The Pilot Approach to Cluster Programming in C

J. Carter, W. B. Gardner, G. Grewal
Department of Computing and Information Science
University of Guelph
Guelph, Ontario, Canada
jcarter@uoguelph.ca
{wgardner,gwg}@cis.uoguelph.ca

***Abstract*—The Pilot library offers a new method for programming parallel clusters in C. Formal elements from Communicating Sequential Processes (CSP) were used to realize a process/channel model of parallel computation that reduces opportunities for deadlock and other communication errors. This simple model, plus an application programming interface (API) fashioned on C's formatted I/O, are designed to make the library easy for novice scientific C programmers to learn. Optional runtime services including deadlock detection help the programmer to debug communication issues. Pilot forms a thin layer on top of standard Message Passing Interface (MPI), preserving the latter's portability and efficiency, with little performance impact. MPI's powerful collective operations can still be accessed within the conceptual model.**

Keywords—MPI; library; parallel programming; deadlock detection; CSP

I. Introduction

The Message Passing Interface (MPI) library [1] has been a popular method of parallel programming for high-performance computing (HPC) clusters for years. Indeed, it is fair to call MPI the mainstay of programmers in the 17 Canadian academic institutions using SHARCNET—Shared Hierarchical Academic Research Computing Network (www.sharcnet.ca)—the consortium with whom this approach is being developed. Popularity notwithstanding, experience shows that training novice programmers to master MPI can be very challenging. While students coming up through a computer science curriculum typically learn something about parallel programming techniques and their hazards—e.g., in operating systems courses—scientific programmers are rarely exposed to such content.

MPI's application programming interface (API) is large and daunting—around 300 functions in the C bindings of MPI 2.1 [1]—and opportunities for misusing the message-passing functions abound. As well, the collective functions are arguably intrinsically confusing: The same function call (e.g., `MPI_Bcast`) has a different effect depending on the caller's identity (MPI "rank"). While this may ease the creation of Single Program, Multiple Data (SPMD) applications, when the code for the ranks is separate, as it may be for the master/worker pattern, the workers execute the "broadcast" function in order to receive data—just the opposite meaning of the function's name. This can be disorienting given the current norms of software engineering.

A common result of either API misuse or faulty program design is deadlock. However, since the deadlocked program is likely to run on uselessly until it exhausts its time limit, detecting the condition, let alone diagnosing its cause, is troublesome for beginning scientific programmers.

The above state of affairs suggests that there is room for a new method of programming HPC clusters that leverages standard MPI while being easier for novice users to understand and utilize. Our C library, called **Pilot,** forms a layer above MPI. It is theoretically based on a few simple abstractions from a classic formal notation, Communicating Sequential Processes (CSP) [2], and presents an API that capitalizes on the minimum knowledge which can be expected of novice C programmers. Parallel programming with Pilot should prove both easier and safer than with MPI alone, yet still provide the benefits of message-passing, the proven scalable mechanism for cluster-based HPC.

The following sections describe related work (Section II), the Pilot API and runtime services (Section III), the library's implementation (Section IV), performance data (Section V), and plans for development (Section VI), which include extending the Pilot approach to Fortran and C++.

II. Related work

Techniques for programming parallel clusters can be classified broadly into two categories: languages and libraries. Pilot falls squarely in the latter, but it is worth noting that, before the advent of MPI, Mazzeo et al. suggested using CSP to program a cluster of heterogeneous workstations [3]. The DISC language was C with CSP-like extensions, and involved a custom parallel compiler, linker, run-time environment, system monitor, profiler, debugger, makefile generator and graphical user interface. DISC computation was performed through a set of processes, in turn composed of events, each representing a step in a calculation. Interprocess communication was performed exclusively via channels, including many-to-one channels. With Pilot, CSP is kept more "under the hood"; for example, "events" are not part of the programmer model.

A number of libraries for concurrent programming, though not necessarily for HPC clusters, essentially provide a tool kit of CSP components so the programmer can implement a process/channel model. Their use is possible without

978-1-4244-6534-7/10/$26.00 ©2010 IEEE

extensive CSP knowledge, but effective and correct usage is facilitated by familiarity with CSP. JCSP [4] was originally invented to improve upon Java's native concurrency constructs, and this approach has been applied to C, C++ [5], and C# [6], as well as other languages. In contrast, Pilot is aimed at novice HPC programmers who could be intimidated by introducing a formal method too explicitly; however, the Pilot library approach does belong to this family.

CSP4MPI [7] also aimed to help programmers avoid communication-related errors by introducing an abstraction layer over MPI. This approach was inspired by the CSP++ [8] software synthesis tool, which generates C++ from a formal CSP specification. However, CSP4MPI requires some user understanding of CSP, and feedback from SHARCNET staff responsible for new user training judged this impractical for the targeted user community.

MPI programmers can presently resort to a third-party package to obtain detection of deadlocks and other programming errors, e.g., inadvertently using up MPI resources. The Umpire tool [9] hooks into an MPI program transparently by means of the MPI profiling layer, so that it receives a record of every MPI function called by the application, including the functions' arguments. Umpire builds and analyzes a dependency graph, and shuts down the program with a diagnostic upon detection of a deadlock. The authors note that the task of deadlock detection for MPI programs is greatly complicated by non-blocking calls, collective calls, and wildcard receives. Hilbrich et al. found that the complications warranted devising a fresh model for MPI deadlocks, called AND ⊕ OR Wait-For Graphs [10]. Their approach, based on sound theory, is implemented within Umpire's framework and has obtained orders of magnitude speed-up for the detection apparatus. Pilot's limited use of MPI functions, and its restricting of collective calls and wildcard receives to specific controlled contexts, mean that Pilot can use simple dependency-based deadlock detection without much elaboration.

Pilot's combination of distinctive characteristics can be summarized as follows: applications programmed using ANSI C (no preprocessor or custom compiler); based on CSP's process/channel model (without exposure to the formal method); inspired by C's formatted I/O for ease of learning; and implemented on top of standard MPI (thus sharing the latter's portability). It is not claimed that individual Pilot *programs* are necessarily portable, since they may be designed to exploit hardware resources on a particular cluster. But the Pilot *library* source code is portable, since it does not use language extensions, does not invoke external libraries, and only relies on a limited set of MPI functions.

III. The Pilot API

Pilot aims to facilitate parallel program design by using high-level abstractions that define a simple parallel computational approach. By shielding the user from dealing with low-level communication issues, such as MPI's tags, communicators, and buffers, programmers are free to concentrate on designing their parallel algorithms. Transferring the algorithms to an implementation based on the same abstractions is a straightforward path. Subsections below introduce the few abstractions that a Pilot programmer is required to learn, followed by the entire API—less than two dozen functions.

A. Abstractions to learn

A Pilot program is based on two simple abstractions: *process,* a locus of execution; and *channel,* the sole means of interprocess communication. Readers familiar with process algebras such as CSP [2] and Pi-calculus [11] will recognize these at once.

A Pilot process is equivalent to an MPI process, that is, a typical "process" in operating system terms, with its own address space and thread of execution. A process is created by associating it with a C function, and arguments may be passed at creation time much like the POSIX threads function `pthread_create`. Just as with MPI programs, the code for all processes is typically packaged into a single executable file, copies of which are run on each node. However, the code for Pilot processes is normally disjoint, residing in individual functions, though it is permissible for any process function to serve multiple Pilot processes (the instances being distinguished by different argument values). This contrasts with the MPI practice of interleaving the code of all processes into one function, and controlling them by means of rank-dependent conditionals. It may be more accurate to say that Pilot is a Multiple Program, Multiple Data (MPMD) approach in the guise of SPMD. Keeping the source code for each process distinct can make for better software quality factors (e.g., readability and maintainability), the trade-off being a bias toward master/worker organizations.

A Pilot channel has three characteristics: (1) point-to-point, being bound to two processes at creation time; (2) one-way, such that one process writes to the channel and the other process reads; (3) synchronous, so that execution of the reader and writer only proceed after the message is passed; (4) not typed, so that a given channel may be used to communicate any type of data. This design choice was made to avoid the proliferation of channels that would occur if they were strongly typed. Thus, the programmer works with a model of synchronous communication, but "synchronous" is not as strict as it sounds. The underlying `MPI_Send` operation may only block until the message is on its way.

There are benefits to restricting interprocess communication to defined channels. With MPI, any process can potentially message any other process. If a message is sent to a process that was not planning to receive it, or a coding error is made in a rank, tag, or communicator argument, runtime errors or even system deadlock can occur. An analogy can be made to UML: Class diagrams show defined relationships between classes as lines. Collaboration diagrams are more specific, depicting the flow of messages between objects. However, unless certain Computer-Aided Software Engineering (CASE) tools are used, these defined relationships present in the design are not enforced in the code. In practice, any object can invoke any other object's public methods. Since Pilot enforces interprocess communication integrity via channels, one category of parallel program logic and coding errors is avoided. Fig. 1(a) illustrates simple mas-

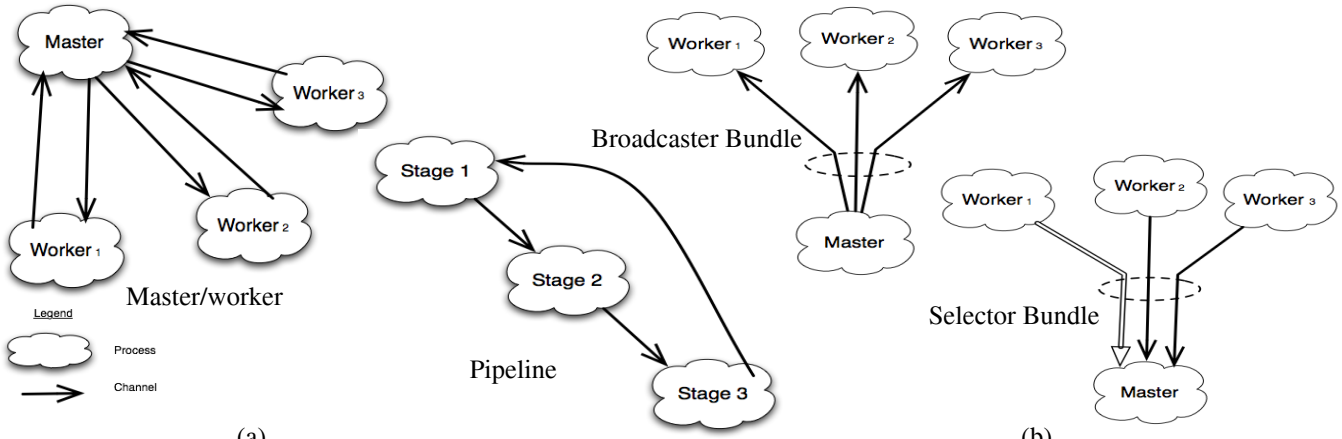

Figure 1. Pilot process/channel architectures.

ter/worker and pipeline architectures using processes (clouds) and channels (arrows).

There is one more abstraction for a Pilot programmer to learn: the *bundle*. This construct represents a group of channels that have a common endpoint. The purpose of introducing bundles is to allow the library to access MPI's efficient collective operations, specifically broadcast and gather, without breaking the theoretical process/channel model. (Pilot support for other commonly-used collectives operations, scatter and reduce, is planned for the future.) Thus, a broadcaster bundle is a group of channels with a common write process, while a gatherer bundle has a common read process. A bundle is created after first creating all its constituent channels.

Another type of bundle, a selector, also has a common read process. It is useful in master/worker architectures where the master wishes to know which worker has a result to read. This is the same sense of "select" as in the standard C library, whereby a program can learn which file descriptor (representing, say, a group of sockets) is ready to read before committing to a blocking read operation. This feature of Pilot echoes the ability of MPI programmers to obtain "nondeterminism" by means of wildcard receives. Fig. 1(b) illustrates a broadcaster and a selector bundle (the white arrow shows Worker_1 ready to read).

B. API function calls

A Pilot program has two distinct phases: (1) configuration, where all processes, channels, and bundles are created in `main()`; and (2) execution, where processes start their individual lives and carry out communication.

1) Configuration phase: During this phase, all processes are actually executing, in SPMD style, the same code in `main()`, with the result that, upon entering the execution phase, all have built up the identical tables of process, channel, and bundle definitions, irrespective of their various node architectures. There is no need for one node to send the tables to the other nodes.

A Pilot program initially calls `PI_Configure`, just as an MPI program calls `MPI_Init`:

```
int PI_Configure(int *argc, char ***argv);
```

Pilot checks for and removes any command line arguments starting with "`-pi`...", and passes the rest to `MPI_Init`. Pilot arguments are used for selecting runtime options, such as error-checking level, deadlock detection, and logging. This function returns the maximum number of Pilot processes that can be created (equal to the `MPI_COMM_WORLD` size). The program may use this information to adjust the number of Pilot processes it goes on to create.

Processes are created using the following function:

```
PI_PROCESS* PI_CreateProcess(func,index,hook);
```

This associates a Pilot process with a C function having the following prototype:

```
int func(int index, void *hook);
```

In the case where a single function serves multiple Pilot processes, the `index` argument, say, an array index, may be used to distinguish the instances. The `hook` argument can be used to pass arbitrary data to the function.

This is not a "fork"; the process function is not executed until the execution phase (see below).

Process, channel, and bundle creation functions all return a pointer to an opaque datatype. `PI_PROCESS*` pointers are not needed after the configuration phase, so they may be kept in variables of any storage class. However, `PI_CHANNEL*` and `PI_BUNDLE*` pointers *are* needed, so must be kept in storage accessible by both `main()` and process functions, or, alternatively, passed to process functions via a `hook` data structure.

Pilot processes are mapped to finite resources, MPI processes, but channels are a software abstraction. They have no practical limit and need not be "conserved" by the programmer. The channel creation function fixes a channel's endpoints and direction, which cannot be changed:

```
PI_CHANNEL* PI_CreateChannel(from, to);
```

The `from` and `to` arguments are both `PI_PROCESS*` variables. The main/master process is designated by the symbol `PI_MAIN` (which simply equates to zero). One may create multiple channels between a given pair of processes, in the same or opposite direction.

The three types of bundles, for three distinct purposes,

are created from arrays of PI_CHANNEL* pointers having a common endpoint:

```
PI_BUNDLE* PI_CreateBundle(usage,chanarray,N);
```

where usage is one of PI_BROADCAST, PI_GATHER, or PI_SELECT. The chanarray argument is an array of type PI_CHANNEL*[N]. These functions verify that the channels in selector and gatherer bundles have a common read end, and that broadcaster channels have a common write end. A further restriction is that no channels have duplicate processes at the other end of the bundle.

After the configuration phase is complete and the execution phase is entered, no further creation of processes, channels, or bundles is allowed.

2) Execution phase: This phase is triggered by calling the following function:

```
int PI_StartAll(void);
```

At this point, each Pilot process starts executing its associated function. In main(), where PI_StartAll returns to, the subsequent code becomes the "master" process, with MPI rank 0, also known as PI_MAIN.

During this phase, processes may freely engage in channel reading and writing. The syntax of these operations has been specially designed to utilize constructs that novice C programmers are sure to know, and thus able to quickly grasp. They are inspired by fprintf and fscanf formatted I/O. Furthermore, the PI_CHANNEL* and PI_BUNDLE* variables of Pilot work like C's FILE* variables. This is simply an idiom for specifying data types; it does not imply that messages are being sent as text.

The two basic channel communication functions are:

```
void PI_Write(chan, format, vars...);
void PI_Read(chan, format, vars...);
```

The chan argument is a PI_CHANNEL* pointer. There is no need to specify the origin or destination processes; they are already bound to the channel. Pilot will verify that the calling process is indeed bound to the referenced channel and is acting in its predefined role as reader or writer.

Just like C formatted I/O, the format string is made of individual formats with the pattern: %*n*T. T is the datatype, using a subset of scanf's designations (see Table I). "l" stands for long, and "h" for "half" (implying short).

Non-scanf additions are very few. "b", standing for byte format, transfers uninterpreted 8-bit bytes. It is suitable for sending (arrays of) structures among homogeneous nodes, where there are known to be no endian, word length, sign convention, floating point format, or character code differences. The IMB benchmarks (see Section V) rely on this datatype. For heterogeneous clusters, where C structures may require conversion assistance from MPI, advanced users may construct user-defined datatypes and send them using the special "m" format. "%m" requires a variable of type MPI_Datatype as the next argument in order, followed by a data pointer argument; i.e., "%m" consumes two arguments. The Pilot API does not directly support the *construction* of user-defined datatypes; it is up to the user to call the necessary MPI functions directly, e.g., MPI_Type_struct and MPI_Type_commit.

TABLE I. PILOT FORMATS VS. C AND MPI DATATYPES

Pilot format	C datatype	MPI datatype
%c	char	MPI_CHAR
%hhu	unsigned char	MPI_UNSIGNED_CHAR
%d, %i	int	MPI_INT
%hd	short int	MPI_SHORT
%ld	long int	MPI_LONG
%lld	long long int	MPI_LONG_LONG
%u	unsigned int	MPI_UNSIGNED
"u" gives corresponding unsigned integer types: hu, lu, llu		
%f	float	MPI_FLOAT
%lf	double	MPI_DOUBLE
%Lf	long double	MPI_LONG_DOUBLE
%b	any	MPI_BYTE
%m	user-defined	MPI_Datatype variable

The optional *n,* a quantity in the same position as printf's "field width," specifies the number of elements in the datatype. If omitted, the data argument is treated as a scalar value. An array may be specified either by hard-coding a literal length, e.g., %250f, or by coding an asterisk, %*f, to supply the length from an argument. (This "*" is a printf field width convention.) For readability, whitespace may be inserted into the string between individual formats. The format argument is simply of type const char* and so may, of course, be supplied from a char array as well as from a string literal.

As with printf and scanf, the sequence of formats drives the interpretation of the variable arguments. Pilot checks that the number of arguments matches the format string. For PI_Write, each argument supplies a scalar value or array address. (Note that "%m" always requires an address.) For PI_Read, each argument must be an address. However, any "*" causes the next argument, in order, to be interpreted as an array length. For example,

```
PI_CHANNEL *data; float x[200]; int k[300];
PI_Write(data,"%200f %*d %c",x,100,k,'w');
```

would send a 200-element float array from x followed by 100 int elements of k, ending with the character "w". This example shows all three length forms: fixed-length array, variable-length array, and scalar.

The above write must be paired with a single PI_Read call. Two of the possible matching reads are shown here:

```
PI_CHANNEL *data;
float input[200]; int num[300], n=100; char c;
PI_Read(data,"%200f %*d %c",input,n,num,&c);
PI_Read(data,"%*f %100d %c",200,input,num,&c);
```

Just as with fscanf, the scalar receptacle c needs an "address of" operator, which the array receptacles input and num do not (since in C the name of an array resolves to the address of its first element). In the first read, the length for %200f is an integer literal, while the length argument

TABLE II. FUNCTION CALLS FOR USE WITH BUNDLES

Process at common end of bundle will call	Multiple processes at other end of bundle will call
PI_Broadcast	PI_Read
PI_Gather	PI_Write
PI_Select & PI_Read	PI_Write

for %*d is supplied in the variable n (100).

This sense of "variable-length array" does not mean the reader is absolved from knowing the incoming length in advance; it may need to be sent in an earlier message. Variable-length array means that the array length can be supplied as an argument rather than hard-coded in the format.

The above examples are for basic writing and reading on channels. One additional function is available for polling a channel's read status. It returns true if the channel is ready to be read (i.e., PI_Read would not block).

```
int PI_ChannelHasData(chan);
```

For collective operations on bundles, the big departure from MPI is that only the process at the common end of the bundle uses the special "collective" function call, while the processes at the bundle's other end use the naturally symmetric PI_Read or PI_Write, as shown in Table II. The argument lists for PI_Broadcast and PI_Gather are the same as for PI_Write and PI_Read.

For example, here is the code for the "master" process (PI_MAIN). It creates two groups of N channels to/from the N worker processes, then fashions them into two bundles:

```
PI_PROCESS *work[N];
  // PI_CreateProcess calls not shown
for (int i=0; i<N; i++) {
  data[i]=PI_CreateChannel(PI_MAIN,work[i]);
  result[i]=PI_CreateChannel(work[i],PI_MAIN);}
PI_BUNDLE *thedata =
  PI_CreateBundle(PI_BROADCAST, data, N);
PI_BUNDLE *allresults =
  PI_CreateBundle(PI_GATHER, result, N);
PI_StartAll();
```

Now, the main process sends the same 100 coefficients to N workers, and then gathers the results as N doubles:

```
float coeffs[100];
double z[N];// room for N results
PI_Broadcast(thedata, "%100f", coeffs);
PI_Gather(allresults, "%lf", z);
```

The workers on the bundles' other ends simply call the usual read/write functions:

```
float coef[100]; double ans;
PI_Read(data[mynode], "%100f", coef);
    // ...calculate...
PI_Write(result[mynode], "%lf", ans);
```

The last function to be introduced is only for use with a selector bundle:

```
PI_PROCESS *work[N]; // create not shown
PI_CHANNEL *result[N];
for (int i=0; i<N; i++)
  result[i]=PI_CreateChannel(work[i],PI_MAIN);
PI_BUNDLE *workers =
  PI_CreateBundle(PI_SELECT, result, N);
...
int w = PI_Select(workers);
PI_Read(result[w], "...", ...);
```

Here, PI_Select is used to interrogate the selector bundle coming from the worker processes. This call will block until at least one channel is ready to read. Its index (in the array originally used to create the bundle), returned from PI_Select, is used here in the subsequent PI_Read.

If multiple channels become ready to read before PI_Select is called, MPI will queue the arrivals. PI_Select will indicate the channel whose message arrived first. Successive calls to PI_Select/PI_Read will receive the messages in order.

Analogous to PI_ChannelHasData, a non-blocking version of PI_Select is available as PI_TrySelect. It returns the same channel index as PI_Select, or -1 if no channel in the selector bundle was ready to read.

Two helper functions are available for use with selector bundles. The first retrieves the number of channels in the bundle. The second can be used in lieu of looking up a channel in the array that was used to create the bundle.

```
int PI_GetbundleSize(bund);
PI_CHANNEL* PI_GetbundleChannel(bund,index);
```

3) Utility functions: A common requirement in master/worker patterns is to have one set of channels going to the workers, another set coming back, and perhaps additional sets for broadcast or gather use. In order to reduce coding, a function is provided to "copy" an array of channels. That is, another array of PI_CHANNEL* pointers will be produced having the same endpoints. An option allows the directions to be reversed, making it convenient to create the channels needed for, say, a gatherer bundle after already creating those for a broadcaster bundle.

```
PI_CHANNEL **PI_CopyChannels(direction,
  chanarray, N);
```

where direction is PI_SAME or PI_REVERSE.

There are a small number of additional functions that do not relate to channel I/O. The first pair allows a process to set or get an optional global-scope alias that can be attached to any process, channel, or bundle, mainly used for diagnostic printouts and logging:

```
void PI_SetName(object,"alias");
const char *PI_GetName(object);
```

Here object can be any PI_PROCESS*, PI_CHANNEL*, or PI_BUNDLE* pointer. In the absence of aliases, Pilot will print diagnostics using default names like "P3" for the pro-

cess of MPI rank 3, or "C4" and "B2" for the fourth channel and second bundle created, respectively.

There are two functions for timing, `PI_StartTime` and `PI_EndTime.` Finally, `PI_Abort` provides a clean way for any process to abort the program.

4) Program termination: When individual process functions are finished, they need only return, which results in their calling `exit(0)`. The integer return value has no effect, but will be entered in the log. The `main()` function, on the other hand, must call `PI_StopMain(status)`. This synchronizes all processes on a barrier, after which `MPI_Finalize` is called, and then Pilot's internal tables are deallocated. The application can have other code following `PI_StopMain`, but no further Pilot calls can be made.

C. Programmer support

A major goal of Pilot is to provide support for the novice programmer—in terms of monitoring, analysis, debugging—that is mostly absent from MPI. The features are calibrated as to the overhead required, and selectable (via command line option or global variable `PI_CheckLevel`) so as to burden program performance with only those desired at any one time. It is expected that programmers would run with Level 1 checking or higher until confident of their code, and then switch to Level 0 for production runs.

Level 0 validates function preconditions that can be trivially checked for usage errors. Level 1, the default, in addition validates internal tables (which may have been corrupted, say, by misuse of dynamic memory) and performs more time-consuming checks.

All Pilot errors are fatal, and result in a diagnostic message being output on `stderr` that pinpoints the source file name and line number where the user program called a library function. In the event of an MPI error, the location of the library call is reported. In the future, additional checks will be provided in a Level 2, such as verifying that read/write formats match their arguments lists.

In addition to error diagnosis, Pilot aims to provide monitoring and reporting features (see Future Work in Section VI). The first feature available is deadlock detection, described next.

D. Deadlock detection

A user may enable the deadlock detection service by specifying the command line option "`-pisvc=d`". The current implementation consumes one MPI process. Deadlock detection was designed to be integrated with a range of Pilot runtime services, including logging of calls to Pilot functions. As of Version 1.1, deadlock detection has been implemented, and the logging infrastructure is in place. The current implementation is not claimed to be efficient, but has the advantage of not depending on third-party packages, and does not need to interface with the MPI profiling layer.

When deadlock detection is running, each Pilot communication function sends an event message to the log via MPI, which is received by the detection process. It records each event in a dependency matrix based on the semantics of CSP channel communication. That is, if process P writes to process Q, a write dependency is inserted. When Q reads from P, the new read dependency satisfies the write, and the dependencies are cleared. Collective operations, e.g., `PI_Select` and `PI_Broadcast`, record multiple dependencies. A selection dependency is special, because it can be satisfied by any write on the related bundle channels; however, the write dependency remains, because it must be satisfied by an explicit channel read. Depending on various factors, such as message length, `MPI_Send` can unblock after dispatching the message from the user's buffer, but before the message is received. Thus, it is possible that a writing process—which, according to CSP semantics ought to be blocked—may continue on to generate another communication event before the first one is formally completed by a receive event. Such "eager" events are queued for processing in order after the earlier dependency is satisfied.

When a deadlock is detected, the detection process prints on `stderr` a record of what each deadlocked process was doing, down to the file name and line number of the Pilot API call, and then aborts execution. Armed with this information, the programmer can track down the cause. Diagnosis is categorized by deadly embrace (e.g., P writes to Q, but Q reads from P on a different channel), circular wait (P reads from Q, Q writes to R, R reads from P), "dead" wait (e.g., P reads from Q, before or after Q exits without writing on the channel), and vain select (P selects on a bundle of channels, but all are/become blocked or exit and cannot write to P). Some of these cases are trivially detected when the communication event is processed. Circular wait is checked for any time a new dependency is inserted.

IV. Implementation

It is acknowledged that MPI is excellent for providing message-passing infrastructure on high-performance clusters, has a standard API, and widespread availability on a multitude of architectures. Thus, there was no obvious advantage in replacing MPI, and no desire or need for Pilot to "reinvent the wheel." In OSI network parlance, MPI is taken as the "transport layer" (layer 4) and Pilot builds a higher-level protocol on top of it. Pilot would be analogous to a combined "presentation" and "session layer" (layers 5 and 6), and, in particular, its channels could be viewed as the "connections" of layer 5. Describing its implementation involves detailing how Pilot utilizes MPI.

Processes are equivalent to MPI processes, therefore each Pilot process is assigned an MPI rank, up to `MPI_COMM_WORLD` size, with 0 reserved for the continuation of `main()` after `PI_StartAll`. If this size is exceeded, an error is issued. A data structure for each process records its rank, default name or alias (from `PI_SetName`), associated function pointer, and the two arguments from `PI_CreateProcess`. `PI_StartAll` dispatches control to the appropriate function according to its rank number, or simply returns to `main()` if running as rank 0.

Channels are an abstraction that represent resolution of a write in one process to a matching read in another process. In MPI, a physical message is associated with a rank, a tag, and a communicator or [R,T,C]. In terms of constraints, R is

unique within a given C, while T is effectively global across all communicators. A reader may wildcard R and/or T, but C must be specified. Collective operations only specify C (not T) and apply to all ranks (with one designated as the "root"). Given the above, Pilot's strategy for implementing channel operations is as follows:

- When a channel is created, its associated endpoint processes are recorded.
- *Point-to-point channel read/write:* The pair of processes know each other's ranks, so they can simply address them (with C=`MPI_COMM_WORLD`). Since multiple channels can exist between the same pair of processes, Pilot assigns tag numbers as channels are created, so that each channel has its own tag. `PI_ChannelHasData` uses a tag with `MPI_Iprobe`.
- *Selector bundle:* `PI_Select` and `PI_TrySelect` are implemented via `MPI_Probe` and `MPI_IProbe`, respectively, with C=`MPI_COMM_WORLD` and mandatory wildcard for R. A tag is necessary to prevent Select from picking up a write from any process with channels outside the bundle. Pilot assigns the same tag number to all the channels in a selector bundle. Ambiguity is prevented by disallowing multiple channels from the same process in one bundle.
- *Broadcaster and gatherer bundles:* These groups of channels are each assigned their own communicator for use with `MPI_Bcast` and `MPI_Gatherv`. A `PI_Write` or `PI_Read` on the non-common end of a bundle calls the relevant MPI collective function instead of simple send/receive.

MPI implementations must support a 15-bit tag, so a minimum of 32,767 are available. Pilot reports the program's quantities of created processes, channels, and bundles.

V. Performance

Since Pilot forms a layer on top of MPI, superior performance is not a goal, but Pilot should not degrade performance with excessive overhead. Our approach for assessing Pilot's performance is to take the Intel MPI benchmarks (IMB 3.1) [12] and "Pilotize" the ones that can be implemented using Pilot's API. So far, this has been done for the pingpong benchmark, preserving all the original IMB code, changing only `MPI_Send` to `PI_Write` and `MPI_Recv` to `PI_Read`. Because of the architecture of the IMB suite, it is necessary to call `PI_Configure` after `MPI_Init` has been invoked. This sequence is detected by `PI_Configure` and automatically puts Pilot into a "bench mode" whereby `PI_StopMain` does not call `MPI_Finalize` as usual, thus permitting IMB to change the message size and number of iterations, and do another round of timing.

Message sizes and iterations are specified in the original IMB code. In Fig. 2, the time for a complete round of iterations is shown against message sizes on the X axis. The data type is `MPI_BYTE` (Pilot %b). The number of iterations for each data point starts at 1000, and begins to decrease (by halves) as the message size passes 2^{15}. With Pilot's level 0 of checking, running on an HP Opteron (2.2 GHz) cluster with Myrinet-2G (GM) interconnect, using HP Linux XC 3.1 and

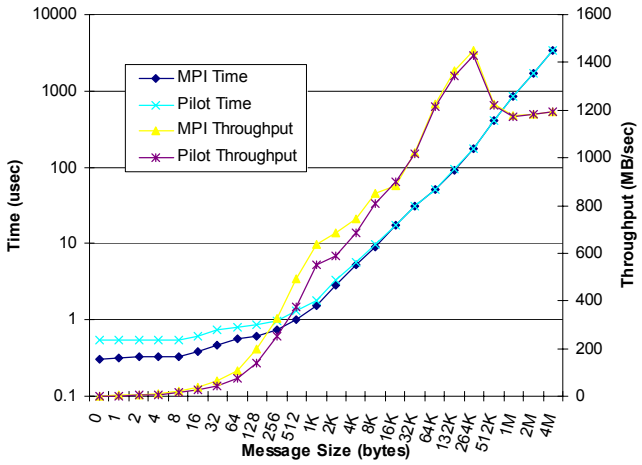

Figure 2. IMB 3.1 pingpong timings and throughput

HP-MPI, it is clear that the time for any Pilot round is only slightly more than for pure MPI, with more noticeable overhead for smaller message sizes, as would be expected. The other pair of lines provide the effective throughput in MBytes/sec. Again, the Pilot throughput is barely below the pure MPI line. In future, the rest of the applicable IMB benchmarks will be converted to Pilot and tested.

In conjunction with a graduate course, Pilot has been used to build parallel cluster applications. Takeva-Velkova demonstrated image reconstruction from parallel magnetic resonance imaging (MRI) [13]. Girard implemented parallel scatter search in both MPI and Pilot to compare their performance on the same hardware [14]. The results bore out what the benchmark data showed, no significant overhead, at least for that algorithm.

VI. Availability and Future Work

This paper describes Version 1.1 of Pilot. The software is copyright by the University of Guelph. The public source code release can be configured and installed onto a variety of platforms without any license fee. Aside from an installation of MPI, no other library is needed to install and use Pilot. See **http://www.carmel.cis.uoguelph.ca/pilot** for current availability, including documentation and training material.

In the near term, future work will concentrate on enhancing Pilot's ability to detect inadvertent usage errors (e.g., mismatch of formats and arguments), and on giving the programmer visibility into the complex world of a parallel cluster application via monitoring and reporting features: statistics and/or logging of Pilot library calls, a trace of completed read/write events (which could be used for formal analysis), and a map of process/channel topology. We plan to add support for `MPI_Scatter`, and investigate how to support reductions (e.g., `MPI_Reduce`). Since Pilot is based on CSP, it should be possible to carry out formal verification on a Pilot program, and uncover potential deadlock scenarios without actually running it. Such work has been carried out for MPI programs—for example, Vo et al. [15]—so the Pilot case ought to be simpler.

We also are interested in making it easier for programmers to apply parallel design patterns, for example, pipelines

that "shift" without accidentally deadlocking, or networks of processes and channels organized into grid or graph topologies. An experimental Fortran API for Pilot, based on the ISO_C_BINDING module, is undergoing trials. There is likely use for a C++ object-oriented API, as well.

Fundamentally, in its concept and initial release, Pilot is aimed firmly at the "novice programmer" user group. This group, in particular, may be quite willing to sacrifice some performance if less programming complexity is obtained, as shown by the use of Java on clusters for the sake of speeding up deployment. If it becomes evident that Pilot is attracting attention as a favourable alternative to pure MPI programming, features can be added to satisfy more advanced users by giving indirect access to more powerful MPI functions, while respecting the formal process/channel model. This approach is already illustrated with the bundle construct, and the ability to utilize user-defined datatypes. Ultimately, we wish to see development driven by HPC users' needs and not for its own sake.

VII. Conclusion

One can make the analogy that Pilot is to MPI as high-level languages are to assembly code. Using its simple process/channel model, programmers can design their algorithms using higher-level abstractions and then transfer them directly to Pilot library constructs. In this way, Pilot aims to be an "easier" means of parallel programming than pure MPI, which is borne out by its API being less than one-tenth the latter's size. Pilot is also "safer," both due to its channel basis inherently ruling out a category of program errors that can be committed using pure MPI, and due to its attention to diagnosing usage errors and detecting deadlocks. For these reasons, Pilot can be useful in teaching introductory courses on parallel programming.

It can be argued that advantages similar to those offered by Pilot could be obtained by simply designating a small, "safe" subset of MPI functions, and teaching only those. The authors feel that this approach is inferior, because (1) trainees do not receive a conceptual model for organizing their parallel program designs; (2) they have to deal with tags and/or communicators in order to obtain any of the collective functionality that Pilot gives without them; and (3) they are sure to encounter mysterious problems which Pilot would have diagnosed.

For some in the target user community—novice scientific programmers—Pilot training may be all they need to accomplish their parallel programming objectives. If others need more power, control, or access to dynamic features of MPI-2, then when they go on to learn MPI, their experience with Pilot's message passing should help them grasp how MPI is used, just as learning a high-level programming language prepares one to understand what assembly language is all about.

Acknowledgment

This research is supported by grants from Canada's Natural Science and Engineering Research Council (NSERC) and by a SHARCNET fellowship.

References

[1] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard Version 2.1 [online]. September 2008 [cited 07/17/2009]. Available from: http://www.mpi-forum.org/docs/mpi21-report.pdf.

[2] C.A.R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, 1978.

[3] S. Russo A. Mazzeo. and G. Ventre. Using CSP languages to Program Parallel Workstation Systems. *Future Gener. Comput. Syst.*, 8(1-3):149–163, 1992.

[4] Nan C. Schaller, Gerald H. Hilderink, and Peter H. Welch. Using Java for parallel computing: JCSP versus CTJ, a comparison. In P.H. Welch and A.W.P. Bakkers, editors, *Communicating Process Architectures*, pages 205–226. WoTUG, IOS Press, 2000.

[5] Neil Brown. C++CSP2: A many-to-many threading model for multicore architectures. *Communicating Process Architectures 2007*, pages 183–205, 2007.

[6] Alex A. Lehmberg and Martin N. Olsen. An introduction to CSP.NET. In Peter Welch, John Kerridge, and Fred Barnes, editors, *Communicating Sequential Architectures 2006*, pages 13–30. IOS Press, 2006.

[7] J. Carter and W.B. Gardner. A formal CSP framework for message-passing HPC programming. *Proceedings of Canadian Conference on Electrical and Computer Engineering (CCECE)*, pages 1466–1470, May 2006.

[8] William B. Gardner. Converging CSP specifications and C++ programming via selective formalism. *ACM Trans. on Embedded Computing Sys.*, 4(2):302–330, 2005.

[9] Jeffrey S. Vetter and Bronis R. de Supinski. Dynamic software testing of MPI applications with Umpire. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 51, Washington, DC, USA, 2000. IEEE Computer Society.

[10] Tobias Hilbrich, Bronis R. de Supinski, Martin Schulz, and Matthias S. Müller. A graph based approach for MPI deadlock detection. In *ICS '09: Proceedings of the 23rd international conference on Supercomputing*, pages 296–305, New York, NY, USA, 2009. ACM.

[11] R. Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, 1999.

[12] Intel. *Intel MPI Benchmarks, Users Guide and Methodology Description*. Intel GmbH, Germany, 2004.

[13] Viliyana Takeva-Velkova and Dhavide Aruliah. Image reconstruction from parallel MRI using Pilot. In *SHARCNET Research Day*, University of Waterloo, May 21, 2009.

[14] Natalie Girard and G. Grewal. Comparison of Pilot and MPI implementations of parallel scatter search. In *SHARCNET Research Day*, University of Waterloo, May 21, 2009.

[15] Anh Vo, Sarvani Vakkalanka, Michael DeLisi, Ganesh Gopalakrishnan, Robert M. Kirby, and Rajeev Thakur. Formal verification of practical MPI programs. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 261–270, New York, NY, USA, 2009. ACM.