Using the Pilot Library to Teach Message-Passing Programming

William B. Gardner and John D. Carter
School of Computer Science
University of Guelph, ON, Canada
{gardnerw,jcarter}@uoguelph.ca

***Abstract*—Message-passing is the staple of HPC codes, and MPI has long occupied the place of HPC's default programming paradigm, thus it would seem to be the natural choice for instructing undergraduates. Nonetheless, MPI is a low-level API, complex and tricky to use, with many pitfalls awaiting the inexperienced. The Pilot library was invented as an alternative HPC programming model for C and Fortran. Pilot-based codes, using a process/channel application architecture borrowed from Communicating Sequential Processes (CSP), can avoid some categories of errors, and the Pilot library with its integrated deadlock detector provides extensive checking and diagnosis of usage problems, which is especially important for students running cluster programs in their typical low-visibility environment with limited debugging tools. This paper gives an overview of programming in Pilot, with its compact API of point-to-point and collective operations. It explains reasons for preferring it as an introductory message-passing technique, describes free resources available to the instructor, and relates experiences of using Pilot with undergraduates over five years, including student reactions. Pilot is now available as free and open source.**

Keywords-Parallel programming; MPI; high-performance computing; CSP; deadlock detection; novice programmers

I. Motivation

Educating students about high-performance computing (HPC) will typically involve training in message-passing programming. The alternative is to utilize a higher-level language or library that hides the message communications, such as Hadoop MapReduce, or a Partitioned Global Address Space (PGAS) language like Unified Parallel C or Co-array Fortran. But these are specialized techniques, whereas message-passing is as fundamental to non-shared memory HPC as global variables (and the methods of preventing corruption and conflicts) are to shared-memory computing. Therefore, it is hard to imagine that HPC education could leave aside message-passing programming.

In a typical North American undergraduate context where students have learned C, C++ and/or Java, either C or C++ gives them immediate access to the standard MPI library [1]. It is more difficult to justify Java for this purpose because (a) presently there does not seem to be a widely-accepted message-passing library for Java on par with MPI, and (b) passing data structures commonly used in HPC such as multi-dimensional arrays is difficult to do efficiently. Consultation with our Southern Ontario HPC consortium, SHARCNET, revealed that the vast majority of codes are in C, with Fortran bringing up second place (stemming largely from legacy codes). Thus, teaching HPC programming in C (or C++) using MPI would appear to be the obvious choice. Textbooks such as the one we use for the Parallel Programming course at the University of Guelph have a chapter on MPI [2].

And yet, for beginning HPC programmers, diving right into MPI presents a number of challenges that educators would do well to note:

1. The standard API is vast and complex. MPI-1 alone has over 120 functions, and MPI-2 brought the total to over 500. One fundamental operation such as MPI_Send can have many subtle variants: Bsend, Ssend, Rsend, Isend, Ibsend, Issend, and Irsend [3]. Students need careful guidance in sticking to a safe and relevant subset of functions to stay out of trouble. For example, McGuire [4], citing MPI's multiplicity of functions, "more than any one person will likely learn and use," with "typically long and relatively complicated" parameter lists, put a simplified interface onto eleven chosen MPI functions. Some have identified only six "golden" functions as essential [5], and placing such restrictions can counteract this challenge.

2. MPI confusingly mixes two programming models, SPMD (single program, multiple data) and MPMD (multiple programs, multiple data). This is responsible for programmers' peppering of codes with conditionals like "if (rank==0)" and for counterintuitive use of collective functions. Students legitimately ponder why a broadcast *receiver* must call MPI_Bcast, and so on.

3. MPI messages are addressed using combinations of ranks, tags, and communicators, and it's up to the programmer how to utilize them in a given problem. Coding mistakes and logic errors can result in messages going astray to processes that are not expecting them, with, at best, some possibly useful runtime error message, and at worst, mystery deadlocks. MPI's "cryptic error messages" and "lack of stable and useful debugging tools" were cited in 2001 as barriers to training inexperienced programmers [6].

4. Deadlock causes the job to hang, wasting cluster resources while running out its time limit, or putting students in a dilemma: is it just running longer than expected, or should it be killed, likely leaving no evidence for what caused the deadlock? In principle, third-party deadlock checkers such

as Umpire [7] are available, but our experience is that students tend to resist using additional tools with their own learning curves (tool resistance issue raised re MPI debugging [8]).

5. At program completion, MPI may not disclose that there are undelivered messages, yet they point to faulty design and/or logic errors that should be corrected.

It was in recognition of the above pitfalls that the Pilot library [9] was invented. It was intended to be, first and foremost, an educational tool to introduce students to message passing, adequate for assignments and projects on HPC clusters. For many students who go on to careers where multicore computers are ubiquitous, Pilot will have provided a sufficient exposure to the principles and practice of message passing, and will have served as a foil for comparing and contrasting that approach with shared-memory programming techniques. For others who go on to write or maintain "production codes" in HPC shops, Pilot will serve as a convenient stepping-stone to MPI programming. Pilot's concepts are readily transferable to MPI, since it represents a subset of what one can accomplish with MPI.

Pilot is not a *de nouveau* message passing library. On the contrary, it is a thin layer on top of conventional MPI; that is, it uses MPI as the transport mechanism. Since MPI is a standard, this makes Pilot very portable and able to work with any underlying MPI implementation that may happen to be installed on one's cluster. Pilot itself is written in C, but it also has a Fortran API that calls the C functions via the ISO_C_BINDING module. The next sections give an overview of the Pilot library and a sample of how to use it. At the end, we will return to the above five challenges and explain how Pilot addresses them.

II. Overview of the Pilot Library

Pilot is not an ad hoc simplification of MPI. Rather, it was intentionally based on Hoare's process algebra Communicating Sequential Processes (CSP) [10], where computation is specified in terms of *processes* that do not share memory, and communicate strictly in terms of synchronous messages over unidirectional *channels*. Imposing this kind of theoretical framework—really a layer of abstraction—onto MPI is easy, and at one stroke it eliminates many ways of inadvertently misusing MPI, though at the cost of giving up some generality and incurring slight overhead. Pilot users need not know anything about CSP; it simply hovers in the background providing rigour to the library's design.

A Pilot programmer deals with just one programming model, MPMD, where each MPI process (standing for a CSP "process" and running as an operating system process) executes a designated C function. This approach can be called function level parallelism. It is precisely how Pthreads works (see pthread_create), a library that students will likely be familiar with. Function level parallelism has also been recently utilized by the fine-grain MPI (FG-MPI) library [11] as its basic unit of "process" execution.

Thus, the first step in a Pilot program is defining which functions should be executed by MPI processes. The next step is defining channels over which the programmer allows the processes to communicate. Channel variables are utilized as arguments to PI_Read and PI_Write functions in lieu of ranks, tags, and communicators. During execution, it is trivial for Pilot to trap any erroneous attempts to circumvent these predefined channels, thus preventing common communication errors. At the same time, Pilot can also log to a file all channel communications as a debugging aid and keep a dependency matrix of process/channel communications to detect deadlocks. The latter is far simpler than the problem of generalized deadlock detection for MPI [12], since the scope of possible process interactions in a Pilot program is greatly circumscribed.

In addition to the functions designated to run as processes, the C main() function continues execution as MPI rank 0, also known as PI_MAIN, and is appropriate to use as a master process in master/worker patterns.

The chief capability missing from a pure process/channel approach is collective functions. In principle, students can get along without those, but teaching them is a cornerstone of HPC programming and is often a doorway into the underlying MPI library's most efficient operations. The Pilot approach to collective functions—which does not break the underlying CSP formalism—is to allow groups of channels that have a common process endpoint (either sending end or receiving end) to be designated as a *bundle*. The bundle then forms an argument to collective functions: One can broadcast onto a bundle or reduce from a bundle. But instead of programmers awkwardly calling "broadcast" or "reduce" in processes on the non-common "rim" of the bundle, they call the usual PI_Read and PI_Write operations on their respective channels.

Even with collective operations—including broadcast and reduce, scatter and gather, plus "select" (allowing, like the POSIX select system call, a process to detect which channel in a bundle is ready for reading)—and a handful of utility functions (such as obtaining timing data) the entire Pilot API comprises a mere 24 functions (see Table I).

Pilot has other distinctive features that will be illustrated below, all designed to ease the learning curve for students writing and debugging their first cluster programs: (1) syntax deliberately based on C's stdio.h fprintf and fscanf, specifying first the channel or bundle pointer (like FILE*), then the data format specification string, and finally the output values or input variables; (2) extensive runtime diagnostics that identify errors to the level of source file name and line number; and (3) integrated deadlock detector enabled by a command line option that prints out what all parties to a deadlock were doing. Programmers have the

TABLE I. COMPLETE PILOT API (V3.0)

Configuration Phase	Execution Phase
PI_Configure	PI_StartAll
PI_CreateProcess	PI_StopMain
PI_CreateChannel	PI_Write
PI_CreateBundle	PI_Read
PI_CopyChannels	PI_ChannelHasData
Utilities	PI_Select
PI_Set/GetName	PI_TrySelect
PI_GetBundleChannel	PI_Broadcast
PI_GetBundleSize	PI_Scatter
PI_Start/EndTime	PI_Gather
PI_Abort	PI_Reduce
PI_SetCommWorld (for running IMB benchmarks)	

option of giving meaningful names to processes, channels, and bundles, which increases the lucidity of error messages.

As will be seen below, the stdio-like syntax with its format string is very powerful: It allows a single Pilot call to accomplish what would take several calls in MPI, and eases communication of variable length data. There is even a format code that automatically allocates an array of the proper length to hold incoming data.

One feature that Pilot lacks, perhaps surprisingly, is a barrier. We were advised by our HPC consortium that beginning programmers tend to use barriers as a crutch, when in most cases there is no need for them. This is because, in contrast to shared memory communication (via global variables), message passing accomplishes communication and synchronization at the same time. Students need to grasp this inherent power of messages, and taking away barriers aids this realization. Furthermore, since collective functions have a group synchronizing effect, they can be used to achieve a barrier's purpose when warranted.

The simplifications of Pilot do result in some weaknesses compared to MPI:

1. Using Pilot it is less natural to create a pure SPMD code in which the "main" process is doing a full share of the work. Furthermore, there are no Pilot functions like MPI_Allreduce which allow sending processes to also get the reduced value, as that would violate unidirectional channels. This effect could be accomplished, albeit less efficiently, by broadcast following reduce.

2. Pilot has (at present) limited asynchronous capabilities with no functions corresponding to MPI_Isend and MPI_Irecv. However, PI_Write does call MPI_Send "under the hood," which does not necessarily block (depending on several factors internal to MPI such as message size). And one need not call PI_Read when there is a chance of its blocking. Instead, the process can probe an individual channel for data using PI_ChannelHasData, block until some channel in a bundle is ready for reading using PI_Select, or merely probe a bundle without blocking using PI_TrySelect. Using these tools, Pilot programmers can achieve a substantial degree of overlapping computation with communication.

3. Pilot uses a static process/channel architecture that cannot be altered at runtime. Also, creating complex patterns of channels (e.g., for a 2D or 3D grid layout of processes) can get messy compared to simply addressing rank numbers as with MPI.

What's more, students may be unaware that lengthy format strings trigger the sending of multiple messages, and that they could reduce communication penalties by packing and sending a single array. This is a matter of training. For beginning HPC programmers, the above were not considered to be significant drawbacks.

The next section walks through a skeleton Pilot program in C. It also lists all the categories of errors that Pilot detects at run time, depending on which level of error checking is enabled.

III. SAMPLE CODE

The program in Figure 1 is adapted from one of the labs that is available from the Pilot website. Including "pilot.h" makes the API available. As written, this program assumes a fixed number (5) of worker processes, and the student is told how to make it scalable by discovering the number of available processes at run time (return value from PI_Configure) and dynamically allocating the arrays of process and channel variables.

Looking first at main(), Pilot applications execute in two distinct phases: The first is the *configuration phase*, initiated by calling PI_Configure (which also interprets command line options), where the static application architecture, comprising processes, channels, and bundles, is defined by calling PI_CreateXXX functions. Creating a process means pointing to a function for it to run during the execution phase, here workerFunc. The same function body can be associated with multiple processes, and an integer index parameter can be passed so it can easily identify its own instance, very much in the same style as POSIX pthread_create. A second void* parameter is available for optional use, here filled with NULL. Channels are created by specifying their "from" and "to" processes. This sample also creates two bundles: one for broadcasting to the workers, and one for "selecting" which result channels have data to read. The configuration phase is concurrently executed by every MPI process in the cluster, resulting in the construction of equivalent internal tables on the various processors, regardless of their respective word length, data alignment, and endian properties, so that Pilot codes can

```
#include <stdio.h>
#include <stdlib.h>
#include "pilot.h"

#define W 5   // fixed no. of Workers

// arrays of process, channel, bundle pointers
PI_PROCESS *Worker[W];
PI_CHANNEL *toWorker[W];
PI_CHANNEL *result[W];
PI_BUNDLE *toAllWorkers, *allResults;

// no. of numbers to add up
#define NUM 10000

int workerFunc( int index, void* arg2 )
{
  int i, workers, size, myshare, mystart,
      sum=0, *buff;

  // get no. of workers, size of data set,
  // and auto-allocated array
  PI_Read( toWorker[index],
      "%d %^d", &workers, &size, &buff );

  // figure out myshare
  myshare = size / workers;
  mystart = index * myshare;
  if ( index == workers-1 )
      myshare += size%workers;

  printf( "Worker #%d signing on to do share of
%d!\n", index, myshare );

  // add up my share and report sum
  for ( i=0; i<myshare; i++ )
    sum += buff[mystart + i];

  free( buff );// allocated by %^d
  PI_Write( result[index], "%d", sum );
  return 0;   // exit process function
}

int main( int argc, char *argv[] )
{
  int i;

  // return no. of processes available
  int N = PI_Configure( &argc, &argv );

  // create Worker processes and channels
  for ( i=0; i<W; i++ ) {
    Worker[i] =
      PI_CreateProcess( workerFunc, i, NULL );
    toWorker[i] =
      PI_CreateChannel( PI_MAIN, Worker[i] );
    result[i] =
      PI_CreateChannel( Worker[i], PI_MAIN );
  }

  // create bundles for broadcasting, selecting
  toAllWorkers = PI_CreateBundle(
      PI_BROADCAST, toWorker, W );

  allResults = PI_CreateBundle(
      PI_SELECT, result, W );

  // start execution (workerFunc gets control
  // on its processor)
  PI_StartAll();

  // PI_MAIN continues

  int numbers[NUM];// each element is 0-999
  for ( i=0; i<NUM; i++ )
    numbers[i] = (double)rand()*999.0/RAND_MAX;

  // broadcast the work; W no. of workers, and
  // numbers array (length NUM)
  PI_Broadcast( toAllWorkers,
      "%d %^d", W, NUM, numbers );

  // collect the results using selection
  int sum, total = 0;
  for ( i=0; i<W; i++ ) {
    // find out which worker is done next
    int w = PI_Select( allResults );
    PI_Read( result[w], "%d", &sum );
    printf( "Worker #%d reports sum = %d\n",
        w, sum );
    total += sum;
  }
  printf( "Grand total = %d\n", total );

  PI_StopMain(0);// end program
  return 0;
}
```

Figure 1. Data-parallel adding numbers

run on hybrid clusters.

The *execution phase* commences with PI_StartAll making the flow of control diverge: Each process invokes its associated function, except for MPI rank 0 (PI_MAIN), which has no additional associated function and simply continues executing statements in the main() function.

In the sample code, PI_MAIN (here serving as the master) fills an array with random numbers and broadcasts it to all the workers. Note how two broadcasts are combined in a single PI_Broadcast format string: The first "`%d`" sends the integer `W`. The "`%^d`" format sends both the length, `NUM`, and the `numbers` array of size `[NUM]` ints.

The master then settles into a result collection loop, using PI_Select to find which result channel is ready for reading next. PI_MAIN finishes by calling PI_StopMain, which coordinates the shutdown of MPI with all workers.

Now looking at the worker function, it is written without assumptions about the number of workers or size of the data set. Note how it receives the broadcasted data by calling PI_Read on its `toWorker[index]` channel. The format "`%^d`" tells PI_Read to allocate (storing the pointer in variable `buff`) a right-sized array of ints to contain the

TABLE II. PILOT FORMATS AND REDUCE OPERATIONS

Pilot format	C datatype	symbol/	Operation
%c	char	max	maximum
%hhu	unsigned char	min	minimum
%d, %i	int	+	sum
%hd	short int	*	product
%ld	long int	&&	logical and
%lld	long long int	||	logical or
%u	unsigned int	^^	logical xor
"u" gives corresponding unsigned integer types: hu, lu, llu		&	bitwise and
%f	float	|	bitwise or
%lf	double	^	bitwise xor
%Lf	long double	mop	user-defined (MPI_Op)
%b	any		
%s	char*		
%m	user-defined (MPI_Datatype)		

data, also storing the length in the `size` argument. It then calculates the start and extent of its share of the array, adds up its share, and reports the result via PI_Write on its `result[index]` channel. The channel is part of the selector bundle that PI_MAIN is monitoring with PI_Select. A worker function exits simply by returning, after which Pilot will terminate its MPI process.

It is easy to change the program to use a reduction: The `allResults` bundle must be created with usage of PI_REDUCE rather than PI_SELECT. In PI_MAIN the collection loop becomes the single call:

```
PI_Reduce( allResults, "%+/d", &total );
```

The reduction is indicated by inserting an operator and slash before the data type. The corresponding PI_Write in the worker function changes slightly to:

```
PI_Write( result[index], "%+/d", sum );
```

The change is needed because the resulting call to MPI_Reduce must specify the reduction operator. Forgetting to make this change will yield an error message.

The full range of format codes and reduce operators is shown in Table II. Note that the format string (which can be supplied by a variable, not necessarily a string literal) is simply a convenient way to describe the data; it does not imply that data is converted to text for transmission. Fixed array length is specified by inserting an integer, e.g., "%100f" stands for 100 floats. Not shown are the special format characters "*" and "^". The star indicates that the variable array length is supplied from the argument list, e.g., (..., "%*f", 100, floatarray), for both reading and writing. The caret was illustrated in the sample code. The "%s" format uses the same mechanism to communicate a NUL-terminated C string with its length being determined automatically via strlen. The reader is responsible for freeing arrays allocated by "^" and "%s".

Utilizing the integrated deadlock detector is as simple as adding the option "-pisvc=d" on the command line (i.e., `mpirun...-pisvc=d`). The deadlock detector does consume an additional MPI process, so the number of processes should be incremented (PI_Configure will not report it as being "available" for a worker's creation). Errors such as circular wait will cause the program to abort with a diagnostic message identifying the deadlocked processes. For example, suppose the programmer omits the PI_Read in PI_MAIN's collection loop. With MPI alone, the program would likely exit normally after printing a bogus sum. But the Pilot deadlock checker will note that there are outstanding messages undelivered and print the following message:

```
   BLOCKED: Process P2(0) called
PI_Write(C2:P2>main, "lab-sample-c:33")
Deadlock detected in Pilot process
main(0)!!! Process exiting leaves
earlier operation hung
```

"P2" is the default name for the second process created in the configuration phase; it was started with argument 0. At line 33, it wrote on "C2" the second channel created, from P2 to PI_MAIN. Since main() exited without reading that channel, the deadlock detector realized that the message can never be received. If the programmer desires, processes and channels can be relabelled with arbitrary names by calling PI_SetName.

Pilot checks for errors depending on the level selected by the programmer. This can be set by a global variable (PI_CheckLevel) or a command line option (`-picheck=`*n*). Level 1 is the default, checking for violations of function preconditions, I/O argument lists that do not match their format strings, and for internal table corruption. During initial debugging, programmers may wish to select Level 2, which requires more CPU and messaging overhead but verifies that each reader and writer uses matching format strings. Level 3, in addition, checks that arguments where addresses are needed appear to be pointers, thus guarding against a common C-type error of coding fscanf(..., var) instead of &var in input contexts, often leading to a segmentation fault. Since data can look like a valid pointer, this check is not fail-safe.

For example, if in the PI_Reduce call above, we accidentally omit "&" the error message will enable the programmer to pinpoint the offending function:

```
   main(0) @ lab-sample-code-red.c:97:
An argument that should be a location
(pointer) looks like a data value
```

Finally, to obtain a bit more run-time efficiency, Level 0 disables most checking except for function preconditions.

IV. USING PILOT IN A PARALLEL PROGRAMMING COURSE

Details concerning the University of Guelph course, including the 12-week schedule of topics, organization (assignments, term project, final exam), textbook, and par-

allel programming platforms, are available elsewhere [13]. Course prerequisites expect students to have passed our operating systems and computer organization courses. These ensure that they understand the concepts of OS processes with disjoint address spaces, the resources they utilize, forking and joining, memory caching, and at least a light exposure to shared-memory programming with Pthreads, including critical sections, mutexes, and the producer/consumer pattern.

Pilot is used for the first assignment (before moving on to shared memory techniques—pthreads and OpenMP). It is introduced in two 90-minute lab sessions using the same Powerpoint slides and hands-on labs that have been successfully utilized in half-day tutorials presented at conferences. This instruction, along with the textbook's first four chapters on parallel computing is sufficient to prepare students to start working on the Pilot assignment. Absent the OS background described above, one should teach at least the concept of processes before running the Pilot tutorials.

Experience shows that the initial challenge for some students is to realize that the Pilot processes do not share memory and that messages are the sole means of communication. They may still have global variables in their programs—for example, PI_CHANNEL* variables assigned by PI_MAIN and used in the process functions—but cannot communicate between processes using them.

The nature of the Pilot assignment varies from year to year, but it always involves a data parallel problem amenable to course-grained breakdown, sometimes with task parallelism thrown in calling for a pipeline pattern of processes and channels. Students are encouraged to use collective functions in their solutions. Some past assignments were:

- Running queries on Transport Canada's public National Collision Database (which is distributed among the workers) such as "In a year on average, how many people typically wreck their new car?" (opportunities for different types of reductions)
- Searching for textual "image" patterns in large text files (presenting several choices for parallelization and scheduling to optimize load balancing)
- Creating thumbnail images from a large set of JPEG files of various sizes (task parallelism in the decoding and encoding steps)

Deliverables apart from the source code include:

1. A process/channel diagram illustrating the architecture adopted for the program, plus a printout showing messages successfully travelling over the channels. This is handed in prior to the solution, to make students create a cycling Pilot "skeleton" program and exercise the channel "plumbing" before going on to fill in the computation logic. Navigating this part of the learning curve early helps them succeed on the entire assignment.
2. A report including three timing graphs based on given benchmark inputs, all plotted versus number of processors: performance (wall clock time as returned by PI_StopTime minus PI_StartTime), speedup (serial time/ parallel time), and efficiency (speedup/number of processors). They must comment on the graphs, try to account for anomalies, and analyze the scalability of their solution (e.g., noting where Amdahl's Law comes into play). They also have to report on what difference compiler optimization made (startling decreases in run time of up to 30% are not uncommon, and yet the effect tends to wash out in the speedup graph).
3. An experience report on cluster programming and their use of Pilot, including what they liked or didn't like.

A major course emphasis, mirroring the textbook authors' thrust, is on writing *scalable* codes that find out the number of processors available at run time and use them all to achieve respectable speedups. Thus the remaining assignments, as well as the term project, reprise the above reporting requirements, as they shift to Pthreads and OpenMP using Intel Parallel Studio tools.

Students also have the option of using Pilot for the term project. The Fortran API was initially created to parallelize a "dusty deck" program used to simulate spectrograms involved in analyzing Martian soil samples for evidences of water. Another project combined Pilot with OpenMP, to take advantage of both coarse- and fine-grained parallelism. Some students have enabled cluster programming using other languages by means of the Pilot approach, either by wrapping Pilot's C API (similar to how Fortran is supported) or creating a "workalike" library mirroring Pilot's capabilities. These include LuaPilot and Pylot (for Python). Pilot++ (for C++), which is truly object-oriented unlike MPI's deprecated C++ binding (removed in MPI-3 [14]), was developed at St. Olaf College (contact Dick Brown).

Student feedback has been important in deciding what enhancements to Pilot would be worthwhile. In general, error reporting has been improved from version to version. The most requested feature—writing and reading variable length data in a single step—was implemented in the V2.1.

Impressions quoted next are from different students who used Pilot for an assignment:

- Pilot was very easy to write code with. The API is simple and has a familiar format for C users. For a novice scientific programmer it simplifies the parallelization immensely.
- Looking at the API that it was based off [MPI] I am extremely happy that we had a simplified API to learn on.

- The Pilot library contains a small amount of functions. Therefore, you don't need much time to be used to it and start working on it. There are only 3 concepts to understand before handling the library properly [referring to processes, channels, and bundles].
- It has its function parameters similar to the C language, which simplifies the learning of the library and increases productivity and makes it intuitive to use.
- It provides enough abstraction so that it is easy to pick up and start using right away, but still has enough functionality that makes it a usable tool.
- I'm sure the deadlock detection saved me lots of time completing this assignment.

These impressions are from a student who used MPI for a project comparing Game of Life on a cluster vs. Pthreads on a multicore, so he has some basis for comparing Pilot and MPI: "Now that I've used MPI, I can't believe how much simpler you've made cluster programming. I definitely didn't realize how crucial Pilot's deadlock detector was until I needed it. Using the [MPI] library isn't as difficult as people make it seem, although its incredible amount of functions can be very overwhelming. I tried my best to just stick with functions I could map to Pilot functions, because those were ones I understood." The last point shows initial Pilot training serving successfully as a "ramp" up to more advanced use of MPI.

The most frequent complaint about Pilot is that more documentation is wanted. This probably does not refer to documentation that can be read systematically. In reality, students (and their professors!) now debug chiefly by doing a web search with a problem, question, or error condition, expecting an answer to pop up on Stack Overflow or the like. With Pthreads, OpenMP, and MPI they can work this way, because there is an adequate body of experience on the Internet. It will be a long time before Pilot can thus compete, but it is hoped that by opening a wiki or similar forum, students may begin to contribute problems and solutions.

To conclude this section, we can now go back to the five objections listed in Section I and explain how Pilot overcomes them in the context of HPC training:

1. Pilot's API is so compact that students can learn about all of it in a short time and realize that there are no complex features lurking in the background.

2. They use only an MPMD programming model based on the process/channel architecture that they devise for a given problem. This makes their design effort more straightforward. By analogy with pthread_create, creating Pilot processes based on parameterized functions is easy to understand. Such functions need not know their MPI rank, nor does Pilot provide that information.

3. All communication functions take a single channel or bundle argument, and all obey familiar fprintf/fscanf syntax, so there is no confusion about rank, tag, and communicator arguments, what order to put them in, and when to use special symbols such as MPI_ANY_SOURCE. Mistaken use of channels or bundles, and mismatched reader/writer formats (including differing read/write lengths), are immediately diagnosed by error messages down to the exact line of source code.

4. If deadlock is suspected, the program can be rerun with an additional MPI process and the command line option for deadlock detection. The program will abort with a printed diagnostic revealing the processes and channels implicated in the deadlock. Other problems may be tracked down by enabling message logging to a file.

5. The deadlock detector will also disclose whether any process exiting creates a situation of undelivered messages (writes that cannot be satisfied by reads), or conversely, reads or selects that will hang because the process(es) that could possibly write to those channels have terminated.

A frequent occurrence with students coming to office hours for help is to see them start with error-riddled codes and to watch how Pilot's diagnostics help them make incremental improvements. Even though the helper may spot a problem by eyeballing the source code, it is better to let them run it, generating a Pilot error message. One can work through its meaning with the student, and how to correct the faulty code. Running again leads to the next error, or perhaps a deadlock is detected, until finally after several iterations the code works properly. This process would be more painful using MPI alone due to its wider opportunities for committing errors and its less-supportive error messages.

V. Status and future work

Pilot is available for free public downloading from its website [15]. Under "all docs" one can find an installation guide, release notes, tutorial, quick reference card, and list of publications. Training slides (3 hours) and source code for the hands-on lab exercises are also there. Pilot can be used for free by anyone without any licensing formalities.

The library is copyright by the University of Guelph and was not originally open source in order to control API bloat, and, more importantly, to avoid ad hoc enhancements that break the CSP formalism. By Version 3.0, we consider the library mature enough to release it under the open source LGPL, meaning that programmers can incorporate it into their proprietary codes without forcing those to become open source themselves. Pilot is written to a high professional standard and comes with a large suite of regression tests. It can be downloaded and built by a user with ordinary privileges; it need not be installed by a system administrator. We have not found a version of MPI with which C

Pilot does not build, and would like to know if such occurs. In contrast, the Fortran API is more sensitive to compilers because interpretations of ISO_C_BINDING vary.

In terms of future work, the following are planned:

- Off-line utilities to help visualize execution-time message flows by analyzing the log (difficult to do by hand)
- Offering the option to let PI_MAIN be a thread, so that a "snoozing master" need not consume an OS process (meaning a dedicated core) while simply waiting for workers to respond
- Providing limited facilities for asynchronous I/O, at least for the common technique of double buffering

There are no plans at present to support any new features from MPI-3. Finally, the Fortran API lags behind at V1.2, not accessing all features of the latest C version. Given sufficient demand, this could be prioritized.

VI. Conclusion

Pilot carries the motto "A friendly face for MPI." We have shown how the Pilot approach to HPC programming differs from MPI's, helping teach the principles of message-passing programming in a well-cushioned environment featuring a simple process-and-channel based MPMD programming model, whose user-defined architecture is enforced at run time, with extensive diagnostics for all kinds of usage errors and an integrated deadlock detector. Instructors can use Pilot as an endpoint for training in an individual course, perhaps targeted at beginning scientific programmers who are breaking into parallel programming, or as a gentle ramp up to full-bodied MPI for advanced users. We welcome contributions from the HPC user community aimed at improving Pilot and adapting it further for educational purposes.

Acknowledgment

Pilot research has been supported by SHARCNET (Shared Hierarchical Academic Research Computing Network) and NSERC (Natural Sciences and Engineering Research Council) of Canada.

References

[1] Message Passing Interface Forum. MPI: A Message-Passing Interface standard version 2.2 [online]. Sep 2009 [cited 08/26/14]. Available from: http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report/mpi22-report.htm.

[2] Calvin Lin and Larry Snyder. *Principles of Parallel Programming*. Addison-Wesley, 2009.

[3] Argonne National Laboratory, Mathematics and Computer Science. MPI's send modes [online]. 2012 [cited 08/23/14]. Available from: http://www.mcs.anl.gov/research/projects/mpi/sendmode.html.

[4] Timothy J. McGuire. A simplified message-passing library. *Journal of Computing Sciences in Colleges*, 19(4):252–256, Apr. 2004.

[5] Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. *Introduction to Parallel Computing*. Addison-Wesley, 2nd edition, 2003.

[6] Lori Pollock and Mike Jochen. Making parallel programming accessible to inexperienced programmers through cooperative learning. In *Proceedings of the thirty-second SIGCSE technical symposium on Computer Science Education (SIGCSE '01)*, pages 224–228. ACM, 2001.

[7] 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*, page 51, Washington, DC, 2000. IEEE Computer Society.

[8] Jayant DeSouza and Jeff Squyres. Why MPI makes you scream! and how can we simplify parallel debugging? Supercomputing '05 (SC05), Birds-of-a-Feather Session, 2005. Available from: http://www.open-mpi.org/papers/sc-2005/debugging-bof-6up.pdf.

[9] John Carter, W.B. Gardner, and G. Grewal. The Pilot approach to cluster programming in C. In *Proc. of the 24th IEEE International Parallel & Distributed Processing Symposium, Workshops and Phd Forum*, Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC-10), pages 1–8, Atlanta, Apr. 23 2010.

[10] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[11] Humaira Kamal and Alan Wagner. An integrated fine-grain runtime system for MPI. *Computing*, 96(4):293–309, 2014.

[12] 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, 2009. ACM.

[13] William Gardner. Third-year parallel programming for CS undergraduates. In *Frontiers in Education: Computer Science and Computer Engineering (FECS '11)*, pages 8–13. Las Vegas, Jul. 18-21 2011.

[14] Jeff Squyres. The MPI C++ bindings: what happened, and why? [online]. Oct 2012 [cited 08/23/14]. Available from: http://blogs.cisco.com/performance/the-mpi-c-bindings-what-happened-and-why. Cisco Blog, High Performance Computing Networking.

[15] Pilot home [online]. Available from: http://carmel.socs.uoguelph.ca/pilot.