Vol.2014-HPC-145 No.13
2014/7/28

Visualizing Collectives over InfiniBand Networks (Unrefereed Workshop Manuscript)

Kevin A. Brown[1,a)] Jens Domke[1,b)] Satoshi Matsuoka[1,c)]

Abstract: As the scale of high performance computing systems increases, optimizing interprocess communication becomes more challenging while being critical for ensuring good performance. Furthermore, the hardware layer abstraction provided by MPI makes it difficult to perform any application optimization that links network utilization with application communication. We overcome this barrier by extending the Peruse utility in Open MPI to track network events within MPI operations from the application layer. We also develop a non-intrusive profiling library to make use of our Peruse enhancement and show how we can use BoxFish with our profiling library to visualize the flow of application traffic over each link within large scale InfiniBand networks. The tool-chain that we describe can be used without any modification to the target application and incurs less than 1% application runtime overhead.

Keywords: Open MPI, InfiniBand, Peruse, Profiling, BoxFish

1. Introduction

High performance computing (HPC) systems are rapidly growing in physical size, with staggering increases in node counts in recent years. The Tianhe-2 supercomputer which tops the Top500 list of fastest supercomputers comprises over 16,000 nodes [2]. In fact, each of the five fastest supercomputers in the world, based on this list, has over 15,000 nodes. Expectedly, the interconnect networks that support communication among the thousands of nodes in these systems are simultaneously growing in complexity. As this complexity increases, inter-process communication becomes an even more significant factor in the overall performance of applications, especially for communication bound applications [4]. It is therefore no surprise that optimizing communication within these large-scale applications is a standard approach in performance tuning on these massive systems.

The Message Passing Interface (MPI) [20] has become the most widely used message passing standard on HPC systems, defining both communication and process management APIs. The actually method of performing data communication over the network hardware is hidden within the MPI library's implementation and is transparent to the application. Unfortunately, the hardware abstraction performed by the MPI library introduces an obstacle in making a connection between the communication routines within the application and data transfer events in the hardware layer. The issue is further complicated when considering collective communication operations, which are critical to some of the most important HPC applications such as those using fast Fourier Transforms (FFTs) [23]. The logic of collectives are completely hidden within the MPI implementation. Internal communication semantics are often dynamically chosen at runtime and, without knowledge of the MPI library's internals, are virtually impossible to track.

The profiling interface provided by the MPI standard, PMPI, allows users to intercept calls to the MPI library. This provides access to the parameters of the intercepted function, allowing the user to monitor arguments being used by the MPI operations. VampirTrace/Vampir [15], Scalasca [6] and other widely used performance analysis tools rely on PMPI. Since this profiling interface does not penetrate the MPI layer, network performance information cannot be gathered in this manner. Therefore, any optimizing strategies relying on tools that use only this interface will ignore the impact of the network layer activities on an application's performance.

BoxFish [12] is one of the few tools that circumvents this abstraction by including network performance metrics in application performance analysis. It can overlay performance metrics from multiple domains, such as application, communication, etc., on a graphical representation of the hardware. However, current research using BoxFish requires the tracking of port counters across the network in order to visualize link traffic. The approach is impractical in complex networks such as TSUBAME2.5's fat-tree network, especially when multiple applications are sharing the system.

To overcome the limitations of currently available tracing and profiling tools, we design a profiling tool that is capable of extracting data from within the MPI library with no modifications to the application source code. Our approach extends the Peruse utility [1] within Open MPI to track data transmission over InfiniBand network interfaces. The profiling library that we created makes use of this extension and tracks the InfiniBand transmis-

[1] Tokyo Institute of Technology, Meguro, Tokyo 152-8552, Japan
[a)] brown.k.aa@m.titech.ac.jp
[b)] domke.j.aa@m.titech.ac.jp
[c)] matsu@is.titech.ac.jp

sion events during the application's run. Finally, we create a visualization module for BoxFish to visualize our fat-tree network. Our method facilitates the comprehensive performance analysis of an application's communication pattern with minimal intrusion while utilizing widely available tools.

The rest of the paper is organised as follows. Section 2 covers the important aspects of the technologies that we used. We give the details of our implementation and describe the experiments we conducted in Sections 3 and 4, respectively. In Section 5 we look at other related research and explain how our work differs from the others. Then we present the conclusion in Section 6.

2. Technology

2.1 InfiniBand

InfiniBand [10] is a channel-based, low latency, high throughput switched networking architecture that is widely used in the HPC industry. The InfiniBand Trade Association proposed the InfiniBand standard in the year 2000 and the technology is currently being used by 44.4%, or 222, of the systems on the Top500 list [11].

InfiniBand hardware allows concurrent data flow over independent incoming and outgoing channels and also supports offloading network communication from the CPU. Applications can bypass the CPU to gain direct access to the interface by using `ibverbs`, the InfiniBand API, resulting in simultaneous end-to-end connectivity with little impact on CPU load. InfiniBand hardware also provides Remote Direct Memory Access (RDMA) facilities, transferring data from a process's memory on the local host to a process on a remote host without involving the CPU.

In an InfiniBand network, a set of interconnected channel adapters and switches are referred to as a subnet. A node that is not a switch may be connected to multiple subnets. Each subnet is managed by a subnet manager process running on a node, such as a compute node or a switch, in that subnet. The `ibdiagnet` utility is used to query the configuration and status of elements in the subnets. Each active port on a channel adapter is assigned a local identifier (LID), which is unique to on that subnet. Additionally, each channel adapter's port is assigned a unique ID called PGUID, and each channel adapter and switch is assigned a unique node ID called NGUID. Messages are produced and consumed at channel adapters and forwarded though the subnet via switches. LIDs are used by switches when forwarding messages to their destination ports.

2.2 Open MPI

The MPI standard defines a communication and process management API for processes on distributed systems. It also defines the portability requirement for all libraries that implement this API. Open MPI is one of the most widely used MPI library and has been chosen for this research because it implements the Peruse interface (see Section 2.3).

Open MPI's code base is separated into three sections: OMPI, ORTE and OPAL. It uses the Modular Component Architecture (MCA) to dynamically search for and load components at runtime. Fig. 1 illustrates the features of the MCA. MCA frameworks provide the API for services that are handled by that framework, but not the implementation of the services. Each framework handles an exclusive set of tasks related to specific area. For example, transmitting data between processes is managed by the `btl` framework and collective communication logics is handled by the `coll` framework. Each component of a given framework contains a unique implementation of the services handled by that framework. Components are selected and loaded by the framework at runtime. Modules are the runtime instantiations of components.

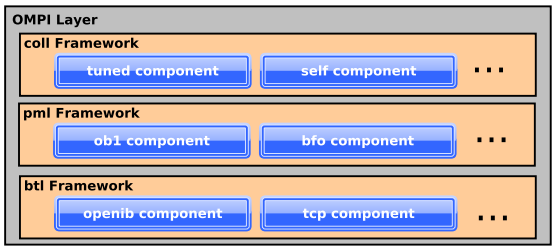

Fig. 1 Figure showing a few of the frameworks in the OMPI layer and their related components

Our focus in this work is on InfiniBand network traffic, hence we are most concerned with the `openib` component in the `btl` framework, which manages the transfers of data between processes via InfiniBand network adapters. The `openib` component uses the `ibverbs` API to interface with the InfiniBand channel adapters. All MPI operations involving the InfiniBand channel adapters are done by this component; it performs all aspects of connection management and data transfer over the InfiniBand network. If there exists multiple active InfiniBand ports on the node, the `btl` framework will instantiate an `openib` module for each port.

2.3 Peruse

The Peruse utility was proposed as a performance revealing extension to the MPI standard that allows the tracking of internal events within an MPI library [1]. It accomplishes this by registering a user-defined callback function to each event of interest within MPI. Two examples of these events in an `MPI_Send` operation are (i) the point when the MPI library begins processing the send request and (ii) the point when the actual data transmission begins. The Peruse standard defines the API for querying the events that are supported by Peruse within an MPI implementation. It also defines the interface for registering callback function, the function prototype for callback function and methods for enabling and disabling events. The callback functions are executed within the MPI library and are passed variables from within the MPI space as arguments.

Keller et. al [13] describes the details of implementing Peruse in Open MPI. Their research reported a 1.7% increase in communication latency when using Open MPI+Peruse versus the native Open MPI on an InfiniBand network. Peruse was integrated in Open MPI through the use of C macros, which are defined in the Peruse code base. The macros are passed various parameters such as message size, the communicator involved and the event being performed in the function from which they are called. A macro is

inserted at each point in the Open MPI code base where an event of interest occurs. The code within the macro checks if the corresponding Peruse event is activate, packages the arguments for the callback function and executes the callback function that was registered for that event if it is active.

The implementation of Peruse in Open MPI is entirely contained within the OMPI code section, and all events are tracked within the `pml` framework's components. `pml` stands for Point-to-point Management Layer and, as its name states, it coordinates the the point-to-point transfer of data in MPI operations. All information necessary for tracking the events of interest is provided in this layer. Furthermore, `pml` components remain hardware agnostic by using `btl` components to perform the actual data transmission; thus, Peruse gains the flexibility of being hardware agnostic.

2.4 BoxFish

BoxFish is a performance analysis tool that is capable of visually representing the physical nodes and links in a network. It uses visualization modules to present performance data in different forms: a `Table` module presents data in a tabular form, a `3D Torus` module constructs a 3D torus network topology and presents performance data as the colours of the network elements, etc. The same performance metrics may be presented simultaneous in multiple modules and filters may be applied to modules individually and in groups.

The core of BoxFish handles the reading or performance data from input files and stores the information in a generic, module-independent form. It passes data to the modules as requested, and manages the linking of performance data across multiple modules.

3. Design

3.1 Extending Open MPI

First, we created a definition for the "`PERUSE_OPENIB_SEND`" event in the Peruse code base. This event corresponds to points when data is sent over InfiniBand adapters. We then created a special Peruse macro to handle these events and properly communicate InfiniBand hardware information to the callback function. Because of reasons mentioned in Section 2.3, the hardware-specific information required by this macro is not available in the `pml` layer where all other events are tracked. The calls to our macro were therefore added to each point in the `openib` component where data is sent to the InfiniBand interfaces. The activation and monitoring of this event results in the tracking of point-to-point messages sent over individual InfiniBand ports from within the MPI library.

3.2 `ibprof` Profiler

Using the enhanced Peruse, we built the `ibprof` tool to record information from the InfiniBand events. The `ibprof` library, which can be linked at compile time or preloaded at runtime, maintains two arrays of traffic counters for each active port on the system, one array for bytes sent and one for bytes received. Separate arrays are used for sent and received data because information transmitted during an RDMA operation is not always recorded at both the sender and receiver. This is due to the fact that the size of message fragments used to initiate RDMA operations may not coincide with the amount of data transmitted during the operation. The length of each array is equal to or greater than the maximum LID value in the network. The index of each array element corresponds to a target LID, and the value of that element corresponds to the amount data sent to/received from that target LID. The array is populated within the user callback function that has been registered on our new Peruse event. Counter values are written to Open Trace Format (OTF) files using the OTF library [14, 16].

`ibprof` registers and activates the "`PERUSE_OPENIB_SEND`" send event. Depending on the status of its environment variables, `ibprof` can capture every "`PERUSE_OPENIB_SEND`" event throughout the entire duration of the program, or limit the scope to only the events within specified collective operations. Furthermore, the library supports manual instrumentation of the user application to specify blocks of operations, or application phases, for which the event should or should not be activated for.

3.3 Fat-Tree Visualization

We wrote a new visualization module for BoxFish that can effectively represent the TSUBAME2.5's fat-tree network. BoxFish is used to visualize our profile because of its ability to link information across multiple domains: application, hardware and communication domains.

Our application profiles provide information on (i) which MPI process is running on which node, (ii) the InfiniBand Port configuration for nodes with MPI processes and (iii) the size and destination of traffic sent per port per application phase. We developed a post-processing script that parses our profiles and `ibdiagnet` output files to extract performance metrics and network configuration information. After parsing all input files, the script creates as connected graph to represent the nodes and links in the network. It then adds weights to the links by tracing application traffic across the network using the port forwarding tables. Finally, output files are written containing position and performance information in a format that can be read by BoxFish and visualized by our module.

4. Experiments

4.1 Overhead Measurements

TSUBAME-KFC was used for experiments to measure the runtime overhead imposed by our profiling library. The system, based at the Tokyo Institute of Technology (Tokyo Tech), has of 40 compute nodes and two 24-port InfiniBand FDR switches. Each compute node is connected to one of the switches and has two Intel Xeon E5-2629v2 processors. The switches are interconnected using 15 links. Profiling was done by running the target code with our library preloaded using the `LD-PRELOAD` command. No other user processes were running on the system while experiments were being conducted.

We used an `MPI_Alltoall` microbenchmark running on 32 compute nodes with message size ranging from 0 bytes to 32 kilobytes. 30 profiled trials and 30 un-profiled trials were ran for each message size, with each trial comprising of 20,000 iter-

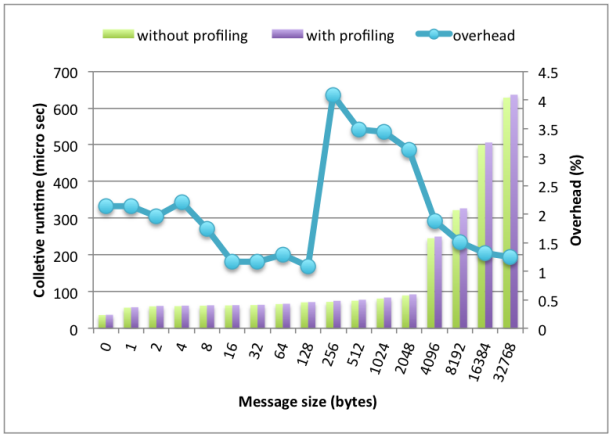

Fig. 2 Chart comparing the runtime of an MPI Alltoall call with various message sizes when profiled by our tool.

ations of the collective call (2 initialization runs + 19,998 timed runs). The minimum of the average of the runs for each trial was taken as the resulting value since this would be the most reproducible result [7]. Results are shown in Fig. 2. The overhead ranged from 1.09% to 4.08% of the collective's runtime.

In addition to the micro-benchmark, we also conducted experiments using the communication bound FT benchmark of the NAS Parallel Benchmark Suite [22, 24]. The FT benchmark makes use of collective communication to solve a partial differential equation using fast Fourier Transforms [3]. We ran 100 profiled runs and 100 un-profiled runs of the benchmark using the class C problem size on 32 nodes. We took the minimum of the average total runtime for each trial. The runtime of the benchmark increased from 12.1849 seconds to 12.1874 seconds when our profiling library is introduced, representing an overhead of less than 0.02%.

Results presented in this section do not include the time for writing output to OTF files, which is usually less than 10 milliseconds in our test environment.

4.2 Large-Scale Application Profile and Visualization

For large scale visualization experiments, we used Tokyo Tech's TSUBAME2.5 system [8]. This system has over 1,400 compute nodes and over 350 switches. It is designed with 2 subnets and each compute nodes has one link connected to each subnet. Based on how the job queues were configured at the time of the experiments, only 256 nodes could be used; our benchmark required that the number of nodes be a power of 2.

Fig 3 shows a visualization of the complete run of the FT benchmark on 256 nodes using the class C problem size. Layer 0 represents all the nodes in the network that possess channel adapters. This includes all compute and storage nodes. Layers 1a, 2a and 3a contains all switches in the first subnet and layers 1b, 2b and 3b contains the switch in the second subnet. All compute nodes are connected to both subnets, while storage and specific management nodes are connected to a single subnet. Links are coloured based on the link utilization colour range shown on the upper-right side of the figure. To ensure we accurately represent the bidirectional flow of traffic, each half of a link is coloured independently based on the traffic sent from the the node connected at that end. For simplicity, we excluded performance data for the nodes and focused on only the performance metrics for the links.

The user can select an appropriate colour scheme for displaying the elements (links and nodes) and then inspect the image for links that transfer the most data or possible hotspots. When a link is selected, it is highlighted in the image; both nodes connected to that link are also highlighted. Additionally, all other links connected to the highlighted nodes are also highlighted. The remaining links and nodes in the network are faded. The lower section of 3 shows the selection of the link that transmitted the most data during the application's execution. The table on the right in this image reflects the highlighted elements (links or nodes) and gives additional information about that element. The colour scheme chosen for 3 allows to quickly identify the most utilized links, by scanning the image for the reddest links. Fig. 4 is a visualization of the same application profile using a different colour scheme for the links. With this scheme, we can quickly identify the links with similar performance characteristics and traffic patterns across the network.

5. Related Work

5.1 Visualization

In their work on visualization approaches for parallel applications, Muelder, Gygi and Ma [21] discussed the drawbacks of many visualization approaches at large scales and presented a new way of visualizing thousands of processes. Their approach allows the user to drill down into regions of interest from an high-level overview, making it more scalable than the other methods they investigated. However, unlike our work, their method does not include any reference to network hardware information.

Bhatele et. al [5] and Isaacs et. al [12] showed how BoxFish can be used to effectively explore performance data on large-scale systems by including the hardware domain. In fact, by using BoxFish in addition to other analysis tools, Bhatele et. al gained a 22% performance improvement for an adaptive mesh refinement library. Their work, however, dealt solely with the 3D torus network of the IBM Blue Gene/P (BG/P) system [9]. Our approach is independent of any topology and can be used on any system that utilizes InfiniBand hardware.

5.2 Tracing and Profiling

In their article on the performance analysis of simulations on BG/P systems, Landge et. al [18] demonstrated how BoxFish can be used to visualise network traffic generated during an application's execution. In addition to the network performance characteristics of MPI collectives and point-to-point operations, they presented several case studies in which they investigate the performance characteristics of a layer and plasma interaction simulator running on different BG/P systems. Communication traffic measurements required for their visualizations were attained by intercepting MPI calls using the PMPI interface, then capturing port counters using BG/P system tools. The application's source code was also manually instrumented to summarize and output the communication metrics that was captured during different application phases. While our work relates very closely to theirs in that we seek to visualize the network performance of applica-

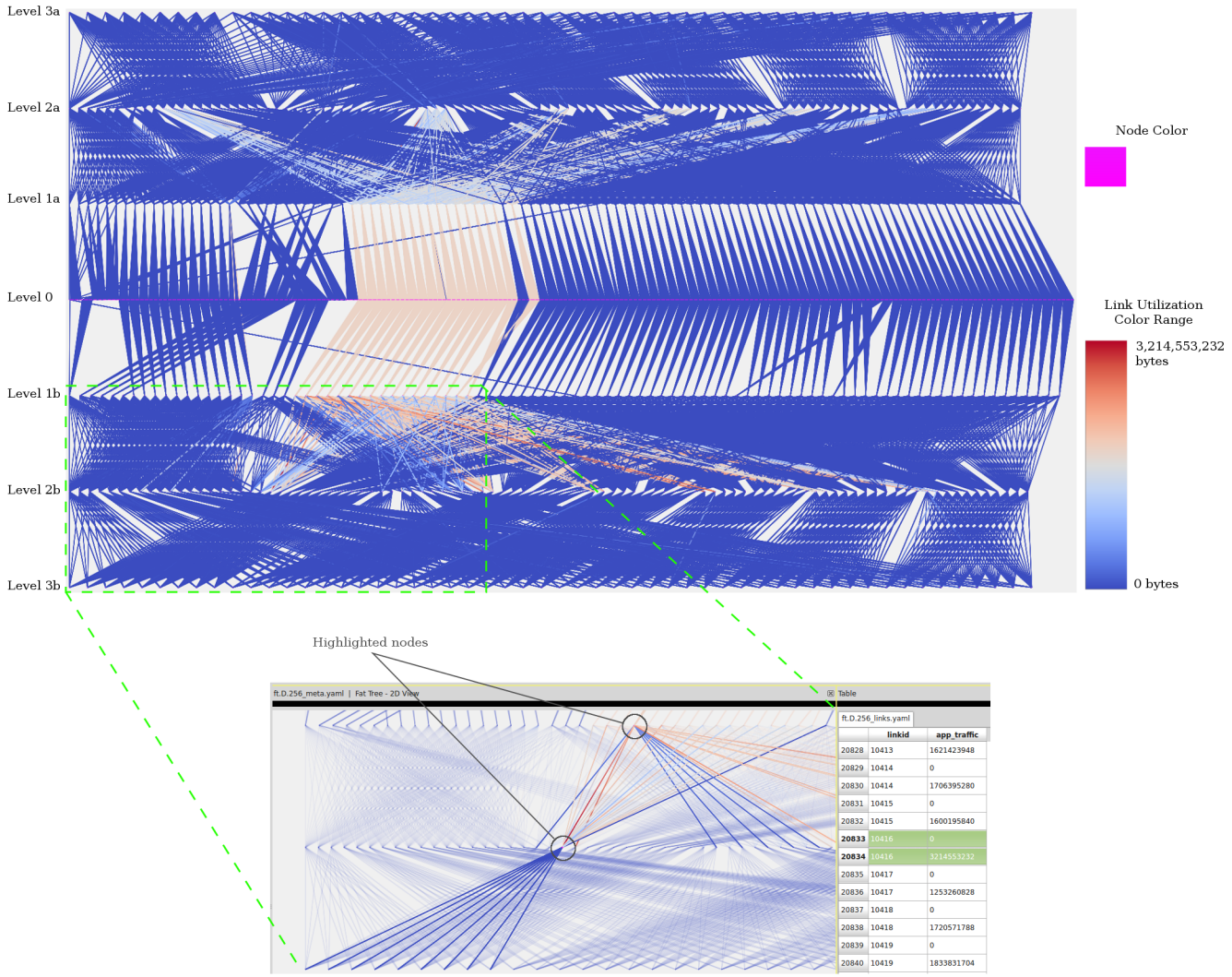

Fig. 3 A BoxFish visualization of the NPB FT benchmark running on 256 nodes of TSUBAME2.5 with the class D problem size. The upper image shows the full view of the network while the lower image shows a close-up of the most utilized link being highlighted.

tions running on large scale systems using BoxFish, our approach to measuring network traffic is very unique. We focus on systems that use InfiniBand networks and extract application traffic metrics from within the MPI layer instead of via port counters. This gives our approach the advantage of measuring only the traffic generated by the application we are profiling, even if the network and systems are being shared with other applications.

Miquel-Alonso, Navaridas and Riruejo [19] described a process of using MPI application traces to conduct simulation-based performance analysis. In order to track the internal point-to-point communication of collectives with their traces, they modified the source code of MPICH2 to expose these point-to-point function calls. They were then able to use the PMPI interface to trace these internal calls. In a similar manner, Kunkel et. al [17] also modified the source code of MPICH to gain access to function calls within collectives via the PMPI interface. Our extension of Peruse, though considered a modification of Open MPI, is an extension supported by the Peruse implementation. Additionally, our modification reveals information of the ports used in the communication while theirs still has no access to information in the hardware layer.

6. Conclusion

MPI libraries, by design, prevent the user from easily seeing the correlation between communication events in the application and data transmission over network links. This limits optimization strategies that incorporate network performance metrics when conducting performance analysis. In this paper, we showed how to overcome this barrier by enabling the tracking of network communication events within the MPI layer. The Peruse utility, which is built into Open MPI, was extended to report whenever the library sent data over InfiniBand interfaces. We built a profiling tool, `ibprof`, which takes advantage of this extended Peruse utility and reports the network traffic generated by each process with minimal intrusion to the application. Our profiler incurs only 0.02% runtime overhead with the NPB FT benchmark and less than 5% overhead with an `MPI_Alltoall` collective call.

Additionally, we designed a fat-tree visualization module for BoxFish and demonstrated its use by visualising the FT benchmark on TSUBAME2.5. Using our approach, the user can ac-

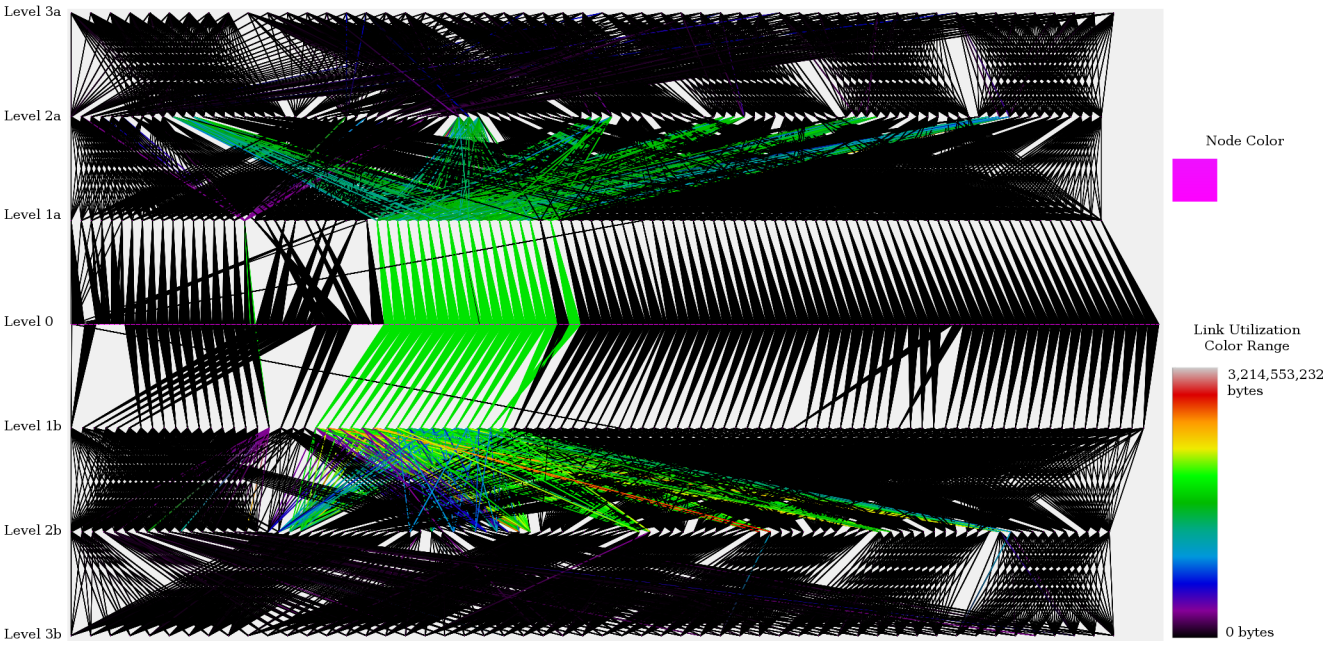

Fig. 4 A BoxFish visualization of the NPB FT benchmark running on 256 nodes of TSUBAME2.5 with the class D problem size. The colour scheme chosen for the links allow us to quickly see communication patters across the network.

cess interactive visualizations of application performance data on large-scale system with minimal intrusion and overhead.

References

[1] MPI PERUSE: An MPI Extension for Revealing Unexposed Implementation Information version 2.0. `http://hcl.ucd.ie/wiki/images/e/ea/Current_peruse_spec.pdf`, Mar 2006. Accessed: 2014-06-29.

[2] Top500 List. `http://www.top500.org/`, Jun 2014.

[3] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. The NAS Parallel Benchmarks (RNR Technical Report RNR-94-007). `https://www.nas.nasa.gov/assets/pdf/techreports/1994/rnr-94-007.pdf`, Mar 1994. Accessed: 2014-04-16.

[4] C. Bell, D. Bonachea, R. Nishtala, and K. Yelick. Optimizing Bandwidth Limited Problems Using One-Sided Communication and Overlap. In *Proc of IPDPS*, 2006.

[5] A. Bhatele, T. Gamblin, K. Isaacs, B. Gunney, M. Schulz, P. Bremer, and B. Hamann. Novel Views of Performance Data to Analyze Large-scale Adaptive Applications. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pages 1–11, Nov 2012.

[6] M. Geimer, F. Wolf, B. J. N. Wylie, E. brahm, D. Becker, and B. Mohr. The Scalasca performance toolset architecture. *Concurrency and Computation: Practice and Experience*, 22:702–719, 2010.

[7] W. Gropp and E. L. Lusk. Reproducible Measurements of MPI Performance Characteristics. In *Proc. of Euro PVM/MPI*, 1999.

[8] GSIC, Tokyo Institute of Technology. TSUBAME2.5 Hardware and Software Specifications. `http://www.gsic.titech.ac.jp/en/node/420`, Nov 2013. Accessed: 2014-06-30.

[9] IBM Blue Gene Team. Overview of the IBM Blue Gene/P project. *IBM Journal of Research and Development*, 52(1.2):199–220, Jan 2008.

[10] InfiniBand Trade Association. `http://www.infinibandta.org/`, Jun 2014.

[11] InfiniBand Trade Association. Worlds Top Supercomputers Deploy InfiniBand. `http://www.infinibandta.org/content/pages.php?pg=press_room_item&rec_id=801`, Jun 2014.

[12] K. E. Isaacs, A. G. Landge, T. Gamblin, P.-T. Bremer, V. Pascucci, and B. Hamann. Exploring Performance Data with Boxfish. In *Proc. of SC Comp.*, 2012.

[13] R. Keller, G. Bosilca, G. Fagg, M. Resch, and J. J. Dongarra. Implementation and Usage of the PERUSE-Interface in Open MPI. In *Proc. of Euro PVM/MPI*, 2006.

[14] A. Knüpfer, R. Brendel, H. Brunst, H. Mix, and W. E. Nagel. Introducing the Open Trace Format (OTF). In *Proceedings of the 6th International Conference on Computational Science - Volume Part II*, ICCS'06, pages 526–533, 2006.

[15] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. Müller, and W. Nagel. The Vampir Performance Analysis Tool-Set. In M. Resch, R. Keller, V. Himmler, B. Krammer, and A. Schulz, editors, *Tools for High Performance Computing*, pages 139–155. Springer Berlin Heidelberg, 2008.

[16] A. Knüpfer, H. Brunst, A. D. Malony, and S. S. Shende. Open Trace Format API Specification Version 1.1. `http://www.paratools.com/otf?action=AttachFile&do=get&target=specification.pdf`, Nov 2006. Accessed: 2014-06-29.

[17] J. Kunkel, Y. Tsujita, O. Mordvinova, and T. Ludwig. Tracing Internal Communication in MPI and MPI-I/O. In *Parallel and Distributed Computing, Applications and Technologies, 2009 International Conference on*, pages 280–286, Dec 2009.

[18] A. Landge, J. Levine, A. Bhatele, K. Isaacs, T. Gamblin, M. Schulz, S. Langer, P.-T. Bremer, and V. Pascucci. Visualizing Network Traffic to Understand the Performance of Massively Parallel Simulations. *IEEE Trans. on Vis. and Computer Graphics*, Dec 2012.

[19] J. Miguel-Alonso, J. Navaridas, and F. Ridruejo. Interconnection Network Simulation Using Traces of MPI Applications. *International Journal of Parallel Programming*, 37(2):153–174, 2009.

[20] MPI Forum. `MPI`: A Message-Passing Interface Standard. `http://www.mpi-forum.org/`, Mar 2014.

[21] C. Muelder, F. Gygi, and K.-L. Ma. Visual Analysis of Inter-Process Communication for Large-Scale Parallel Computing. *Visualization and Computer Graphics, IEEE Transactions on*, 15(6):1129–1136, Nov 2009.

[22] NASA Ames Research Center. Nas parallel benchmarks. `https://www.nas.nasa.gov/publications/npb.html`, Mar 2012. Accessed: 2014-04-16.

[23] A. Nukada, K. Sato, and S. Matsuoka. Scalable multi-GPU 3-D FFT for TSUBAME 2.0 Supercomputer. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 44:1–44:10. IEEE Computer Society Press, 2012.

[24] F. Wong, R. Martin, R. Arpaci-Dusseau, and D. Culler. Architectural Requirements and Scalability of the NAS Parallel Benchmarks. In *Proc. of SC*, 1999.