opposed to a binary tree).

Note that terminal symbols that denote action types or parametrize actions, such as *send*, *recv*, *requ*, *exec*, *to*, *with*, *from* and *user* are not allowed as identifiers. An arbitrary number of whitespaces (newline, tab, space) is allowed between any tokens in the language. Everything between a # sign and the end of a source line is ignored (e.g., for comments).

```
<GOAL> ::= ( (<operation> | <dependency>) ';' )+
<letter> ::= 'a'|'b'|..|'y'|'z' | 'A'|'B'|..|'Y'|'Z'
<digit> ::= '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'
<integer> ::= <digit>+
<identifier> ::= <letter> { <letter> | <digit> | '_' }
<function> ::= (* will be defined below *)
<buffer> ::= <integer> ',' <integer>
<target> ::= <integer> (* unique identifier for other processes *)
<sendop> ::= 'send' <buffer> 'to' <target>
<recvop> ::= 'recv' <buffer> 'from' <target>
<task> ::= 'exec' <function> 'with' <buffer> ',' <buffer>
<operation> ::= [ <identifier> ':' ] ( <sendop> | <recvop> | <task> )
<dependency> ::= 'requ' <identifier> '->' <identifier>
```

Fig. 1. Extended Backus-Naur Form of GOAL

The semantics of a send/receive pair is clear, they simply transmit a message from one process to another. Local operations can either be chosen from a set of predefined operations or from arbitrary user-supplied functions. The former have the advantage that they only define semantics and therefore allow execution on all units that support this operation. For predefined operations, both buffers must contain the same number of basic elements. This restriction does not apply to user-defined operations where the user is free to use the two buffers arbitrarily. The set of element-wise reduction functions is predefined as follows.

```
<opcode> ::= ('max'|'min'|'sum'|'prod') |
             ('l'|'b')('and'|'or'|'xor') | 'copy'
             (* logical, bitwise and copy *)
<datatype> ::= ('Int'|'UInt'|'Float')
               ('8'|'16'|'32'|'64')
<function> ::= <opcode> <datatype> |
               'user' <integer> (* function ident. *)
```

The user can define own functions that accept two buffer arguments with the keyword `user`. The interpreter will then ensure that the function at the supplied pointer will be executed by the main CPU with the arguments `(*func)(void *buf1, int size1, void* buf2, int size2)`.

If we assume 1 byte integer data items beginning at address 500 and `sum` as operation, Example 1 could be expressed in textual GOAL on the three participating processes as follows:

Example 2.

```
rank #0 {
  r1: recv 503,1 from 1;
  r2: recv 504,1 from 2;
  e1: exec sumInt8 with 501,1 503,1;
  e2: exec sumInt8 with 502,1 504,1;
  requ e1 -> r1;
  requ e2 -> r2;
}
rank #1, #2 {
  send 501,1 to 0;
}
```

B. GOAL Library Interface

Our EBNF description can be turned into a library interface that retains all optimization and transformation possibilities of the original language. However, there are many ways to implement such an interface. Our implementation in LibNBC [6] uses the following functions:

```
goal GOAL_Create() = creates an opaque GOAL object
fid GOAL_Reg_func(goal, func) = register user function
id GOAL_Send(goal, sbuf, size, dest) ≡ <sendop>
id GOAL_Recv(goal, rbuf, size, src) ≡ <recvop>
id GOAL_Exec(goal, fid, buf1, s1, buf2, s2) ≡ <task>
GOAL_Requ(goal, A, B) ≡ <dependency>
sched GOAL_Compile(goal) = assembly step
GOAL_Sched_run(sched) = execute schedule
```

Example 3. *Example 2 could be expressed as follows:*

```
GOAL_Create()
if (rank = 0) {
  r1 = GOAL_Recv(g,503,1,1)
  r2 = GOAL_Recv(g,504,1,2)
  e1 = GOAL_Exec(g,sumInt8,501,1,503,1)
  e2 = GOAL_Exec(g,sumInt8,502,1,504,1)
  GOAL_Requ(g,e1,r1)
  GOAL_Requ(g,e2,r2)
}
elif ((rank = 1) or (rank = 2)) {
  GOAL_Send(g,501,1,0)
}
sched = GOAL_Compile(g)
```

C. The GOAL Schedule

The GOAL language (both the textual version as well as the library version) define an architecture-independent way of expressing group operations. For performance reasons, GOAL is translated into a machine-dependent format, just like other languages, such as C/C++ and Java which are compiled into machine language (or bytecode respectively). This final representation, called schedule, should be carefully adapted to execute efficiently on the target (network) architecture. We call the particular description for a specific architecture

Network Instruction Set Architecture (NISA). In this paper we define the NISA that we used to implement our first prototype for execution on a separate CPU core. This NISA can be extended or adapted to different network architectures (e.g., InfiniBand [10] targets could split a send operation into a memory-registration task and an RDMA write operation, thus enabling more parallelism and automatic/pipelined pre-registration such as in [11]).

All group communication operations in a GOAL description form a Directed Acyclic Graph (DAG) $\mathcal{G} = (V, E)$ where the set of vertices V includes all actions and the set of edges E includes all dependencies. Such a DAG can be serialized into a stream of actions that retains all properties of the DAG. Multiple different streams are often possible and thus enable dynamic scheduling. We use the following binary format to hold the resulting schedule:

```
<SCHEDULE> ::= <independent> <action>+
<count> ::= (* unsigned 32-bit integer *)
<offset> ::= (* unsigned 32-bit integer *)
<num sources> ::= (* unsigned 16-bit integer *)
<type> ::= (* unsigned 16-bit integer *)
<independent> ::= <count> <offset>+
<dependent> ::= <count> <offset>*
<action> ::= <num sources> <operation> <dependent>
<operation> ::= <type> <parameters>
<parameters> ::= (* operation-specific parameters *)
```

This representation embeds all topological sorts of the DAG, thus enabling efficient dependency-driven scheduling. All independent actions (i.e., actions that can be started immediately without waiting for anything) are listed at the beginning of the schedule (offsets are given relative to the start of the schedule). All other actions in the schedule depend on at least one other action. Actions start with a counter representing the number of incoming edges in the DAG, followed by the actual operation with its specific parameters, and close with references to all dependent actions.

Such techniques have been explored for DAG-driven computations in the past [12], [13], and we want to establish an equivalent for the more general group communication operations. Our execution model represents the data-driven execution in the established models. Demand-driven execution would cause many control messages (each process would have to request data from remote processes instead of just receiving it) and is thus not considered.

Example 4. *Figure 2(a) shows our Example in form of a DAG representation. The independent actions $r1$ and $r2$ ($num\ sources = 0$) have $e1$ and $e2$ as dependent actions respectively. The actions $e1$ and $e2$ both depend on exactly one other action and therefore have a sources counter of 1 in the schedule. Figure 2 (b-e) shows some possible orders of execution of this graph. The online scheduler can choose the best execution order based on the availability of either data item c or d (cf. Example 1). Also note, that (e.g., on a multi-core CPU) e1 and e2 can be executed in parallel, and (e.g., on a multi-port network), r1 and r2 can received in parallel. Therefore, the scheduler is free to execute the whole DAG based on the available resources.*

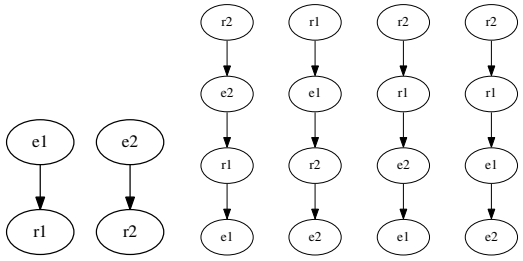

Fig. 2. Dependency DAG (a) and some possible execution orders (b-e)

Local optimization is enabled in that the associated computation can be executed immediately as soon as any receive is completed. This is possible because the scheduler can choose from any valid topological ordering of the graph, and can thus (dynamically) select a suitable path based on the current progress of the individual elements.

Global optimization can be performed in a superordinate compilation step. If all processes of a group are known at compile time, then the compiler can collect the communication schedules from all processes and compose a global communication graph. Now, the compiler can map this graph to a particular network architecture and compute a better communication schedule. After the graph mapping, a good scheduling could be achieved with multi-graph coloring as described in [5]. The schedule transformation can be done by adding more dependencies which limits the dynamic scheduler's flexibility. For example, if the compiler knows that all processes can communicate with process 0 independently, an artificial dependency that makes the communication with process 0 dependent on some other communication (which needs to be executed first then) might reduce the load at process 0 and improve overall performance. Such scheduling algorithms are an open field of research similar to scheduling of DAG-driven computations [14]. However, GOAL enables and encourages the use of such techniques for group communication operations.

D. Execution of a GOAL schedule

The dependency-driven execution of the schedule can be implemented with well-known techniques from task-scheduling systems. The scheduler can either be centralized and assign work to the network and the main CPU, or it can be distributed and work-stealing techniques, well-known from TBB [15] or Cilk [16], could be employed. These greedy ("keep everything busy") scheduling strategies perform well in practice, and are even theoretically competitive (e.g., the List Scheduling algorithm published in 1966 by Graham is $(2 - \frac{1}{p})$-competitive).

Today's computer systems typically have a single network interface but many CPU cores. Thus, we propose to run a centralized scheduler (e.g., one on each compute node in a cluster), which interprets the schedule and assigns work to the network and other CPU cores, on a separate CPU core. This

technique can be changed if it turns out to limit performance (e.g., if the number of cores or network interfaces becomes too large to be managed by a single thread).

We also pay special attention to a simple execution in minimalistic environments. This enables the implementation of a scheduler in hardware (for example on a network card or another appliance with access to the host memory). In the following, we show how a simple and efficient scheduler can be implemented that interprets GOAL schedules.

Our proposed centralized scheduling scheme, representing `GOAL_Sched_run()`, is described in the following. Let $\mathcal{R}$ be the set of running (or outstanding) action items, and let α and δ represent single actions. λ is an integral number.

```
1  R = ∅
2  foreach independent action α
3    start action α
4    R ⇐ R ∪ α
5  end for
6  while (R ≠ ∅)
7    α ⇐ next completed action in R
8    R ⇐ R \ α
9    foreach dependent action δ of α
10     λ ⇐ decrease sources counter of δ
11     if (λ = 0)
12       start action δ
13       R ⇐ R ∪ δ
14     end if
15   end for
16 end while
```

Listing 1. Pseudocode for a centralized scheduler

The scheduling/execution algorithm in Listing 1 starts independent actions, waits for any completion and starts a dependent action when all its dependencies are satisfied. A blocking implementation waits until a next action is completed. A nonblocking implementation, however, needs to remember $\mathcal{R}$, leaves the loop at Line 7, and resumes at the same position whenever the operation is progressed.

With some modifications, a schedule can be executed in a limited memory environment by using a sliding window, much like a superscalar CPU uses an instruction window. However, some additional memory is needed in the case where an action gets completed but its corresponding dependent actions are not yet in the current window.

Theorem 3. *A schedule with n actions can be executed using a sliding window of size $\mathcal{O}(1)$ and $\mathcal{O}(n)$ additional space.*

Proof: The number of edges f from finished actions to another action a that is not in the window yet can be stored in a table $H[a] \rightarrow f$. When the action a enters the window, its incoming count is decremented by f. The maximum size of H is limited by the total number of actions n. With a schedule resulting from an adversary DAG where for $1 \leq i < n$, action i has an edge to action $i+1$, and for $1 \leq j \leq n/2$, action j has an edge to action $j + n/2$, it can be shown that this bound is tight. Such an adversary DAG is shown in Figure 3 for 10 actions. We observe that with a window size of 2, storage of $n/2$ is needed to remember the completed actions for all items that are to the right of the sliding window. ■

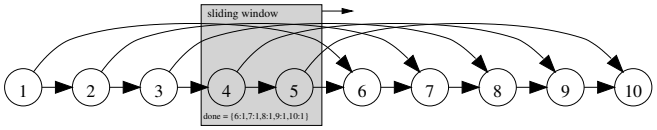

Fig. 3. Illustration of a schedule with 10 actions and a window size of 2

This might cause concern in a hardware implementation, e.g., an implementation where a schedule is streamed to a network interface for asynchronous execution. We argue that every schedule can be adapted to run with constant space.

Theorem 4. *The space requirements to execute a schedule can be reduced to $\mathcal{O}(1)$ if dummy actions are added.*

Proof: Inserted dummy actions, which are not executed (i.e., they represent NOPs), can be used to introduce additional dependencies, such that:

- all actions between two consecutive dummy actions i and j ($i < j$) depend on the completion of i, and
- dummy action j depends on all actions between i and j.

Such a transformed schedule needs to remember at most $j - i - 1$ action items (equidistant dummies). The order of actions must be a valid order according to the topological sort of the original graph. All spare dependencies crossing the dummy actions can now be removed safely while retaining (restricting) the original dependencies. Thus, it is possible to limit any window-based scheme to have a maximum number of unreachable actions during execution. ■

This transformation allows every schedule to be broken up into smaller pieces by the compiler. Our modified example schedule is show in Figure 4. However, this restriction in parallelism is likely affecting performance adversely, and our experience suggests to support at least a number in the order of $\log p$ where p is the targeted number of processes.

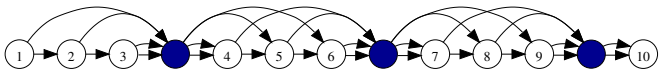

Fig. 4. Transformed DAG with a maximum memory demand of 3 items, independent of the window size

IV. Common collective algorithms in GOAL

In this section, we explain how to implement some classes of algorithms in GOAL that reflect commonly used group communication operations. GOAL itself is a simple domain-specific language to express such operations. However, it is not mighty enough to define these operations in a dynamic way, i.e., independent of a particular data and process layout. Thus, a definition of an algorithm in GOAL has to be generated in a preprocessing step before it can be compiled. If the preprocessing language is Turing complete, like for example the template system of C++, then arbitrary communication and computation operations can be expressed with GOAL.

A. Tree Algorithms

Tree-based algorithms are important to minimize the running time of latency-bound operations. Typical examples are broadcast and reduction on binomial trees [17]. The following pseudocode can be used to generate a broadcast operation in GOAL (i = process rank and p = # of processes, see Def. 1).

```
recv = -1   // valid identifiers are ≥ 0
for r = 0 to ⌈log2 p⌉-1
  send = -1
  if ((i+2^r < p) and (i < 2^r))
    send = GOAL_Send(g, buf, size, i+2^r)
  end if
  if ((send ≥ 0) and (recv ≥ 0))
    GOAL_Requ(g, send, recv)
  end if
  if ((i ≥ 2^r) and (i < 2^(r+1)))
    recv = GOAL_Recv(g, buf, size, i-2^r)
  end if
end for
```

Listing 2. Generating a broadcast tree rooted at process 0 in GOAL

A reduction tree can be built in a similar way. The main difference is that data has to be transformed after it is received and before it is sent. This replaces the *requ* clause (`GOAL_Requ`) with a function that requires the receive to finish and must complete before the send is started.

B. Dissemination Algorithm

The dissemination algorithm is often used to implement operations that do not have a single root but where information can be computationally reduced or combined during the communication [17]. We show how to generate a general n-way dissemination [18] pattern with a reduction operation. The n-way dissemination algorithm degenerates to the original algorithm by Hengsen [19] for $n = 1$.

```
for r = 0 to ⌈log_{n+1} p⌉-1
  for w = 1 to n
    send[w-1] = GOAL_Send(g,localbuf,size,
                         (i + w·(n+1)^r) mod p)
  end for
  if (r > 0)
    prev = recv[0]
    for w = 1 to n-1
      red = GOAL_Exec(g,op,buf[w-1],size,buf[w],size)
      GOAL_Requ(g,red,recv[w])
      GOAL_Requ(g,red,prev)
      prev = red
    end for
    red = GOAL_Exec(g,op,buf[n-1],size,localbuf,size)
    GOAL_Requ(g,red,recv[0])
    GOAL_Requ(g,send[n-1],prev)
    for w = 1 to n
      GOAL_Requ(g,send[w-1],red)
    end for
  end if
  for w = 1 to n
    recv[w-1] = GOAL_Recv(g,buf[w-1],size,
                         (i - w·(n+1)^r) mod p)
  end for
end for
```

Listing 3. Generating a n-way dissemination pattern with a reduction operation

C. Other Algorithms and Schedule Size

The size of the schedule clearly depends on the implemented algorithm. For dissemination and tree based algorithms, the size is $s_{tree} = \mathcal{O}(\log p)$. For other algorithms, such as a pipelined broadcast operation of with k data fragments, the schedule size scales with $s_{pipe} = \mathcal{O}(k)$, which could be problematic. Although this number of fragments is typically rather small for a reasonable segment size (i.e., one that leads to a good performance). The maximum size of a useful schedule is also trivially bounded by the proof of Theorem 1 to $s = \mathcal{O}(k \cdot p)$. Average schedules are considerably smaller, often bounded by $\mathcal{O}(\sqrt{n} + \log p)$ for some message size n.

We note that many algorithms generate well-structured GOAL schedules. For example in a pipelined chain broadcast with a message size of n, every node sends $\lceil n/s \rceil$ fragments with a segment size of s to its neighbor. It is possible to find domain-specific compression schemes for such regular schedules with a much lower resource usage even during execution. We are considering such features as worthwhile candidates for future GOAL extensions.

Those theoretical considerations show that only a minimal overhead is added for storing schedules. We will show with practical experiments in the next section that storing, creating, and interpreting the schedules only adds negligible runtime overhead.

V. Implementation and Results

In order to show the applicability of our approach, we implemented a compiler/assembler, a library interface and the scheduler/executor outlined in Listing 1 with and without a separate communication thread. We compared our binomial tree broadcast implementation that is similar to Listing 2 with the optimized tuned collective implementation in Open MPI 1.3 [3]. The tuned collective implementation in Open MPI chooses a binomial broadcast tree for small messages. Figure 5 shows the MPI_Bcast latency for both implementations. We see that the addition of our scheme is very comparable to highly optimized implementations.

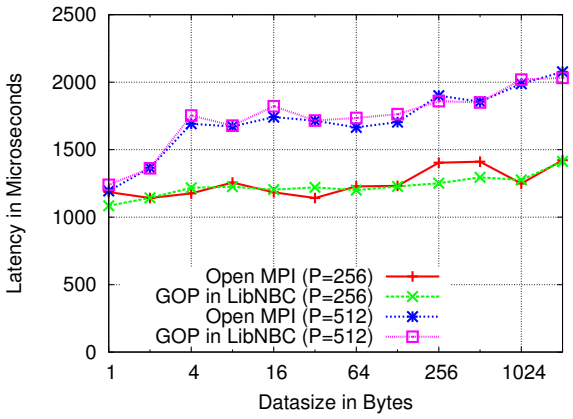

Fig. 5. Comparison of Open MPI and GOP on 256/512 processes on 64/128 InfiniBand nodes

Our abstract implementation can also be executed on a separate core, independently of the main CPU. This is very interesting for the implementation of nonblocking collective operations because group operations expressed in GOAL can be executed either blocking or nonblocking and either in the main thread or in a separate thread. We also compare our results to the only available open-source implementation for nonblocking collective operations LibNBC [6]. We compared the minimal CPU overhead (i.e., the fraction of communication that cannot be overlapped) of LibNBC with a manually progressed GOP implementation (i.e., the main thread has to call the scheduler from time to time) and an implementation in a separate thread. We used the time-based overlap benchmark as described in [20].

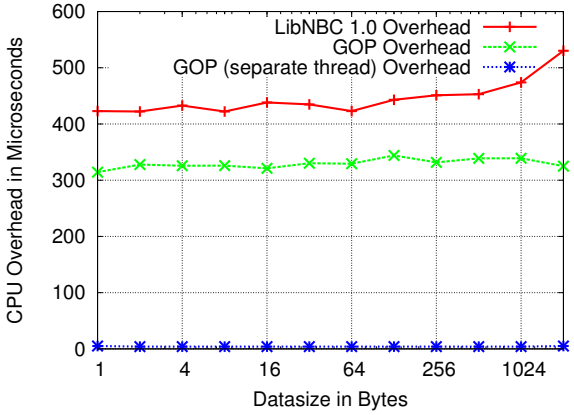

Fig. 6. Comparison of LibNBC and GOP on 512 processes on 128 InfiniBand nodes

We showed that GOAL and the proposed framework has the potential to express blocking and nonblocking group operations similarly and also enables execution on separate processing elements. We also demonstrated that offloading the scheduler to a separate thread can easily lead to a low-overhead implementation.

VI. Conclusions and Future Work

We have shown that an abstract definition of group communication operations enables easy definition of such operations for blocking and nonblocking execution. We outlined how such a scheme can be used to implement all MPI collective operations in a blocking and nonblocking interface with minimal performance impact. On the contrary, our language-based approach enables compiler-based transformations, which can more easily make use of many general optimizations. Communication schemes can be further enhanced with a better mapping to the underlying communication network (global optimization) and can be more efficiently executed due to the utilization of dynamic scheduling (local optimization). To express maximum parallelism, we employed a data-driven DAG model which is well known from the architecture and functional programming world. We suspect that our work is a first step towards defining a high-level language to express optimized group communication algorithms.

Our future work aims to analyze the use of different scheduling strategies and compiler optimizations that can be applied to improve the execution. We will also investigate different compression schemes in order to enable smarter resource consumption. We showed that an implementation of the scheduler in a separate thread allows for fast execution, high overlap, and asynchronous progression. We will also investigate the implementation of schedulers on network interface cards to make those cards capable of handling such group operations directly.

References

[1] D. Meyer, "Administratively Scoped IP Multicast," RFC 2365 (Best Current Practice), Jul. 1998.

[2] MPI Forum, "MPI: A message-passing interface standard. version 2.1," September 4th 2008, www.mpi-forum.org.

[3] E. Gabriel *et al.*, "Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation," in *Proceedings, 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, September 2004.

[4] N. S. Arora, R. D. Blumofe, and C. G. Plaxton, "Thread Scheduling for Multiprogrammed Multiprocessors," in *Proc. of the 10th Symposium on Parallel Algorithms and Architectures (SPAA)*, 1998, pp. 119–129.

[5] T. Hoefler and J. L. Traeff, "Sparse Collective Operations for MPI," in *Proceedings of the 23rd IEEE International Parallel & Distributed Processing Symposium (IPDPS), HIPC Workshop*, May 2009.

[6] T. Hoefler, A. Lumsdaine, and W. Rehm, "Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI," in *In proceedings of the 2007 International Conference on High Performance Computing, Networking, Storage and Analysis, SC07*, 11 2007.

[7] S. Kumar *et al.*, "Architecture of Component Collective Messaging Interface," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 15th European PVM/MPI Users' Group Meeting*, vol. LNCS 5205. Springer, 9 2008.

[8] G. E. Moore, "Cramming more components onto integrated circuits," in *Readings in computer architecture*, San Francisco, CA, 2000, pp. 56–59.

[9] J. White III and S. Bova, "Where's the Overlap? - An Analysis of Popular MPI Implementations," 1999.

[10] InfiniBand Trade Association, *Infiniband Architecture Specification Volume 1, Release 1.2*, 2003.

[11] G. M. Shipman *et al.*, "High performance RDMA protocols in HPC," in *Proceedings, 13th European PVM/MPI Users' Group Meeting*, ser. Lecture Notes in Computer Science, Bonn, Germany, September 2006.

[12] P. C. Treleaven, D. R. Brownbridge, and R. P. Hopkins, "Data-driven and demand-driven computer architecture," *ACM Comput. Surv.*, vol. 14, no. 1, pp. 93–143, 1982.

[13] Arvind and K. Gostelow, "An Asynchronous Programming Language for a Large Multiprocessor Machine," Dept ISC, UC Irvine, Tech. Rep. TR114a, Dec 1978.

[14] C. Fu, X. Jiao, and T. Yang, "Efficient sparse lu factorization with partial pivoting on distributed memory architectures," *IEEE Trans. Parallel Distrib. Syst.*, vol. 9, no. 2, pp. 109–125, 1998.

[15] J. Reinders, *Intel Threading Building Blocks*. O'Reilly, 2007.

[16] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the Cilk-5 multithreaded language," in *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, Montreal, Quebec, Canada, Jun. 1998, pp. 212–223.

[17] J. Pjesivac-Grbovic, T. Angskun, G. Bosilca, G. E. Fagg, E. Gabriel, and J. J. Dongarra, "Performance Analysis of MPI Collective Operations," in *Proceedings of the 19th International Parallel and Distributed Processing Symposium*, Denver, CO, April 2005.

[18] T. Hoefler, T. Mehlan, F. Mietke, and W. Rehm, "Fast Barrier Synchronization for InfiniBand," in *Proceedings, 20th International Parallel and Distributed Processing Symposium IPDPS 2006 (CAC 06)*, April 2006.

[19] D. Hengsen *et al.*, "Two Algorithms for Barrier Synchronization," *Int. J. Parallel Program.*, vol. 17, no. 1, pp. 1–17, 1988.

[20] T. Hoefler and A. Lumsdaine, "Message Progression in Parallel Computing - To Thread or not to Thread?" in *Proceedings of the 2008 IEEE International Conference on Cluster Computing*, 10 2008.