2017 IEEE International Parallel and Distributed Processing Symposium Workshops

Log Visualization Tool for Message-Passing Programming in Pilot

Tianyi Bao and William B. Gardner
School of Computer Science
University of Guelph
Guelph, Ontario, Canada
{baot,gardnerw}@uoguelph.ca

***Abstract*—The Pilot library is aimed at novice high-performance computing (HPC) programmers and has been used for years to teach message-passing programming to undergraduates. While built on top of standard Message Passing Interface (MPI), it offers a compact application programming interface (API) based upon simple abstractions from the process/channel model of Communicating Sequential Processes (CSP), extensive error-checking, and an integrated deadlock detector. This work enhances Pilot with a log visualization facility adapted from MPI Parallel Environment (MPE) and Jumpshot-4. This new feature is a pedagogical tool helping beginners to understand actual run-time message-passing between processes, and a debugging tool for diagnosing logic that impedes parallelism.**

Keywords—MPI; parallel programming; message-passing programming; high-performance computing; log visualization

I. Introduction

Parallel programming in the Bachelor of Computing degree at the University of Guelph has not yet been integrated into core computer science (CS) courses. Our approach is to offer a third/fourth-year elective course that covers both shared memory and cluster programming, while emphasizing underlying hardware characteristics (e.g., high cost of non-local memory references) and principles (e.g., Amdahl's Law) that strongly influence a student's success in obtaining meaningful speedup. The course assumes that students have had light exposure to Pthreads programming in Operating Systems under the topic of concurrency. This means that they are unfamiliar with the technique of message passing, which does not depend on shared memory.

An important tool for teaching the latter style of programming is the Pilot library [1], developed at Guelph. Billed as "A friendly face for MPI," it was designed, firstly, to give students a way to think about organizing their parallel programs: a programming model based on easy-to-grasp concepts of processes and channels taken from Communicating Sequential Processes (CSP) [2]. Secondly, it gives them an easy-to-use programming toolkit: a small set of application programming interface (API) functions hiding low-level MPI features (such as ranks, tags, and communicators), made easy to learn by borrowing C's well-known fprintf and fscanf format syntax, and with extensive support for finding and fixing mistakes: elaborate error-detection for any abuse of the API plus an integrated deadlock detector not reliant on any third-party tools. Pilot uses a pure Multiple Program, Multiple Data (MPMD) style exactly like the Pthreads that students already know, contrasting with MPI's mixed approach that presents them with apparent contradictions such as calling MPI_Bcast in processes that are *receiving* data. For example, in Pilot the broadcasting process would call PI_Broadcast, and the receivers would all call PI_Read, just as if reading a point-to-point message.

Pilot is suitable for educational settings blessed with an HPC consortium partnership such as Guelph has, but also for installation on students' Linux multicore laptops, requiring only a version of MPI (e.g., OpenMPI), the open source Pilot library, and a C compiler. (Pilot also has a Fortran API.) Naturally, an N-core laptop will only be able to deliver up to an N-fold speedup, but this is enough for learning message-passing programming.

Even if avoiding the hazards of shared memory, parallel programs present special debugging challenges due to lack of observability and nondeterministic execution sequence. Pilot has always had a logging feature, enabled on the command line, for recording API calls from all processes into a single file, that could be resorted to for help with debugging. However, the text-based log had three major shortcomings: (1) the timestamps were not accurate, since they recorded the moment of arrival of API events at a central logging process (the same one running the deadlock detector which analyzes those events); (2) events from all processes were conglomerated and painful to separate; and (3) the output was scarcely human readable. As such, it was no wonder that this feature was rarely used, even when it would have been helpful.

In this paper, we describe the creation of a superior logging tool for Pilot, one that allows full visualization of all API calls. Section II gives previous work on Pilot, and research consulted in trying to make an "adopt, adapt, or build" decision for a log visualization tool. Section III describes the software solution that was ultimately created. Section IV shows practical uses of the new tool, both as a pedagogical aid and as a help for diagnosing problems with students' software. Finally, Section V brings the conclusion and future work.

II. Previous and Related work

The Pilot library (V1.1) was introduced [3] along with its rationale, programming abstractions, implementation, and performance. Since then, proposed additions have been carefully screened to avoid both gratuitous API bloat and

978-0-7695-6149-3/17 $31.00 © 2017 IEEE
DOI 10.1109/IPDPSW.2017.52

violations of the underlying CSP formalism. V2.0 added support for more collective operations, including broadcast, scatter, gather, and reduce. In V2.1, the ability to receive arrays of unknown length in a single call was added. V3.0, the first open source release under the GNU Lesser (or Library) General Public License (LGPL), featured two more command-line selectable levels of error checking: verifying that reader and writer format strings match, and that pointer arguments seem to be valid. The parallel programming course mentioned above was described [4] and with advocacy for teaching with Pilot [5] including student feedback.

Turning to the problem of log visualization, considerable effort was made to find an off-the-shelf solution that could simply be adopted by Pilot, or failing that, adapted, as a way of avoiding needless independent development. To guide our search, we set out criteria for a good solution:

- Must depict Pilot's functions on their own terms and not expose the underlying layer of MPI operations: For example, a single Pilot function may result in multiple MPI calls "under the hood" which would simply confuse programmers if portrayed. The visual log must relate directly to their source codes.
- Prioritize debugging over profiling: Some visualization tools focus more on large data analysis and performance profiling. Since Pilot is targeted at novice parallel programmers, debugging is more needed.
- Freeware to go along with free/open source Pilot: A commercial tool may not be suitable for an educational environment, and may generate a long-term commitment to renewing licenses.
- Minimize dependency on additional libraries: Pilot has not depended on any third-party libraries and is not bound to a particular implementation of MPI, therefore, the fewer new dependencies the better, and those should be optional at installation time.
- Prefer software that is widely used and reasonably current: A tool used by many people usually has a long life span, and one may be able to get more guidance in using it and solving problems that arise.
- Able to be used "as is": Any third-party software should not require modification to meet Pilot's needs so there is no future maintenance commitment.

After an initial scan of the literature, including a helpful survey paper [6], it was found that many current visualization tools for parallel computing are no longer limited to basic functions like debugging with a small number of processes. Instead, their main focus has shifted to performance profiling, such as analyzing large data, mapping 3D graphs, and manufacturing animation, but profiling is out of our scope. Therefore, we proceed to present tools that are more suitable for a debugging emphasis.

Eleven tools concerned with visualizing MPI parallel programs were found which could have capabilities to be applied to Pilot. They can be classified into three categories (see [7] for details):

1. Visualization tools: Pajé [8], Vampir [9], ViTE [10], Triva [11], Viva [12], Jedule [13], and Jumpshot-4 [14] (referred to as Jumpshot hereafter).
2. Debugging tools: DDT [15] is a debugging tool with visualization facility.
3. Measurement infrastructures generating a logfile to support visualizers: TAU [16] with its affiliated visualizer ParaProf [17], Score-P [18], and MPE [19].

Vampir and DDT are commercial products. The only tools that met all six search criteria were MPE and Jumpshot.

A. MPE

The Multi-Processing Environment (MPE) library was developed by University of Chicago and Argonne National Laboratory [19]. Based on the characteristics of MPI, MPE supplies various extension functions for debugging, graphics, and some common utility routines. Its main purpose is to generate logfiles of MPI calls automatically, but it also allows customized logging via its API. The Pilot library is itself an MPI program, so in principle these functions can be invoked from within the Pilot library on behalf of the user's Pilot application. MPE assumes that MPI is running and uses MPI for some internal operations.

MPE can generate SLOG-2 format logfiles, or CLOG-2 format which is convertible to SLOG-2 for input to Jumpshot. Even though it requires an extra step, the literature calls the conversion approach "preferred" for two reasons: (1) The program may turn out to be "non well-behaved" and produce a defective SLOG-2 file that cannot be properly displayed. (2) The conversion step can be useful for (a) diagnosing problems with the log contents, say, due to improper use of MPE's API, and (b) adjusting conversion parameters that affect the subsequent display such as the "frame size" (the amount of data initially displayed by the visualization tool).

B. Jumpshot

Jumpshot [14] was developed by Argonne National Laboratory as a visualization program for the SLOG-2 logfile format. It permits seamless scrolling at any zoom level of an entire logfile plus dragged-zoom, grasp and scroll, zoom in/out, vertical expansion of timelines, and timeline cut and paste. Jumpshot has a search-and-scan facility that helps locate graphical objects which are hard to find. It can also draw a picture from user-selected duration which allows for ease of data analysis on the statistics of a logfile. For example, it enables easy detection of load imbalance across processes among timelines. The legend table categorizes different objects, such as arrows, states, and events, and provides manipulation on visibility and searchability. All known SLOG-2 convertible logfile formats, including CLOG-2, can be transformed by an integrated logfile converter. In this way, Jumpshot satisfies many users' expectation of look and feel for a standard visualization tool.

We were able to adapt these off-the-shelf tools for Pilot's desired use, as described next.

III. Log Visualization Solution

Regarding basic MPE logging concepts, the user has a choice of recording events that have *duration,* called "state" (having start time and end time), and those that do not, called (by us) "solo events" (having only event time). Recording a

particular event in the log can be thought of as instantiating one or another "event ID" (an MPE-generated integer). States require a pair, one for the start and one for the end. State and solo event IDs are defined and given properties of name and displayable colour. Event instances inherit those properties and also add time(s) and optional text (limited to 40 bytes). Therefore, one must anticipate all the kinds of events that want to be recorded, and define each one by generating an event ID at initialization time.

After initialization, calling MPE_Log_event in pairs will log the start and end points of a state. This will result in Jumpshot displaying a coloured rectangle from the start to the end time. Nested states are possible: If state A runs from time 3 to 20, and state B from 5 to 8, then state B is fully nested within A. In Jumpshot, A will be shown as an outer rectangle and B as another rectangle within A.

MPE_Log_event can be called singly to record solo events, displayed as a small coloured circle or "bubble." Rectangles and bubbles can be right-clicked in Jumpshot to display times and additional text (if any). In order to show the relationship of message passing, MPE_Log_send and MPE_Log_receive should be called in pairs with matching tag number and length of data as parameters. This will result in a white arrow being drawn from the timeline corresponding to the sending process to that of the receiving process.

At the program's end, MPE_Log_sync_clocks is called to synchronize or recalibrate all MPI clocks to minimize the effect of time drift, and MPE_Finish_log is called to write out the single merged CLOG-2 logfile.

Jumpshot's displays are drawn on coordinates axes presenting processes and global time (in seconds) on Y and X axes, respectively. The process number (rank) starts from zero representing the main process (known as PI_MAIN). Jumpshot also has a "legend" button that brings up a listing of all the events and states in use by the logfile being displayed. For each state or event, it gives the coloured icon, the name, and some simple statistics (which can be sorted): a "count" of the number of instances (which could represent the number of times a function was called) and two durations marked "incl" and "excl". Inclusive means the sum of the duration of its state instances (equal to adding the widths of all its state rectangles, e.g., all the states named "Reduce"). Exclusive is the inclusive time minus any nested states, i.e., subtracting interior rectangles, which amounts to the time spent computing purely in the state and not in its substates. These statistics are potentially useful for performance purposes in the absence of special-purpose profiling tools.

Next, a visual design for displaying a log of Pilot calls in Jumpshot is described. The plan is in terms of an overall colour scheme and the use of graphical objects, which are observed most clearly in Section IV's Fig. 3.

A. *Plan for using colours*

Colours are not used in an ad hoc, arbitrary fashion, but a meaningful system was devised based on breaking Pilot functions into four categories: output, input, administrative (i.e., not performing message I/O), and other. The last category is functions not significant enough to warrant displaying in Jumpshot, either because they belong to the brief, one-time configuration phase (which is already displayed in total as PI_Configure), or because they are simple utilities with no communication implications.

The first principle is that all the functions in the same category should have similar colours. The second principle is that within a category we can distinguish simple channel-based communication (PI_Write and PI_Read) versus collective communication (e.g., PI_Reduce, PI_Broadcast) by light versus dark shades of the same colours.

We decided to adopt a red theme for input and a green theme for output.[1] This is the same plan used by Pajé. Symbolism is intended: The word "red" is similar to "read," and since reading always blocks in Pilot, it can also be associated with "red means stop." In contrast, "green means go" can readily be connected with writing because sending a message has an interprocess synchronization effect—signalling to wake up a waiting reader—as well as a communication effect. However, even if some user does not feel inspired by this symbolism, he or she will at least recognize right away that input-type functions can easily be distinguished visually from output-type functions. Then, to apply the second principle, it is intended that collective Pilot I/O functions should use darker shades of PI_Read or PI_Write. For example, PI_Read and PI_Write are red and green. PI_Broadcast and PI_Gather are ForestGreen and IndianRed.

To implement and simplify the colour design, we created a header file for color assignments. If users are not satisfied with the default colours, they can alter them by modifying this file and recompiling Pilot. Moreover, in Jumpshot one can adjust the colours to individual taste by pressing one of the icon buttons of objects in the legend window, but this setting only persists for the current Jumpshot session.

B. *Plan for using graphical objects*

A state rectangle is used to display, on a given process's timeline, each instance of its calling of a Pilot function. The time at the left edge of the rectangle will be just after the function is invoked, and the time at the right edge just before it returns. The associated popup shows the line number where it is called in the original .c file, the name of the calling process, and its work function's index argument. Each Pilot process has a default simple name like "P3" meaning the third process created (equivalent to MPI rank 3); however, programmers can call PI_SetName to assign a more meaningful name, and they may wish to do so precisely for the purpose of logging and debugging. The reason for showing the first argument (an integer) is because it is common in master/worker patterns for all workers to execute the same function, distinguishing their instances only by the value of the first argument, which often serves as an array index. Therefore, the argument denotes which of the workers issued the call. If there is a bundle[2] argument involved in a Pilot collective function call, the name of the bundle (e.g., "B4")

[1]The colours used could be distinguishable even for viewers who have some colour blindness, since the lead author is himself red-green colour blind, but still sees these pictures clearly.

will be shown. As with processes, the programmer can assign meaningful names to any channel or bundle.

Event bubbles will be used to annotate the state rectangles with the precise times of, and information about, important "milestones" during the execution of Pilot functions. For example, a PI_Read call can spend considerable time waiting for a message to arrive, so the rectangle could be rather long. The bubble in its rectangle will mark the moment when the message arrives, and its associated popup will display detailed information about the name of the channel involved in this PI_Read (e.g., "C3"). (On the output side, e.g., PI_Write, the data length and the value of the first element are also shown.) Furthermore, since a single PI_Read may involve multiple messages (e.g., the format "%d %100f" sends two MPI messages, one for a single integer and another for an array of 100 floats), there will be a bubble inside the rectangle indicating when each message arrives.

A message arrow will be drawn between any output source and its input destination. In the case of collective operations like PI_Broadcast, a bundle with N channels will result in N arrows being drawn. The popups associated with message arrows always display the start and end times of the transmission, its duration, the MPI tag, and message size. No way was found to attach additional data.

PI_Select is a slight exception to the above patterns. On the one hard, it acts like PI_Read in that it blocks until a message is received on any of its bundle's channels, therefore it should be represented as state. On the other hand, no message is actually received until a subsequent call to PI_Read, therefore it does not have an event bubble. Its information popup gives the index of the channel that is ready to read.

The above descriptions apply to output/input categories. For administrative functions, bubbles are used to show related information and message arrows are not used at all. These function calls are mandatory in every Pilot program (shown clearly in Fig. 3's source code):

1. PI_Configure: This is used to initialize lower level MPI operations and must be called before any processes, channels, and bundles are created. The Configuration Phase stretching from PI_Configure to PI_StartAll will be represented by a bisque coloured state rectangle.
2. PI_StartAll and PI_StopMain (Compute): PI_StartAll triggers each created process to execute its work function, while code following it continues under the identity of PI_MAIN (normally filling the role of the master process). This marks the start of the Execution Phase, which ends when each work function returns and when PI_MAIN calls PI_StopMain. Thus, PI_StartAll and PI_StopMain bracket a clear execution time period and can be represented by a gray coloured state rectangle, named as Compute.

[2]A "bundle" is a set of channels having a common process endpoint. They are created by the programmer during the configuration phase to serve as arguments to collective functions, just as channels are for point-to-point communication. Pilot does not support all-to-all communication.

The following optional functions are treated as independent events: PI_ChannelHasData, PI_TrySelect, PI_Log, PI_StartTime, and PI_EndTime. They are represented as bubbles with their return values shown in the related popup window. In all cases, the line number where the function was called is reported.

The PI_Abort function is used by the programmer, or by Pilot itself, to halt execution of the program on all nodes, typically after a fatal problem has been detected by one process. Ideally, we would like to enter PI_Abort in the MPE log, but there is a problem: PI_Abort calls MPI_Abort, which shuts down all MPI message infrastructure. There is no other way for the programmer (and Pilot) to terminate a cluster program's execution except to call MPI_Abort, which enlists MPI in killing the program executing on every rank. However, MPE uses MPI messages to collect the per-node logs and combine them into a single file written by rank 0. Thus when MPI_Abort is called, there is no way to avoid the loss of the MPE log. One might imagine that this is a recognized problem for MPE, yet we were unable to find literature dealing with the issue. So we do not currently provide any logging treatment for PI_Abort. Pilot's existing native log does not have this vulnerability because it writes each log entry onto a disk file when it is received.

C. *Integrating MPE into Pilot*

Since Pilot now constitutes a "legacy code," it was important to respect its existing software architecture, source code organization, and user interface. In particular, the existing data pipeline of API events that flow to the dedicated logging and deadlock detection process was not disturbed, nor was it utilized: The timestamp problem was mentioned above. Beyond that, only one event per API call was reported, which is not enough to establish state duration.

Pilot uses command line arguments to turn on run time options such as error check levels and logging. To be consistent, we added another option to the "-pisvc=" command line argument: "j" standing for "Jumpshot log". This is the same way that Pilot's native call logging is enabled presently: -pisvc=c. Options can be combined, e.g., -pisvc=cj. MPE logging will not be enabled by default because of the possible overheads (measured and reported in Section E).

In Pilot source files, all the calls and parameters related to the MPE logging facility were set up to make building Pilot with MPE optional by means of conditional compilation macros. The administrator can choose whether or not to install MPE at any site. In practice, the inserted MPE functions are wrapped by "#ifdef" and "#endif". If the user asks for an MPE log (-pisvc=j) but without MPE being built in their Pilot installation, a warning will be printed to show that logging for Jumpshot is not available. In this way, Pilot does not gain a hard dependency on the MPE library.

Some problems emerged during the integration experiments, the main one being distortions of bubble and arrow positions. When event bubbles and arrows are created within an extremely short time period, which can happen in drawing multiple arrows for collective operations, it was found that they could end up superimposed upon each other. This

condition can also raise a warning message called "Equal Drawables" when converting the CLOG-2 file to SLOG-2. The message indicates that two or more graphical objects having the same event ID also have identical start and end times. This can result from the limited resolution of MPI_Wtime (returning wallclock time in double precision seconds) [20].

To prevent this problem of superimposed objects, a compromise is to artificially spread the time of each arrow creation by inserting delays using usleep (found in unistd.h). With just 1 ms of delay per arrow, the problem is eliminated resulting in an even fanout of arrows, and yet the injected delay hardly impacts the program's execution.

Another thorny problem was information appearing garbled and out of order in popup windows whose text strings utilize printf-style substitutions. For example, if a string was "%d lines", the extra information in the popup window would come out as "lines *number*". Suspecting that this was an MPE bug, we checked the ASCII content logged in both CLOG-2 and SLOG-2 files, and found the sequence there was correct, so we thought the problem might be caused by an internal operation of Jumpshot. Rather than attempt to change the related code of Jumpshot, we discovered the workaround of starting any string with some literal text; thus "Lines: %d" would display in the desired order.

D. Demonstration Application

A JPEG thumbnail application was used as a course assignment and gives a realistic view of the log of a well-designed HPC program. Its purpose is, given more than one thousand JPEG files, to rapidly produce a corresponding set of thumbnail images (also JPEG). It is intended to use a pipeline of three kinds of processes which are PI_MAIN, multiple D_i (standing for decompressor), and single C (compressor). PI_MAIN is responsible for opening each .jpg file in the specified input directory, inputting the file as binary data, and shipping it off to the next available worker D_i. A D process does the work of decompressing the JPEG data, cropping out the center 32% of the pixel array, and then down-sampling the array so as to send only 32% (i.e., every third one) of the pixels to the compressor process C. The C process obtains each thumbnail image as pixels from a D_i worker, compresses it back into JPEG format, and ships the binary data back to PI_MAIN which writes it to the output directory. A constraint is that only PI_MAIN is permitted to do disk I/O. The application scales by adding additional data parallel D processes, since this is the most time-consuming stage of the whole task parallel pipeline.

Fig. 1 shows the timeline of the application in Jumpshot running with PI_MAIN (rank 0) plus 10 work processes, i.e., the compressor (rank 1) and 9 decompressors (ranks 2-10). Since the resulting SLOG-2 file can be successfully read by Jumpshot after calling thousands of Pilot functions without any conversions errors from CLOG-2, it demonstrates that the MPE logging calls are robust in a reasonably large and complex Pilot application. At this time scale, the screenshot is not intended for detailed interpretation—for that purpose see Section IV—but it can be stated that the apparent yellow

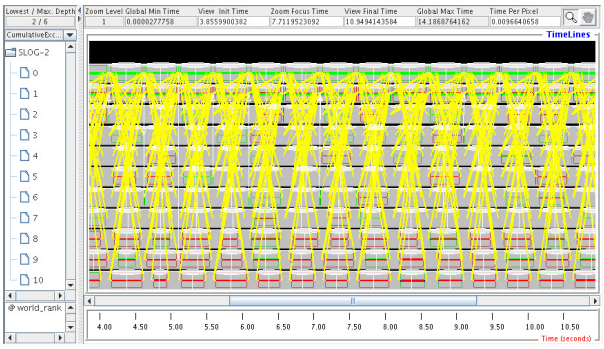

Figure 1. Thumbnail application shown in Jumpshot

"lines" are actually patterns of event bubbles, and the vertical white lines are nearby (in time) almost-vertical message arrows to/from rank 0. As for the horizontal rectangles that appear in outline form (rather than solid colours), this is how Jumpshot portrays state changes in a zoomed-out interval (the rectangle) that are too numerous to show individually: Within the rectangle it draws horizontal stripes such that the widths of the stripes indicate the relative proportions of each colour—here, red, green or gray—within that interval.

Fig. 2 shows a zoomed-in portion of Fig. 1 which is much more distinguishable. We can see at a glance that Pilot I/O functions only take a small proportion of the time, since the colours red and green (many instances too narrow to stand out in this screenshot) are tiny in comparison to the amount of gray. In contrast, most of the execution time is used for computation (the gray state rectangles), namely decompressing, cropping, and recompressing the pictures instead of being idle. This means the parallel application program is well-designed and can be a good experimental case study for measuring the overhead of MPE logging calls.

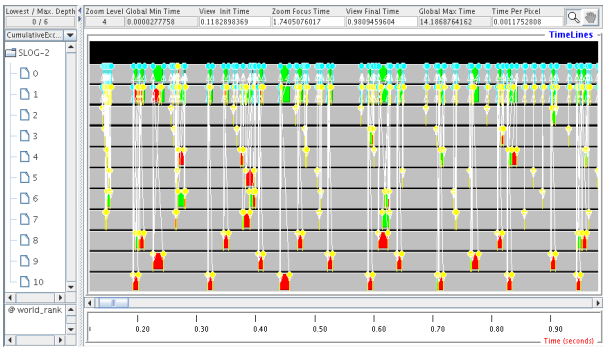

Figure 2. Thumbnail application in Jumpshot zoomed in

E. Overhead measurement

The thumbnail program was run for 1058 input files using 5 or 10 work processes (plus one for PI_MAIN) and varying combinations of Pilot error and deadlock checking. See [7] for details. Each case was run ten times and the median execution time calculated [variance shown in brackets]. The significant comparison is between no logging (30.97s [0.24] for 5 workers, 14.42s [1.40] for 10, which shows nice speedup) versus MPE logging (30.03s [0.23] for

```
#define W 5        // fixed no. of workers
#define NUM 10000  // size of data array
PI_PROCESS* Worker[W];
PI_CHANNEL* toWorker[W], *result[W];
int workerFunc(int index, void* arg2)
{
  int myshare, sum=0, *buff;
  PI_Read(toWorker[index], "%d", &myshare);
  buff = malloc(myshare * sizeof(int));
  PI_Read(toWorker[index], "%*d", myshare, buff);
  for (int i=0; i<myshare; i++) sum += buff[i];
  free(buff);
  PI_Write(result[index], "%d", sum);
  return 0;// exit process function
}
int main(int argc, char *argv[])
{
  PI_Configure(&argc, &argv);
  for (int 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);
  }
  PI_StartAll();// Workers launch, PI_MAIN continues
  /** Omitted -- fill numbers array with random nos. **/
  for (int i=0; i<W; i++) {
    int portion = NUM/W;
    if (i == W-1) portion += NUM%W;
    PI_Write(toWorker[i], "%d", portion);
    PI_Write(toWorker[i], "%*d", portion,
  &numbers[i*(NUM/W)]);
  }
  int sum, total = 0;
  for (int i=0; i<W; i++) {
    PI_Read(result[i], "%d", &sum);
    printf("Worker #%d reports sum = %d\n", i, sum);
    total += sum;
  }
  printf("Grand total = %d\n", total);
  PI_StopMain(0);// Workers also cease
  return 0;
}
```

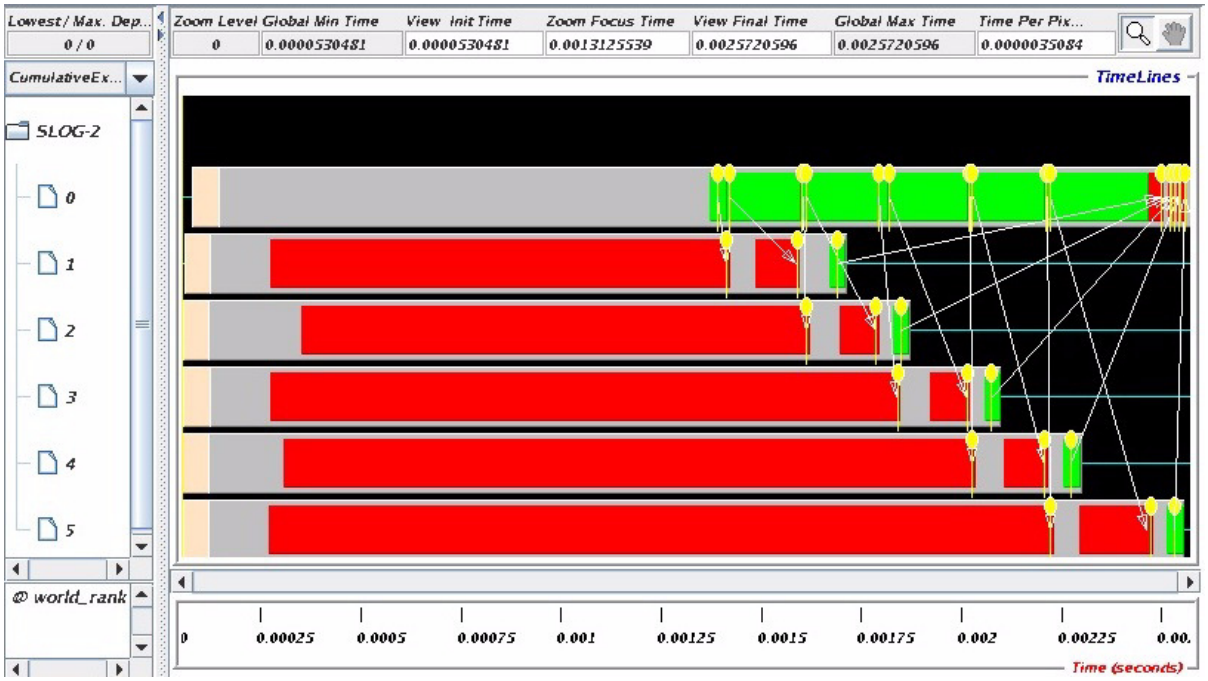

Figure 3. Hands-on exercise "lab 2" source code (top) and visual log (bottom)

5, 14.42s [0.87] for 10), both with maximum level 3 error checking. (This same study showed that the error checking level was essentially inconsequential in terms of added overhead.) Thus, as the number of processes is scaled up, MPE logging calls added only extremely slight overhead to this Pilot program of a compute intensive nature (which is the norm for HPC production codes). Note, however, that this disregards log wrap-up time.

The native Pilot logging feature writes to the disk continually, so its overhead is amortized with the program's execution. However, it does consume an additional MPI rank, and this is reflected in the times for native logging (40.64s and 16.2s), since one worker is displaced. In contrast, MPE pays a cost at program termination to collect, merge, and output the log. This wrap-up time was measured as 0.74s for 5 workers and 0.84s for 10. This is certainly a bearable cost.

IV. Practical Uses

During the Fall semester of 2016, Pilot's new visualization feature showed its usefulness in two primary ways.

A. *Pedagogical aid to the instructor*

The message-passing unit of the course opens with two lab sessions devoted to Pilot training. This same content has been presented at various half-day tutorials offered at conferences, and is available from the Pilot website [1]. There are

three hands-on exercises, one shown in Fig. 3, which involve compiling and running a given C file, observing the output, and then introducing changes to accomplish certain tasks. For the first time, it has been possible to show students a graphical representation of exactly what these simple codes are doing. By walking through the Jumpshot display in conjunction with the source code, the entire multiprocessor execution and message communication pattern become quite clear. This helps students mentally establish the process/channel model of Pilot and navigate the learning curve involved in becoming comfortable with HPC programming. One could alternatively draw a diagram on the board, but the log visualization is automatic to prepare, more attractive, and one can interact with the display.

Fig. 3 lists lab2.c (colour-coded) along with its visual log when executed with six processes. Total execution time is under 3 ms. Observe how process 0 represents PI_MAIN, and 1 through 5 are instances of workerFunc. It is easy to see from the latter's red bars how each worker waits with two PI_Read calls to get the size of its work allocation (integer myshare), and then its data (buff array).[3] After the addition loop shown in gray, the short green bar shows it reporting its subtotal to PI_MAIN. Each of those communications has its counterpart in PI_MAIN with green bars representing the PI_Write calls, and red showing the subtotals being received. White arrows stand for messages. Coloured bars and yellow bubbles can be clicked for detailed information.

B. *Debugging aid to the programmer*

Another Pilot programming assignment was to read, in parallel, a 316MB .csv file of data on automotive collisions in Canada, with different worker processes starting from different file offsets, and then carry out a series of queries in parallel, merging the results. Two student submissions were notable for failing to exhibit any speedup. By generating the logs and displaying them in Jumpshot to the class, it took a matter of moments for the students to realize the mistakes. To start with, that something is wrong is obvious from the unfavourable ratio of gray computation to red blocking-read.

In Fig. 4, instance A, file reading runs from 0 to 1.1 seconds, then query processing continues on to 2 seconds. During file reading, the partial overlapping of gray bars show that the program was unable to fully parallelize the I/O. But more seriously, during query processing, it looks like pairs of PI_Write and PI_Read were called for each worker in a loop instead of all the PI_Writes (to distribute the work parcels) followed by all the PI_Reads. Thus, the program inadvertently serialized the calculations and the workers never did query processing in parallel at all.

In Fig. 5, instance B, the workers were kept waiting till PI_MAIN did 11 seconds of initialization, showing that the

[3]As of V2.1, one can replace the two PI_Reads and malloc call with a single PI_Read(..., "%^d", &myshare, &buff). (The PI_Writes must also be changed to match.) Pilot will automatically send the array length, store it in myshare and malloc the right-length array, returning the pointer in buff. The programmer is still responsible to eventually free buff. This change will be accurately reflected in the visual log, despite the fact that multiple MPI calls are made internally.

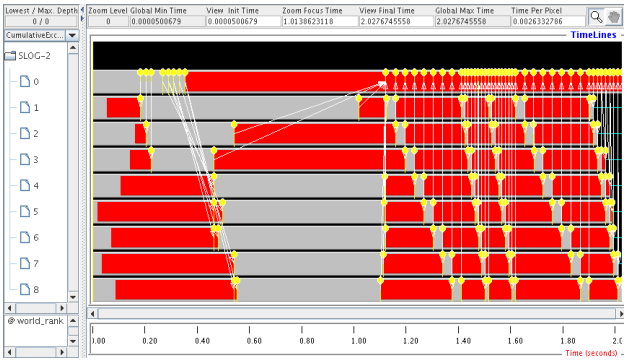

Figure 4. Instance A of student's program in Jumpshot

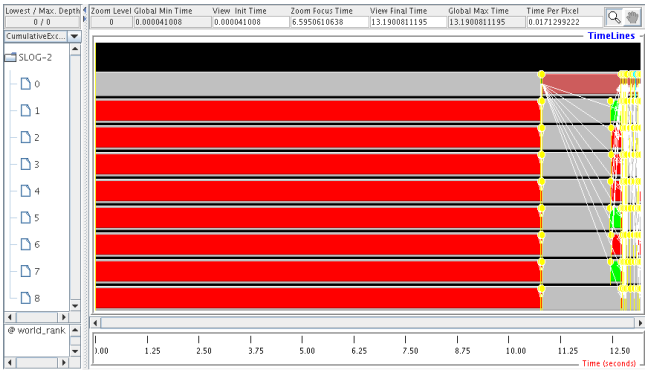

Figure 5. Instance B of student's program in Jumpshot

program did not succeed in parallelizing the reading of this big file, so the total run time always stayed nearly the same (since the calculations were fast).

These were not "bugs" in the sense of causing incorrect results, but they were bugs in parallelization that undermined the whole purpose of parallel programming: obtaining correct results in reduced time. Log visualization could also expose load imbalances among the worker processes and help the programmer, for example, adjust work granularity to provide a more even distribution, or perhaps switch from a static to a dynamic work allocation scheme.

V. Conclusion and Future Work

This work has succeeded in providing Pilot programmers with a greatly enhanced call logging feature, such that logfiles are automatically generated with a single command-line option, and are easy to view and manipulate in Jumpshot. This is useful both for instructors of message-passing programming as a pedagogical tool for illustrating the complex parallel interactions of processes at run time, and for novice programmers in debugging logic that impedes parallelism.

An important lesson learned is that because the visualization feature represents "another tool," and by nature it seems that "users resist tools" ([21] raised in context of MPI debugging), the class should have been given an exercise where they have to generate a log, display it in Jumpshot, and submit the screenshot or have it viewed by a lab assistant. Once convinced about its ease of use and potential helpfulness, probably no further encouragement would be needed.

Pilot can be obtained from its website [1] and MPE/ Jumpshot from Argonne National Lab [22]. Note that Jumpshot should be run with Java JDK 1.8 or higher, otherwise font problems were noticed.

In terms of future work, we would like to solve the problem of losing the MPE logfile if the program aborts. This problem is not considered grave because when Pilot detects an error (and it also intercepts MPI errors), diagnostics are printed that pinpoint the problem right to the line of source code, thus the programmer is likely not relying on the log to fix such reported errors. Nonetheless, it would be better if the MPE log could be finalized in all cases, and this will be a subject of future efforts.

ACKNOWLEDGMENT

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

REFERENCES

[1] William Gardner. Pilot homepage [online, cited 4/Nov/16]. Available from: http://carmel.socs.uoguelph.ca/pilot/.

[2] C. A. R. Hoare. Communicating Sequential Processes. In Per Brinch Hansen, editor, *The Origin of Concurrent Programming: From Semaphores to Remote Procedure Calls*, pages 413–443. Springer New York, New York, NY, 2002.

[3] William B. Gardner, John D. Carter, and Gary Grewal. The Pilot approach to cluster programming in C. In *IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 1–8, Atlanta, April 2010. IEEE Press.

[4] 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.

[5] William Gardner and John Carter. Using the Pilot library to teach message-passing programming. In *Proc. of the Workshop on Education for High-Performance Computing (EduHPC)*, pages 1–8, New Orleans, Nov. 2014. IEEE Press.

[6] Athanasios I. Margaris. Log file formats for parallel applications: A review. *International Journal of Parallel Programming*, 37:195–222, Feb. 2009.

[7] Tianyi Bao. Visualization tool for debugging Pilot cluster programs. Master's thesis, School of Computer Science, University of Guelph, Dec. 2016. Available from: http://hdl.handle.net/10214/10193.

[8] J. Chassin de Kergommeaux, B. Stein, and P.E. Bernard. Pajé, an interactive visualization tool for tuning multi-threaded parallel applications. *Parallel Computing*, 26(10):1253–1274, Aug. 2000.

[9] Wolfgang E. Nagel, Alfred Arnold, Michael Weber, Hans-Christian Hoppe, and Karl Solchenbach. VAMPIR: Visualization and analysis of MPI resources. *Supercomputer*, 12(1):69–80, May 1996.

[10] Kevin Coulomb, Mathieu Faverge, Johnny Jazeix, Olivier Lagrasse, Jule Marcoueille, Pascal Noisette, Arthur Redondy, and Clément Vuchener. Visual trace explorer [online, cited 11/ Nov/16]. Available from: http://vite.gforge.inria.fr/.

[11] Lucas Mello Schnorr, Guillaume Huard, and Philippe O.A. Navaux. Triva: Interactive 3D visualization for performance analysis of parallel applications. *Future Generation Computer Systems*, 26(3):348–358, Mar. 2010.

[12] Youn Kyu Lee, Jae young Bang, Joshua Garcia, and Nenad Medvidovic. ViVA: A visualization and analysis tool for distributed event-based systems. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 580–583, Hyderabad, India, 2014. ACM.

[13] S. Hunold, R. Hoffmann, and F. Suter. Jedule: A tool for visualizing schedules of parallel applications. In *39th International Conference on Parallel Processing Workshops (ICPPW)*, pages 169–178. IEEE, 13-16 Sep. 2010.

[14] Anthony Chan, David Ashton, Rusty Lusk, and William Gropp. *Jumpshot-4 Users Guide*. Mathematics and Computer Science Division, Argonne National Laboratory, Nov. 2007. Available from: ftp://ftp.mcs.anl.gov/pub/mpi/slog2/js4-usersguide.pdf.

[15] Allinea Software. *DDT User Guide: Version 2.5*, 2009. Available from: http://geco.mines.edu/compilerDocs/allinea/ddt/userguide.pdf.

[16] Sameer S. Shende and Allen D. Malony. The Tau parallel performance system. *Intl. Journal of High Performance Computing Applications*, 20(2):287–311, May 2006.

[17] Robert Bell, Allen D. Malony, and Sameer Shende. ParaProf: A portable, extensible, and scalable tool for parallel performance profile analysis. In Harald Kosch, László Böszörményi, and Hermann Hellwagner, editors, *Euro-Par 2003 Parallel Processing*, volume 2790 of *Lecture Notes in Computer Science*, pages 17–26, Klagenfurt, Austria, 26-29 Aug. 2003. Springer Berlin Heidelberg.

[18] Felix Schmitt, Robert Dietrich, René Kuß, Jens Doleschal, and Andreas Knüpfer. Visualization of performance data for MPI applications using circular hierarchies. In *First Workshop on Visual Performance Analysis (VPA)*, pages 1–8, New Orleans, 21 Nov. 2014. IEEE.

[19] Anthony Chan, William Gropp, and Ewing Lusk. *User's Guide for MPE: Extensions for MPI Programs*, 1998. ANL/MCS-TM-ANL-98/xx. Available from: ftp://ftp.mcs.anl.gov/pub/mpi/mpeman.pdf.

[20] Anthony Chan. [mvapich-discuss] clog2TOslog2 fails on MPI_Test [online]. 1 Apr. 2011 [cited 20/Nov/16]. Available from: http://mailman.cse.ohio-state.edu/pipermail/mvapich-discuss/2011-April/0032%80.html.

[21] 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.

[22] Argonne National Laboratory. Performance visualization for parallel programs: Download [online]. Available from: http://www.mcs.anl.gov/research/projects/perfvis/download/.