FCNP: Fast I/O on the Blue Gene/P

John W. Romein

romein@astron.nl
Stichting ASTRON (Netherlands Institute for Radio Astronomy), Dwingeloo, The Netherlands

Abstract— *This paper describes the* Fast Collective-Network Protocol (FCNP). *FCNP is a low-overhead, high-bandwidth network protocol that we developed for fast communication between the Blue Gene/P compute nodes and I/O nodes. The CPU cores in this system are hardly able to keep up with the high-speed internal network, and any protocol overhead significantly slows down the achieved bandwidths. FCNP minimizes overhead and approaches the link speed for large messages.*

FCNP is of critical importance to the correlator of the LOFAR radio telescope, that will process hundreds of gigabits of real-time telescope data per second. Without FCNP, the correlator would not even achieve the required data rates. However, FCNP allows bandwidths over 50% beyond the LOFAR requirements, so that the telescope can observe proportionally more sources or frequencies and becomes a much more efficient instrument.

Keywords: low-overhead network protocol, IBM Blue Gene/P, LOFAR radio telescope

1. Introduction

I/O is one of the most precious resources in a supercomputer [6], and the IBM Blue Gene is no exception to this rule. Although the Blue Gene/P uses much faster I/O hardware than its predecessor Blue Gene/L, we found that it is still difficult to efficiently stream large amounts of data from outside the system to the final destination CPUs.

The Blue Gene architecture consists of compute nodes that run a parallel application and a much smaller number of I/O nodes that perform I/O operations on behalf of the compute nodes. The I/O nodes and compute nodes are internally connected by what is called the *tree* or *collective network*. The system software forwards I/O operations like `read()` and `socket()` from a compute node to its associated I/O node where the operation is factually performed.

By default, the I/O nodes work transparently. However, the performance of I/O-intensive applications can improve significantly if a select part of the application or communication library (like PVFS) runs on I/O nodes rather than compute nodes [6].

We use the Blue Gene to process real-time radio-telescope data. Part of the application runs on the I/O nodes, while the compute-intensive processing is done on compute nodes. The application on the I/O node needs to communicate large amounts of data with the application on the compute nodes.

In 2007, IBM unveiled the Blue Gene/P (BG/P) [11] as successor to the Blue Gene/L (BG/L). Per rack, the BG/P has 2.43 times as much compute power as the BG/L. However, the (maximum) number of I/O nodes per rack has *halved*, due to the change from 1 to 10 Gb/s Ethernet technology. Hence, to scale with the computational performance, each I/O node has to handle 4.86 times as much data. This is challenging, since the tree is only 2.43 times faster. However, the main challenge is that the processor cores that drive the tree are merely *21%* faster. Fortunately, the I/O nodes have four cores rather than two, but at most two cores can be used for internal communication: one to read and one to write the tree. The remaining cores can be used for other tasks.

We found that a core is hardly able to read or write packets at link speed from or to the tree; even an "empty" function call per packet already ruins performance. The overhead of any protocol on top of the packet interface would decrease the obtained bandwidth significantly.

This paper describes the *Fast Collective Network Protocol* (FCNP), that is designed to provide bandwidths at link speed between the I/O nodes and the compute nodes. The key design feature of FCNP is that it maximizes bandwidth by minimizing protocol overhead to an absolute minimum. By reducing the protocol overhead, the processor cores have more time to transport user data. We will show performance results and characterize the performance in terms of bandwidth, latency, and overhead.

FCNP is heavily used to process LOFAR telescope data. LOFAR [4], [12] is the first of a new generation of telescopes, that combines the signals of tens of thousands of simple, cheap, omni-directional antennas rather than using expensive dishes. In several ways, it will be the largest telescope of the world. Another novel feature is that the data are processed in *software* on a supercomputer [9], [10], where traditionally custom-built hardware is used. The data are streamed at high bandwidths into the system and processed in real time. I/O nodes receive the data and internally forward them to compute nodes. Standard system software does not provide sufficient bandwidth to handle 2.1 Gb/s input and 0.58 Gb/s output per I/O node, needed to meet the LOFAR specifications [1]. In contrast, FCNP achieves 3.1 Gb/s input and 1.2 Gb/s output bandwidth per I/O node. The improved input data rate matches the absolute maximum that the telescope hardware can generate, which is *over 50% beyond* the specifications. This implies that LOFAR will be able to observe a 50% wider part of the radio spectrum instantaneously,

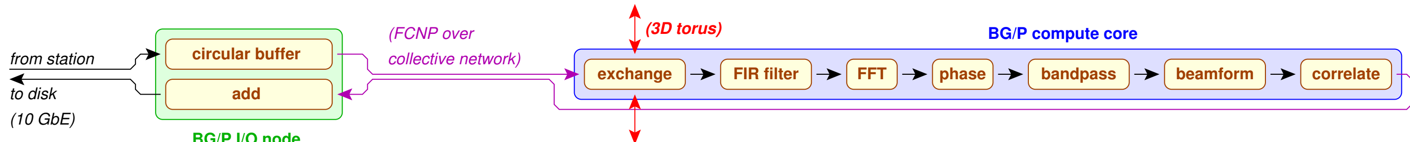


Fig. 2: LOFAR real-time signal processing.

or to observe 50% more sky sources concurrently, increasing the usage possibilities of the instrument. The doubled output data rate allows new observation modes, where even higher numbers of sky sources can be observed concurrently (albeit at lower precision), increasing the flexibility of the instrument.

We developed FCNP to support streaming LOFAR data, but the ideas of this protocol are more widely applicable.

This paper is structured as follows. In Section 2, we describe the relevant parts of the LOFAR processing pipeline. Then, after an introduction to the BG/P hardware (Section 3), we describe FCNP (Section 4). Next, we discuss related work (Section 5). We compare the performance of FCNP to competing protocols and analyze the performance impact on the LOFAR application (Section 6). Section 7 concludes.

2. LOFAR processing

LOFAR is a new type of radio telescope, that combines the signals of tens of thousands of antennas and processes the data centrally on a BG/P supercomputer. We briefly explain how LOFAR data are processed, other papers provide more details [10, Sec. 2], [9, Sec. 4–6].

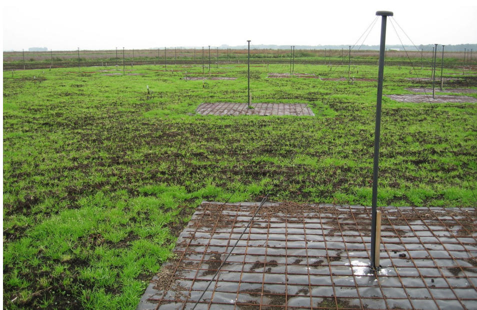


Fig. 1: The low-band antennas of a LOFAR station.

Co-located groups of 48 or 96 dual-polarized low-band antennas (see Figure 1) and high-band receivers form a *station*, i.e. a virtual telescope. Construction of 36–54 Dutch stations and 8–20 European stations is well underway. Each station digitizes the antenna voltages and pre-processes the data using FPGAs [5]. The FPGAs send UDP packets with station data over dedicated Wide-Area Network links to the BG/P, where the data are centrally processed. Initially, LOFAR used a 6-rack IBM BG/L supercomputer for real-time processing of the station data, but the system was recently replaced by an equally powerful 2.5-rack BG/P.

The application that runs on the BG/P is called the *correlator*, although it does much more processing than correlating only. Figure 2 shows a simplified scheme of one of the processing pipelines: the standard imaging pipeline that creates sky images. An I/O node receives 48,828 UDP packets per second of up to 8 KiB from one station. It copies the samples into a circular buffer that holds the most recent three seconds of data. The buffer is used to synchronize the stations (the travel times over the WAN are higher for remote stations than for nearby stations) and to prevent data loss due to small hiccups in the remainder of the processing pipeline. The buffer is also used to set the observation direction by “delaying” the stream of station samples for each station differently, to compensate for the fact that a celestial wave hits stations at different times [10, Sec. 2.1].

The buffered data are sent to the compute nodes for further processing. There, the data are first exchanged over the torus network, to collect pieces of data that can be processed independently. Then, the data are filtered and Fourier transformed to split each subband into narrow frequency channels. Next, a phase correction fine-tunes the observation direction. Then, a per-channel bandpass correction flattens a ripple introduced by a station filter. Optionally, a group of stations can be beam formed (by weighted addition of their samples) to form a more sensitive, virtual “super station”. Finally, the data are correlated (by multiplying the samples of all station pairs) to filter out noise and integrated to reduce the amount of output.

The compute node sends the correlated data back to the I/O node. The I/O node optionally adds data from other compute nodes, and sends the result (asynchronously) to external systems that temporarily store the data on disk. After an observation has finished (not shown in Figure 2), bad data (due to interference) are removed, and the resulting data are calibrated [8] and imaged.

Two tasks are of particular interest to this paper: the transport of buffered data from the I/O nodes to the compute nodes (at 3.1 Gb/s) and the transport of correlated data in the opposite direction (at up to 1.2 Gb/s). Moreover, each I/O node also needs to receive 3.1 Gb/s of UDP data from the stations and send 1.2 Gb/s of TCP data to external systems, and are therefore already quite busy processing the data through the IP protocol stack. The two available mechanisms to communicate between the I/O nodes and compute nodes (TCP and Unix domain sockets) do not provide sufficient bandwidth (see Section 6) and consume too many CPU resources, necessitating the need for a light-weight protocol for internal communication.

3. The Blue Gene/P

The BG/P is built using SoC (System-on-a-Chip) technology that integrates all processing and networking functionality on a single die. A node contains four PowerPC 450 cores, running at a modest 850 MHz clock speed to reduce power consumption and to increase the package density. Each core is extended by two Floating-Point Units that provide support for operations on complex numbers; each FPU can sustain one (real) double-precision, fused multiply-add per cycle. The four cores share 2 GiB of main memory. One BG/P rack contains 1,024 compute nodes, providing 13.9 TFLOP/s peak processing power, and up to 64 I/O nodes.

The BG/P contains several networks. A fast *three-dimensional torus* connects all compute nodes and is used for point-to-point and all-to-all communications. The *tree* or *collective network* is used for MPI scatter and gather operations, but also for communication between compute nodes and I/O nodes. A *10 Gb/s Ethernet device* (only enabled on I/O nodes) is used for external communication. A *global interrupt network* provides support for fast barriers. Additional networks exist for initialization, diagnostics, and debugging.

The compute nodes run a fast, simple kernel (Compute Node Kernel, CNK) that can execute one thread or process per core. An I/O node uses the same hardware as a compute node, but runs a modified Linux kernel.

3.1 The Tree Network

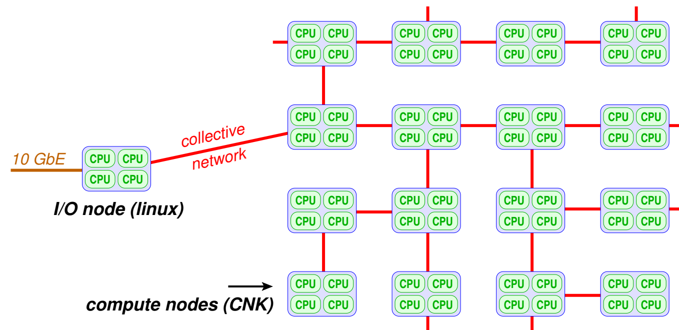


Fig. 3: A pset.

Each compute node is (indirectly) connected to one I/O node via the tree (see Figure 3). All I/O-related system calls on the compute nodes to external systems are forwarded to a daemon program (CIOD), that runs on the I/O node and performs the real operation. A group of one I/O node, its 10 Gb/s Ethernet interface, and the compute nodes that are connected to the I/O node is called a *pset*. Since LOFAR generates much data, our system is configured with the maximum number of 1 I/O node per 16 compute nodes (64 cores); typical installations have 1 I/O node per 32 compute nodes. Our system has 160 psets in total.

The original BG/L design did not support the idea of running user applications on the I/O nodes, but in earlier work [6], we showed that doing so significantly improved performance, flexibility, and costs. Unfortunately, this required major changes to the BG/L system software [3], [6]. In contrast, the BG/P supports user applications on I/O nodes, but the existing communication protocols between the compute nodes and the I/O nodes provided insufficient bandwidth for the LOFAR correlator.

The tree uses bi-directional links at 6.8 Gb/s per direction. The network has a tree topology with a complex physical structure. A node is connected to at most three other nodes. A packet can be routed via several nodes. The processor cores of in-between nodes are not interrupted by the routing process. The hardware provides two separate virtual channels. One is used by CIOD; the other is typically only used on the compute nodes: by the MPI library for some of the collective MPI operations. However, the runtime environment can be changed so that MPI uses the 3-D torus for these collectives instead of the tree, leaving one of the virtual channels unused. This virtual channel is accessible from user space.

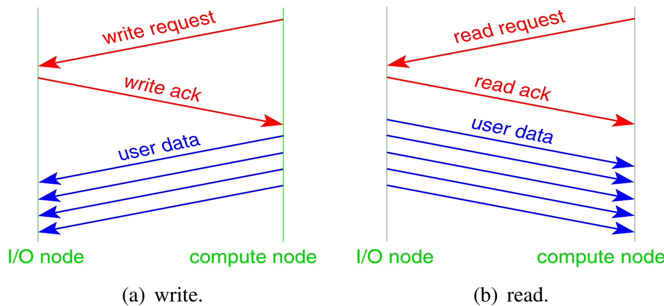
The LOFAR correlator does not perform collective MPI operations. However, other applications that critically depend on collectives might see performance differences (positive or negative) by using the 3-D torus instead of the tree. On the BG/L, the collective network was specifically designed to support collective operations. However, the BG/P is very well capable of doing collective operations using the torus, since the BG/P has DMA support for the torus (but not for the tree). Therefore, FCNP can use the free virtual channel, without significantly slowing down collective operations.

A processor can send and receive fixed-size packets over a virtual channel. A packet consists of a 4-byte header (used for routing) and 256 bytes payload. The 16-byte load and store instructions from the double FPU are used to efficiently transfer data from memory to the (memory-mapped) device and vice versa, but these impose a 16-byte alignment restriction. There is no DMA hardware available for the tree. The four cores from one processor share the same link. Each node has an 8-packet send FIFO and an 8-packet receive FIFO per virtual channel. An attempt to send a packet to a processor with a full receive FIFO blocks the virtual channel, filling up the sender's send FIFO, and eventually stopping the sender. Therefore, received packets should be consumed as fast as possible.

4. The Fast Collective Network Protocol

CIOD (the daemon that runs on the I/O node and handles the I/O requests from the compute nodes) is intended to provide communication between compute nodes and external systems. It is also possible to use TCP or Unix domain sockets internally, between an application on the compute nodes and an application on the I/O node. However, the obtained bandwidth (see Section 6) is insufficient for the LOFAR correlator. Therefore, we developed a new protocol, *Fast Collective Network Protocol* (FCNP), that uses the free virtual channel of the tree.

Since one compute core is barely able to keep up with the link speed, any heavy-weight protocol overhead would decrease the obtained bandwidth. FCNP reduces the protocol overhead to a single bit test in the normal case. FCNP distinguishes control packets (requests and acknowledgments) from data packets by setting the Irq (interrupt-on-receipt) bit in the header. Typically, most packets (for our application roughly 99.98%) are data packets that do not contain any metadata in the payload part.



(a) write. (b) read.

Fig. 4: The FCNP protocol.

To send data from the compute node to the I/O node, the application on the compute node calls *fcnp_write(ptr, size)* and the application on the I/O node calls *fcnp_read(ptr, size, core)*. The compute node sends a write request packet to the I/O node (see Figure 4(a)). The I/O node acknowledges a write request as soon as the application called a matching *fcnp_read()* for that particular core *and* when no other compute node is sending data to this I/O node. This way, the I/O node knows at all times from which compute core data packets originate, and can receive data packets consecutively in memory without additional checks. Only the Irq bit must be checked to see if the packet is indeed an expected data packet, and not a request packet from another compute core. The actual amount of bytes sent in a message is negotiated by the compute node and the I/O node by taking the minimum of the requested sizes on both sides. A read request (see Figure 4(b)) is handled similarly. At any time, up to one read and one write per pset can be active (although multiple unacknowledged request can be outstanding).

```
struct RequestReplyPacket {
  enum {READ, WRITE, RESET} type;
  unsigned                  node;
  unsigned short            core;
  unsigned short            rankInPSet;
  unsigned                  size;
  char                      msgHead[240];
};
```

Fig. 5: Format of the payload part of request and acknowledgment packets.

Figure 5 shows the format of request and reply packets. A request is a *read*, *write*, or a *reset* packet (reset requests are only sent during startup, to drain the FIFOs that may hold lingering packets from previous runs). The *node*, *core*, and *rankInPset* fields are the node number, the compute core number (between 0 and 3), and the rank within the pset respectively. These are necessary for the I/O node to determine the source of a request and to put routing information in the header of a reply. The *size* field in a request contains the requested message size; the *size* field in an acknowledgment contains the agreed size, which may be smaller than the requested size if the I/O-node application does not want to read or write as much data. The last 240 bytes of a read acknowledgment or a write request are used to send up to 240 bytes of data, such that the remaining amount of data is a multiple of 256 bytes, and can be sent in an integer number of packets.

To avoid deadlocks, it is always a *compute core* that initiates communication by sending a (read or write) request packet to the I/O node. Compute nodes are not always willing or able to receive data, and if the I/O node would send a packet to a compute node that does not listen, the tree could block. The I/O node, in contrast, runs a daemon thread that continuously polls the receive FIFO, waiting for an incoming request packet. This way, the tree cannot stall. The polling thread is suspended while another thread receives data packets from a compute node.

Each of the four cores in a compute node can post a read or write request. To avoid race conditions, only one of them is allowed to read the receive FIFO. As long as none of the request is acknowledged, one of the cores that await a reply polls the FIFO until it receives an acknowledgment. If the acknowledgment is addressed to another core, it transfers the packet (via shared memory) to the other core and releases its access rights. A compute core that receives a read acknowledgment gains exclusive access to the receive FIFO until the message is completely received. The sequence of data packets might be interrupted by a write acknowledgment for another core, which is then transferred to the addressed core. This interruption is detected by testing the Irq bit in the header. The stream cannot be interrupted by a read acknowledgment, since the I/O node makes sure that only one read can be active at a given time.

On the compute cores, we use fast hardware mutexes to synchronize the cores. On the I/O nodes, the same hardware mutexes are physically present, but not exposed by the Linux kernel, so we use atomic instructions to implement spin locks. Since these are noticeably slower and since the write FIFO has to be protected, the implementation writes up to eight consecutive packets before releasing and re-obtaining a lock. This way, the amortized locking overhead is negligible.

FCNP strongly encourages 16-byte aligned data and message sizes, but does not enforce this. Unaligned transfers are supported at the expense of a copy to an intermediate buffer.

FCNP is not fair, in the sense that one core can claim all network bandwidth. This is intentional, since sharing bandwidth between multiple cores implies that all cores proceed at reduced speed. Figure 6 illustrates this with an example, where all CPUs

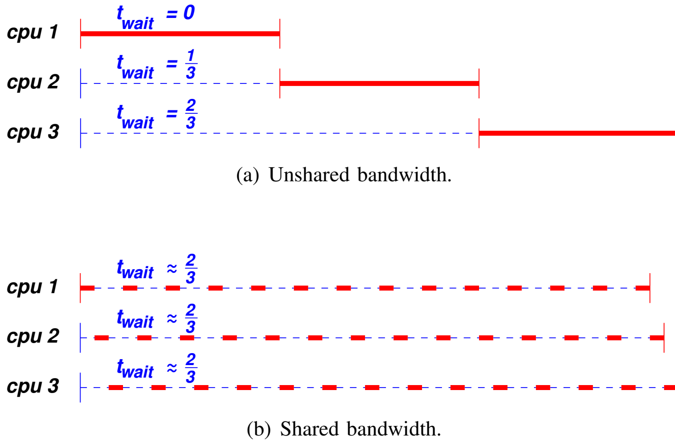


Fig. 6: Unfair communication is faster (see text).

concurrently start communicating the same amount of data (the red, fat bars; blue, dashed lines represent waiting CPUs). If the bandwidth is not shared (Figure 6(a)), some CPUs finish much earlier than others, and can continue doing other work. Even the latest CPU does not finish later than in the shared case (Figure 6(b)), where all CPUs finish approximately at the same time. In general, fairness is an issue for interactive applications and multiple applications run by different users, but FCNP is designed to support a single, cooperative, distributed application, of which we want to minimize the execution time.

4.1 Interrupts

The polling thread on the I/O node consumes a lot of CPU resources: even if it does not receive any data, one out of four cores is continuously busy. A second core is fully used by CIOD to poll the other virtual channel of the tree, leaving few resources for application processing. To reduce the CPU usage, IBM modified the Linux device driver of the tree to handle interrupts from both virtual channels, and adapted CIOD to take advantage of the interrupts. Likewise, we adapted FCNP's polling thread to block until a request packet (with the Irq bit set) is received in the receive FIFO.

An interrupt from the receive FIFO does not necessarily imply that it is the *first* packet in the queue that has the Irq bit set, nor that the packet is still in the FIFO (it might have been read before the interrupt is processed), so care must be taken to avoid race conditions and handle spurious interrupts. The device driver does not use (posix-style) signals to pass on an interrupt to an application. Instead, a thread can do a dummy read() system call on the device driver's file descriptor, and is blocked as long as there are no packets in the receive FIFO that have the Irq bit set. We think that this is a much more elegant solution, since signal handlers and threads do not coexist well. Moreover, an implementation based on signal handlers would need to perform additional system calls to avoid potential race conditions, thus the dummy-read mechanism is also more efficient.

The application can choose whether FCNP should use interrupts or not. In interrupt mode, the thread that handles new requests does not immediately suspend itself, but polls the receive FIFO for up to 50 µs. Since applications often send multiple messages in bursts, it is generally beneficial to try to receive a next request by polling, to avoid the interrupt overhead [7]. This significantly reduces the latency if a message arrives within 50 µs, but hardly wastes CPU resources if no message arrives within this time.

On the compute node, there is no need to interrupt the kernel (and application) for acknowledgment packets. Since the kernel allows only one thread per core, the core cannot be used to run another application thread while waiting for an acknowledgment. Only if the FCNP interface would support asynchronous reads and writes, this would be useful, but we did not implement this, mainly because the absence of DMA hardware would limit the use of asynchronous I/O anyway.

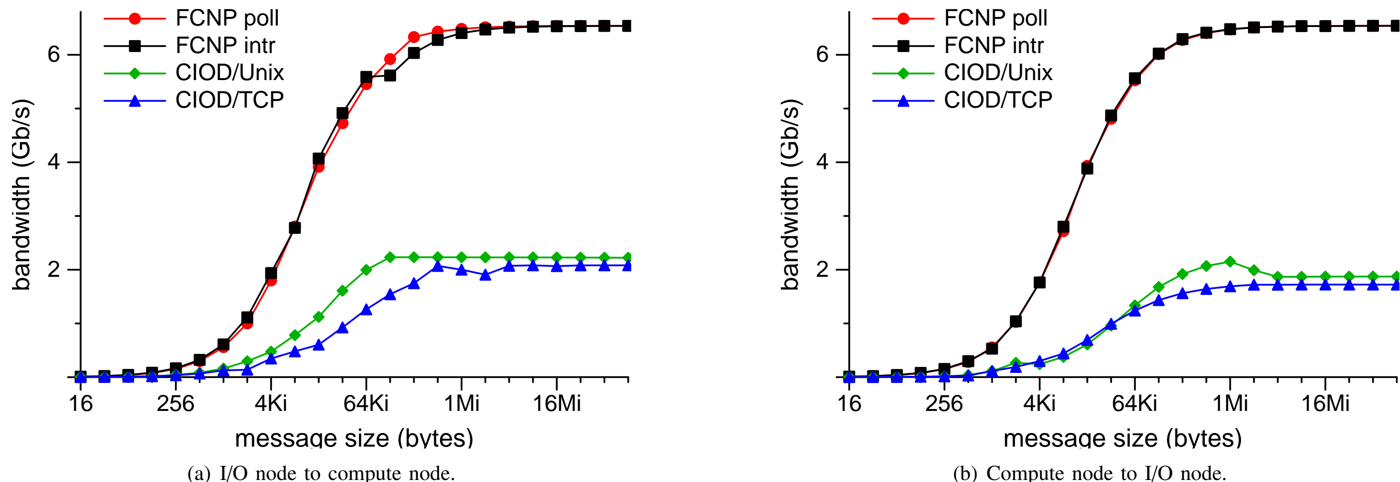
5. Related Work

CIOD (Control and I/O Daemon) [11] is part of the standard Blue Gene system software and performs two major tasks: it is responsible for booting the compute node and starting jobs (control), and for file and socket communication from applications that run on the compute nodes. It uses a function-forwarding mechanism to forward I/O-related system calls from a compute node to its I/O node, where the operations are really performed. This function forwarding is transparent to the user. On the BG/L, its performance was initially poor [10], but improved significantly later [6]. On the BG/P, CIOD is multi-threaded and made open source.

ZOID (ZeptoOS I/O Daemon) [6] is a communication framework that improves I/O performance on the Blue Gene. ZOID is a replacement for CIOD, aimed to provide even better performance and more flexibility. ZOID is extensible: a small daemon on the I/O node provides a basic function forwarding facility from the compute nodes to the I/O nodes. On top of this, plug-ins (in the form of shared objects) that implement some functionality are loaded by the daemon and perform the real work. A standard plug-in is the *Unix* plug-in that implements the Unix I/O related system calls. On the compute nodes, the glibc library was adapted to replace the Unix system calls by stubs that forwards calls like *socket()* and *read()*, and a shared object on the I/O node implement the actual calls. A stub generator creates code that marshals and unmarshals function arguments and results on the compute nodes and I/O nodes.

ZOID's extensibility allows arbitrary application code to be run on the I/O node; something that, on the BG/L, was not possible before. Before the LOFAR BG/L was replaced by a BG/P, we used ZOID to run LOFAR-specific application code on the I/O nodes [6].

ZOID was ported to the BG/P, but runs only with ZeptoOS kernels on the compute nodes, not with the CNK. ZeptoOS did not yet support the 3-D torus, which is of critical importance



(a) I/O node to compute node.

(b) Compute node to I/O node.

Fig. 7: Measured bandwidths, as function of message size.

to the LOFAR application. Once ZeptoOS fully supports the torus, we might use it instead of CNK.

A fundamental difference between ZOID and FCNP is that ZOID integrates system control and application I/O into the same process. ZOID applications (in the form of shared objects) are plugged in into the address space of the daemon, to avoid expensive context switches during communication. A disadvantage of this approach is that a crashing application can crash the control daemon as well. Due to the availability of a free virtual channel on the BG/P, FCNP can separate control I/O from application I/O, without performance penalty.

6. Performance

Figure 7 shows the measured bandwidths for FCNP (with interrupts enabled or disabled) and for TCP and Unix domain socket communication using CIOD. The benchmark communicates data between the I/O node and one of the compute nodes as fast as possible, using messages of various sizes. For large messages, FCNP approaches the link speed.

CIOD peaks at a bandwidth that is slightly over 2 Gb/s. In theory, this equals the required LOFAR data rate, but we found that the bandwidth is not stable over long times, and provides too little headroom for real-time processing.

FCNP is significantly faster than CIOD, because its overhead is much lower. CIOD cannot be blamed for this, since CIOD was designed to provide external, not internal communication. Internal communication happens to work, but a system-call interface with a heavy-weight TCP/IP protocol is obviously less efficient.

The discontinuity in the curve for interrupt-driven FCNP in Figure 7(a) is caused by the fact that the polling thread polls the tree for 50 µs after receiving a request packet, before suspending itself (see Section 4.1). Messages of 64 KiB or more take over 50 µs to send. Therefore, a request for a large message causes an interrupt that increases the latency. In contrast, requests for smaller messages will be received through polling. This effect is not seen in traffic from the compute node to the I/O node, since in this direction, the 50 µs timer starts running after the last data packet was received (rather than the request packet). Consequently, in this benchmark all subsequent requests will be received through polling.

We use a simplified LogGP model [2] to determine the latency, bandwidth, and overhead. We define the approximate time to send a message as the latency plus the message size multiplied by the time to send a byte.

The latency is dominated by the receipt and handling of a request packet on the I/O node (and would be overhead in terms of LogGP). The latency varies depending on several circumstances. Using the benchmark described above, the latency is 13.3 µs for read requests and 13.9 µs for write requests. We also used a different benchmark that measures the latency for the case that the I/O-node application is already waiting for a compute-node request, so that the handshake on the I/O node can complete immediately after a request packet is received. In this case, the latency is only 2.1 µs.

Using interrupts increases the latency if the polling daemon must be awakened. The penalty depends on which cores the polling thread and application thread run. The hardware interrupt is always handled on core 0. If neither thread runs on core 0, the additional latency is 8 µs. If the polling thread runs on core 0, the additional latency is 5 µs. However, if the application thread runs on core 0, the latency is increased by 26 µs, so the application had better avoid affinity to core 0 (there is another reason why one may not want to run an application thread on core 0: Ethernet interrupts are handled there as well).

In both directions, FCNP obtains a bandwidth of 6.54 Gb/s for large messages, hence the time per byte is 1.22 ns. This is as fast as a protocol-less benchmark, that simply sends packets on one side and receives them on the other side, and thus the maximum that the hardware can practically achieve.

Since reading and writing is done in separate threads, this bandwidth can be achieved in both directions simultaneously. A separate benchmark confirmed this claim.

6.1 Application Performance

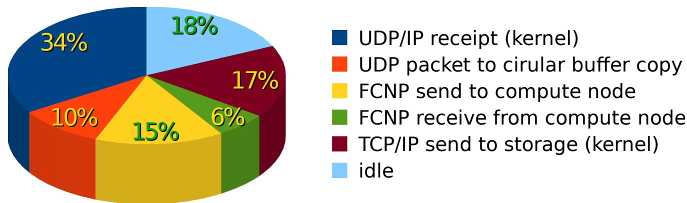


Fig. 8: Performance breakdown on the I/O node.

We also measured how the LOFAR correlator benefits from FCNP. Figure 8 shows a breakdown of the processor utilization on the I/O node. Apart from other tasks, each I/O node sends 3.1 Gb/s and receives 1.2 Gb/s from the compute nodes. The data are sent directly from the circular buffer, without additional copying (obeying the alignment restrictions, though, was rather complicated). The Linux real-time scheduler (SCHED_RR) runs both the sending and receiving threads at the highest priority, to assure that the correlator can always proceed. These threads consume 15% resp. 6% of the total CPU power. This is slightly more than the theoretical minimum of 12% resp. 4.6%, mainly caused by the presence of other threads that compete for the same resources (cache, memory, etcetera). With a CPU utilization of 82%, the I/O node cannot handle much more without starting dropping data. Without FCNP, the required LOFAR data rates are not achieved; with FCNP, the correlator runs smoothly over 50% beyond the requirements.

7. Conclusions

This paper described *Fast Collective Network Protocol (FCNP)*, a very low-overhead network protocol for communication between the I/O nodes and compute nodes on the Blue Gene/P. The relatively slow processor cores on this system are hardly capable to keep up with the fast internal network, hence any protocol overhead significantly slows down the obtained bandwidths. Moreover, a low-overhead protocol is all the more important, because I/O nodes on the Blue Gene/P are much heavier loaded than I/O nodes on its predecessor Blue Gene/L. To scale up, Blue Gene/P I/O nodes must forward data 4.86 times faster than Blue Gene/L I/O nodes, over links that are only 2.43 faster, using processor processor cores that are a marginal 21% faster.

FCNP is critically important to the correlator of the LOFAR radio telescope. One of the novel features of this telescope is that it does all real-time, central processing in *software* rather than hardware, but the processing and bandwidth requirements demand the use of a supercomputer. The standard system software is, however, not designed to provide user-level bandwidths between I/O nodes and compute nodes at the required data rates. FCNP, on the contrary, allows the LOFAR correlator to achieve bandwidths that are even over 50% beyond the requirements, keeping up with the absolute maximum data rates that the LOFAR stations can produce. As a consequence, a 50% wider part of the spectrum can be observed instantaneously, or alternatively, the number of concurrent observations can be increased by a half. Hence, with FCNP, the LOFAR telescope can be used much more efficiently than it was ever designed for.

Acknowledgments

Chris Broekema, Jan David Mol, and Rob van Nieuwpoort made useful comments to a draft version of this paper. We thank Kamil Iskra and Kazutomo Yoshii from Argonne National Labs and Bruce Elmegreen, Todd Inglett, Tom Liebsch, and Andrew Tauferner from IBM for their support.

LOFAR is funded by the Dutch government through the BSIK program for interdisciplinary research and improvement of the knowledge infrastructure. Additional funding is provided through the European Regional Development Fund and the innovation program EZ/KOMPAS of the Collaboration of the Northern Provinces (SNN). ASTRON is part of the Netherlands Organization for Scientific Research, NWO.

References

[1] http://www.lofar.org/p/astronomy_spec.htm.
[2] A. Alexandrov, M.F. Ionescu, K.E. Schauser, and C. Scheiman. LogGP: Incorporating Long Messages into the LogP Model — One Step Closer Towards a Realistic Model for Parallel Computation. In *ACM Symposium on Parallel Algorithms and Architectures (SPAA'95)*, pages 95–105, 1995.
[3] P. Boonstoppel. Semi-Transparent Dual-Processing on Blue Gene/L I/O Nodes. Master's thesis, Dept. of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, May 2008.
[4] H.R. Butcher. LOFAR: First of a New Generation of Radio Telescopes. *Proceedings of the SPIE*, 5489:537–544, October 2004.
[5] A.W. Gunst and M.J. Bentum. Signal Processing Aspects of the Low Frequency Array. In *IEEE International Conference on Signal Processing and Communications*, pages 600–603, Dubai, United Arab Emirates, November 2007.
[6] K. Iskra, J.W. Romein, K. Yoshii, and P. Beckman. ZOID: I/O-Forwarding Infrastructure for Petascale Architectures. In *ACM SIGPLAN Symposium on Principles and Practice on Parallel Programming (PPoPP'08)*, pages 153–162, Salt Lake City, UT, February 2008.
[7] K. Langendoen, J.W. Romein, R.A.F. Bhoedjang, and H.E. Bal. Integrating Polling, Interrupts, and Thread Management. In *Proceedings of Frontiers'96*, pages 13–22, Annapolis, MD, October 1996.
[8] R. J. Nijboer and J. E. Noordam. LOFAR Calibration. In R. A. Shaw, F. Hill, and D. J. Bell, editors, *Astronomical Data Analysis Software and Systems (ADASS XVII)*, number 376 in ASP Conference Series, pages 237–240, Kensington, UK, September 2007.
[9] J.W. Romein, P.C. Broekema, J.D. Mol, and R.V. van Nieuwpoort. Processing Real-Time LOFAR Telescope Data on a Blue Gene/P Supercomputer, 2009. Under review.
[10] J.W. Romein, P.C. Broekema, E. van Meijeren, K. van der Schaaf, and W.H. Zwart. Astronomical Real-Time Streaming Signal Processing on a Blue Gene/L Supercomputer. In *ACM Symposium on Parallel Algorithms and Architectures (SPAA'06)*, pages 59–66, Cambridge, MA, July 2006.
[11] IBM Blue Gene team. Overview of the IBM Blue Gene/P Project. *IBM Journal of Research and Development*, 52(1/2), January/March 2008.
[12] M. de Vos, A.W. Gunst, and R. Nijboer. The LOFAR Telescope: System Architecture and Signal Processing. *Proceedings of the IEEE*. To appear.