Multiprocessing for Pd

Miller Puckette
University of California, San Diego
msp@ucsd.edu

Abstract

A new `pd~` *object has been added to Pd (version 0.42) to allow users to embed separate Pd processes inside each other, so that the OS can schedule the processes on separate CPUs. This paper discusses the design of pd~in the context of a general reflection on real-time media computation.*

1 Introduction

Now that it's getting hard to buy a fast uniprocessor, it becomes interesting to be able to run Pd (or anything else CPU intensive) efficiently on multiprocessors. Pd is an especially tricky case because it has to run in real time, and also because the applications it is used in are so widely varied. On the other hand, the "dataflow" character of Pd makes parallelization somewhat easier than it would be in an environment (such as a database system or even a traditional programming environment) in which transactions have return values.

2 Pd programming model and scheduler

The basic orientation of Pd is computing audio signals, and its design reflects that closely. Pd's fundamental unit of time is a tick, defined to be 64 audio samples. What Pd does, when it runs, is to run a loop in which it (1) checks for messages from input devices or timeouts; (2) reads a tick's worth of audio input; (3) runs its DSP objects forward one tick; and (4) writes a tick's worth of audio output. Anything that is not audio (video, for example) is dealt with under the audio-driven scheduler, and so in effect it is resampled, with some irregularity, to the audio clock (i.e., frames look like a sequence of audio sample time stamps.)

Audio ticks have time stamps, but time resolution for messages in Pd is finer than that, being at least fine enough to distinguish individual audio samples. (Time stamps in Pd are implemented as double floating point numbers and so should have at least sample resolution for the first few thousand years of running time). Messages occur at specific time stamps which may be equal to a tick's time stamp or not; although all such messages are processed between ticks of audio processing, they are nonetheless processed in order of increasing time stamp, and objects such as `vline~` can use time stamps to offer sub-sample-accurate control over audio computation.

Inputs from I/O devices may generate messages in Pd; in this case, it is up to the device handler to assign an appropriate time stamp to the message. This is usually only done to the accuracy of a tick. (In Pd's implementation today, this assignment of time stamps is quite sloppily done and needs improvement!) Messages resulting from timeouts (such as from the `delay` object) are timed "exactly" except for the possibility of truncation error in the time stamp. To avoid these truncation errors the time stamps are scaled so that samples and milliseconds are both realized as integers, so that any combination of ticks and millisecond delays can be represented exactly.

Pd patches can measure time using `timer` objects. These do not measure "real" time intervals, but differences between time stamps of messages. As a result of this and some other precautions, Pd patches run deterministically in that, if one is run twice with inputs (if any) that are the same and receive the same time stamps, the two results will be identical.

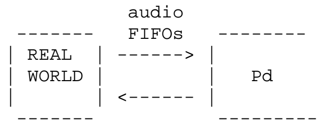
2.1 Batch processing; an aside

The real world need not be considered as a real-time device tied in real time to Pd via FIFOs. If we get rid of the real world, what is left is batch processing. As of Pd 0.42, the "-batch" flag is available in Pd to allow a single Pd process to run without audio input or output. This can be useful for writing shell-based scripts that process soundfiles through Pd patches.

2.2 FIFOs to the real world

The Pd scheduler makes sure that computation corresponding to a given time stamp occurs approximately at the corresponding "real" (external) time. This is accomplished by tying Pd's audio sample inputs and outputs to the "real world"

over a pair of FIFOs:



Here the "real world" generates samples of audio input and receives samples of audio output at a continuous rate. The Pd process, on each scheduler tick, reads one tick of audio input and writes one tick of audio output. The sum of the number of sample frames in the two FIFOs is thus a constant that may be specified at startup. This amount, which can be converted to units of time, is the *latency* (L) of the Pd process.

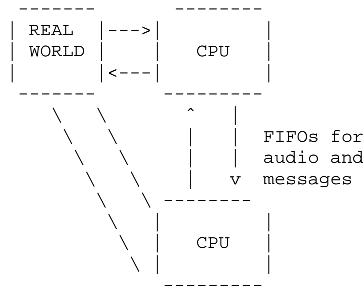
Viewed in these terms, if the input FIFO has at least a tick's worth of samples in it, the Pd process is runnable, in the sense that it may read input and write output without having to wait. When the FIFO contains less than one full tick, Pd can't run and has to wait for more input samples. In this situation the output FIFO contains nearly the entire latency's worth of samples (up to about one tick less). Put another way, Pd is then computing samples L time units ahead of when they will be heard, or yet again, Pd's time stamp has advanced to real time plus L, which is as far in advance of real time as Pd will compute audio.

On the other hand, Pd can get behind in its computation, and this has the effect that the number of input samples available increases and the number of samples in the output FIFO decreases. This is no problem unless the output FIFO empties entirely, so that there is no sample available to provide to the real world. In this case Pd's computation is late; the sound output either skips or repeats (depending on the audio hardware and driver). If Pd's current time stamp is one tick ahead of real time, Pd is not late yet. So when things are running correctly, the currently computing time stamp can wander as far as L ahead of real time.

Even if things aren't running correctly in the real-time sense, Pd still maintains deterministic audio and message computation - in Pd's programming model, everything is fine except that the real world is running at the wrong speed.

2.3 The Max/FTS way

The FTS ("faster than sound") system ran on IRCAM's ISPW system, from about 1990 to 1995 (Puckette 1991). The ISPW had 6 processors each running at 40 MHz. FTS was perhaps the first real-time multiprocessing system for audio that could mix real-time control and audio computations deterministically, and it did so using an extension of the FIFO model shown above. In FTS, the real world and the six processors were joined in a 7-node complete graph as shown (with only two CPUs instead of the six):



Each processor was assigned a latency that the user believed was at least as large as the amount that that processor could get behind in its calculations. The round-trip delay between any processor and the real world was simply that processor's latency; the round-trip delay between two given processors was the sum of their two latencies.

In FTS, the FIFOs between processors carried not only audio signals but also Max-style messages. On each tick, each processor read all the messages going to it for that tick; carried out all the message processing for the tick, possibly engendering messages to the FIFOs to other processors; and then read all the audio signals for the tick, did one tick of DSP computation, and wrote the resulting audio to each of its output FIFOs.

The number of audio signals streaming down any given FIFO was variable, because the user could add or delete audio connections while the system was running. This was tricky to code because the addition and deletion of audio signals in any FIFO had to be carefully synchronized between the two processors at the ends of the FIFO.

The "real world" was different from other CPUs only in that, first, it had no assigned latency, and second, it only sent and received audio, not messages. To be totally correct, control I/O should have also been sent to the processors over the FIFOs, but it was much easier in practice to handle control I/O *ad hoc*.

The Max/FTS system had a centralized patch editor (Max) that resided on an entirely different processor (that of the NeXT machine host) whereas, in Pd, the editor resides on the processor (and in the process) that also does the real-time computation. Since there is only one such process, there is no need to make a multi-processor distributed editor, which would probably have been prohibitively hard to write.

2.4 Can't it just be automatic?

In Max/FTS, the allocation of different parts of a multi-window patch was handled by the user, who was responsible for deciding which window should run on which processor. All the objects in any particular window ran on the same processor; i.e., there was no fine-grained assignment of objects

to CPUs, but only assignment of windows to CPUs.

Since at least 1990, users and critics of Max/FTS have observed that it would be desirable for objects to be automatically allocated to processors in a way that would minimize the bandwidth of interconnections between the objects. This would free the user from the cumbersome task of understanding the actual flow of data between objects in the patch; the software would automatically assess that.

This didn't prove practical, for two reasons. First, as has long been well known, one can't compute the quantity of data that will flow between any given pair of objects in a patch (at least, not if the patching language is able to solve arbitrary computing problems). Predicting how much data will flow where is hopeless.

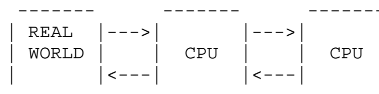
The second problem is that nobody has been able to make an expressive patching language that doesn't depend on objects sharing data. In Max/FTS (and in Pd as well) this takes the form of "named" objects such as arrays. Any automatic distribution of patches that allows accessing arrays would have to place every object that accesses any particular array on the same processor, or else use some kind of locking mechanism that would be unlikely to work in real time. Also, any situation in which there is of recombination of message fanout would require that both message paths be synchronized, i.e., that both message paths go through the same itinerary of processors or be otherwise delay-equalized. In combination, these constraints would require that, for complete transparency, almost any interesting patch would have to reside on a single processor. It appears to be an inescapable fact that multiprocessing has non-hideable effects on the execution of "patches" and can't effectively be carried out without the user's active participation.

3 Embedding Pd in Pd

Pd was originally intended to run on single-processor systems. This was a good choice for the first ten years of Pd's existence (1996-2006), but lately the processor pushers are making it hard not to buy multiprocessors—the fastest uniprocessors on the market are apparently slower than even a single processor on the current 2-processor lines. The last three years have seen no growth to speak of in uniprocessor performance. Although the CPU manufacturers are making louder and louder claims about their CPUs' increasing performance per watt, in fact they are building machines that function primarily as high-wattage memory architectures optimized to serve multiprocessors at the expense of uniprocessors. Wattage is increasingly offloaded from the CPU onto the motherboard, which must of course be designed to deal with the maximum possible number of "cores" that are compatible with any particular socket specification—no matter that anyone who only needs a uniprocessor will also pay (especially in wattage) for a support infrastructure designed for a multiprocessor.

A good response to this state of affairs might be to embrace multiprocessing, but only somewhat cautiously, as one would embrace a porcupine. The `pd~` object is an attempt to extend Pd to allow using the extra CPU cycles that a multiprocessor offers, but in a way that doesn't assume that multiprocessing is the great answer to all our computing problems. The approach taken is partly inspired by a desire to change Pd itself as little as possible.

The `pd~` object simply embeds one Pd process inside another, allowing the OS to determine what CPU to run which process on. (If the processes are CPU intensive, presumably the scheduler will tend to run them on different CPUs.) The embedded Pd sub-process has no direct access to audio devices. Instead, `adc~` and `dac~` objects in the sub-process read and write audio from the audio inlets and outlets of the `pd~` object in the originating super-process. This is managed by setting up a pair of FIFOs from the super-process to the sub-process, so that a setup with one `pd~` object would look like this:



```
 -------        -------        -------
| REAL  |--->|        |--->|        |
| WORLD |    |  CPU   |    |  CPU   |
|       |<---|        |<---|        |
 -------        -------        -------
```

Adding more `pd~` objects and their sub-processes would expand this into a rooted tree structure. Most likely, a typical application would have only one super-process with the desired number of sub-processes immediately hanging from it, making a star shape. Compared with the FTS picture, this has the disadvantage of higher audio latencies between the sub-processes and the Real World, and also in communications between sub-processes, all of which traffic must pass through the super-process which then adds its own latency. (There is also an extra stage of copying the signals, adding CPU load, compared with Max/FTS.)

The sub-process gets its own editor (unless this is suppressed using the -nogui flag), and the user has to manage separate Pd documents for separate processes. This wasn't necessary in Max/FTS because, there, the editor was a separate process that managed all the distributed parts of a global patch.

The Max/FTS complete-graph model would be hard to reproduce in Pd, for the above reason and one other practical one: it is very hard in off-the-shelf OSes to simultaneously control FIFO fill points between the audio driver and more than one process. On the ISPW this was feasible only because the audio driver was designed (by Bennett Smith) to work with Max/FTS.

3.1 Messages in the FIFOs

The FIFOs connecting the main Pd process with the "Real World" carry audio only and are maintained by the operating system; but the FIFOs between Pd super-processes and sub-processes carry both audio and messages. In this way, the ordering between audio and message computation is maintained across FIFOs, and so is message order specified by "trigger" objects and the like. The result is fully deterministic (with the proviso, as before, that the Real World act consistently).

At the user level, in the super-process, messages can be sent to the `pd~` object that specify any "receive" or other named object on the sub-process, and a message to forward to it. In the sub-process, a "stdout" object gives an explicit portal to the super-process; messages sent to "stdout" appear at a message outlet on the `pd~` object on the super-process.

3.2 Implementation

The implementation of `pd~` only requires a standard external object and a plug-in scheduler. (However, it was necessary to slightly adjust the scheduler API to make it all work; the changes appeared in Pd version 0.42.) The implementation code lives in the "extra" directory, thereby leaving open the possibility of introducing a better mechanism someday without losing backward compatibility. Two files, "pd~.c" and "pdsched.c", supply the object itself and the plug-in scheduler for the sub-process.

The FIFOs are implemented as regular Unix pipes. (The current implementation therefore only runs on OSes that offer pipes; how to get `pd~` running in Microsoft's OS is unclear). The audio signals and messages are simply converted to ASCII for transmission down the pipes. Although, strictly speaking, the messages should be individually time-tagged, in this implementation the messages are simply assigned the logical time of the most recent audio block (in effect quantizing logical time). The messages are followed by an empty message (an extra ASCII semicolon) which heralds the beginning of the next audio block. The audio block is also terminated by a semicolon, which announces the next tick's worth of messages.

The workings of the `pd~` object can be described by what it does at startup, on receiving messages, at DSP ticks, and at shutdown, as follows:

```
Startup:
  create send and return FIFOs (pipes);
  create sub-process, passing user-supplied arguments;
  write L ticks' worth of zeroes to the "send" pipe;
  read and dispatch a tick's messages from "return" pipe

Message:
  write the message to the "send" pipe

DSP:
  write signal inputs to "send" pipe;
  read an audio tick from "return" pipe to signal outlets;
  read the next tick's worth of messages from "return" pipe
     and save them as NEXT-MESSAGES;
  set a timer (clock_delay(0)) to the present logical time
    (to get called back right after the DSP tick is done)

Timer:
  copy NEXT-MESSAGES to the message outlet

Shutdown:
  close the two pipes;
  wait() for subprocess to exit
```

The scheduler for the sub-process acts as a single routine that dispatches messages and DSP ticks, blocking as necessary to wait for input data. The routine works as follows:

```
start GUI (unless disabled)
while (not EOF on input pipe)
begin
  read a (semicolon terminated) audio tick from input FIFO
  advance logical time one tick (runs DSP and clock timeouts)
  poll for messages to or from GUI;
  print DSP audio output (dac~ buffers) to output FIFO;
  read one tick's messages from input FIFO and dispatch them
end
```

3.3 Current status

The `pd~` object compiles and runs as a Pd or Max object (the Max version is fun: it runs Pd patches inside Max ones). There are still some problems that need ironing out, though. Since the sub-process's scheduler is slaved to DSP ticks in the super-process, the sub-process freezes whenever DSP is turned off in the super-process. More seriously, there is a possibility of deadlock if the data in the FIFOs ever exceeds their combined sizes; pipes in UNIX are typically buffered to somewhere around 8K bytes. Even if only one of the two FIFOs blocks, real-time performance suffers.

It is too bad that the user has to specify at startup whether the sub-process is to have a GUI or not. It is best to suppress it for performances, but if something goes wrong it would be good to be able to start the GUI up on the fly. A restartable GUI would be desirable for other reasons anyway.

It's unclear how the ideas described here would hold up if Pd were being used to process images instead of (or in addition to) audio. One ramification of making image processing work the same as audio would be the loss of a simple and centralized mechanism for measuring time and scheduling.

In its current state, `pd~` is most easily used for offloading things like voice banks that are highly CPU intensive but have relatively simple communications with the rest of a patch. It is less clear how more "intelligent" sorts of processing can be offloaded to other processes.

References

Puckette, M. S. (1991). Fts: A real-time monitor for multiprocessor music synthesis. *Computer Music Journal 15*(3), 58–67.