Timestamp flow in GWB and correction
version 1.1

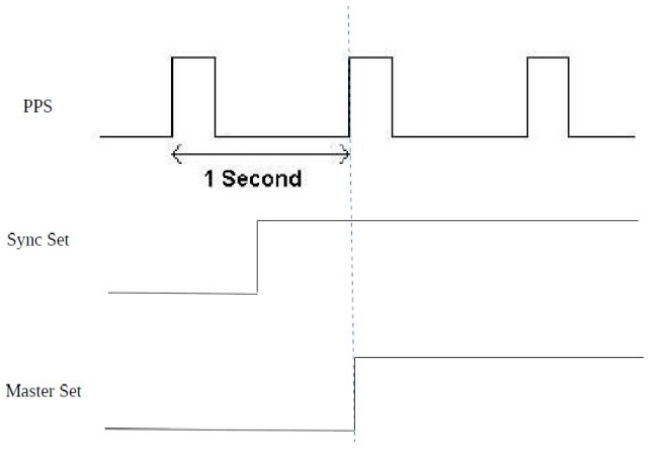
S.Harshavardhan Reddy

22/10/2019

In GWB release mode, timestamp which flows with visibility and beam has a time difference with the actual time at which the data is sampled. For visibility, the difference is not constant and it varies with the integration (LTA) in GWB. For beam data, the difference is constant. This report explains the flow of timestamp in GWB release mode and the time difference with the actual time.

1. Synchronizing the start of observation with GPS PPS

In the GWB, the start of an observation is synchronized with a 1-PPS signal derived from the observatory GPS receiver. Initially, all the ROACH boards are programmed, such that they wait for a “master set” signal in the FPGA to go high for starting the digitisation and packetisation. Then, an initialization (init) command is given to the GWB through an online control server by the operator. Five seconds after this ‘init’ command, the “sync set” signal in the FPGA is made high through a telnet port. After the “sync set” signal is made high, the “master set” signal goes high at the occurrence of the next 1-PPS pulse. Once the “master set” becomes high, packets are formed with the sampled data and sent to Compute nodes through 10GbE link. Along with the sampled data, packet number information is sent for every packet. Packets are received on Compute node through 10GbE NIC. In the first Compute node, upon receiving the first packet, the local time (rounded to the second) is stored in a timeval structure(start time). This start time is shared with the host node (gwbh6) and the rest of the nodes.



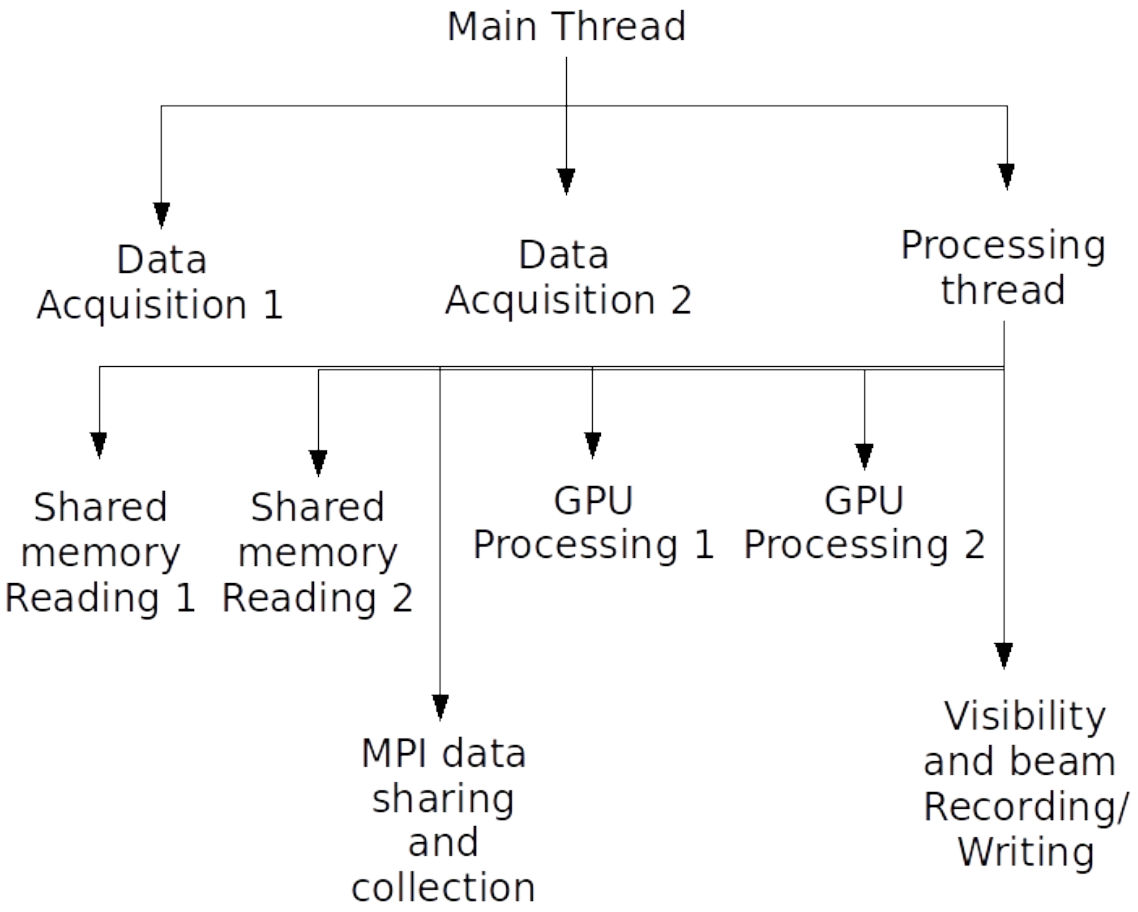
Synchronizing in the GWB using the GPS 1-PPS signal

2. Timestamp calculation for delay and fringe correction

The compute nodes process 256 MB of data in one iteration, which translates to 0.67108864 seconds (buffer time) of data. The timestamp sent for calculating delay and fringe values is obtained by incrementing buffer time to the start time at every iteration.

3. Data flow in GWB

Different sections of GWB are performed in parallel using OpenMP threads. Two threads (read threads) are assigned to read raw data from shared memory, each thread for two inputs. One thread (MPI thread) for doing MPI transfers i.e; time slicing and sharing of raw data between compute nodes and collection of results by host machines. Two threads (GPU threads) for GPU processing by both GPUs, each GPU processing half the time slice data. One thread (write thread) for writing results on to shared memories for being recorded. Double buffering ping-pong scheme is used to prevent overlap of processing and reading/writing the buffers.



OpenMP threads structure in GWB

In the first iteration, first buffer (256 MB) sampled at the start of observation (T0) is read from the shared memory by read threads. At the same time, MPI threads, GPU threads and read threads performs processing/IO on junk data. In the second iteration, read threads read the second buffer's (T1) data, MPI threads perform time slicing and sharing on first buffer's data (T0), and the rest of the threads performs processing/IO on junk data. In the third iteration, read threads read third buffer's data (T2), MPI threads perfrom time slicing and data sharing on T1 data and GPU threads perform correlation and beam formation processing on T0 data. In the fourth iteration, read threads read fourth buffer (T3), MPI thread performs time slicing and sharing on T2 data, GPU threads process T1 data and MPI threads perform results collection on T0 data. In the fifth iteration, read threads read fifth

buffer (T4), MPI thread perform time slicing and sharing on T3 data and results collection on T1 data, GPU threads process T2 data and write thread performs writing on T0 data's results. The above mentioned scenario is for integration of one buffer i.e; LTA = 1. For higher LTA values, the GPU threads process and hold the visibility data for the number of buffers to be integrated (LTA). MPI collection of results and writing of results is distributed among LTA number of iterations.

	Read threads (both threads)		MPI Sharing (Time slicing and data sharing)		GPU processing (both GPU threads)		MPI Gathering (Visibility and beam data)		Recording (Writing the results into SHM)
Iteration 0	Read buffer 0	Buffer 0 (T0)	MPI buffer 0	junk	visi buffer 0	junk	output buffer 0	junk	junk
	Read buffer 1	junk	MPI buffer 1	junk	visi buffer 1	junk	output buffer 1	junk	
Iteration 1	Read buffer 0	Buffer 0 (T0)	MPI buffer 0	Buffer 0 (T0)	visi buffer 0	junk	output buffer 0	junk	junk
	Read buffer 1	Buffer 1 (T1)	MPI buffer 1	junk	visi buffer 1	junk	output buffer 1	junk	
Iteration 2	Read buffer 0	Buffer 2 (T2)	MPI buffer 0	Buffer 0 (T0)	visi buffer 0	Buffer 0 (T0)	output buffer 0	junk	junk
	Read buffer1	Buffer 1 (T1)	MPI buffer 1	Buffer 1 (T1)	visi buffer 1	junk	output buffer 1	junk	
Iteration 3	Read buffer 0	Buffer 2 (T2)	MPI buffer 0	Buffer 2 (T2)	visi buffer 0	Buffer 0 (T0)	output buffer 0	Buffer 0 (T0)	junk
	Read buffer 1	Buffer 3 (T3)	MPI buffer 1	Buffer 1 (T1)	visi buffer 1	Buffer 1 (T1)	output buffer 1	junk	
Iteration 4	Read buffer 0	Buffer 4 (T4)	MPI buffer 0	Buffer 2 (T2)	visi buffer 0	Buffer 2 (T2)	output buffer 0	Buffer 0 (T0)	Buffer 0 (T0)
	Read buffer 1	Buffer 3 (T3)	MPI buffer 1	Buffer 3 (T3)	visi buffer 1	Buffer 1 (T1)	output buffer 1	Buffer 1 (T1)	
Iteration 5	Read buffer 0	Buffer 4 (T4)	MPI buffer 0	Buffer 4 (T4)	visi buffer 0	Buffer 2 (T2)	output buffer 0	Buffer 2 (T2)	Buffer 1 (T1)
	Read buffer 1	Buffer 5 (T5)	MPI buffer 1	Buffer 3 (T3)	visi buffer 1	Buffer 3 (T3)	output buffer 1	Buffer 1 (T1)	
Iteration 6	Read buffer 0	Buffer 6 (T6)	MPI buffer 0	Buffer 4 (T4)	visi buffer 0	Buffer 4 (T4)	output buffer 0	Buffer 2 (T2)	Buffer 2 (T2)
	Read buffer 1	Buffer 5 (T5)	MPI buffer 1	Buffer 5 (T5)	visi buffer 1	Buffer 3 (T3)	output buffer 1	Buffer 3 (T3)	
Iteration 7	Read buffer 0	Buffer 6 (T6)	MPI buffer 0	Buffer 6 (T6)	visi buffer 0	Buffer 4 (T4)	output buffer 0	Buffer 4 (T4)	Buffer 3 (T3)
	Read buffer 1	Buffer 7 (T7)	MPI buffer 1	Buffer 5 (T5)	visi buffer 1	Buffer 5 (T5)	output buffer 1	Buffer 3 (T3)	
Iteration 8	Read buffer 0	Buffer 8 (T8)	MPI buffer 0	Buffer 6 (T6)	visi buffer 0	Buffer 6 (T6)	output buffer 0	Buffer 4 (T4)	Buffer 4 (T4)
	Read buffer 1	Buffer 7 (T7)	MPI buffer 1	Buffer 7 (T7)	visi buffer 1	Buffer 5 (T5)	output buffer 1	Buffer 5 (T5)	

indicates block that is being processed

Data flow between OpenMP threads in GWB (LTA = 1)

4. Flow of timestamp along with visibility data

In the first iteration, start of observation time (start time) is shared by first compute node to all the other nodes (host and compute). Visibility host calculates the timestamp for delay and fringe value calculations. The timestamp and its corresponding delay and fringe values are calculated one iteration in advance and shared with the compute nodes and applied in the next iteration in phase shifting kernel in GPU processing. Hence, in the third iteration, timestamp of T1 is calculated, its delay and fringe values calculated and shared by MPI thread. GPU threads process T1 buffer in the fourth iteration and the delay and fringe values are applied correspondingly.

The timestamp calculated is stored in a timestamp buffer. The timestamp buffer is an array of three elements of timeval structure. The iteration count ('run count') runs from 0, and the timestamp calculated in the current iteration is stored in timestamp buffer at (run

count % 3). The visibility data integrated in the GPUs and stored in global memory of GPUs, is copied into the CPU memory once in LTA number of iterations at the end of every (run count + 1) % LTA = 0 iteration. Collection of visibility data by MPI thread starts at next (run count + 1) % LTA = 0 iteration and is completed LTA buffers later. At the end of next (run count + 2) % LTA = 0 iteration which is LTA buffers later, visibility data and timestamp are written into the visibility shared memory in visibility host node.

Tracking the actual time at which the visibility data integration started and the timestamp that is sent to the shared memory buffer gives the difference between the timestamp (sent to visibility shared memory) and actual time. As the T0 buffer GPU processing starts at third iteration (run count = 2), and the visibility data integration starts at every run count % LTA = 0 iteration, the actual time associated with the visibility buffer is T(run count - 2). Now, assume that the iteration at which the integration started as 'ref count', then its corresponding actual time is T(ref count - 2). At the end of next (run count + 1) % LTA = 0 iteration i.e; at ref count + LTA – 1 iteration, visibility data (say V) is copied into CPU memory from GPU memory. At the start of next (run count + 1) % LTA = 0 i.e; at ref count + (2 x LTA) – 1 iteration, MPI collection of visibility data V starts and LTA buffers later collection finishes i.e; at ref count + (3 x LTA) – 2 iteration. At the end of next (run count + 2) % LTA = 0 iteration i.e; at ref count + (4 x LTA) – 2 iteration, visibility data V is written into visibility shared memory. The timestamp from the timestamp buffer written into the shared memory is from the ((run count - 2) % 3) element in the array. Now depending on the LTA, the difference between the actual time (of start of integration of visibility buffer V) and timestamp written into the visibility shared memory varies.

Timestamp buffer (run count % 3)	Timestamp	Iteration number		Processing description
0		Iteration 0		
1	T0	Iteration 1		
2	T1	Iteration 2		
...	...	...		
...	...	...		
(ref count % 3)	T(ref count – 1)	Iteration 'ref count'	----->	Start of integration of visibility 'V'
...	...	...		
...	...	...		
(ref count + LTA – 1) % 3	T(ref count + LTA)	Iteration 'ref count + LTA – 1'	----->	End of integration of visibility 'V'
...	...	...		
...	...	...		
(ref count + 2 x LTA – 1) % 3	T(ref count + 2 x LTA)	Iteration 'ref count + 2 x LTA – 1'	----->	Start of MPI collection of visibility 'V'
...	...	...		
...	...	...		
(ref count + 3 x LTA – 2) % 3	T(ref count + 3 x LTA – 1)	Iteration 'ref count + 3 x LTA – 2'	----->	End of MPI collection of visibility 'V'
(ref count + 3 x LTA – 1) % 3	T(ref count + 3 x LTA)	Iteration 'ref count + 3 x LTA – 1'	----->	Start of collection of visibility 'V' into LTA buffer
...	...	...		
...	...	...		
(ref count + 4 x LTA – 4) % 3	T(ref count + 4 x LTA – 3)	Iteration 'ref count + 4 x LTA – 4'		
(ref count + 4 x LTA – 3) % 3	T(ref count + 4 x LTA – 2)	Iteration 'ref count + 4 x LTA – 3'		
(ref count + 4 x LTA – 2) % 3	T(ref count + 4 x LTA – 1)	Iteration 'ref count + 4 x LTA – 2'	----->	End of collection of visibility 'V' into LTA buffer and written into visibility shared memory. Timestamp written into the shared memory buffer is (run count – 2) % 3

Visibility data and timestamp flow with iterations

So, for LTA = 1, the timestamp is advanced by one buffer (0.671 seconds) to the actual time. For LTA = 2, the timestamp is advanced by five buffers (3.35 seconds), for LTA = 4, timestamp is advanced by 13 buffers (8.72 seconds), for LTA = 8, the timestamp is advanced by 29 buffers (19.46 seconds), for LTA = 16, the difference is 61 buffers (40.93 seconds) and LTA = 32, it is 125 buffers (83.88 seconds) offset.

5. Flow of timestamp along with beam data

For beam data, LTA does not matter. For every buffer (256 MB) beam data is written into shared memory. At the third iteration, T0 buffer is processed by GPU threads, and in the fourth iteration T0 buffer MPI collection is performed by MPI thread. For IA/PA beam, T0 buffer beam data is written to beam data shared memory in fifth iteration. Along with beam data, timestamp is written into beam data shared memory. The timestamp is taken from the timestamp buffer from (run count - 1) % 3 element of the timestamp buffer which has timestamp of T2. Hence, for IA/PA beam data the timestamp is advanced by 2 buffers (1.34 seconds).

For voltage beam (CD Pipeline), after T0 buffer MPI collection in the fourth iteration, because of bug in the code, T0 buffer beam data is written to beam data shared memory for CD pipeline operation in sixth iteration. Hence, for voltage beam data, the timestamp is advanced by 3 buffers (2.01 seconds).

6. Correction of timestamp for visibility data

For correcting the timestamp difference, timestamp calculated (as mentioned in section 2) is stored in a new timestamp buffer (array of four elements, each element is a timeval structure). The timestamp is stored once in every LTA buffers at (run count % LTA) = 0 iteration. At every (run count % LTA) = 0 iteration, a new iteration counter (‘visi count’) is incremented. This new counter ‘visi count’ is used to address the new timestamp buffer at (visi count % 4) into which timestamp is stored. The timestamp stored is from the previous iteration as the timestamp is calculated one buffer in advance (to get delay and fringe values and to be shared to all the nodes for correction). At start of every run count % LTA = 0 iteration, visibility integration starts and ends LTA buffers later i.e; at next (run count + 1) % LTA = 0 and the visibility data is copied into ‘visibility buffer’. At the next ‘run count’, MPI collection of visibility data starts and ends LTA buffers later i.e; at next (run count + 1) % LTA = 0. At the next ‘run count’, visibility data collection into ‘LTA buffer’ starts and LTA buffers later i.e; at next (run count + 1 % LTA) = 0 collection into ‘LTA buffer’ ends and is written into shared memory. Along with visibility data, timestamp is picked from new timestamp buffer and written into shared memory. The timestamp picked from the new timestamp buffer is addressed at (visi count -2) % 4.

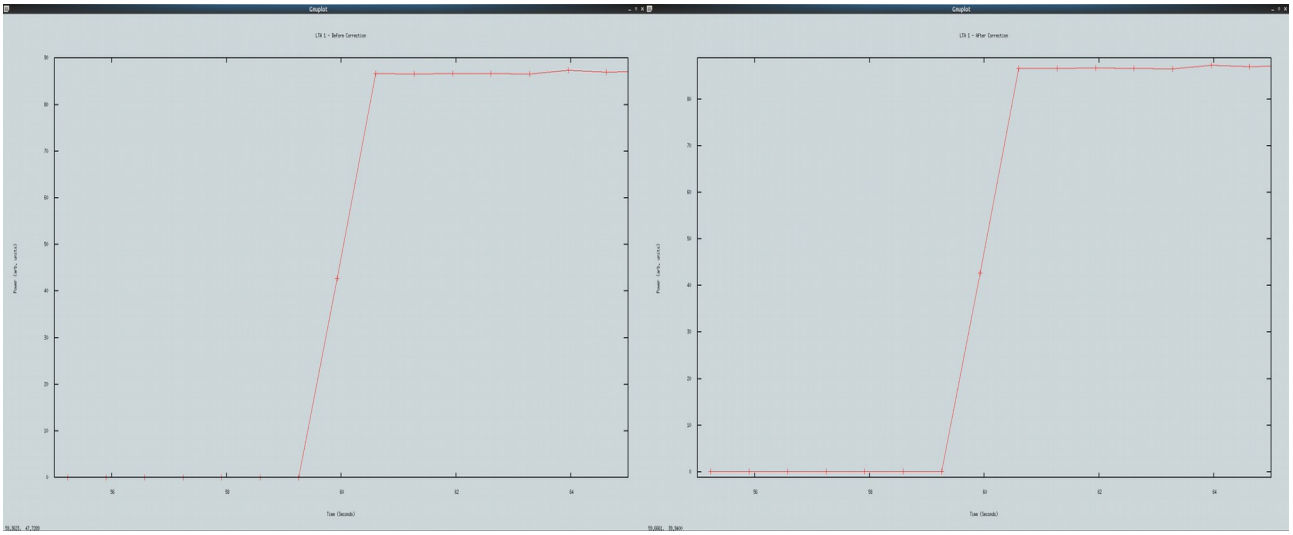
Timestamp buffer (run count % 4)	visi count	Timestamp	Iteration number		Processing description
0	0		Iteration 0		
1			Iteration 1		
2		T0	Iteration 2		
...	...	...	...		
...		...	...		
(visi count % 4)	visi count	T(ref count – 2)	Iteration 'ref count'	----->	Start of integration of visibility 'V'
...		...	...		
...		...	...		
		T(ref count + LTA – 3)	Iteration 'ref count + LTA – 1'	----->	End of integration of visibility 'V'
(visi count + 1) % 4	visi count + 1	T(ref count + LTA – 2)	Iteration 'ref count + LTA'	----->	Start of MPI collection of visibility 'V'
...		...	...		
...		...	...		
		T(ref count + 2 x LTA – 3)	Iteration 'ref count + 2 x LTA – 1'	----->	End of MPI collection of visibility 'V'
(visi count + 2) % 4	visi count + 2	T(ref count + 2 x LTA – 2)	Iteration 'ref count + 2 x LTA'	----->	Start of collection of visibility 'V' into LTA buffer
		...	...		
...		...	...		
...		T(ref count + 3 x LTA – 3)	Iteration 'ref count + 3 x LTA – 1'	----->	End of collection of visibility 'V' into LTA buffer and written into visibility shared memory. Timestamp written into the shared memory buffer is (visi count % 4)

Visibility data and timestamp flow with iterations after correction

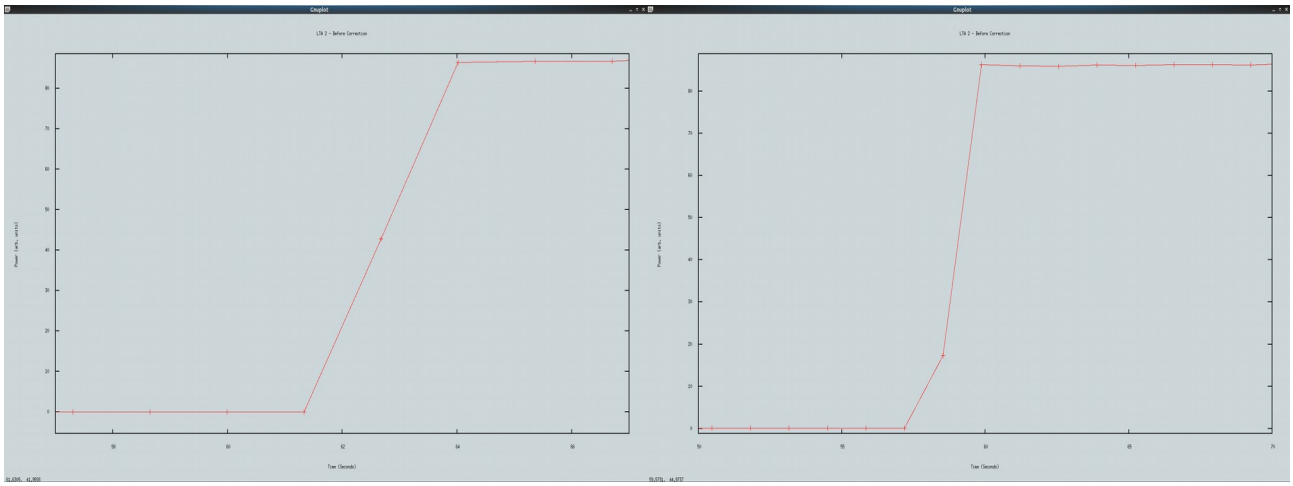
7. Experimental verification of time offset in visibility data

For this experiment, input used was broadband noise modulated by GPS PPM. PPM modulated noise was fed as input to GAB RF input of a polarization of an antenna (C00 pol1). GAB LO was set to 250 MHz, Band-3 RF bandpass filter was selected, 200 MHz Baseband Low pass filter was selected. The noise power level was adjusted such that ON pulse and OFF pulse noise is clearly visible. GWB settings are : 200 MHz bandwidth, 2048 spectral channels, total intensity mode. LTA is varied from 1 to 32 in steps of 2 i.e; 1,2,4,8,16,32. Integration is LTA value x 0.67108864 seconds.

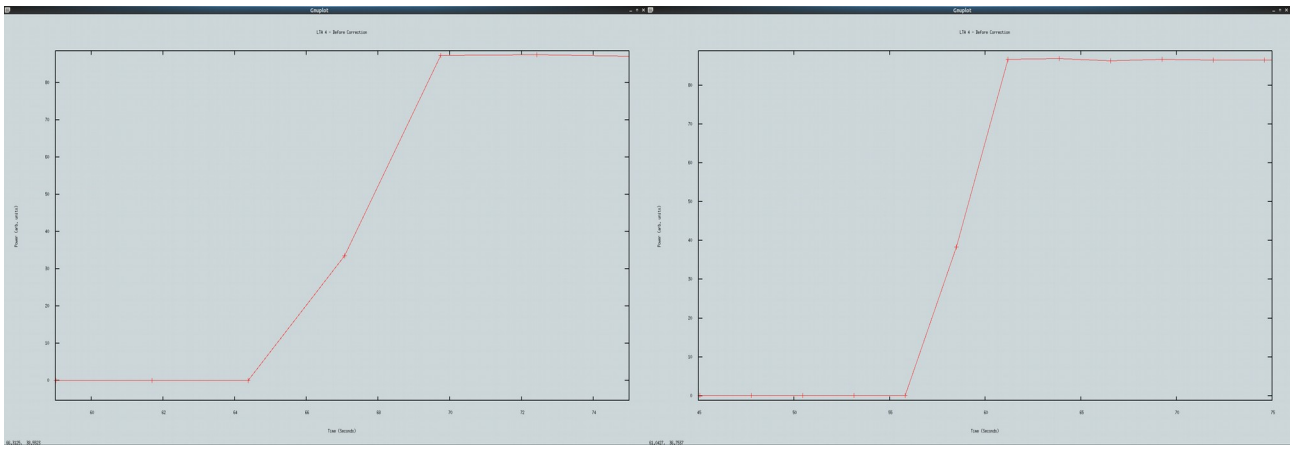
Using ‘xtract’ tool a single channel time series was extracted and plotted after extrapolating the timeseries from the start time of recording. Plots below show the time series of a single channel (800 channels) timeseries plotted with respect to extrapolated time from start time. As mentioned in section 4, it can be seen (on left side plots) that there is a timeoffset which is varying with LTA and it has been corrected (on right side plots). Exact amount of offset and the timestamp’s correctness is difficult to be seen as LTA increases as sample time increases.



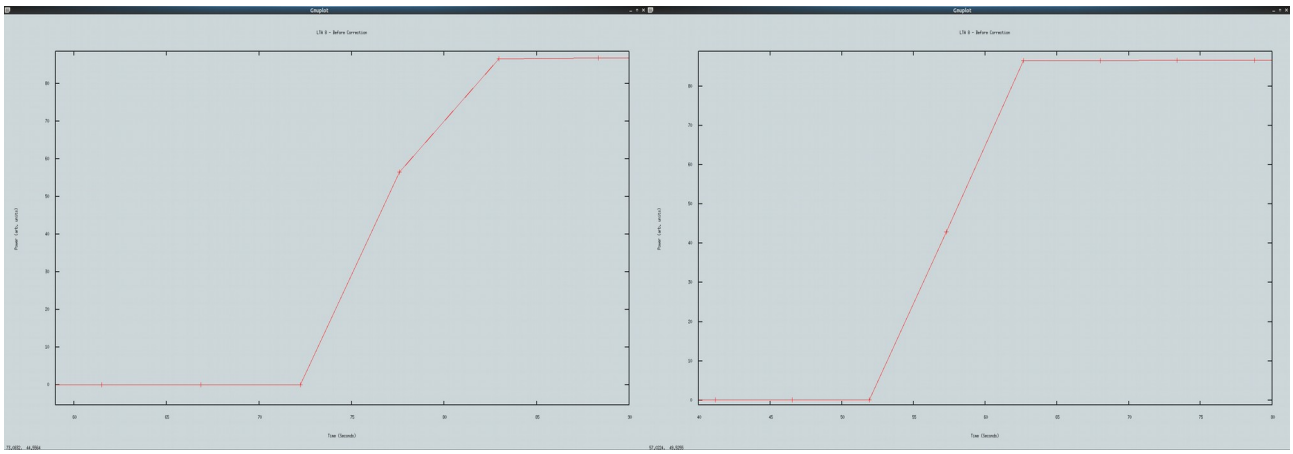
Plot showing timeseries of a channel from visibility data for LTA 1. Left side – Before time offset correction, Right side – Post time offset correction. Time offset before correction is 1 LTA buffer i.e; 0.67108864 seconds.



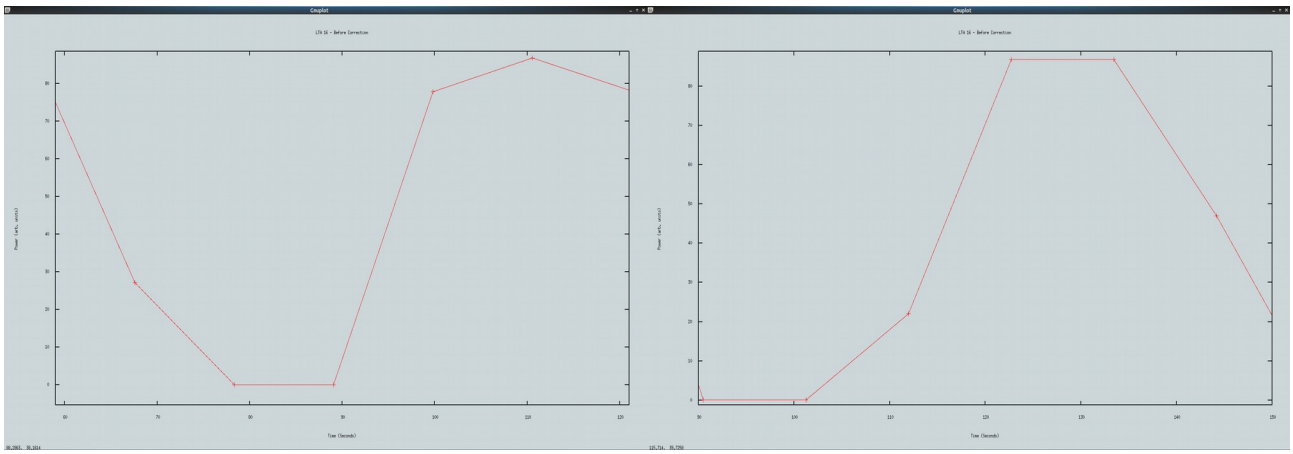
Plot showing timeseries of a channel from visibility data for LTA 2. Left side – Before time offset correction, Right side – Post time offset correction. Time offset before correction is 5 LTA buffers i.e; 3.35 seconds.



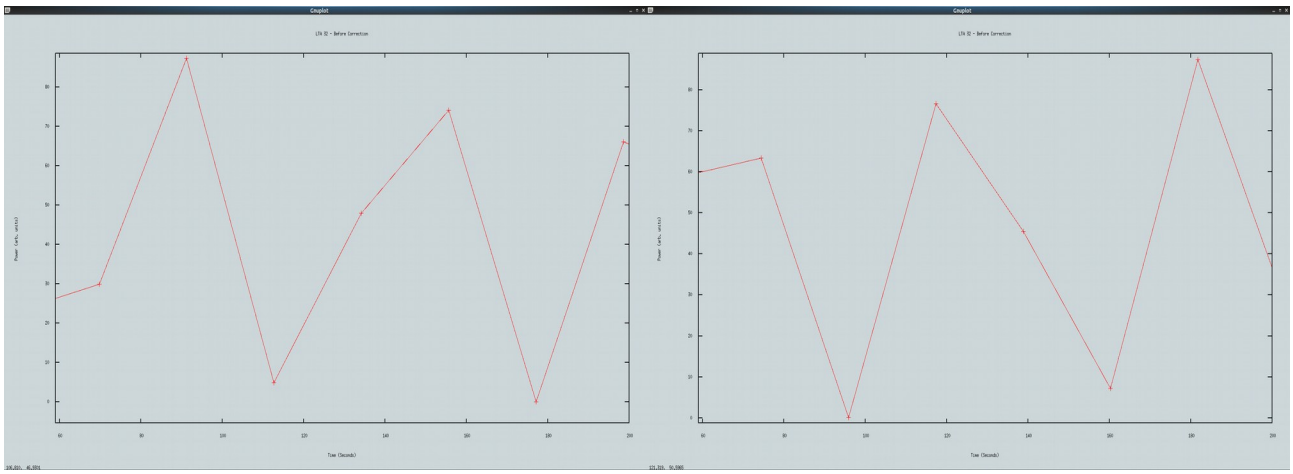
Plot showing timeseries of a channel from visibility data for LTA 4. Left side – Before time offset correction, Right side – Post time offset correction. Time offset before correction is 13 LTA buffers i.e; 8.72 seconds.



Plot showing timeseries of a channel from visibility data for LTA 8. Left side – Before time offset correction, Right side – Post time offset correction. Time offset before correction is 29 LTA buffers i.e; 19.46 seconds.



Plot showing timeseries of a channel from visibility data for LTA 16. Left side – Before time offset correction, Right side – Post time offset correction. Time offset before correction is 61 LTA buffers i.e; 40.93 seconds.



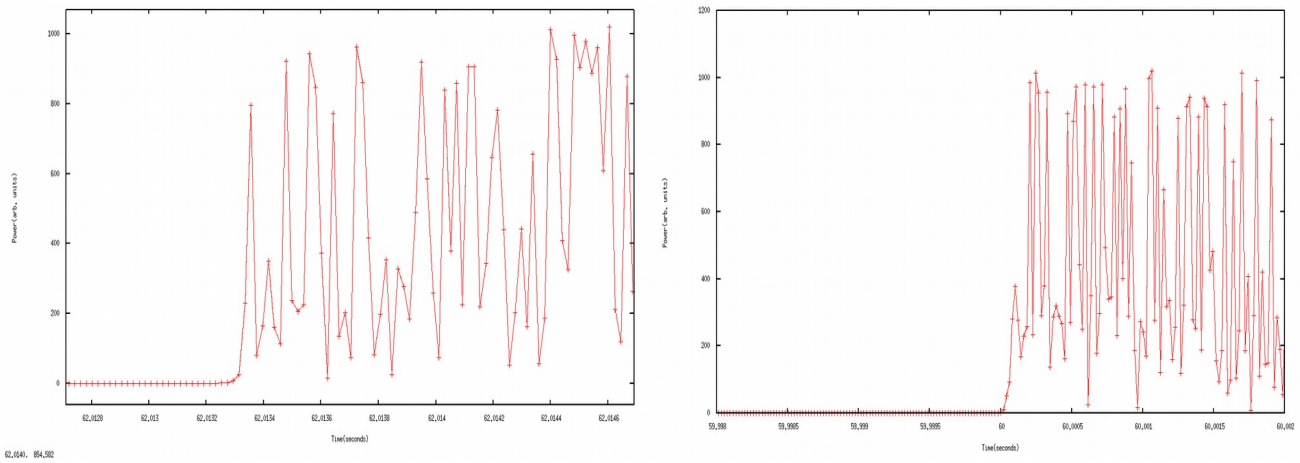
Plot showing timeseries of a channel from visibility data for LTA 32. Left side – Before time offset correction, Right side – Post time offset correction. Time offset before correction is 125 LTA buffers i.e; 83.88 seconds.

8. Correction of timestamp for beam data

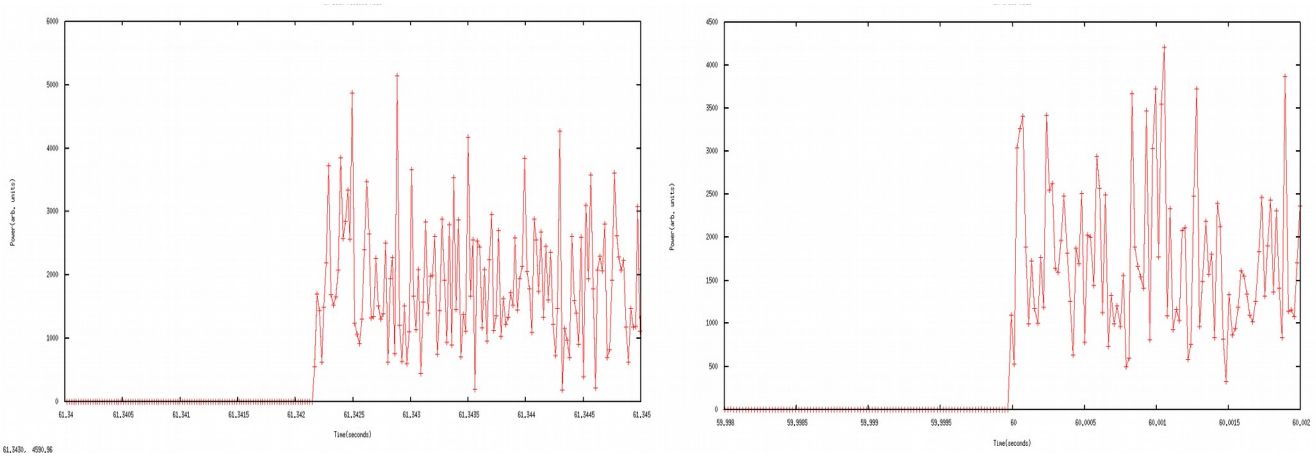
For beam data, there is no integration and every buffer's beam data is sent to shared memory after GPU processing and MPI collection of beam data from all compute nodes. Hence, the pipeline delay with respect to timestamp calculated is three buffers i.e; one for timestamp calculation (calculated in advance), one for GPU processing and one for MPI collection of results. Hence, when the beam data is written to shared memory the timestamp is picked from the old timestamp buffer addressed at (run count – 3) % 4.

9. Experimental verification of time offset in beam data

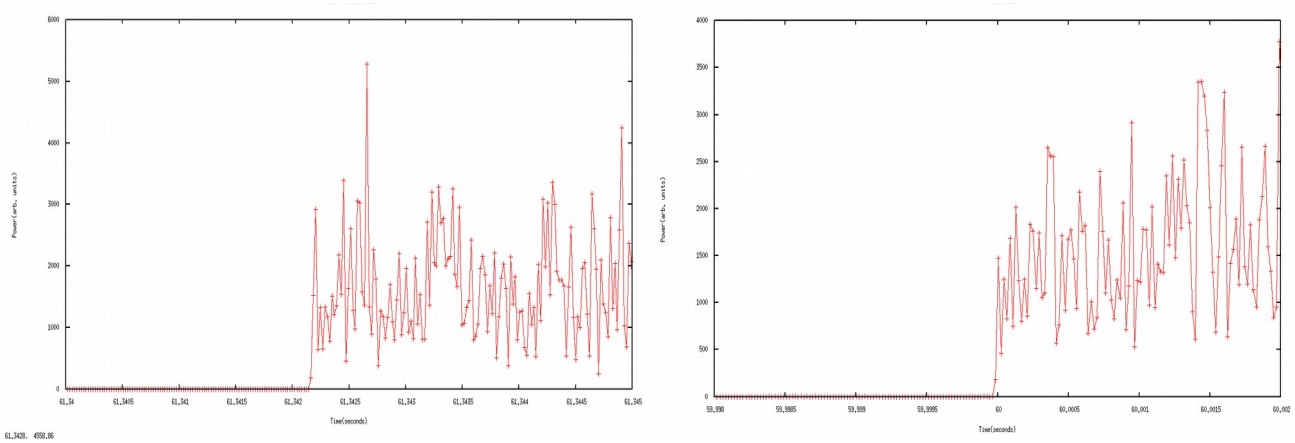
For this experiment, input used was broadband noise modulated by GPS PPM. PPM modulated noise was fed as input to GAB RF input of a polarization of an antenna (C03 pol1). GAB LO was set to 250 MHz, Band-3 RF bandpass filter was selected, 200 MHz Baseband Low pass filter was selected. The noise power level was adjusted such that ON pulse and OFF pulse noise is clearly visible. GWB settings are : 200 MHz bandwidth, 1024 spectral channels, total intensity mode, beam integration – 4 FFTs (20.48 microseconds). In the GAC config, only the input where PPM modulated noise was connected was selected (C03 pol1 here). Beam data was recorded for few minutes.



Plot showing timeseries of a channel from CD pipeline data. Left side – Before time offset correction, Right side – Post time offset correction.



Plot showing timeseries of a channel from IA beam data. Left side – Before time offset correction, Right side – Post time offset correction.



Plot showing timeseries of a channel from IA beam data. Left side – Before time offset correction, Right side – Post time offset correction.

A program was written in C-language, to extract a single channel as a time series from the beam data and written to a file. Also, along with the time series, timestamp was extrapolated from the start time of recording obtained from the header file and written to a file. For the extrapolation of timestamp, only the seconds and nanoseconds were considered and extrapolated. So, in the case when the timestamps are corrected (no offset), the ON pulse should start at multiple of 60 seconds (eg : 60, 120, 180 ----) and OFF pulse should start at multiple of 30 seconds (eg : 30,90,120,150 ----). Plots attached above shows the timeseries of a channel extracted from CDP data, IA beam data and PA beam data.