Using the Pilot Library to Teach Message-Passing Programming

Bill Gardner and John Carter
School of Computer Science
University of Guelph, Ontario, Canada
15 min. to “sell” Pilot for teaching (in lieu of or before MPI)

• What is the Pilot library?
• How does it ease the learning curve for HPC?
  • Theoretical, practical, familiar, safety net
• Let’s look at a Pilot program
• Student reactions
• Other advantages
• Resources available
What is the Pilot library?

• Thin abstraction layer on top of MPI
• Provides an alternative “slimmed down” C and Fortran API for message-passing
  • Does away with complications of ranks, tags, communicators
  • Uses MPI for message transport
  • Supports point-to-point communication and collective operations
• Not tied to any version of MPI
• Not tied to any compiler (except Fortran works best with Intel)
• Standalone installation, does not require any other libraries
• Fully open source under LGPL (license for libraries)
  • Incorporating in your code does not force it to be open source
Eases learning curve 4 ways

1. Theoretical
2. Practical
3. Familiar
4. Safety net
1. Theoretical

- Provides an MPMD programming model (akin to pthreads) that helps students organize their use of parallel processes.
- Model of **processes** and unidirectional message **channels** is taken from CSP (Communicating Sequential Processes) in order to have formal rigour.
  - Students need not know that.
  - Not simply an arbitrary ad hoc simplification of MPI.
- Pilot processes = MPI processes (ranks).
- Channels are an abstraction.
  - Implemented with tags “under the hood”.
- Collective operations utilize groups of channels called **bundles**.
  - Implemented with communicators.
2. Practical

- Instead of MPI’s enormous API with lengthy argument lists, Pilot has 2 dozen carefully targeted function calls
- Small enough to grasp in its entirety
- Large enough to encompass what you want to teach, the heart of HPC:
  - Point-to-point messages and common collective operations

<table>
<thead>
<tr>
<th>PI_Configure</th>
<th>PI_StartAll</th>
<th>PI_Select</th>
<th>PI_Set/GetName</th>
</tr>
</thead>
<tbody>
<tr>
<td>PI_CreateProcess</td>
<td>PI_StopMain</td>
<td>PI_TrySelect</td>
<td>PI_GetBundleChannel</td>
</tr>
<tr>
<td>PI_CreateChannel</td>
<td>PI_Write</td>
<td>PI_Broadcast</td>
<td>PI_GetBundleSize</td>
</tr>
<tr>
<td>PI_CreateBundle</td>
<td>PI_Read</td>
<td>PI_Scatter</td>
<td>PI_Start/EndTime</td>
</tr>
<tr>
<td>PI_CopyChannels</td>
<td>PI_ChannelHasData</td>
<td>PI_Gather</td>
<td>PI_Abort</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PI_Reduce</td>
<td>PI_SetCommWorld*</td>
</tr>
</tbody>
</table>

*Just for running Intel IMB benchmarks*
3. Familiar

- Pilot's messaging functions all use C's `fprintf/fscanf` syntax to specify data types and lengths

```c
PI_CHANNEL *result;
float buff[500];
double total;
PI_Read( result, "%500f %lf", buff, &total );
PI_Read( result, "%*f %lf", 500, buff, &total );
```

- Data is not translated into characters!
  - This is a compact metaphor (16 C datatypes)
  - Each format % results in one MPI message
  - Reduction notation: %+/... (11 operations)
4. Safety net

• The theoretical design already eliminates categories of communication errors from occurring
  • Programmers do not encounter ranks, tags, or communicators, hence cannot misuse them
• Library's functions all do extensive error-checking on their arguments to help students diagnose misuse of API
• Integrated **deadlock detector** is enabled via a simple command line option
  • Stops on and diagnoses deadly embraces, circular waiting, and unreceived messages involving collective operations as well as point-to-point communication
  • **Cost:** consumes an additional MPI process
• **Message log** can be produced (command line option)
Let’s look at a Pilot program

Two phases:

1. **PI_Configure**: *Configuration phase* specifies static architecture of processes, channels, and bundles
   - Processes are bound to functions (like pthread_create)
   - Channels are bound to processes (*from P to Q*)
   - Bundles are formed from arrays of channels with common endpoint
   - Executed in parallel on all MPI ranks

2. **PI_StartAll**: *Execution phase* launches all Pilot processes
   - Each starts executing its function
   - main() continues as “PI_MAIN” often used as master process
   - Function processes end by returning
   - PI_MAIN ends by calling **PI_StopMain** (barrier coordinates shut down)
Configuration phase

```
#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;

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 );
```
PI_MAIN “master” process

// 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);      // syncs end of all Pilot processes
Process work function

```c
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
}
```
Student reactions

- 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.

**Main complaint:** “want more documentation”
Other advantages

- Easy to download, build, and install with virtually any version of MPI and C compilers
  - No sysadmin action or special configuration needed
  - Fortran API can optionally be built
- Thin layer adds little overhead on top of MPI, so it is suitable for production use with typical master/worker and pipeline patterns for scalable parallel cluster computation
  - For production efficiency, error-checking level can be turned down, not run deadlock detector
- After learning Pilot, students who go on to MPI will understand its basic operations and be able to transfer their programming knowledge
Resources available

- Pilot website
  - carmel.socs.uoguelph.ca/pilot
- Latest source code (V3.0)
- **Training materials** used in Guelph course and elsewhere
  - PDF slides (Powerpoint upon request) \(\leftarrow\) to be recorded as video
  - **Quick Reference Card** for API
  - Source code for 3 hands-on lab exercises
- Downloads of Pilot-related publications
- Contact me about Pilot ports to other languages (gardnerw@uoguelph.ca)
  - **Pylot** (Python)
  - **Pilot++** (C++, developed at St. Olaf College)
  - **LuaPilot**
Conclusion, future work

• Experience shows Pilot is a viable method of introducing students to message-passing for cluster programming
• Enables fast uptake to tackle substantial assignment or project

• Status of code base:
  • High quality “industrial strength” code
  • No outstanding bugs
  • Feature set now complete “enough” as of V3.0
  • Anxious to avoid “API bloat” or breaking underlying formalism

• Current work:
  • Visualization tools to make message logs easier to interpret

16 Nov. 2014
EduHPC, New Orleans
Thanks to...

Students who contributed to Pilot

- John Carter
- Natalie Girard
- Jordan Goldberg
- Kevin Green
- Ben Kelly
- Emmanuel MacCaull
- Daniel Recoskie