Postprint: Efficient Real-time Pipeline Data Stream Transmission and Processing Technology for Radio Astronomy

Authors: Zhang Meng, Zhang Hailong, Wang Jie, Li Jian, Ye Xinchen, Wanqiong Wang, Li Jia, Wang Boqun, Yazhou Zhang

Date: 2021-03-30T00:00:00+00:00

Abstract

To address the challenges of real-time and efficient transmission and processing of massive astronomical signals in ultra-wideband and multi-beam receiving systems, testing and analysis were conducted on software systems for mainstream FPGA+GPU-based terminal equipment. Ultra-wideband receiving devices require terminal system software capable of real-time data stream transmission and processing under conditions of wider bandwidth and higher temporal and frequency resolution. In conjunction with future development directions for large-aperture radio observation facilities, a design approach is proposed that utilizes high-speed parallel circular buffers for data stream caching, GPU clusters for real-time data stream processing, and BeeGFS for distributed parallel data storage, to modularly construct a software pipeline for radio astronomy signal transmission.

Full Text

Efficient Real-Time Pipeline Data Stream Transmission and Processing Technologies for Radio Astronomy

Zhang Meng[12], Zhang Hailong[1234]*, Wang Jie[124], Li Jian[1], Ye Xinchen[14], Wang Wanqiong[1], Li Jia[1], Wang Boqun[12], Zhang Yazhou[12]

[1] Xinjiang Astronomical Observatory, Chinese Academy of Sciences, Urumqi, Xinjiang 830011, China
[2] University of Chinese Academy of Sciences, Beijing 100049, China
[3] Key Laboratory of Radio Astronomy, Chinese Academy of Sciences, Nanjing, Jiangsu 210008, China
[4] National Astronomical Data Center, Beijing 100101, China

Abstract: Aiming at the challenges of real-time, efficient transmission and processing of massive astronomical signals in ultra-wideband and multi-beam receiving systems, this paper tests and analyzes the software systems of mainstream backend equipment based on FPGA+GPU architectures. Ultra-wideband receiving equipment demands that backend system software achieve real-time data stream transmission and processing under conditions of wider bandwidth and higher temporal and frequency resolution. Combining this with future development directions for large-aperture radio observation facilities, we propose a design concept for constructing radio astronomy signal transmission pipeline software in a modular fashion, utilizing high-speed parallel ring buffers for data stream caching, GPU clusters for real-time data stream processing, and BeeGFS for distributed parallel data storage.

Keywords: Data transmission and processing; Radio astronomy; Real-time; FPGA+GPU

0 Introduction

The advancement of multi-beam and Phased Array Feed (PAF) receiving technologies has continuously enhanced the data acquisition capabilities of radio astronomical instruments. Radio astronomy observation data is now growing in a real-time, massive, and continuous mode. The efficient transmission and real-time processing of massive astronomical signals in backend systems require support from heterogeneous distributed computing and storage platforms. During data transmission, the received signals are first digitized to implement preprocessing such as channelization and Radio Frequency Interference (RFI) mitigation, which can be executed on development boards such as ROACH[1], SNAP, and RFSoC[2]. Hardware boards are connected to server platforms via high-speed networks, which packetize the data and transmit it to GPU servers for complex algorithmic processing. The hybrid FPGA+GPU architecture platform imposes higher requirements on software performance. High-speed real-time transmission of astronomical data packets necessitates addressing packet loss issues in high I/O environments. The Linux kernel incurs significant overhead when processing broadband network traffic, and since the network protocol is UDP, the system must handle lost, out-of-order, and duplicate packets; otherwise, phase information will be missing. For pulsar data, packet loss can severely impact pulsar period prediction and folding. When implementing high-speed data flow between CPU and GPU, the communication rate between CPU and GPU may become a performance bottleneck that also requires consideration and resolution.

In radio astronomy digital backend systems, programs that achieve high-rate real-time data packet transmission and preprocessing are called pipelines[3]. A pipeline is a programming implementation where data is chained together and flows simultaneously within it. The pipeline contains multiple processing units,

each performing specific tasks. The most fundamental and primary function is to provide a caching module that temporarily stores data to enable real-time data flow. Pipeline data transmission for radio astronomy heterogeneous backend systems mainly includes three stages: data reception, CPU/GPU-based processing, and data output. Buffer modules enable efficient transmission from data reception to disk writing, while calling different stage program modules implements real-time processing during data stream transmission.

This article studies and analyzes data pipelines that can be deployed on radio telescope backend systems, proposing a pipeline design approach based on FPGA+GPU for dynamic, high-performance real-time transmission, processing, and monitoring of massive radio astronomy data streams. This provides a reference for developing customized multifunctional real-time data stream transmission and processing pipeline software for the future Xinjiang 110-meter Radio Telescope (QTT)[4].

---

1 Existing Radio Astronomy Data Stream Pipeline Software

For real-time high-speed transmission of radio astronomy data, foreign researchers have achieved a series of results and provided open-source CPU/GPU data stream pipeline software, including GUPPI_{daq}[2], PSRDADA[3], Hashpipe, Bifrost, Kotekan[4], Pelican[5], and Cobalt. Domestic research in this field is currently in its initial stages.

1.1 GUPPI_{daq}

GUPPI_{daq}, applied at the Green Bank Telescope (GBT), is the data acquisition software for the pulsar digital backend GUPPI. GUPPI_{daq} implements real-time data stream reception and processing, recording files in PSRFITS format. Real-time processing options include base mode, folding mode, and subband mode.

GUPPI processes 800 MHz bandwidth, 8-bit quantized data based on CASPER FPGA boards, packetizing it into UDP packets in a 10Gb Ethernet environment for transmission. GUPPI_{daq} is responsible for receiving this data and saving it to disk. During transmission, it implements interaction with the control subsystem, provides a human-machine interface, and records monitoring parameters saved to output files.

GUPPI_{daq} operates in multi-threaded mode, with threads transmitting data through shared memory-based ring buffers. In base mode, GUPPI_{daq} uses only two processing threads and one data buffer. The network processing thread receives UDP packets generated by the FPGA, checks custom sequence numbers to determine packet loss, saves received data to the ring buffer, fills lost data with zeros, and simultaneously reads header information from the status

buffer to copy it into the data block header. The disk writing thread reads full buffers, parses data block header information, and saves it to disk in PSRFITS format.

At high data rates, GUPPI_{daq} faces limitations in disk writing speed. When receiving data through high-speed Ethernet at 800 MB/s, a small amount of packet loss occurs[9].

1.2 Hashpipe

Hashpipe is a derivative of the early GUPPI_{daq}, originally developed as an efficient shared pipeline engine for NRAO VEGAS[10] by David Macmahon. It can be used for FX correlators, beamformers, pulsar observations, Fast Radio Bursts (FRB), SETI, and has also been applied to China' s FAST[6] and "Tianlai" project[11].

Hashpipe establishes shared memory segments and semaphore mechanisms, dynamically constructing pipelines at runtime based on command-line parameters. Its overall structure is shown in Figure 1, adopting a modular architecture design. The blue components represent plugins—Hashpipe plugins are shared libraries that define application-specific thread modules, shared data buffers, and other functional modules, allowing users to create plugins for invocation during Hashpipe runtime. The red components represent executable files that dynamically call defined plugins during Hashpipe runtime.

The core of Hashpipe is a flexible ring buffer that simulates continuous memory blocks, enabling data flow and sharing among multiple threads. It uses CPU control to start and stop operations, temporarily stores and passes UDP packets through ring buffers, ensures rapid data capture, and distributes data in the correct order.

Hashpipe distributes tasks to separate threads, with each thread sharing memory buffers and using semaphore mechanisms to achieve mutual exclusion. It defines three threads: data reception, processing, and output, each implementing its own tasks. The data reception thread (net_{thread}) receives high-speed network packets from the server' s 10Gb NIC, extracts file headers and payload data according to packet format, and parses packet headers. Data packets from the FPGA contain timestamps, allowing reordering into proper time sequences if they arrive out of order. Data is written to the first input data buffer, and once a continuous data block is full, it is handed to the next thread for processing[12]. The processing thread (gpu_{thread}) retrieves data from the input data buffer, transfers it to the GPU for complex computation, and writes results to the output data buffer. The output thread (output_{thread}) retrieves data from the output data buffer and writes it to files stored on disk. By defining a monitoring thread to listen for UDP packets from the FPGA and a transpose thread to reformat data, time samples are correctly arranged in memory for backend processing.

The backend is written in C/C++, with multiple structure variables defined in header files for convenient invocation. It defines the data buffer hashpipe_{databuf}, containing basic information such as data types to be transmitted, buffer space size, buffer ID, number of data storage blocks, and block ID. The hashpipe_{{{thread}}*{{desc}}} structure stores metadata describing Hashpipe threads, while the hashpipe{{{udp}}*{{params}}} structure variable saves network connection parameters including port number, IP address, and packet size. The hashpipe{{{clean}}*{{shmem}}}() function clears status buffer occupancy, while hashpipe{{{status}}*{{unlock}}}{safe}() and hashpipe_{{{status}}*{{lock}}}{safe}() functions provide thread-safe locking and unlocking of the status buffer to ensure it remains in a locked state. Using existing Hashpipe functions basically meets data transmission needs, and can be implemented and called according to actual requirements.

For interface monitoring, David Macmahon wrote the ruby program rb-Hashpipe, which serves as a visual frontend for Hashpipe. It can display information such as packet reception, thread status, and buffer status, abstracting the pipeline at a higher level to provide a concise and intuitive interface for monitoring data. Both Hashpipe and rb-Hashpipe can be deployed on server platforms, with main environment configuration steps as follows: (1) Update GNU toolchain to the latest version to provide Hashpipe compilation environment; (2) Install Ruby version manager rvm; (3) Use RubyGems to update required library files; (4) Obtain Hashpipe version information and complete installation; (5) Configure gnome terminal to launch under shell.

The data transmission status monitoring interface is shown in Figure 2.

1.3 PSRDADA

PSRDADA, developed and maintained by Swinburne University of Technology in Australia, supports distributed acquisition and data analysis of pulsar signals. It is mainly used for pulsar baseband data recording and processing, managing the entire data flow process from ADC signal sampling to GPU cluster data analysis. PSRDADA enables management and monitoring of baseband recording data distribution for APSR, BPSR, and CASPSR, and is currently primarily applied to terminal systems for ASPSR, BPSR, CASPSR, and HIPSR at Parkes in Australia[13], as well as PuMa-II and other radio telescopes in Europe[7].

At the basic level, PSRDADA is a flexible and manageable ring buffer whose core function is to pass data streams between ring buffers. Each buffer is divided into a header storage block describing file information and multiple sub-blocks storing data. When data is received, it is written sequentially to sub-blocks, and when a sub-block is filled, a flag is triggered indicating that data in the sub-block can be read. Ring buffers implement enqueueing and dequeueing of datasets, with multiple datasets able to queue in parallel across multiple ring buffers using multithreading, primarily communicating between threads through shared memory and semaphore mechanisms.

At the high-level configuration, PSRDADA can be launched on each node in a cluster, with visualization interfaces monitoring related processes and managing data transmission and archiving.

PSRDADA is implemented based on modular design, including independent processes for performing specific tasks, cleanly separating different operations such as data transmission, commands, control instructions, and data analysis. It creates ring buffers for data transmission between processes; uses configuration files and scripts to complete various basic configurations in distributed workflows; provides a web-based user interface for monitoring via web browsers; and can send data to remote computing devices via gridbus for remote monitoring.

PSRDADA implements corresponding developments for different devices including APSR, BPSR, CASPSR, and PUMA2 based on the C language. It executes through multiple command lines: dada_{db} can create or delete a buffer, dada_{dbmonitor} monitors opened buffers, dada_{dbdisk} writes buffer data to local disk, dada_{header} obtains data file header information, and dada_{dbcopydb}, dada_{dbdisk}, dada_{diskdb} implement data copying between buffers and between buffers and disk[8].

PSRDADA configuration and basic usage steps are as follows: (1) Install or update GNU tools required by PSRDADA: autoconf (version 2.59 and above), automake (version 1.9.3 and above), libtool (version 1.5.8 and above), m4 (version 1.4 and above); (2) Obtain data in dada format; (3) Create a ring buffer with a specific key value: $dada_{db}$–$kkey_{value}$ 4 dada_{diskdb} -k key_{value} -f filename.dada; (5) Read data from the ring buffer: $dada_{dbdisk}$–D.–$kkey_{value}$–s 6 dada_{dbmonitor} -k key_{value}; (7) Delete the ring buffer with a specific key value: $dada_{db} -k key_{value} -d.

Buffer creation and data transmission status are shown in Figure 3. A buffer structure with shared memory key 100 is created, including 4 data storage buffers of size 524,288 bytes each, with a total capacity of 2 MB, and 8 header storage buffers of size 4,096 bytes each, with a total capacity of 32 KB. The dada_{dbmonitor} command opens a monitoring interface to monitor read/write status of file headers and data buffers.

1.4 Bifrost

Bifrost is specifically designed for radio astronomy digital signal processing, enabling rapid development of high-performance pipelines for stream data processing. It is currently used for processing data from the LWA, with applications in beamforming and correlators[14].

Bifrost utilizes Python to implement high-level pipeline interfaces for data transmission and C++ to implement a high-performance backend supporting GPUs. This includes a series of high-performance data structures and algorithms related to astronomical data processing, library functions designed for interferometry,

pulsar dedispersion, timing, and transient signal search applications, used for data computation and processing on CPUs and GPUs[15].

Both interfaces and functions are called blocks, which can communicate simultaneously through flexible ring buffers while executing multiple threads, enabling asynchronous block operation. There are three types of data blocks: tasks, which read data from rings, implement data transformation, and write results to output ring buffers; sources, which generate data blocks, obtain data from files or Ethernet data streams outside the pipeline, and write data to output buffers; and sinks, which can directly read data from input ring buffers, plot data, or write it to files^{9}.

Bifrost has concise installation steps suitable for rapid deployment. After installation, modules can be called via `import bifrost`. For example, bifrost.pipeline handles pipeline construction, automatically creating ring buffers and building directed graphs through these ring buffers to provide users with a high-level view. The `pipeline.run` statement executes the pipeline, creating a thread for each block. The bifrost.blocks module loads functional blocks, calling FFT, FDMT, quantization, transpose, etc., to implement data processing. The network-related module bifrost.udp captures UDP packets and can count and analyze packet transmission information. If intermediate operations are needed during data stream transmission, users can customize modules^{10}.

Taking wav-format audio files as an example, Bifrost execution steps generally include: reading the wav file to a ring buffer; using GPU FFT to implement data channelization and other operations; and writing filterbank files to disk. The specific steps are: (1) Read .wav format file; (2) Copy raw data to GPU; (3) Divide time axis into small blocks and perform FFT on newly generated blocks; (4) Take the square of the modulus of these FFTs; (5) Transpose data to sigproc-compatible format and normalize; (6) Copy data to CPU; (7) Convert data to integer type and store data as filterbank.

1.5 Kotekan

Kotekan[16] is a software framework developed by Andre Recnik using C/C++ combination, primarily applied to radio astronomy telescope data transmission. It was initially developed for the CHIME backend to achieve higher efficiency and throughput^{11}.

Kotekan pipelines are modular, consisting of a series of units performing different operations on data. They implement real-time management of data flow on X-engines, including receiving UDP packets from F-engines, real-time processing of data streams on GPU nodes, and result storage. The core structure is a FIFO ring buffer that enables communication between CPU and AMD GPU through ring buffers in system memory, as well as data inspection and temporary storage before transmission to GPU for processing[17].

Kotekan creates network threads, GPU threads, GPU callback threads, GPU

post-processing threads, and output threads. Each thread is interfaced with one or more ring buffer objects, with received data temporarily stored through ring buffers to enable simultaneous reading and writing. These threads and buffers form the data path shown in Figure 4, where pipelines can generate multiple result datasets in parallel through multithreading[18].

Buffered data is written and read by Kotekan::stage processing modules, which obtain data from networks or external devices, implement data operations through data processing algorithms, and finally transmit processed data through networks or store it locally.

1.6 Pelican

Pelican[19] is a real-time data stream processing framework developed by the Oxford e-Research Centre, featuring efficiency, modularity, lightweight design, and calibration capabilities. It is mainly applied to LOFAR and BEST-II, performing parallel data processing on GPU servers.

Pelican provides highly abstracted APIs suitable for static, quasi-real-time processing. Pelican implements reusable modular components in C++, separating data acquisition and processing with flexibility and versatility. It has mechanisms for reading UDP data, passing data streams to computing servers, completing data buffering, processing, and final file writing. It can perform real-time processing of telescope-received data and can be used for any application requiring processing of incoming data streams[20].

Pelican can be used for data preprocessing in pulsar searches. The SKA' s early plans used Pelican for non-imaging, GPU-based real-time radio pulse detection[21]. The workflow is shown in Figure 5[Figure 5: see original paper]: first, UDP data is read through software servers, peaks of narrowband spectral interference are removed, further narrowband division is implemented in GPU, complex data is converted to power data, and data is transmitted to GPU processing modules to execute dispersion search algorithms[22].

Short-term imaging leads to very large correlator output rates. To achieve data transmission rates and generate updated calibration coefficients, output data streams must be processed in real time. Pelican can be used to implement data calibration and imaging equation solving before converting radio telescope data into scientific images. Pelican' s framework for scientific imaging is shown in Figure 6[Figure 6: see original paper], receiving subband data from ROACH2, providing initial calibration through real-time updating of correlator phase and amplitude coefficients, then generating dirty images based on GPU 2D-FFT, performing cleaning, executing image difference comparison based on specific step sizes, using threshold detectors to discover transient events, and finally using stacking modules to form high dynamic range images^{12}.

1.7 Cobalt

Cobalt is a GPU-based software correlator and beamformer for the LOFAR telescope[23]. The software component implements a data pipeline for managing data flow through networks and CPU/GPUs, and completing data storage.

Figure 7[Figure 7: see original paper] shows the schematic diagram of data flow from external systems to Cobalt' s internal data flow. Received data is stored in input buffers, subband processing is completed, and then storage is performed, with modules communicating in an all-to-all pattern.

Cobalt' s software consists of several interacting subsystems that manage data flow from the network, achieve high-rate sustainable data transmission, and persistently store data; it also performs control, monitoring, logging, and metadata management. Figure 8[Figure 8: see original paper] shows Cobalt' s components, dependencies, and data flow. Cobalt processes a large number of independent data streams, with each AntennaFieldInput receiving UDP packets from 10GbE ports and forwarding payload data to TransposeSender, which performs delay compensation by moving integer multiples through ring buffers. These data streams combine MPI for parallel processing without mutual dependencies. The signal processing pipeline GPUpipeline on GPU contains TransposeReceiver components that transmit subband data, combining MPI_{Isend} and MPI_{Irecv} non-blocking sends and receives to transmit correlation or beam data. Output components receive GPU subband data and ultimately store it in MeasurementSet or HDF5 format using casacore[24].

Cobalt can process offline tasks including automatic flagging, calibration, averaging, coherent dedispersion of pulsar data, dynamic spectrum generation, as well as online folding and searching of pulsar data. Cobalt is scalable, supporting subarray parallel observations, interrupt response, and other observation modes. It provides significant additional capacity in network bandwidth and computing power, with notable efficiency improvements, such as parallel observations of low-band and high-band antennas. Generally, correlators can only process data from a single project, but Cobalt' s enormous computing and throughput capabilities allow simultaneous processing of multiple project data^{13}.

An updated version, Cobalt 2.0, now exists with better flexibility and throughput, receiving data rates exceeding 1 TB/s, maximizing GPU advantages to achieve high-performance computing and real-time data processing.

---

2 Current Challenges

Current data stream pipeline software faces key difficulties and focal points:

(1) Existing radio astronomy data stream pipeline software is designed for specific observation equipment to meet particular requirements and is not applicable to all types of observation equipment or modes. For specific

observation equipment, corresponding software functions should be implemented according to the requirements of each receiving system. Although some software systems can be ported to different equipment terminal systems, they do not require all their functions during use, necessitating in-depth study of underlying code to complete secondary development and debugging, resulting in high development costs.

(2) Existing large radio telescope terminal systems mostly adopt Xilinx FPGA-based ROACH series and SNAP series hardware boards. In recent years, the more highly integrated RFSoC has been introduced, providing high-speed Ethernet data transmission interfaces that impose higher requirements on server-side real-time data stream reception capabilities. Pipelines deployed on servers need to meet higher bandwidth data reception requirements based on the RFSoC platform to improve data transmission efficiency.

(3) Real-time efficient transmission and processing capabilities are the direction for pipeline software performance improvement. Data processing platforms mostly use GPU clusters for large-scale data processing; however, data distribution and collection have not been effectively addressed. Efficiently implementing data stream distribution and aggregation across multiple GPUs will greatly improve transmission and computing efficiency.

(4) To meet the data processing requirements of multifunctional digital backend systems, a flexibly configurable system software should be formed. The constructed pipeline software should implement algorithms required for data processing and analysis, provide callable libraries for different observation modes, and satisfy the needs of multiple observation projects.

---

3 Design Concepts for Large-Aperture Radio Telescope Data Stream Pipeline Software

As an important component of multifunctional terminal system software, pipeline software for large-aperture radio telescope data stream transmission and processing will complete data stream acquisition, transmission, management, and monitoring. It must efficiently capture and temporarily store network packets from FPGA, achieve high-speed data flow and related processing between CPU and GPU, provide GPU internal data processing algorithm calls, implement conversion from raw data format to standard file format and data type matching, and finally write data efficiently to disk.

3.1 Data Transmission Based on Zero-Copy and Ring Buffers

The QTT multifunctional digital backend adopts an FPGA+GPU cluster architecture, as shown in Figure 9[Figure 9: see original paper][25]. RFSoC implements data sampling and preprocessing, with a single RFSoC chip capable of

achieving 8 channels, each with a maximum sampling rate of 4.096 GSPS and 12-bit quantization. The data volume per channel after sampling is approximately 48 Gbps. After preprocessing by RFSoC, the sampled data is divided into multiple subbands, and the RFSoC provides a 100 Gb data transmission interface for transmission via high-speed Ethernet to the GPU cluster for data correlation and preprocessing.

Network data transmission uses the UDP protocol, where data in NIC buffers is transmitted to user applications through the Linux kernel protocol stack. Data packets are copied between kernel space and user space, resulting in large system overhead and transmission bottlenecks. To effectively reduce system resource occupancy and meet low packet loss rate and low latency requirements for GPU server-side transmission, GPU servers can implement high-performance data acquisition based on zero-copy technology in high I/O environments. The comparison is shown in Figure 10[Figure 10: see original paper]. Pipeline software implements a zero-copy-based data transmission module, creating parallel network input threads that read packets from NIC buffers in user space through polling, writing them in real time to dynamically created ring buffers. This avoids standard kernel protocol stack processing and reduces the number of data copies from NIC to pipeline software, thereby improving data access efficiency.

3.2 Real-Time Data Processing Based on GPU Clusters

The QTT multifunctional terminal system based on hybrid architecture plans to implement multiple observation modes including pulsar, transient source, molecular spectral line, total power, VLBI, and baseband data. The internal data flow in the GPU cluster is shown in Figure 10. After FPGA preprocessing, data is transmitted to the GPU cluster for processing. The data flow undergoes decoding, RFI mitigation, and then completes corresponding processing for different observation modes, finally being stored in formats such as VDIF, PSRDada, or PSRFITS.

Pipeline software should provide GPU internal data processing modules and fully integrate astronomical algorithms to implement common functions such as decoding, RFI identification, calibration, folding, and standard format output, achieving conversion to standard file formats and data type matching. It should provide a friendly access interface based on Web, offer better flexibility and support for expansion in a modular manner, and provide user-defined module functionality, allowing users to call modules or write custom modules according to actual needs.

For the QTT multifunctional digital terminal data stream, the top-level structure of pipeline software data transmission is shown in Figure 11[Figure 11: see original paper]. Network threads capture data from RFSoC and store it in input data ring buffers. Pipeline software runs on multiple GPU servers, with each GPU server receiving one frequency channel subband. To achieve multithreaded parallel and distributed execution on GPU clusters, efficient and timely opera-

tion and data block forwarding, cluster nodes combine MPI to implement data distribution and acquisition, using MPI' s non-blocking message reception and send call interfaces MPI_{Isend} and MPI_{Irecv} to enable overlapping computation and communication, improving data processing efficiency. Processed data from GPU clusters is temporarily stored in output data buffers and transmitted to the storage system through parallel output threads.

3.3 Data Storage Based on BeeGFS

As data volume increases rapidly, HPC systems require more complex and efficient methods to manage and store massive data. By deploying distributed file systems, data is distributed across numerous storage devices for distributed storage and management, enabling distributed read/write. Using the parallel distributed file system BeeGFS^{14} to build a more efficient data storage backend improves I/O performance and achieves high performance and scalability.

The overall framework of the data storage system based on BeeGFS is shown in Figure 12[Figure 12: see original paper]. Data processed by the GPU cluster is integrated and encapsulated into specific formats, then transmitted through building parallel output management threads that simultaneously transfer one or more data streams. Data of the same format is written to the same ring buffer and transmitted to the file storage system. Data is read and written through BeeGFS-provided Clients, utilizing multiple independent I/Os to achieve high-concurrency read/write. The BeeGFS-based distributed file system achieves metadata separation, using local file systems of MDS and OSS to carry metadata and data storage respectively. File headers containing observation information such as date, time, and observation environment parameters serve as the main components of metadata information and are stored on metadata servers. The scalability of MDS, OSS, and Clients enables the BeeGFS-based data storage system to meet large-scale data storage expansion requirements.

Radio astronomy telescope digital backend systems increasingly rely on real-time processing to overcome data storage and analysis bottlenecks. The astronomical community has developed and studied data stream processing pipelines to improve data transmission and processing efficiency and simplify the difficulty of developing data processing code for astronomers. This article reviews existing radio astronomy data transmission pipeline software, introduces the structure and functions of radio astronomy data transmission pipeline software, and conducts tests on GPU servers. Based on summarizing existing pipeline software, we present general design ideas for future large-aperture radio telescope multi-functional terminal systems to design data transmission pipelines according to actual observation needs. Future multifunctional digital terminal systems based on FPGA+GPU should develop toward easy upgrading, modification, flexibility, and scalability.

Acknowledgments

This work is supported by the National Natural Science Foundation of China (11873082, 11803080, 12003062), the Youth Innovation Promotion Association of the Chinese Academy of Sciences, the National Key R&D Program of China (2018YFA0404704), the Astronomical Finance Special Project, and the National Astronomical Data Center, Chinese Academy of Sciences Science Data Center System. This paper has received data resources and technical support from the China Virtual Observatory, the National Astronomical Data Center, and the Chinese Academy of Sciences Science Data Center System.

---

5 References

[1] Zhang Meng, Zhang Hailong, Wang Jie, Li Jian, Tohtonur. The Construction of Digital Backend Experiment Platform Based on ROACH2[J]. Astronomical Research & Technology, 2020,17(02):244-251.

[2] Farley B, Mcgrath J, Erdmann C. An All-Programmable 16-nm RFSoC for Digital-RF Communications[J]. IEEE Micro, 2018, 38(2):61-71.

[3] Paine D, Lee C P. Producing data, producing software: Developing a radio astronomy research infrastructure[C]//2014 IEEE 10th International Conference on e-Science. IEEE, 2014, 1: 231-238.

[4] Wang N. Xinjiang Qitai 110 m radio telescope[J]. Scientia Sinica, 2014, 44(8):783.

[5] Zhang Hailong, Wang Jie, Wang Wanqiong, Nie Jun, Ye Xinchen, Zhu Yan, Tohtonur. Construction and Application of the Data Center in Xinjiang Astronomical Observatory[J]. Astronomical Research & Technology, 2017,14(01):94-102.

[6] Srikanth S, Norrod R, King L, et al. An overview of the Green Bank Telescope[C]// Antennas & Propagation Society International Symposium. IEEE, 2002.

[7] Mei Li, Su Yan, Zhou Jianfeng. The History and Development of The Low-Frequency Radio Observation[J]. Astronomical Research & Technology, 2018,15(02):127-139.

[8] Bandura K, Addison G E, Amiri M, et al. Canadian hydrogen intensity mapping experiment (CHIME) pathfinder[J]. Proceedings of SPIE - The International Society for Optical Engineering, 2014, 9145.

[9] Duplain R, Ransom S, Demorest P, et al. Launching GUPPI: the Green Bank Ultimate Pulsar Processing Instrument[J]. 2008, 7019:70191D.

[10] Prestage R M, Bloss M, Brandt J, et al. The versatile gbt astronomical spectrometer (vegas): Current status and future plans[C]//2015 USNC-URSI

Radio Science Meeting (Joint with AP-S Symposium). IEEE, 2015: 294-294.

[11] Chen-Hui Niu, Qun-Xiong Wang, David MacMahon, Feng-Quan Wu, Xue-Lei Chen, Ji-Xia Li, Hai-Jun Tian, Guillaume Shippee, Dan Werthimer, Xiao-Ping Zheng. The design and implementation of a ROACH2+GPU based correlator on the Tianlai dish array[J]. Research in Astronomy and Astrophysics, 2019,19(07):135-144.

[12] Macmahon D H E, Price D C, Lebofsky M, et al. The Breakthrough Listen Search for Intelligent Life: A Wideband Data Recorder System for the Robert C. Byrd Green Bank Telescope[J]. Publications of the Astronomical Society of the Pacific, 2017, 130(986).

[13] Price D C, Staveley-Smith L, Bailes M, et al. HIPSR: A Digital Signal Processor for the Parkes 21-cm Multibeam Receiver[J]. Journal of Astronomical Instrumentation, 2016, 5.

[14] Cranmer M D, Barsdell B R, Price D C, et al. Bifrost: a Python/C++ Framework for High-Throughput Stream Processing in Astronomy[J]. Journal of Astronomical Instrumentation, 2017.

[15] Kaszyk K, Wagstaff H, Spink T, et al. Full-system simulation of mobile cpu/gpu platforms[C]//2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). IEEE, 2019: 68-78.

[16] Recnik A, Bandura K, Denman N, et al. An Efficient Real-time Data Pipeline for the CHIME Pathfinder Radio Telescope X-Engine[C]// 2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP). IEEE, 2015.

[17] Pinsonneault Marotte T. Towards precision measurements of the Hubble constant with the Canadian Hydrogen Intensity Mapping Experiment[D]. University of British Columbia, 2018.

[18] Denman N, Amiri M, Bandura K, et al. A GPU-based Correlator X-engine Implemented on the CHIME Pathfinder[J]. 2015.

[19] Mort B, Dulwich F, Williams C, et al. Pelican: Pipeline for Extensible, Lightweight Imaging and CAlibratioN[J]. Astrophysics Source Code Library, 2015.

[20] Karastergiou A, Chennamangalam J, Armour W, et al. Limits on fast radio bursts at 145 MHz with ARTEMIS, a real-time software backend[J]. Monthly Notices of the Royal Astronomical Society, 2015, 452(2): 1254-1262.

[21] Trefethen A E. Extreme Computing: Challenges, Constraints and Opportunities[J]. 2010.

[22] Karastergiou A. SKA NON IMAGING PROCESSING CONCEPT DESCRIPTION: GPU PROCESSING FOR REAL-TIME ISOLATED RADIO PULSE DETECTION[J]. 2011.

[23] Broekema P C, Mol J J D, Nijboer R, et al. Cobalt: A GPU-based correlator and beamformer for LOFAR[J]. Astronomy and computing, 2018, 23: 180-192.

[24] Prasad P, Huizinga F, Kooistra E, et al. The AARTFAAC all-sky monitor: System design and implementation[J]. Journal of Astronomical Instrumentation, 2016, 5(04): 1641008.

[25] Zhang Hailong, Zhang Meng, Nie Jun, et al. Pulsar digital backend technologies review[J]. SCIENTIA SINICA Physica, Mechanica & Astronomica, 2019(9): 15-28.

Note: Figure translations are in progress. See original paper for figures.