Abstract—We describe the design of a lightweight library using MPI to support stream-processing on acyclic process structures. The design can be used to connect together arbitrary modules where each module can be its own parallel MPI program. We make extensive use of MPI groups and communicators to increase the flexibility of the library, and to make the library easier and safer to use. The notion of a communication context in MPI ensures that libraries do not conflict where a message from one library is mistakenly received by another. The library is not required to be part of any larger workflow environment and is compatible with existing MPI execution environments. The library is part of MarketMiner, a system for executing financial workflows.

I. INTRODUCTION

Stream-processing constitutes an important class of parallel applications that are widely used in all types of engineering systems [1]. The basic structure underlying stream-processing is an acyclically connected collection of components. Although there are many technologies that can be used as the “glue” to connect together these components, one such approach is to use the Message Passing Interface (MPI) library [2].

Since its inception in 1992, MPI has become widely used in the high performance computing community for the implementation of scientific computing applications. The success of MPI can be partly attributed to striking a good balance between library abstractions and what can be efficiently supported on the latest parallel architectures. Although critics sometimes consider it too low level, one innovation introduced by MPI was the support for creating higher level libraries to support particular classes of applications. There are several library routines with middleware support for information hiding and the creation of an independent context for messaging within the library. With the use of these routines, MPI can be used to create libraries which hide the non-essential complexities of maintaining the underlying structure while providing a high-level abstract interface to the user [3], [4].

We consider the design of a lightweight MPI-based library to support stream-processing. The library has been used in the implementation of MarketMiner [5], a real-time stream-processing system targeted towards financial engineering workflows. It consists of a number of core computational components that can be connected together to build customized workflows to compute analytics on financial data.

Real-time computation of financial analytics is challenging because it is both data and compute intensive and the ability to perform more analysis in less time provides a technical advantage over the competition. MarketMiner takes advantage of MPI to create a flexible and scalable system for gluing together highly-parallel modules to realize financial workflows.

There are numerous advantages to using MPI. First, it is a well-established technology for the implementation of parallel software and there already exists a software base of programs and libraries that can be integrated into the system. Second, most existing equipment vendors support MPI and the use of MPI, rather than a custom ad-hoc communication mechanism, simplifies the porting of applications to new architectures as they become available. MPI is an open-standard and the latest version is MPI 2.1 with MPI 3 in the planning stages. There are several open-source implementations of the library with active ongoing development [6], [7]. These developers work closely with universities and industry partners to ensure MPI is well-supported on the latest architectures. As a widely accepted standard, one expects that as an MPI library we can continue to take advantage of other MPI software and advances in MPI middleware for large scale parallel architectures.

The main contribution of this paper is an illustration of the use MPI to build a simple library for stream-processing applications. We show how the underlying workflow structure can be hidden from the user. Modules can be added by considering only its local connections and protocol adapters can be used to further tailor the inter-module communication. The modules themselves can be highly parallel and we use MPI inter and intra communicators to provide a messaging environment that reduces the possibilities of incorrect messaging. The construction of the inter-module communication structures uses MPI collectives to create the necessary communication contexts. Since the communication contexts can overlap, the operations need to be coordinated to avoid deadlock. We describe distributed algorithms to build communication contexts for workflows consisting of an arbitrary collection of acyclically connected components.

In Section II we briefly contrast our approach to related work in the area. In Section III we give a general overview of the system. We also discuss details of MPI relevant to the use and implementation of our library. Section IV describes the design of the library including the MPI execution environment.
and the user-level API to initialize and communicate using the library. In Section V we show the implementation of a prime number sieve to illustrate the use of the system. Section VI describes the techniques and algorithms used for creating the communication contexts needed by the library. Finally Section VII has the conclusions and a discussion of possible extensions.

II. RELATED WORK

The focus of our paper is on the lower level implementation details of the workflow environment. Our work is orthogonal to work on description languages and richer runtime grid-environments for the execution of distributed or stream-based systems [8], [9]. Our system does not provide a complete runtime environment like commercial products such as streambase [10]. Runtime environments are not discussed since it is somewhat orthogonal to the library. The advantage of using MPI is its wide adoption with almost all cluster manufacturers, who provide their own runtime environment. This allows the library to be used on almost any cluster, including the Grid with MPICH-G [11], and any open-source or proprietary workflow environment that can integrate with MPI.

Our work is closest to that of P-RIO [12], which defines connections with imports and exports similar to our own use of the terms. P-RIO was implemented in PVM and there is no MPI implementation that takes advantage of MPI’s mechanisms for the safe use of libraries. There are also systems like Dryad [13], which describes a rich interconnection system between modules, but is based on threads and not MPI. With regards to MPI, there is the work by Squyres [14], who describes an MPI library using the concept of a port. However, the design and implementation of the libraries are not given. The notion of ports also appears in the Common Component Architecture (CCA) [7], a standard for component-based architectures for High Performance Computing. Our library is similar to the CCA compliant CAFFEINE framework [7] in that the library is lightweight, without all the features of more heavy-weight component frameworks, however, unlike CAFFEINE the library is restricted to stream-based computing rather than the composition of SPMD modules.

III. OVERVIEW

Our goal was to create a simple MPI library to support workflow computation where the underlying structure is a directed acyclic graph (DAG). Figure 1 shows an example of a financial workflow comprised of a variety of modules connected together in a DAG. In Figure 1 there is a live data source and a database source that flow into the first two modules of the system, called the collector modules. The collector modules retrieve data for different assets (e.g., stock bid/ask quotes) and pass the data onto the technical analysis engines which create a stream of technical analysis indicators (e.g., price) based on each of the assets. The technical analysis engines pass the stream of data onto the correlation engines which periodically compute the correlation matrix using the time-series streams for all of the assets. The correlation components that do the bulk of the computation are themselves implemented in parallel using an arbitrary number of processes. The correlation engines pass the computed correlation matrices onto trend monitoring and risk analysis tools that use the correlation information for real-time measurement and forecasting of volatility and risk. The trend and risk analysis components are data sink modules and either pass the results outside the system for visualization, storage or input to an automated trading system. Each of the modules operates with respect to a time interval where data is pumped into the system, periodically triggering a computation, with computed results being passed onto the next component in the workflow.

Although there is a variety of different middleware that could be used to implement workflows, we focused on MPI because it is the de-facto standard for implementing parallel scientific software. Financial workflow systems like Market-Miner can take advantage of the large number of high quality numerical libraries that already use MPI. As a result, our library needed to ensure that it could integrate with and use existing MPI libraries and programs.

A. Groups and communicators

A communicator is an MPI object containing group information and a unique identifier called the context. In MPI, a group is a collection of processes where the processes in a group of size $N$ are identified by their rank, an integer from 0 to $N-1$. The context is used to uniquely identify the group or groups for the communication. Group information and the context make it possible to guarantee that messages are matched by the intended communication routine. Essentially, communicators provide a scoping mechanism for messages and can be used to structure the communication to support libraries, simplify programming and eliminate a common source of messaging errors.

There are two types of communicators: intra-communicators and inter-communicators. An intra-communicator is used to communicate inside a group and an inter-communicator is used to communicate between two groups. With respect to a given process, an inter-communicator contains information about the local group to which the process belongs, the remote group to which it can communicate, as well as the context. New communicators can only be created with respect to an outer enclosing communicator that contains all of the processes
in the communicator. All communicators are directly or indirectly derived from MPI_COMM_WORLD, the pre-defined communicator associated with the group of all processes.

MPI routines that create new communicators are collective operations with respect to the group associated with the outer communicator. A collective operation is one that requires communication between all of the members of the group and therefore the routine must be invoked by all members of the group at the same time. These requirements are necessary to ensure that a unique context identifier can be chosen for the communicator. Because creating communicators is a collective operation, care must be taken when groups are overlapping to avoid deadlock.

We take advantage of both inter and intra communicators inside our library. A module that is part of the workflow has its own intra-communicator that is the basis for all intra-module communication. Inter-communicators are used for inter-module communication where conceptually these inter-module connections are viewed as I/O ports with an agreed upon protocol specifying the type of data and manner of reading and writing the data to the port. Each component in the system has one or more inports and one or more outports. Ports can be many-to-one or one-to-many.

### IV. Design of the Library

Our library consists of the following routines: workflow_Getenvironment(), workflow_Init(), workflow_Finalize(), and various user defined workflow_Send() and workflow_Recv() routines. The following three sections describe the main parts of the library. Section IV-A describes workflow_Getenvironment() and initial set-up of the intra-communicators. Section IV-B describes the inter-communicators that are set-up by workflow_Init(). Finally, Section IV-C describes the different types of protocol adapters and support for implementing custom user-defined ones.

#### A. MPI Execution Environment

The mpiexec command can be used to start the execution of a MPI program. Users specify on the command line, or as a separate configuration file, the separate processes and number of processes of each type to start executing. Each separate process can have its own command line parameters.

We define three new command line parameters for every process: name, inports, and outports. Parameter name is a required parameter for every process. Parameters inports and outports are comma delimited strings from the same set of process names. The names are used to identify the set of all processes that belong to a module. The inports and outports specify the data flow connections between the modules. The resulting structure must be acyclic and all the processes belonging to the same module must have the same set of inports and outports (see Figure 3).

Specifying the structure on the command line is a very simple and flexible technique for defining the workflow. The name parameter makes it possible for users to create modules consisting of one or more processes and it also makes it possible to use the same executable in different modules of the workflow. The workflow_Getenvironment() routine is used to retrieve the command line parameters and store the values into an environment variable that is an argument of workflow_Init(). The one disadvantage to this approach is the potential for conflict with previously defined command line parameters for the process, which would require some modification to the program for the process. The functionality of workflow_Getenvironment() was separated from workflow_Init() to make it possible to extend the library with other techniques for obtaining the information needed to initialize the environment.

The workflow_Init() routine uses the list of names in the environment argument to create two new intra-communicators with respect to the outer communicator MPI_COMM_WORLD. We create an intra-communicator for all the processes in the same module. The intra-communicator acts as a localized MPI_COMM_WORLD, which simplifies porting stand-alone MPI programs into the workflow. We also select a leader from the processes within each module and create an intra-communicator for the group of all leaders. The leaders intra-communicator is used internally as the basis for the inter module communication that is described later in Section VI.

The first step in creating the two new communicators is to use the collective operation MPI_Allgather() to gather the names of all the processes. After execution, all processes will have a list of names where the index of the list is a mapping of process names to process ranks in MPI_COMM_WORLD. Each process uses the list to determine a module leader. If $k$ is the index of the first occurrence of name $A$ in the list, then process $k$ is the leader for module $A$. All processes can now call

```cpp
MPI_Comm_split(MPI_COMM_WORLD, myleader 0,&localcomm);
```

which uses myleader as the key for partitioning MPI world and returning a new intra-communicator (localcomm) for each partition. Routine workflow_Init() will return localcomm to the calling process so that each module has its own intra-communicator. Similarly,

```cpp
MPI_Comm_split(MPI_COMM_WORLD, key, 0,&leadercomm);
```

creates an intra-communicator for all leader processes by either setting key to one for leader processes and MPI_UNDEFINED otherwise.

#### B. Inter-communicator connections

Consider the connection between modules in the workflow. As shown in Figure 1, the outports of one or more modules are connected as inports of another module. There are various ways to use inter-communicators to connect the ports from the modules on the left to the module on right. Figure 2 shows several different possibilities of how to connect two modules $A$ and $B$ with module $C$. Although shown with only two inports,
In general an arbitrary number of outports may be connected to a given module. When a module has several outports, the module will obtain an inter-communicator connection for each target import. Each of the configurations has its own advantages and disadvantages with regards to defining protocol adapters (Section IV-C). In general, there is a trade-off between the ability to easily compose modules and the richness of the communication patterns that can be supported.

In the case of (a), we combine the leaders of A and B into its own group and then create an inter-communicator between the A-B group and the MPI_COMM_SELF group of C’s leader. This configuration is appropriate when each module uses the “root” process for I/O and the root is responsible for distributing and coalescing the data. The advantage of putting the leaders of A and B into the same group is that we can use MPI wildcards on the receive rather than looping over several inter-communicators. Case (b) is similar to (a), except now leaders coalesce the data but any process in group C can directly receive data from the leaders of A and B and avoid forwarding the data via the root. Case (c) is the most general where any member of A or B can forward data to any member of C. A disadvantage of (c) is that because a new group was created to contain both A and B the rank of a process in the combined A-B group may not to be the same as the rank of the process in A or B. Case (d), unlike (c) where the ranks are with respect to the combined A-B group, the ranks remain the same but will require looping over several inter-communicators. In general, although configuration (a) potentially introduces additional overhead for forwarding, in terms of encapsulation, modules can completely hide their internal communication structure. Routine workflow_Init() can configure either of these types, but all the modules of the workflow must use the same configuration strategy. The details for constructing type (a) configurations are given in Section VI. Construction of the configurations for the other types is very similar to (a).

Similar techniques can be used to create intra-communicators rather than inter-communicators. A potential advantage for intra-communicators is the ability to do collective communications over the group. However, intra-communicators would allow communication between processes in the same module along with the communication between modules, whereas inter-communicators only allow communication between the modules.

C. Protocol Adapters

The final type of routines in the library are the inter-module communication routines. The two most simple types of routines to provide are workflow_Send() and workflow_Recv() (see Figure 5, lines 5 and 12). Users only need to specify the data, data type, and the local intra-communicator. The send routine sends the data on all of the outports and workflow_Recv() returns when it receives data on any of its imports.

These routines need access to the inter-communicators associated with the imports and exports of the process. We use MPI attributes to attach the information to the local intra-communicator, thereby hiding the inter-communicator structures from the user. MPI attributes implement a simple dictionary API that uses a unique key to bind data to a communicator. By adding the appropriate copy and deletion functions, communicators with the attached attributes act exactly like other communicators and can be duplicated, copied and deleted. In implementing the workflow versions of send/receive the workflow simply retrieves the appropriate inter-communicators and calls the MPI communication routine using the inter-communicators.

Although in general this works for simple structures, we also need to ensure that we use a protocol that matches the configuration of the inter-communicators. Thus, if we wish to receive from any of the imports then MPI_Recv() should use a wildcard to receive from MPI_ANY_SOURCE and MPI_Send() needs to loop over all of the outport inter-communicators. We provide a simple set of protocol adapters mirroring the standard MPI_Send() and MPI_Recv() as well as specific ones for simple communication of integers and strings.

The library also provides routines for retrieving and testing inter-communicators. Other MPI routines can be used to query and determine the size and ranks of processes in the remote group. These routines allow users to implement their own customized routines that adapt the protocol to their particular application in cases when the standard communication routines are insufficient.

V. EXAMPLE OF A SIMPLE PROGRAM

As a way of illustrating the use of the system we present a simple parallel pipeline implementing Eratosthenes’ prime number sieve. The first module generates all odd numbers starting from two and each subsequent module keeps the first
number it is given, which is a prime, and passes only those that do not divide by the number it keeps. The result is an output stream of numbers which are not multiples of the $k$ primes in the preceding $k$ modules.

Figure 3 shows an MPICH configuration file for a pipeline of five modules all consisting of the same executable.

```
-n 1 wfmodule -n stage0 -o stage1
-n 1 wfmodule -n stage1 -o stage2
-n 1 wfmodule -n stage2 -i stage1 -o stage3
-n 1 wfmodule -n stage3 -i stage2 -o stage4
-n 1 wfmodule -n stage4 -i stage3
```

Fig. 3. MPICH config file for Prime Number Sieve Pipeline, one process per stage.

The single program called *wfmodule* is used for all the stages and consists of an MPI program with a few added routines. Figure 4 shows the basic setup of the workflow. Routine *getEnvironment()* (line 7) obtains the list of input and output ports from the command line and stores them into the environment structure. The main work is done by *workflow_Init()* (line 11), which takes the communicator for the program and returns a replacement communicator that should now be used for all intra-module MPI communication. The rest of the program remains the same, except that where ever MPI_COMM_WORLD appears the new context should be used instead. As Figure 4 shows very few routines need to be added to a MPI program to make it into a workflow module.

The final call to *workflow_Finalize()* (line 16) frees all the resources attached to the new context. The main part of the program for the prime number sieve is shown in Figure 5. The first stage simply starts a flow of numbers (lines 3–9), the middle stages (lines 10–20) receive a number and forward it when the number is not divisible by *prime*, and the last stage (lines 22–25) simply prints out the stream it receives. A special *stop* value is used to terminate the pipeline.

```
1. while ( TRUE ) {
2.   if ( env.numin == 0 ) { /* no inports */
3.     // First Pipeline Stage
4.     if ( num++ < maxnum )
5.       workflow_Send(&num, MPI_INT, comm);
6.   } else {
7.     workflow_Send(&stop, MPI_INT, comm);
8.     break;
9.   }
10. } else if ( env.numout != 0 ) {
11.   // Middle Pipeline Stages
12.   workflow_Recv(&num, MPI_INT, comm);
13.   if ( num == stop ) {
14.     workflow_Send(&stop, MPI_INT, comm);
15.     break;
16.   } else if ( first )
17.     { prime=num; first=FALSE; }
18.   else if ( num &#8211; prime != 0 )
19.     workflow_Send(&num, MPI_INT, comm);
20.   else /* not a prime */
21.   }
22. } else if ( first )
23.   { prime=num; first=FALSE; }
24. } else if ( num == stop ) break;
25. printf("number is %d\n",num);
26. }
```

Fig. 5. Main part of program with two simple protocol adapters.

VI. CREATING COMMUNICATION CONTEXTS

The routine *workflow_Init()* sets up the appropriate contexts within MPI. As mentioned, the objective of the routine is to create MPI inter-communicators between a set of modules on the “left” that send data to a module on the “right”. In the rest of the discussion we restrict the discussion to the inter-communicator structure between leader processes as shown in Figure 2(a). This can be easily extended to the other three cases.

The basic MPI routine that is used to create the inter-communicators is:

```c
int MPI_Intercomm_create (   MPI_Comm local_comm, int local_leader,
    MPI_Comm outer_comm, int remote_leader,
    int tag, MPI_Comm *comm_out )
```

*MPI_Intercomm_create()* is a collective operation with respect to *local_comm*. The routine uses message-passing in *outer_comm* between the leaders in both groups to create a unique context identifier for the inter-communicator, which is then distributed to the other members of the group. This operation can succeed only when all members of the groups on both sides of the communicator call the operation at the same time.

There is one additional distributed operation used to gather and distribute information between the nodes. In the next section we define a sweep operation, which takes advantage of the acyclic structure of the workflow to setup the necessary structures and parameters needed to create the inter-communicators.
A. Sweep Operation

Given that the underlying workflow is acyclic, the program fragment shown in Figure 6 can be used to make a single sweep of the nodes from sink to source. In particular, the code

```c
for (i=0; i< num_outports; i++) {
    MPI_Recv(&value,1,MPI_INT,
             0,out[i],outer,&status);
    if ( value > level ) level = value;
    level = level+1;
}
```

Fig. 6. Create a topological ordering of the workflow.

fragment in Figure 6 topologically sorts the workflow nodes starting from 0 for the data sinks (nodes with no outports). Lines 5 and 7 are specific to creating a topological ordering of the nodes and these instructions can be replaced to distribute other information in topological order. Similarly, by swapping the imports and outports at lines 2, 4 and 8, 9, we can sweep in the opposite direction. This simple operation is used for distributing or gathering information from a node’s import or export neighbors. The execution time of the operation depends on the depth of workflow, the maximum distance from source to sink or vice versa.

B. Inter and Intra communicator Creation

The creation of intra and inter communicators needs more coordination because of the collective nature of the operations and the need to avoid deadlock. Given a directed acyclic graph \((G,E)\), i.e., a workflow, we define a labeling of the directed edges of the graph where the label of edge \(l(e_i)\) is a number from 1 to \(k\). The labeling must have the following two properties:

1) all incoming edges to a node must have the same label,
2) all outgoing edges from a node must be distinct.

If a labeling satisfies these two conditions, then the two phase algorithm to be described is deadlock free and constructs the inter and intra communicators as illustrated in Figure 2(a). An example of a proper labeling of a directed acyclic graph is shown in Figure 7. Later in Section VI-C we give a coloring algorithm to create a proper labeling of any DAG.

The creation of the inter and intra communicators is performed in two separate phases. In Phase I, the intra-communicators among the leaders are created, and in Phase II we create the inter-communicators. Assume there is a properly labeled graph \(G\) among all of the leader nodes in the workflow. With respect to the labeling of \(G\), before execution each leader process \(u\) has received the following information:

(a) the maximum number of rounds \(\text{maxrounds}\), which is computed by the coloring algorithm in Section VI-C,
(b) \(\text{inlabel}\), the value of the incoming labels to node \(u\), and
(c) an array where \(\text{round}[i]\) is either \(\text{MPI}_\text{UNDEFINED}\), when the node is not active in this round, or the rank of process \(v\) where edge \((u,v)\) is labeled \(i\).

The algorithm for constructing the communicators is shown in Figure 8. In Phase I (lines 2–5), all the necessary intra-communicators are created where \(\text{intracomm}[i]\) is an intra-communicator shared by the collection of processes whose outports point to the same process. As indicated in Figure 2(a), these intra-communicators form the left-side of an inter-communicator with a single leader process on the right-side. In Phase II, the rounds are defined by the labeling so that the appropriate inter-communicator and leader process perform the left and right side parts on the same round thereby synchronizing with each other to create an inter-communicator.

The program is deadlock free because of the properties of the labeling. In Phase I, since all incoming edges \((u,v)\) to process \(v\) have the same label (i.e., active on the same round) they will all execute \(\text{MPI}_\text{Comm}_\text{split}()\) at the same time. Since every outgoing edge has a different label, they do not conflict on a round. When there is no outport with label \(i\), then \(\text{round}[i]\) is \(\text{MPI}_\text{UNDEFINED}\), which indicates that the node is not part of any intra-communicator on that round.

Using the DAG in Figure 7 the creation of the intra-communicators is shown in Figure 9 (Phase I). The brackets

```c
1. // Phase I
2. for (i=1; i<= maxrounds; i++) {
3.   MPI_Comm_split(leadercomm, round[i],0,
4.                  \&(intracomm[i]));
5. }
6. // Phase II
7. for (i=1; i<= maxrounds; i++) {
8.   // left-side of inter-communicator
9.   if ( round[i] != MPI_\text{UNDEFINED} ) {
10.      MPI_\text{Intercomm}_\text{create}(\text{intracomm}[i],0,
11.                                     leadercomm,0, tagv, \&intercomm[i]);
12.  }
13. // right-side of inter-communicator
14. if ( i == inlabel ) {
15.   MPI_\text{Intercomm}_\text{create}(\text{MPI}_\text{COMM}_\text{SELF},0,
16.                                    leadercomm,0, tagv, \&intercommR);
17.  }
18. }
```

Fig. 7. Example of a proper labeling needed for creating inter-communicators.

Fig. 8. Two phase algorithm for creating inter-communicators.
in each column show the nodes which will become a member of a new intra-communicator on that round.

In Phase II we construct the inter-communicator from the right (Figure 8, lines 15-16) and left (lines 10-11) intra-communicators. The labeling ensures that there is a matching between the nodes on the left side with those on the right side. As in Phase I the appropriate nodes are active at the same time. Since an incoming label can equal an outgoing label it is possible that a node must act as both the right and left side of MPI_Intercomm_create(). In this case (Figure 8, lines 9-16), we arrange for the collective communication to occur first on the left and then on the right. This ensures there is always at least one set of nodes that can complete the operation, thus eventually completing the round.

The brackets in Phase II of Figure 9 show the role of the node in constructing the inter-communicator where “L” indicates that it is a member of the intra-communicator on the left and “R” indicates it is the intra-communicator for the right. If there are no inports to a node, then the node will never be on the right-hand side of MPI_Intercomm_create(). A node is on the right in the round corresponding to the label of its imports.

At the end of these two phases we have the desired configuration of contexts to support arbitrary DAG communication. All the imports to a node are part of the same context which is connected via an inter-communicator to the node.

C. Creating a deadlock-free edge labeling

The success of the previous parallel algorithm for creating the different contexts relies on the existence of a proper labeling of the DAG. We show that a proper labeling exists whenever there is coloring of an associated graph, which we call the conflict graph.

Given a DAG $G(V, E)$ the associated conflict graph $C$ is the undirected graph with nodes $V$ where there is edge between two nodes $v, w$ in $C$ whenever there exists a node $u$ in $V$ of $G$ such that $(u, v)$ and $(u, w)$ are in $E$ (i.e., the nodes share a common source).

Given a coloring of $C$ the label of the directed edge $(u, v)$ in $E$ of DAG $G$ is simply the color of node $v$ in $C$. The coloring conditions ensure there is a proper labeling of the DAG.

For example, Figure 10 shows a coloring of the conflict graph associated with Figure 7. Using the previous rule, the graph in Figure 10 can be used to construct the proper labeling shown in Figure 7.

Graph coloring is NP-complete [15], however, there are good heuristics for this problem including distributed algorithms for computing a coloring in parallel [16]. Although a more sophisticated heuristic algorithm could be used, we currently use a simple parallel greedy algorithm based on the fact that every graph can be colored with at most maximum degree plus one colors. At most, this can result in $|V|$ colors, leading to $|V|$ rounds, or a sequential algorithm that linearly constructs the communicator structures one node at a time. The parallel algorithm consists of the following two phases: 

**Phase I:** By swapping the imports and outports in Figure 6 we can sweep through the DAG from the data sources (no inports) to the data sinks (no outports). Each node sends its outport list to each of the nodes in the list. For example, in Figure 7, node 9 sends (4, 7, 8) to nodes 4, 7 and 8. The union of the lists determines a node’s adjacencies in the conflict graph. For example, node 7 receives (4, 7, 8) and (6, 7) implying that 7 is adjacent to 4, 6 and 8 in the conflict graph.

**Phase II:** In the second phase every node performs three actions with respect to the conflict graph. First, in MPI_COMM_WORLD or outer rank order, each node calls MPI_Recv() to obtain a color (integer from 1 to $|V|$) from all adjacent nodes of lesser rank. From the list of colors a node receives, the node chooses the smallest $k$ not in the list. The node then calls MPI_Send() to send $k$ to all nodes of greater rank. For example, in Figure 10, node 7 first receives $k = 2$ from node 4, then $k = 2$ from node 6. The node chooses $k = 1$, and then sends $k = 1$ to node 8. Initially, nodes 1, 3, 5, 9 and 10 are all active and choose $k = 1$.

There are two remaining actions to be performed. Every node needs to know the number of rounds, which is simply the maximum color used. The maximum color is obtained by a MPI_Allreduce() call on the outer context using the MAX operator where each node provides its chosen color as the parameter to the collective routine. Finally, every node must...
communicate the color it has chosen to every node in its list of inports, thus distributing the coloring to complete the edge labeling. All the data is available now to execute the algorithm in Figure 8.

D. Multi-stage Pipelines

One special case of an acyclic structure that can be configured differently is a multi-stage pipeline. We define a multi-stage pipeline to be any DAG that is 2-colorable. If a DAG is 2-colorable, then the longest path from source to sink can be used to label the stages of the pipeline from 1 to \( K \). Figure 1 is an example of a 4 stage pipeline with 2 modules in each stage.

If the previous algorithm is used, then the conflict graph for a multi-stage pipeline consists of \( K \) components that can be independently colored. The two phases for constructing the intra and inter communicators are bounded by the maximum number of colors used in any of the components. Two collective calls are needed to construct each inter-communicator.

There is another technique that can be used in this case which results in fewer collective calls. By using the stage inside a single call to MPI_Comm_split() we can construct an intra-communicator for each stage. After which, the parity of the stages can be used to create inter-communicators between each pair of stages where odd stages are first on the left and then on the right (vice versa for the even stages) and appropriately omitting the ends. The stage inter-communicators can now be used in MPI_Comm_split() to create the new inter-communicators (see [17] for a discussion of creating inter-communicators in this way). MPI_Comm_split() is called first on the odd-even inter-communicator stages and then between the even-odd ones. For each stage inter-communicator we avoid conflicts on the left by peeling off one communicator at a time using each node on the right in turn. This technique does not reduce the overall number of steps but does eliminate some of the collective communication.

VII. Conclusions

Our intent was to provide a simple library that reduces the complexity of implementing a stream-based system. Our goal was the design of an MPI library where each module could itself be highly parallel and the library could take advantage of existing MPI libraries and programs that are available. As an MPI library it will be able to execute on the wide variety of parallel machines that already support MPI. No assumptions have been made about the runtime environment and the library with any of the open-source or vendor-specific runtime environments for MPI.

A major contribution was the use of MPI communicator creation collectives to improve its interoperability with other programs. By using communicators for the inter-module communication, the library can be safely used with other MPI programs and, although the complexity of constructing a properly scoped communication context is hidden, we still allow finer control over the protocol between ports. Where possible we have taken advantage of MPI’s features for creating safe libraries and for encapsulating the complexities of library communication to provide a high level abstract library for stream-based systems.

Future work will continue to extend our collection of protocol adapters. The library, with a few minor modifications, can be used to recursively build up larger stream-based systems from smaller ones. We are currently extending the algorithms to handle more general directed graphs with cycles. Although, not discussed, we added a control system, which is specific to MarketMiner, for initializing a wider variety of environment parameters as well as interactive control over the components.

REFERENCES