StressBench: A Configurable Full System Network and I/O Benchmark Framework

Dean G. Chester*, Taylor Groves†, Simond D. Hammond‡, Tim Law§, Steven A. Wright¶, Richard Smedley-Stevenson§, Suhaib A. Fahmy‖, Gihan R. Mudalidge* and Stephen A. Jarvis**

*Department of Computer Science, University of Warwick, Coventry, United Kingdom
†Advanced Technologies Group, NERSC, Berkeley, USA
‡Sandia National Laboratories, Albuquerque, USA
§AWE PLC, Aldermastion, United Kingdom
¶University of York, York, United Kingdom
‖School of Engineering, University of Warwick, Coventry, United Kingdom
**University of Birmingham, Birmingham, United Kingdom

***Abstract*—We present StressBench, a network benchmarking framework written for testing MPI operations and file I/O concurrently. It is designed specifically to execute MPI communication and file access patterns that are representative of real-world scientific applications. Existing tools consider either the worst case congestion with small abstract patterns or peak performance with simplistic patterns. StressBench allows for a richer study of congestion by allowing orchestration of network load scenarios that are representative of those typically seen at HPC centres, something that is difficult to achieve with existing tools. We demonstrate the versatility of the framework from microbenchmarks through to finely controlled congested runs across a cluster. Validation of the results using four proxy application communication schemes within StressBench against parent applications shows a maximum difference of 15%. Using the I/O modeling capabilities of StressBench, we are able to quantify the impact of file I/O on application traffic showing how it can be used in procurement and performance studies.**

I. Introduction

Predicting the performance of supercomputers is vitally important in evaluating their suitability for applications and for informing the procurement process. Time to solution is usually the primary metric of consideration, and can be impacted by OS jitter [1], network contention [2], and resource allocation [3]. While there exist a variety of simulators focused on modeling the computational aspects of supercomputers, consideration of the networking infrastructure has been less thorough. Since network contention can cause variability in communication time [4]; it is prudent to develop a benchmarking tool that is capable of reproducing traffic patterns commonly seen in scientific applications, thereby allowing for more faithful replication of these workloads on new and existing machines. By benchmarking systems using higher level communication patterns, such a tool enables applications to be evaluated on a variety of architectures without source code being released, which is beneficial in the case of commercially sensitive and/or restricted codes, where the underlying application architecture cannot be exposed. Common MPI benchmark tools either take a generic approach to network congestion [5] or focus on performance of individual MPI operations [6], [7]. This generic approach to network congestion may not be representative of what can be expected from a shared multi-user system.

Understanding the interactions and impact between applications on a multi-user system can be used to improve both resource scheduling and allocation, and the communication patterns themselves, e.g., the development of communication avoiding algorithms such as those used within linear solver applications [8].

In this paper we address these shortcomings with the development of a novel network replication framework called StressBench, that is capable of executing complex communication patterns concurrently and reproducing application workflows. StressBench allows for **user-defined**, **customisable** communication patterns and interactions that can generate representative, realistic congestion.

Specifically, this paper makes the following contributions:

- We document the development of a customisable network and I/O benchmarking tool that uses traffic patterns to evaluate architectures;
- We evaluate the tool against commonly used network microbenchmarks and validate application workflow replication with four proxy applications all within 15% difference;
- We replicate a full system run and use this to demonstrate the impact of network contention on the time-to-solution of multiple proxy applications;
- Finally, we extend our full system replication to present a novel case study assessing communication performance while in contention with common I/O strategies.

The remainder of this paper is structured as follows: Section II explains the research area; Section III describes the system architecture of StressBench; Section IV outlines a validation study of StressBench; In Section V we present two performance studies performed with StressBench; finally, Section VI concludes this paper.

II. Related Work

The usual approach to assessing the performance of massively parallel systems consists of executing a large set of benchmarks with a variety of differing communication patterns, often sequentially. How concurrently running applications interact with a machine's shared resources (i.e., the interconnect and parallel file system) is usually difficult to understand. As a result, these benchmarks fail to provide an accurate picture of application performance as they are unable to capture realistic network usage and highlight potential issues such as load imbalances that may affect the performance of collective operations [9].

ScalaBenchGen provides a way to automatically trace and replay applications as a synthetic MPI benchmark [10]. This approach uses the MPI Profiling layer (PMPI) to capture the MPI events which are stored chronologically; these events are then replayed through a custom tool. One limitation of ScalaBenchGen is that it provides no capability to scale communication sizes as multiple application traces must be captured with varying sizes.

File I/O can often interfere with MPI communications as large amounts of traffic are sent and received over the network. Dickson et al. have studied the I/O characteristics of large applications by replicating I/O workloads with MACSio [11]. This is achieved by capturing Darshan [12], [13] logs of applications, parsing the log files and generating input parameters for MACSio.

Common MPI Benchmarks include the Intel MPI Benchmarks (IMB) [6], OSU Microbenchmarks (OSU) [7], and SKaMPI [14]. These microbenchmarks focus on the performance of singular MPI operations: either point-to-point or collective operations. They are useful when trying to diagnose application performance issues as they often report the average time for MPI operations. SKaMPI is no longer under active development but was extended to cater for complex communication patterns [15].

The NAS Parallel Benchmarks replicate commonly used application patterns to benchmark systems [16]. While this benchmark suite comprises a wide variety of parallel patterns it does not orchestrate them to show how the patterns can interact with or affect one another.

More recent benchmarks such as GPCNeT look at testing network performance in isolation and under load [5]. GPCNeT provides artificial noise in a network with four congestor patterns and is designed to stress a system rather than provide representative communication of a specific workload.

Task Bench is a parameterised benchmark for evaluating parallel systems [17]. It allows for rapid replication of a variety of programming models and applications. Configurable parameters allow the tuning of the length of the benchmark; the degree of parallelism; the type of kernel (such as a stencil or sweep) and tuning of any potential imbalance. This task-based approach is a novel idea that allows for flexibility and customisation. One drawback of this tool is that it only focuses on one task at a time meaning that it is difficult to understand how the chosen benchmark will perform in a production environment.

I/O studies looking at improving performance are not new [18]–[20]. These studies typically focus on tuning I/O parameters for specific applications and systems. They often fail to consider the I/O subsystem being a shared resource and as such what contention may be affecting their performance.

Wright et al. have investigated the effects of I/O performance in relation to contention of the I/O nodes within a system [21]. They note that contention within the I/O subsystem can result in a 13% performance decrease on a multi-user system.

Our proposed StressBench tool overcomes the limitations discussed above by allowing a configurable workload to be run tailored to the applications that interest the system evaluator.

III. StressBench

StressBench was designed to be flexible and applicable to all parallel workloads, this is done through a portable interface that allows for extension and additions to the communication patterns. Motifs are small implementations of communication patterns. Multiple motifs can be chained together to create a *job* which can resemble a production application. A motif is applied in three phases:

Decomposition In this phase, StressBench breaks up the global MPI communicator world into the relevant MPI groups which each have multiple communication patterns associated with them. Once the MPI groups have been constructed, the communication patterns themselves perform a decomposition if required to establish their nearest neighbours in the case of a halo exchange.

Perform During the perform phase the communication patterns execute as if they were a standalone application using their MPI communicator to communicate. Each pattern is timed individually and the total job is timed.

Cleanup The cleanup phase allows the patterns to safely clean up any resources that have been consumed. Each job also collates the timings from each of its group's ranks and then prints these to standard output.

These three phases are separated by global barriers to ensure they begin at the same time such that the patterns under test are controlled tightly to ensure that they are performed concurrently.

Listing 1 shows an example TeaLeaf iteration with I/O write after.

StressBench reports aggregated timings this includes the total job time and the time of each motif within a job. These aggregated statistics include the minimum, average and maximum time per job and motif. Further timing can be collected inside of a motif to provide further insight in to behaviour, for example in the I/O motifs additional timing information is captured so that the I/O bandwidth can be calculated and reported in the output.

A. Communication Patterns

Motifs can be written by providing implementations for each of the key phases of a given application. The example

Listing 1: Example StressBench Input

```
[JOB_NAME] TeaLeaf_CG
[NID_LIST] 6,9,17,18,33,41,58,67,
    72,75,83,84,87,90,102,103
[MOTIF] AllReduce
[MOTIF] Compute -m 350000
[MOTIF] AllReduce
[MOTIF] Compute -m 350000
[MOTIF] halo2d -x 4000 -y 4000
[MOTIF] MPIIO -s 1500000 -i 1 -m 256 -f <file_path>
    -n MPI_File_write_all
```

motifs are taken from mini-applications and can be seen in production applications. As these patterns try to interact with the network (a shared resource) contention increases which can degrade application performance. These examples have been taken from the following proxy applications; TeaLeaf [22], CloverLeaf [23], Sweep3D [24], LULESH [25] and Hardware/Hybrid Accelerated Cosmology Code (HACC) [26].

A "compute" motif is provided so that more intricate workloads can be built. The compute motif includes the capability to emulate load imbalance. This has been achieved by generating a value from a specified distribution. In the case of a Gaussian distribution the mean and standard deviation are parameters passed to the transform function which generates a normal deviate. The normal deviate is used as the intended computation time.

The design of the emulated patterns depends on the communications and computation pipeline. This communication computation pipeline can be extracted using an MPI tracing tool such as Intel Trace Analyzer and Collector (ITAC) [27]. Once the communication/computation pipeline has been extracted the building blocks can be constructed for StressBench. In the case of TeaLeaf there is no overlap between the communications and computation pipeline and could be implemented independent motifs. For Sweep3D the communication and computation pipeline are tightly coupled so the computation has to be integrated into the design of the motif. By understanding the communication and computation pipeline the information can then be built in to a motif by providing implementations to the 3 phases. For the implementation of the 2D halo-exchange the problem decomposition was extracted from TeaLeaf and then rebuilt inside of the decompose functionality. The perform functionality involved inspecting how the 2D halo-exchanges take place and replicating the MPI library calls based upon the provided communicator. In the deletion phase all buffers used are freed; this is independent of the pattern being replicated.

Multiple motifs have already been implemented in StressBench:

Halo Exchanges	A 2D structured halo exchange and 3D unstructured halo exchange are provided.
AllReduce	Support for two reduction operations; sum and minimum operations.
Computation	A computation motif is provided to emulate computation. A distribution can be provided to generate a load imbalance.
Incast	A file I/O motif providing N-1 communications.
AllToAll	An AllToAll pattern is provided in the default package.
Sweep3D	A Sweep3D motif is provided offering a Sweep communications pattern.
PingPong	A PingPong style motif is also provided for measuring the latency while in contention.

B. I/O Patterns

I/O motifs have been integrated in to StressBench to allow the I/O subsystem to be benchmarked simultaneously to the network. This gives the ability to understand the interactions between I/O traffic and MPI application traffic; while these may be configured to avoid interaction, the underlying network has fixed resources that are shared by both types of traffic.

At present, two I/O strategies are implemented, namely N-1 and N-N, where N is processes. These two approaches are the most widely used within HPC applications. For example, StressBench uses N-1 for reading input files.

Currently, these two I/O strategies can be executed through HDF5, MPI-IO or POSIX file operations. Writes are typically of more interest than reads but both have been developed for StressBench.

The I/O bandwidth reported by the motifs is calculated with Equation (1).

$$BW = \frac{\text{Bytes Read/Written}}{\text{Time Taken}} \tag{1}$$

In an effort to validate this implementation, StressBench was run with Darshan profiling the I/O operations. The cumulative timings were taken from the Darshan log for the MPI-IO operations and compared to the cumulative timings from the the output of the motif. The 'MPIIO_F_WRITE_TIME' counter from the Darshan log was used for the comparison. The difference in the timings was less than 0.5% for 1GB files and less than 0.01% for 10GB files. This difference is caused by two factors: the resolution of the timers used and the position in which the timer is placed. Darshan provides wrappers around the I/O function calls which insert the timers within the call while the MPI-IO motif places the timing calls around the I/O function call which includes the calls to capture the information for Darshan; resulting in a marginal difference between the times reported by Darshan and MPI-IO motif.

IV. Validation

A. Evaluation Hardware

We evaluate StressBench on two clusters: Tinis and Isambard, that have different network architectures. Table I shows the hardware configurations for these machines.

StressBench was compiled with GCC (8.3.0) and OpenMPI (4.1.3) on Tinis. Isambard used the default programming environment; Cray Compiler (9.1.3) and Cray MPI (v. 7.7.12).

Aries is Cray's previous network generation found in Cray XC systems. The interconnect is highly configurable and

TABLE I: Hardware Specifications

System	Isambard	Tinis
Node Architecture	2 × 32-core Marvell ThunderX2 2.1 GHz	2 × 8-core Intel Xeon E5-2630 v3 2.4 GHz
Memory per node	Phase1: 256GB Phase2: 512 GB	64GB
Interconnect	Cray Aries	Q-Logic Infiniband
Topology	Dragonfly	Tapered Fat Tree
OS	CLE	Red Hat

scalable to many millions of endpoints. Aries has three ranks of networks [28]. The first rank connects four nodes to a router. The second connects four nodes inside of a group using electrical connections; the topology inside of this is a 2D all-to-all. The third network connects groups together using optical links. A group is typically 384 nodes.

The fat tree in Tinis is a two level tree with a tapering of 2:1. Tapering inside of fat trees has been studied previously [29]. Tinis comprises 212 nodes in total. Two I/O storage nodes provide 500TB with a GPFS file system. The I/O storage nodes sit off an additional network switch outside of the fat tree topology. Therefore, I/O traffic must hit the root switch in order to reach the I/O nodes.

Isambard compute nodes fit within one group connected via electrical connections. Isambard contains 6 I/O Nodes providing a total of 900TB of Lustre storage via a Cray Sonexion 3000 storage cabinet.

B. Microbenchmarks

Traditional MPI benchmarks focus on peak performance. For example, IMB and OSU are designed to run on a quiet system.

One of the motifs built inside of StressBench is 'PingPongAllLatencies' which runs from 0 to 4MB message sizes in the same way as PingPong in IMB and Latency in OSU. This motif was used in the comparison against IMB and OSU. It is possible to measure the latency of a specific message size inside of StressBench.

The default compiler and linker flags were used for building the microbenchmark suites (IMB and OSU). In the case of StressBench the default optimisation level is -O0. The latency benchmarks were scheduled to use one core across two nodes; each of these values were repeated 10 times across different days to establish the average latency the benchmark may achieve.

StressBench performed similarly to the existing MPI microbenchmarks such as PingPong resulting in slight increase in the returned latencies.

For Tinis the average increase in the reported latency was 3.9% (maximum 11.4%) for IMB and 2.75% (maximum 8.2%) for OSU. On Isambard this average difference in the latency was a decrease of -7.9% (maximum -26.3%) for IMB and -29% (maximum -49.2%) for OSU. The large variability on Isambard comes from the adaptive routing inside of the network as each PingPong message may take a different route to reach the desired endpoint, thus varying the latency.

Stressbench demonstrates a negligible difference between traditional microbenchmarks and is a suitable replacement for these traditional microbenchmarks.

C. Representative Workloads

To build a representative workload we extracted key characteristics from a proxy application workflow in order to replicate these patterns with StressBench. These key characteristics were captured with Intel ITAC and were instrumented to capture computation timings, using Caliper [30]. Caliper allows for application source code to be annotated and records snapshots during application execution.

To verify the communication patterns match they were traced with Intel ITAC [27] and the point-to-point message profiles captured. All point-to-point message profiles matched the StressBench emulated version for all four proxy applications and have been left out for brevity. To further validate the workload we compare the times captured inside of StressBench with the timings from the proxy application. The compute timings were generated using a Gaussian distribution function parameterised to model OS jitter.

TeaLeaf is a linear solver proxy application that has a variety of solvers. TeaLeaf solves the heat conduction equations in both 2D and 3D using a 5 and 7 point stencil respectively. The temperatures are cell-centred. Problem set 5 was chosen for the selected problem and has been strong-scaled. This problem is the crooked pipe problem in which a pipe has a lower density than its surroundings and therefore heat travels faster through this part of the problem domain. A Conjugate Gradient (CG) iteration in TeaLeaf consists of two reductions with computation and a 2D halo exchange with post computation; this was confirmed by tracing the application with Intel ITAC and reading through the application source code. In the case of TeaLeaf we emulate one CG iteration.

The difference in the measured and emulated runtimes for TeaLeaf was less than 11% difference, the average difference was -2.7% and maximum was -10.4%. When the communication patterns were traced with Intel ITAC the message profiles matched. The computational motif runtime provided less than 1% of the variability compared to the measured TeaLeaf run. The large variations came from the communications, most notably the halo exchange. In our emulation the message packing was treated as additional computation rather than part of the communication directly. When TeaLeaf is strong scaled like this at larger scales the communications can dominate the execution of an iteration; the computation roughly halves as MPI ranks double. This occurs because the problem size is fixed and distributed over more MPI ranks. The largest difference between the proxy application and the emulation is less than 11% for Tinis in the comparison.

Sweep3D is a discrete ordinates transport code [24], [31]. We have emulated a typical application run which consists of 12 iterations. The problem was weak scaled for each rank to

have a grid size of 50x50x800. The measured compute time was 7.82ms per octant.

The average difference was -3.2% with the largest difference -6% for 16 nodes. As the input deck was weak scaled; we felt this was the best approach to use the average computation time. The point-to-point message profiles for both the measured and emulation matched when compared.

LULESH is a 3D Unstructured Lagrangian Explicit Shock Hydrodynamics proxy application [25]. As with TeaLeaf we have emulated one iteration; we chose to emulate the 50th iteration to provide some warm up iterations. We have performed a weak scaling study across a mesh of 81^3. The computation time was 0.81s for all runs. The maximum difference was 15% between the measured and emulated; this was for the a single node with 8 MPI ranks. For multi-node runs the average difference was -2.2%, maximum was 15% for 1 node; the maximum multi-node difference was -3.7% for 4 nodes. The measured message profiles matched the emulated message profiles for LULESH.

SWFFT is a fast Fourier transform (FFT) proxy application. This type of communication pattern features heavily in astronomy related simulation codes for example Hardware/Hybrid Accelerated Cosmology Code (HACC) [26]. The FFT implementation in SWFFT is from HACC; this operates on 1D FFT steps which are interleaved as transposition and sequential steps. This approach reduces communication overhead. We have emulated a forwards FFT and backwards FFT during the emulation of the pattern. The MPI communications were traced and the number of bytes for the point-to-point communications were recorded. The measured and emulated runtimes for SWFFT differed by as much as 11.4% for a strong scaled problem of 160x160x160. The average error was -7.8% for this problem.

Wu et al. show that their application generated MPI benchmarks from traces achieve differences within 22.1% [10]. Barrett et al. show their Message Rate MPI benchmark shows a 30% difference when compared to OSU microbenchmarks [32]. The differences presented from StressBench are below and comparable to similar studies. This demonstrates that StressBench can be used to effectively emulate communication patterns found in production applications.

V. Performance Studies

A. Full System Orchestration

The validated applications can be composed to mimic a representative system workload. The mimicked workload allows exploration of potential slowdowns as a result of job placement and communication interactions affecting performance.

The workload we have examined utilised the communication patterns in TeaLeaf and Sweep. Applications often perform I/O to either checkpoint or to output a visualisation.

Existing benchmark suites [5], [33] use an Incast-like communication pattern to replicate I/O traffic. While this communication pattern can induce similar network traffic it often lacks the ability to replicate I/O bandwidth which means that it could artificially clear the network. When evaluating a mimicked workload we have used both Incast traffic and real I/O traffic through the use of the Incast and I/O motifs.

I/O traffic patterns were added to these workloads after a number of iterations to emulate this. The size of the I/O has been approximated using Equation (2).

$$\text{I/O Message Size} = \frac{\text{Domain Size} \times \text{variables}}{\text{MPI Ranks}} \tag{2}$$

The system was randomly distributed to have eight jobs of sixteen nodes each; two Sweep3D with an Incast, two TeaLeaf with an Incast, two repeated file I/O and finally two AllToAll Traffic. The problem domain for Sweep3D was configured as 100^3 cube and 5 iterations. The blocking factor for the Z dimension was set to 10 and Incast message size was set to 7,111,111 bytes per rank.

For TeaLeaf the problem was configured similarly to that used in the above validation: a domain size of 4000×4000; five iterations were performed with an Incast motif at the end having a total message size of 16,000,000. The Incast jobs used a size of 480,469 bytes per MPI rank was configured for five iterations. The AllToAll jobs used two message sizes 2048 and 4096 bytes. The file I/O was performed using an 'MPI_File_write_all' call with the MPI-IO motif.

Each job was run in isolation for 20 runs under the same resource allocation of 128 nodes with all other nodes not being provided jobs. All jobs were then combined to execute concurrently across the 128 jobs 30 times.

Fig. 1 shows how each of the jobs performed in isolation and under contention with I/O traffic on Tinis. The mean is shown with the diamond inside of the box plots and the line represents the median. For Sweep3D the worst case slowdown was 1.2×; for TeaLeaf we show a 1.4× slowdown when the applications are trying to run in contention with other applications on the system. The AllToAll applications were slowed down by 1.2× and the Incast application on Tinis was slowed down by 1.1×.

Fig. 2 shows the how Sweep3D and TeaLeaf performed in isolation and under contention with I/O traffic on Isambard. The worst case slowdown for Sweep3D was 1.02×, TeaLeaf 1.02×, the Incast application was slowed down by 1.7× and AllToAll was 1.03×.

Chunduri et al. present a congestion impact (CI) metric [5], shown in Equation (3).

$$\text{CI} = \frac{\text{t}_{\text{congested}}}{\text{t}_{\text{isolated}}} \tag{3}$$

As discussed previously, an Incast pattern can provide a similar communication pattern to file I/O. As such we ran the same jobs with the Incast motif rather than an MPI-IO Write motif. Table II shows how the CI differs between Incast and the use of file I/O on both systems. It is clear that the impact of using real file I/O is greater than using an Incast motif.

Due to the transient nature of the traffic hotspots it is possible that messages are unaffected by network contention resulting in no congestion impact for motifs.

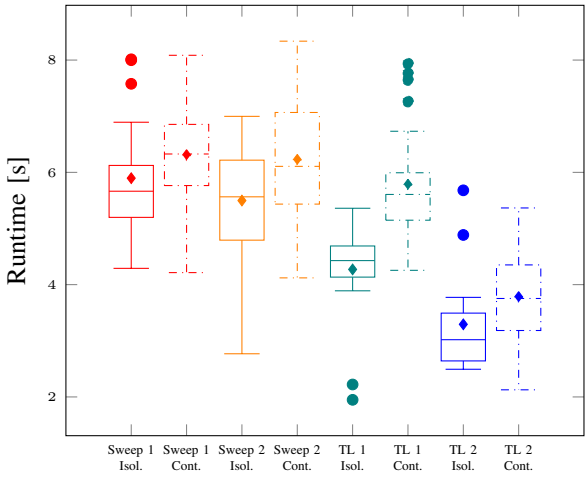

Fig. 1: Application Runtimes in Isolation and in Contention on Tinis

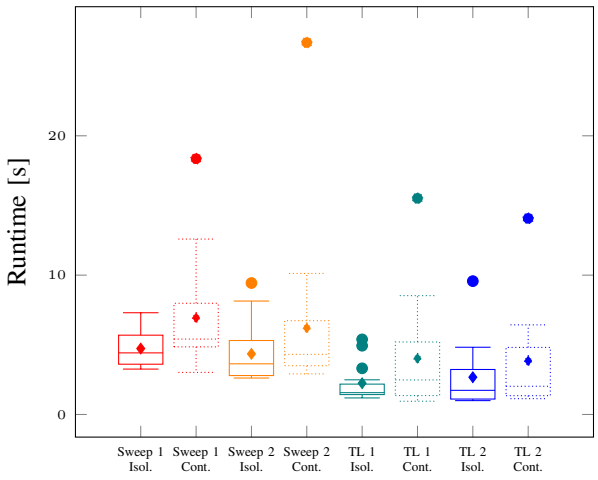

Fig. 2: Application Runtimes in Isolation and in Contention on Isambard

B. I/O Study

This study examines the interactions between I/O traffic and application traffic, and the performance degradation of both of these with StressBench.

There is a presumption that I/O traffic interferes with application traffic yet no such study exists quantifying this interference. As such, we have designed this study to cover breadth rather than depth into a specific interaction. The study focuses on some common communication patterns from applications; we use the validated patterns mentioned above.

To understand the interactions between application communications and I/O traffic we have run a range of application patterns against some large file sizes which would cause congestion inside of the network. In order to ascertain suitable file sizes we analysed the file sizes on four storage systems at NERSC. The data was collected using RobinHood policy

TABLE II: Comparison of CI for Incast and File I/O for Applications

	Tinis		Isambard	
Pattern	Incast	MPIIO	Incast	MPIIO
Sweep 1	1.0068	1.0744	1.0146	1.4664
Sweep 2	1.0010	1.1356	1.0331	1.4266
TeaLeaf 1	1.0172	1.4094	1.0157	1.7936
TeaLeaf 2	1.0707	1.1345	1.0196	1.4392
Incast 1	1.0088	1.1314	1.0004	1.6109
Incast 2	1.0426	1.1161	1.0021	1.6868
All To All 2K	1.0157	1.1917	1.0137	1.0327
All To All 4K	1.0587	1.0000	1.2327	1.0014

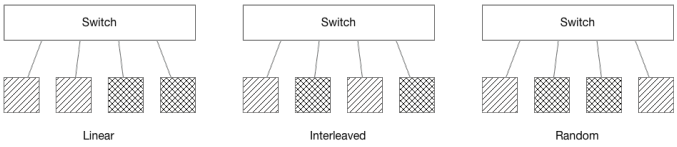

Fig. 3: Job Placement, Diagonal and Hash lines represent different applications

engine [34] and inserting the POSIX information in to a mySQL database [35], [36]. The data provided consisted of the size of files as reported by the inode for each file [36]. We grouped the data in to 5 buckets: 0GB, 1GB, 10GB, 100GB and 500GB. The largest amount of files reside in the less than 1GB bucket.

In an effort to negate background network noise the runs performed the pattern in isolation and then in contention; each job then completed this 10 times. This was done to ensure that the isolated and contended runs performed as close as possible to each other such that they would have an equivalent background noise. These jobs were repeated at differing times across a week to achieve best and worst case background network noise. Each motif in the run was configured to run for at least 30 seconds so that the network can get fully congested and links can be exhausted. This ensures that any adaptive routing algorithms has time to take affect on communication patterns. Previous work has shown that system load can interfere with latency sensitive messages [37].

The study not only looked at the the effects of pattern and file size but also the effects on job placement. Three job placement schemes were used; linear, interleaved and random; Figure 3 shows how the three placement schemes differ.

For runs on Tinis 32 nodes were utilised while on Isambard 256 node runs were used. The small runs on Tinis is sufficient to stress the network given that the fat tree is tapered and the I/O nodes sit off the root switch. Both sets of runs consisted of a 50:50 split between the communication pattern of interest and offending I/O traffic.

To assess the performance degradation we compare the CI. Table III shows how the CI differs against the four file sizes tested for Tinis and Isambard.

In the case of Tinis the file size seems to have negligible difference in the impact on the performance. Rather, the job locality has a greater impact. This is due to the fat tree network

topology deployed in Tinis. Traffic that can be routed between nodes across the same switch are unlikely to suffer because of the file I/O traffic, such as the linear job placement. This results in a slowdown of communication time thus increasing the application runtime.

I/O traffic generated on Isambard has a greater effect on application communication traffic (shown in Table III). This is most notable with a linear placement; with this network it is also observed that I/O traffic size impacts the application communication traffic.

VI. Conclusions

This work demonstrates a reconfigurable tool for benchmarking network performance for MPI applications. The approach presented allows for domain complexity to be abstracted away so that the underlying network performance can be studied using real-world communication patterns. The patterns being studied can be implemented with MPI directly requiring no external infrastructure. These patterns can then be connected together to look at how applications utilise the network while in contention with other communication patterns.

By chaining multiple motifs together applications can easily be replicated within StressBench, we demonstrate that runtime differences are less than 15% for a variety of applications.

The presented orchestration of several applications running concurrently shows that StressBench is a suitable tool for evaluating network performance; with applications such as TeaLeaf running 1.4× on a fat tree slower while in contention with other applications. We also show that real I/O traffic has a greater impact on application communication performance resulting in larger slowdowns when compared to Incast like application traffic. These slow downs results longer time to solutions and more computation resources being used to facilitate the runs as wallclock times are increased to ensure jobs complete. This can impact on research budget where the computation resource is fixed.

We have shown how on systems with adaptive routing that file size affects the performance rather than the placement of the jobs; while on systems with static routing such as those in fat trees we have shown that job location is more likely to cause issues with contention. To mitigate this contention jobs could be scheduled to avoid being placed linearly and instead interleaved; this may improve the performance of the application communication patterns.

Possible extensions to this work include more validated communication patterns from other scientific disciplines and machine learning applications, as well as the addition of I/O libraries to allow for better replication of applications.

Acknowledgements

The authors would like to acknowledge the Scientific Computing Research Technology Platform, University of Warwick, for assistance in the research described in this paper.

This work used the Isambard UK National Tier-2 HPC Service (http://gw4.ac.uk/isambard/) operated by GW4 and the UK Met Office, and funded by EPSRC (EP/P020224/1).

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

References

[1] P. De, V. Mann, and U. Mittaly, "Handling os jitter on multicore multithreaded systems," in *2009 IEEE International Symposium on Parallel & Distributed Processing*. IEEE, 2009, pp. 1–12.

[2] A. Bhatele, A. R. Titus, J. J. Thiagarajan, N. Jain, T. Gamblin, P.-T. Bremer, M. Schulz, and L. V. Kale, "Identifying the culprits behind network congestion," in *2015 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2015, pp. 113–122.

[3] A. Bhatele, K. Mohror, S. H. Langer, and K. E. Isaacs, "There goes the neighborhood: performance degradation due to nearby jobs," in *SC'13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 2013, pp. 1–12.

[4] S. Chunduri, K. Harms, S. Parker, V. Morozov, S. Oshin, N. Cherukuri, and K. Kumaran, "Run-to-run variability on xeon phi based cray xc systems," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017, pp. 1–13.

[5] S. Chunduri, T. Groves, P. Mendygral, B. Austin, J. Balma, K. Kandalla, K. Kumaran, G. Lockwood, S. Parker, S. Warren *et al.*, "Gpcnet: designing a benchmark suite for inducing and measuring contention in hpc networks," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 1–33.

[6] Intel, "Intel MPI Benchmarks," https://software.intel.com/en-us/imb-user-guide (accessed September 20, 2020), 2020.

[7] Ohio State University, "OSU Micro-Benchmarks," http://mvapich.cse.ohio-state.edu/benchmarks/ (accessed September 20, 2020), 2020.

[8] E. Carson, N. Knight, and J. Demmel, "An efficient deflation technique for the communication-avoiding conjugate gradient method," *Electronic Transactions on Numerical Analysis*, vol. 43, no. 125141, p. 09, 2014.

[9] P. Marendić, J. Lemeire, T. Haber, D. Vučinić, and P. Schelkens, "An investigation into the performance of reduction algorithms under load imbalance," in *European Conference on Parallel Processing*. Springer, 2012, pp. 439–450.

[10] X. Wu, V. Deshpande, and F. Mueller, "Scalabenchgen: Auto-generation of communication benchmarks traces," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium*. IEEE, 2012, pp. 1250–1260.

[11] J. Dickson, S. A. Wright, D. Harris, S. Maheswaran, J. Herdman, M. C. Miller, and S. A. Jarvis, "Enabling portable i/o analysis of commercially sensitive hpc applications through workload replication," *Cray User Group*, pp. 1–14, 2017.

[12] P. Carns, K. Harms, W. Allcock, C. Bacon, S. Lang, R. Latham, and R. Ross, "Understanding and improving computational science storage access through continuous characterization," *ACM Trans. Storage*, vol. 7, no. 3, Oct. 2011. [Online]. Available: https://doi.org/10.1145/2027066.2027068

[13] P. Carns, R. Latham, R. Ross, K. Iskra, S. Lang, and K. Riley, "24/7 characterization of petascale i/o workloads," in *2009 IEEE International Conference on Cluster Computing and Workshops*, 2009, pp. 1–10.

[14] R. Reussner, P. Sanders, L. Prechelt, and M. Müller, "Skampi: A detailed, accurate mpi benchmark," in *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer, 1998, pp. 52–59.

[15] M. Haller and T. Worsch, "Skampi—including more complex communication patterns," in *High Performance Computing in Science and Engineering'03*. Springer, 2003, pp. 455–466.

[16] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber *et al.*, "The nas parallel benchmarks," *The International Journal of Supercomputing Applications*, vol. 5, no. 3, pp. 63–73, 1991.

TABLE III: Comparison of Application Workload against Congestion Impact

		Tinis			Isambard		
		AllReduce	Halo Exchange	Sweep3D	AllReduce	Halo Exchange	Sweep3D
Interleaved	1GB	1.026	1.006	1.001	1.000	1.100	1.001
	10GB	1.154	1.007	1.000	1.000	1.015	1.036
	100GB	1.172	1.003	1.006	1.122	1.018	1.070
	500GB	1.222	1.004	1.001	1.145	1.059	1.045
Linear	1GB	1.028	1.006	1.000	1.031	2.026	1.189
	10GB	1.023	1.006	1.002	1.050	1.159	1.139
	100GB	1.015	1.004	1.000	1.104	1.125	1.286
	500GB	1.024	1.000	1.002	1.212	1.157	1.118
Random	1GB	1.015	1.008	1.002	1.101	1.073	1.001
	10GB	1.132	1.000	1.004	1.119	1.096	1.046
	100GB	1.152	1.010	1.003	1.262	1.138	1.064
	500GB	1.135	1.008	1.001	1.304	1.137	1.199

[17] E. Slaughter, W. Wu, Y. Fu, L. Brandenburg, N. Garcia, W. Kautz, E. Marx, K. Morris, Q. Cao, G. Bosilca, S. Mirchandaney, W. Lee, S. Treichler, and U. Patrick McCormick pat@lanl.gov Los Alamos National Laboratory, "Task bench: A parameterized benchmark for evaluating parallel runtime performance," in *2020 SC20: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. Los Alamitos, CA, USA: IEEE Computer Society, nov 2020, pp. 864–878. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SC41405.2020.00066

[18] A. Acharya, M. Uysal, R. Bennett, A. Mendelson, M. Beynon, J. Hollingsworth, J. Saltz, and A. Sussman, "Tuning the performance of i/o-intensive parallel applications," in *Proceedings of the fourth workshop on I/O in parallel and distributed systems: part of the federated computing research conference*, 1996, pp. 15–27.

[19] W. Yu, J. S. Vetter, and H. S. Oral, "Performance characterization and optimization of parallel i/o on the cray xt," in *2008 IEEE International Symposium on Parallel and Distributed Processing*. IEEE, 2008, pp. 1–11.

[20] B. Behzad, H. V. T. Luu, J. Huchette, S. Byna, R. Aydt, Q. Koziol, M. Snir *et al.*, "Taming parallel i/o complexity with auto-tuning," in *SC'13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 2013, pp. 1–12.

[21] S. A. Wright and S. A. Jarvis, "Quantifying the effects of contention on parallel file systems," in *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, 2015, pp. 932–940.

[22] S. McIntosh-Smith, M. Martineau, T. Deakin, G. Pawelczak, W. Gaudin, P. Garrett, W. Liu, R. Smedley-Stevenson, and D. Beckingsale, "Tealeaf: a mini-application to enable design-space explorations for iterative sparse linear solvers," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2017, pp. 842–849.

[23] A. Mallinson, D. A. Beckingsale, W. Gaudin, J. Herdman, J. Levesque, and S. A. Jarvis, "Cloverleaf: Preparing hydrodynamics codes for exascale," *The Cray User Group*, vol. 2013, 2013.

[24] K. R. Koch, R. S. Baker, and R. E. Alcouffe, "Solution of the first-order form of the 3-d discrete ordinates equation on a massively parallel processor," *Transactions of the American Nuclear Society*, vol. 65, no. 108, pp. 198–199, 1992.

[25] I. Karlin, A. Bhatele, J. Keasler, B. L. Chamberlain, J. Cohen, Z. DeVito, R. Haque, D. Laney, E. Luke, F. Wang, D. Richards, M. Schulz, and C. Still, "Exploring traditional and emerging parallel programming models using a proxy application," in *27th IEEE International Parallel & Distributed Processing Symposium (IEEE IPDPS 2013)*, Boston, USA, May 2013.

[26] S. Habib, V. Morozov, N. Frontiere, H. Finkel, A. Pope, and K. Heitmann, "Hacc: Extreme scaling and performance across diverse architectures," in *SC'13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 2013, pp. 1–10.

[27] Intel, "Intel Trace Analyzer and Collector," https://software.intel.com/en-us/intel-trace-analyzer (accessed June 18, 2021), 2021.

[28] G. Faanes, A. Bataineh, D. Roweth, T. Court, E. Froese, B. Alverson, T. Johnson, J. Kopnick, M. Higgins, and J. Reinhard, "Cray cascade: a scalable hpc system based on a dragonfly network," in *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 2012, pp. 1–9.

[29] P. Taffet, S. Rao, E. León, and I. Karlin, "Testing the limits of tapered fat tree networks," in *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. IEEE, 2019, pp. 47–52.

[30] D. Boehme, T. Gamblin, D. Beckingsale, P.-T. Bremer, A. Gimenez, M. LeGendre, O. Pearce, and M. Schulz, "Caliper: performance introspection for hpc software stacks," in *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2016, pp. 550–560.

[31] G. R. Mudalige, M. K. Vernon, and S. A. Jarvis, "A plug-and-play model for evaluating wavefront computations on parallel architectures," in *2008 IEEE International Symposium on Parallel and Distributed Processing*. IEEE, 2008, pp. 1–14.

[32] B. W. Barrett and K. S. Hemmert, "An application based mpi message throughput benchmark," in *2009 IEEE International Conference on Cluster Computing and Workshops*, 2009, pp. 1–8.

[33] Sandia National Laboratories, "Ember Communication Pattern Library," https://github.com/sstsimulator/ember (accessed December 17, 2020), 2018.

[34] T. M. Declerck *et al.*, "Using robinhood to purge data from lustre file systems," *Proceedings of the 2014 Cray User Group, Lugano*, 2014.

[35] G. K. Lockwood, K. Lozinskiy, L. Gerhardt, R. Cheema, D. Hazen, and N. J. Wright, "A quantitative approach to architecting all-flash lustre file systems," in *International Conference on High Performance Computing*. Springer, 2019, pp. 183–197.

[36] Glenn Lockwood, "Inode sizes on NERSC's production file systems," https://zenodo.org/record/2530940# (accessed December 17, 2020), 2019.

[37] D. G. Chester, S. A. Wright, and S. A. Jarvis, "Understanding communication patterns in hpcg," *Electronic Notes in Theoretical Computer Science*, vol. 340, pp. 55–65, 2018.