



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

ULB

# **Capturing the impact of external interference on HPC application performance**

Shah, Aamer  
(2020)

DOI (TUprints): <https://doi.org/10.25534/tuprints-00011774>

Lizenz:



CC-BY-SA 4.0 International - Creative Commons, Attribution Share-alike

Publikationstyp: Ph.D. Thesis

Fachbereich: 20 Department of Computer Science

Quelle des Originals: <https://tuprints.ulb.tu-darmstadt.de/11774>

---

# Capturing the impact of external interference on HPC application performance

Zur Erlangung des akademischen Grades Doktor-Ingenieur (Dr.-Ing.)  
Genehmigte Dissertation von M.Sc. Aamer Shah aus Abbottabad, Pakistan  
Tag der Einreichung: 03.06.2020, Tag der Prüfung: 15.07.2020

1. Gutachten: Prof. Dr. Felix Wolf
2. Gutachten: Prof. Dr. Matthias Müller  
Darmstadt



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

Computer Science  
Department  
Parallele Programmierung

Capturing the impact of external interference on HPC application performance

Accepted doctoral thesis by M.Sc. Aamer Shah

1. Review: Prof. Dr. Felix Wolf
2. Review: Prof. Dr. Matthias Müller

Date of submission: 03.06.2020

Date of thesis defense: 15.07.2020

Darmstadt

Bitte zitieren Sie dieses Dokument als:

URN: urn:nbn:de:tuda-tuprints-urn:nbn:de:tuda-tuprints-117747

URL: <http://tuprints.ulb.tu-darmstadt.de/id/eprint/11774>

Dieses Dokument wird bereitgestellt von tuprints,  
E-Publishing-Service der TU Darmstadt  
<http://tuprints.ulb.tu-darmstadt.de>  
[tuprints@ulb.tu-darmstadt.de](mailto:tuprints@ulb.tu-darmstadt.de)

Die Veröffentlichung steht unter folgender Creative Commons Lizenz:

Namensnennung – Weitergabe unter gleichen Bedingungen 4.0 International (CC BY-SA 4.0)

<https://creativecommons.org/licenses/by-sa/4.0/deed.de>

To  
**My father**



---

## Erklärungen laut Promotionsordnung

### **§8 Abs. 1 lit. c PromO**

Ich versichere hiermit, dass die elektronische Version meiner Dissertation mit der schriftlichen Version übereinstimmt.

### **§8 Abs. 1 lit. d PromO**

Ich versichere hiermit, dass zu einem vorherigen Zeitpunkt noch keine Promotion versucht wurde. In diesem Fall sind nähere Angaben über Zeitpunkt, Hochschule, Dissertationsthema und Ergebnis dieses Versuchs mitzuteilen.

### **§9 Abs. 1 PromO**

Ich versichere hiermit, dass die vorliegende Dissertation selbstständig und nur unter Verwendung der angegebenen Quellen verfasst wurde.

### **§9 Abs. 2 PromO**

Die Arbeit hat bisher noch nicht zu Prüfungszwecken gedient.

Darmstadt, 03.06.2020

---

M.Sc. Aamer Shah



---

# Abstract

---

HPC applications are large software packages with high computation and storage requirements. To meet these requirements, the architectures of supercomputers are continuously evolving and their capabilities are continuously increasing. Present-day supercomputers have achieved petaflops of computational power by utilizing thousands to millions of compute cores, connected through specialized communication networks, and are equipped with petabytes of storage using a centralized I/O subsystem. While fulfilling the high resource demands of HPC applications, such a design also entails its own challenges. Applications running on these systems own the computation resources exclusively, but share the communication interconnect and the I/O subsystem with other concurrently running applications. Simultaneous access to these shared resources causes contention and inter-application interference, leading to degraded application performance.

Inter-application interference is one of the sources of run-to-run variation. While other sources of variation, such as operating system jitter, have been investigated before, this doctoral thesis specifically focuses on inter-application interference and studies it from the perspective of an application. Variation in execution time not only causes uncertainty and affects user expectations (especially during performance analysis), but also causes suboptimal usage of HPC resources. Therefore, this thesis aims to evaluate inter-application interference, establish trends among applications under contention, and approximate the impact of external influences on the runtime of an application.

To this end, this thesis first presents a method to correlate the performance of applications running side-by-side. The method divides the runtime of a system into globally synchronized, fine-grained time slices for which application performance data is recorded separately. The evaluation of the method demonstrates that correlating application performance data can identify inter-application interference. The thesis further uses the method to study I/O interference and shows that file access patterns are a significant factor in determining the interference potential of an application.

This thesis also presents a technique to estimate the impact of external influences on an application run. The technique introduces the concept of intrinsic performance characteristics to cluster similar application execution segments. Anomalies in the cluster are the result of external interference. An evaluation with several benchmarks shows high accuracy in estimating the impact of interference from a single application run.

The contributions of this thesis will help establish interference trends and devise interference mitigation techniques. Similarly, estimating the impact of external interference will restore user expectations and help performance analysts separate application performance from external influence.






---

# Zusammenfassung

---

Anwendungen im Bereich Hochleistungsrechnen (HPC) sind große Softwarepakete mit hohen Rechen- und Speicheranforderungen. Um diesen Anforderungen gerecht zu werden, entwickeln sich die Architekturen von Supercomputern ständig weiter, dabei nimmt ihre Leistungsfähigkeit ständig zu. Heutige Supercomputer erreichen Rechenleistungen im PetaFLOPs-Bereich, indem sie Tausende bis Millionen von Rechenkernen nutzen, die über spezialisierte Kommunikationsnetzwerke miteinander verbunden sind und über ein zentralisiertes E/A-Subsystem mit Petabytes an Speicherplatz ausgestattet sind. Eine solche Struktur erfüllt zwar die hohen Ressourcenanforderungen von HPC-Anwendungen, bringt aber auch eigene Herausforderungen mit sich. Anwendungen, die auf diesen Systemen laufen, haben exklusiven Zugriff auf die Ressourcen für Berechnungen, allerdings müssen sie sich Kommunikationsverbindungen und das E/A-Subsystem mit gleichzeitig laufenden Anwendungen teilen. Der gleichzeitige Zugriff auf diese gemeinsamen Ressourcen verursacht Konflikte und Interferenz zwischen den Anwendungen, was zu einer verminderten Anwendungsleistung führt. Interferenz zwischen Anwendungen ist eine Quelle von Leistungsvariationen zwischen Ausführungen. Andere Variationsquellen, wie z.B. vom Betriebssystem verursachte Fluktuationen, wurden bereits untersucht. Deshalb wird in dieser Doktorarbeit gezielt Interferenz zwischen Anwendungen aus deren Perspektive betrachtet. Schwankungen der Ausführungszeit verursachen nicht nur Unsicherheit und beeinflussen die Erwartungen der Benutzer (insbesondere bei der Leistungsanalyse), sondern führen auch zu einer suboptimalen Nutzung der Ressourcen eines Hochleistungsrechners. Daher zielt diese Arbeit darauf ab, Interferenz zwischen Anwendungen zu evaluieren, Tendenzen zwischen in Konflikt stehenden Anwendungen zu ermitteln und die Auswirkungen externer Einflüsse auf die Ausführungszeit von Anwendungen zu approximieren. Zu diesem Zweck wird in dieser Arbeit zunächst eine Methode vorgestellt, um die Leistung von gleichzeitig laufenden Anwendungen zueinander in Beziehung zu setzen. Die Methode unterteilt die Laufzeit eines Systems in global synchronisierte, feingranulare Zeitabschnitte, für die die Leistungsdaten jeder Anwendung separat aufgezeichnet werden. Die Auswertung der Methode zeigt, dass durch die Korrelation von Leistungsdaten Interferenz zwischen Anwendungen identifiziert werden kann. Weiterhin zeigt die Untersuchung von Interferenz beim Dateizugriff mithilfe der Methode, dass Dateizugriffsmuster ein wichtiger Faktor bei der Bestimmung des Interferenzpotentials einer Anwendung sind. In dieser Arbeit wird weiterhin eine Technik zur Abschätzung des Einflusses von externen Einflüssen auf die Ausführung einer Anwendung vorgestellt. Die Technik führt das Konzept der intrinsischen Leistungsmerkmale zum Gruppieren ähnlicher Segmente der Anwendungsausführung ein. Externe Interferenz führt hierbei zu Anomalien in den Gruppen. Die Auswertung von mehreren Benchmarks zeigt eine hohe Genauigkeit bei der Abschätzung von interferenzbedingten Auswirkungen mit nur einer einzigen Ausführung der Anwendung. Diese Arbeit wird dazu beitragen, Interferenztendenzen zu ermitteln und



---

Techniken zur Vermeidung von Interferenz zwischen Anwendungen zu entwickeln. Ebenfalls werden durch die Abschätzung der Auswirkungen von externen Einflüssen die Leistungserwartungen der Benutzer wiederhergestellt und Leistungsanalytikern wird dabei geholfen, die Anwendungsleistung von externen Einflüssen zu trennen.

---

# Acknowledgements

---

First and foremost, I would like to praise and thank the God Almighty for His blessings, for giving me the ability to undertake this research work, and to carry it to completion.

Special thanks to my mother and my family for continuously supporting me throughout this prolonged process. I would also like to thank my wife and kid for sacrificing their family time for the completion of this dissertation.

I am greatly indebted to Prof. Dr. Felix Wolf for supervising my research work and for helping me through the highs and lows of the work. He was easily reachable, even in the busiest of times, and always had the most pertinent advice. I would also like to thank Prof. Dr. Matthias Müller for supporting me in seeing my research work to completion.

A large number of people, family and friends, have helped me in more than a number of ways. My gratitude to all of them.



---

# Contents

---

|          |                                                                     |           |
|----------|---------------------------------------------------------------------|-----------|
| <b>1</b> | <b>Introduction</b>                                                 | <b>1</b>  |
| 1.1      | Supercomputer architecture . . . . .                                | 1         |
| 1.1.1    | Memory architecture . . . . .                                       | 1         |
| 1.1.2    | Communication interconnect . . . . .                                | 2         |
| 1.1.3    | I/O subsystem . . . . .                                             | 4         |
| 1.2      | Variation in application performance . . . . .                      | 5         |
| 1.2.1    | Operating system jitter . . . . .                                   | 7         |
| 1.2.2    | Process-to-node mapping . . . . .                                   | 8         |
| 1.2.3    | Inter-application interference . . . . .                            | 8         |
| 1.3      | Motivation . . . . .                                                | 9         |
| 1.3.1    | Estimating application execution time . . . . .                     | 9         |
| 1.3.2    | Analyzing application performance . . . . .                         | 10        |
| 1.3.3    | Wasting system resources . . . . .                                  | 10        |
| 1.4      | Thesis contributions . . . . .                                      | 10        |
| 1.4.1    | Identifying and evaluating inter-application interference . . . . . | 10        |
| 1.4.2    | Quantifying inter-application interference . . . . .                | 11        |
| 1.5      | Thesis structure . . . . .                                          | 11        |
| 1.6      | Statement of Originality . . . . .                                  | 11        |
| <b>2</b> | <b>Identifying inter-application interference</b>                   | <b>15</b> |
| 2.1      | Approach . . . . .                                                  | 16        |
| 2.1.1    | Requirements . . . . .                                              | 17        |
| 2.1.2    | Design . . . . .                                                    | 18        |
| 2.1.3    | Deployment . . . . .                                                | 22        |
| 2.2      | Evaluation . . . . .                                                | 23        |
| 2.2.1    | Overhead . . . . .                                                  | 24        |
| 2.2.2    | Interference . . . . .                                              | 26        |
| 2.3      | Summary . . . . .                                                   | 32        |

---

|          |                                                               |           |
|----------|---------------------------------------------------------------|-----------|
| <b>3</b> | <b>Influence of file access patterns on I/O interference</b>  | <b>33</b> |
| 3.1      | Parallel File Systems . . . . .                               | 34        |
| 3.1.1    | Lustre . . . . .                                              | 35        |
| 3.1.2    | GPFS . . . . .                                                | 36        |
| 3.2      | Approach . . . . .                                            | 36        |
| 3.2.1    | File access patterns . . . . .                                | 36        |
| 3.2.2    | Capturing interference . . . . .                              | 38        |
| 3.2.3    | Server-side imbalance . . . . .                               | 39        |
| 3.3      | Evaluation . . . . .                                          | 41        |
| 3.3.1    | Environment . . . . .                                         | 41        |
| 3.3.2    | Experimental setup . . . . .                                  | 42        |
| 3.3.3    | Micro-benchmarks . . . . .                                    | 42        |
| 3.3.4    | Applications . . . . .                                        | 50        |
| 3.4      | Conclusion . . . . .                                          | 56        |
| 3.5      | Contributions . . . . .                                       | 57        |
| <b>4</b> | <b>Visualizing traffic on high-dimensional torus networks</b> | <b>59</b> |
| 4.1      | Visualization . . . . .                                       | 61        |
| 4.1.1    | Detailed plane view . . . . .                                 | 62        |
| 4.1.2    | Composite plane view . . . . .                                | 62        |
| 4.1.3    | Polygon view . . . . .                                        | 62        |
| 4.2      | Example . . . . .                                             | 64        |
| 4.3      | Interference identification . . . . .                         | 64        |
| 4.4      | Conclusion . . . . .                                          | 65        |
| 4.5      | Contributions . . . . .                                       | 65        |
| <b>5</b> | <b>Estimating the impact of interference</b>                  | <b>67</b> |
| 5.1      | Approach . . . . .                                            | 68        |
| 5.1.1    | Methodology . . . . .                                         | 68        |
| 5.1.2    | Profiling methodology . . . . .                               | 75        |
| 5.1.3    | Estimating interference . . . . .                             | 79        |
| 5.1.4    | Interference estimation in depth . . . . .                    | 80        |
| 5.2      | Evaluation . . . . .                                          | 86        |
| 5.2.1    | Experimental setup . . . . .                                  | 86        |
| 5.2.2    | Overhead . . . . .                                            | 87        |
| 5.2.3    | Evaluation methodology . . . . .                              | 89        |
| 5.2.4    | Results . . . . .                                             | 93        |
| 5.3      | Conclusion . . . . .                                          | 96        |

---

|          |                                                                   |            |
|----------|-------------------------------------------------------------------|------------|
| <b>6</b> | <b>Related work</b>                                               | <b>97</b>  |
| 6.1      | Performance analysis tools . . . . .                              | 97         |
| 6.1.1    | Automatic screening of applications . . . . .                     | 97         |
| 6.1.2    | Segmented profiles . . . . .                                      | 98         |
| 6.1.3    | System monitoring . . . . .                                       | 98         |
| 6.2      | Visualizing multi-dimensional data . . . . .                      | 99         |
| 6.3      | Run-to-run variation . . . . .                                    | 101        |
| 6.3.1    | Operating system . . . . .                                        | 101        |
| 6.3.2    | Inter-application interference . . . . .                          | 101        |
| 6.3.3    | Other sources . . . . .                                           | 103        |
| 6.4      | Estimating the impact of inter-application interference . . . . . | 103        |
| 6.5      | I/O access patterns and benchmarking . . . . .                    | 104        |
| <b>7</b> | <b>Summary</b>                                                    | <b>107</b> |
| 7.1      | Outlook . . . . .                                                 | 108        |





---

# List of Figures

---

|     |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |    |
|-----|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----|
| 1.1 | Increase in core count per socket has largely offset the stagnation in single-thread performance [122] © 2015 IEEE. . . . .                                                                                                                                                                                                                                                                                                                                                                                              | 2  |
| 1.2 | The architecture of a typical supercomputer. . . . .                                                                                                                                                                                                                                                                                                                                                                                                                                                                     | 3  |
| 1.3 | The architecture of a compute node. . . . .                                                                                                                                                                                                                                                                                                                                                                                                                                                                              | 3  |
| 1.4 | A non-blocking fat-tree topology. . . . .                                                                                                                                                                                                                                                                                                                                                                                                                                                                                | 4  |
| 1.5 | A three-dimensional torus network with a $(4 \times 3 \times 3)$ network topology. The loop back links are shown as dashed lines. . . . .                                                                                                                                                                                                                                                                                                                                                                                | 5  |
| 1.6 | A dragonfly network topology consisting of three groups. Each group has eight nodes connected via four routers. . . . .                                                                                                                                                                                                                                                                                                                                                                                                  | 6  |
| 1.7 | Architecture of a typical I/O subsystem. . . . .                                                                                                                                                                                                                                                                                                                                                                                                                                                                         | 6  |
| 1.8 | Average message passing rate for a laser-plasma interaction application on IBM Blue Gene/Q (Mira), Cray XE6 (Hopper) and IBM Blue Gene/P (Interpid) systems [8] © 2013 IEEE. . . . .                                                                                                                                                                                                                                                                                                                                     | 7  |
| 2.1 | Operation of a cluster system as a time-space grid. The picture shows four nodes, each running four processes (denoted by colored squares) at a time. During the interval represented by the x-axis the system runs four jobs (A-D). The brightness of each square indicates the intensity of a performance metric. Synchronized time-slice boundaries allow the correlation of performance phenomena across jobs. In the highlighted time slice, high intensity in job A coincides with low intensity in job B. . . . . | 17 |
| 2.2 | Comparison of applications performance with the performance background. The performance background of an application is the sum of the metric values collected for all applications (A, B, C) running at the same time. . . . .                                                                                                                                                                                                                                                                                          | 19 |
| 2.3 | The steps taken by the heartbeat thread at the time slice boundary. . . . .                                                                                                                                                                                                                                                                                                                                                                                                                                              | 21 |
| 2.4 | Depicting the functionality of the integrated setup deployed on the GraphIT system at MSU. Every batch job would get screened by LWM <sup>2</sup> . The resulting profile files were pushed into a database, which was accessed by a web frontend to generate a job digest for users. . . . .                                                                                                                                                                                                                            | 23 |

|      |                                                                                                                                                                                                                                                                                                     |    |
|------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----|
| 2.5  | Runtime overhead of LWM <sup>2</sup> vs. uninstrumented execution for single-threaded MPI (SPEC MPI2007), also in comparison to IPM. The high and low marks indicate the spread after the removal of outliers. . . . .                                                                              | 25 |
| 2.6  | Runtime overhead of LWM <sup>2</sup> vs. uninstrumented execution for Hybrid MPI (NAS), also in comparison to IPM. The high and low marks indicate the spread after the removal of outliers. . . . .                                                                                                | 26 |
| 2.7  | Inter-application interference when sharing I/O resources: continuous I/O benchmark vs. periodic I/O noise. . . . .                                                                                                                                                                                 | 27 |
| 2.8  | Inter-application interference when sharing network resources: 128.GAPgeofem vs. periodic network noise. . . . .                                                                                                                                                                                    | 28 |
| 2.9  | Inter-application interference when sharing node resources: communication interference during point-to-point communication. . . . .                                                                                                                                                                 | 29 |
| 2.10 | Inter-application interference when sharing GPGPU resources: two GPU benchmarks sharing one PCI Express bus. . . . .                                                                                                                                                                                | 30 |
| 3.1  | A typical Lustre configuration, with separate I/O servers for metadata and file storage. . . . .                                                                                                                                                                                                    | 35 |
| 3.2  | Three I/O access patterns [72, 113]. . . . .                                                                                                                                                                                                                                                        | 37 |
| 3.3  | Mapping one file to one OST reduces the runtime imbalance among processes. . . . .                                                                                                                                                                                                                  | 39 |
| 3.4  | Write throughput of I/O servers. The performance of server 3 is degraded, leading to a longer execution time of I/O operations and hence the application. . . . .                                                                                                                                   | 40 |
| 3.5  | Drop in I/O throughput when the patterns are executed in pairs against each other. The pattern at the top shows the probe, while the x-axis shows the signal patterns. A lower bar means lower passive interference and higher throughput. . . . .                                                  | 44 |
| 3.6  | Effect of chunk size on throughput degradation. . . . .                                                                                                                                                                                                                                             | 45 |
| 3.7  | Drop in I/O throughput when the patterns are executed in pairs against each other. The pattern at the top shows the probe, while the x-axis shows the signal patterns. A lower bar means lower passive interference and higher throughput. The signal pattern is executed in periodic mode. . . . . | 46 |
| 3.8  | Effect of process count on the passive degradation produced by patterns on GPFS and Lustre. . . . .                                                                                                                                                                                                 | 47 |
| 3.9  | Drop in I/O throughput in MPI shared-file when the patterns are executed in pairs against each other. The pattern at the top shows the probe, while the x-axis shows the signal patterns. A lower bar means lower passive interference and higher throughput. . . . .                               | 49 |
| 3.10 | Throughput degradation of OpenFOAM when run against the patterns. . . . .                                                                                                                                                                                                                           | 50 |
| 3.11 | Execution of OpenFOAM against periodic occurrences of open-write-close and aggregate-write on GPFS. The time-slice view shows low throughput during the active phases of the micro-benchmarks. . . . .                                                                                              | 52 |

|      |                                                                                                                                                                                                                                                                                                                   |    |
|------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----|
| 3.12 | Throughput degradation of MadBench2 when run against different patterns.                                                                                                                                                                                                                                          | 53 |
| 3.13 | Throughput degradation of HACCIO when run against different patterns. . .                                                                                                                                                                                                                                         | 54 |
| 3.14 | Throughput degradation when the applications are run against each other. .                                                                                                                                                                                                                                        | 55 |
| 4.1  | A three-dimensional torus network with a $(4 \times 3 \times 3)$ network topology. The loop-back links are shown as dashed lines. . . . .                                                                                                                                                                         | 60 |
| 4.2  | An interactive method to visualize traffic on a high-dimensional torus. . . .                                                                                                                                                                                                                                     | 61 |
| 4.3  | Visualizing a synthetic example application occupying a $(4 \times 4 \times 4 \times 4 \times 2)$ network section. The network traffic alternates along the $d$ dimension for even- and odd-ranked nodes. . . . .                                                                                                 | 63 |
| 5.1  | Application iterations and their histograms. . . . .                                                                                                                                                                                                                                                              | 69 |
| 5.2  | Execution segments and wait-state propagation. . . . .                                                                                                                                                                                                                                                            | 69 |
| 5.3  | Clustering of segments based on relative and absolute thresholds. . . . .                                                                                                                                                                                                                                         | 71 |
| 5.4  | Evaluation of clustering algorithms using SPEC MPI2007 benchmark suite. .                                                                                                                                                                                                                                         | 73 |
| 5.5  | Grouping of segments based on communication and file I/O features. . . . .                                                                                                                                                                                                                                        | 74 |
| 5.6  | Combining groups based on communication and file I/O features, and clusters based on computation features into hybrid clusters. . . . .                                                                                                                                                                           | 75 |
| 5.7  | An example demonstrating data compression at runtime in LWM <sup>2</sup> . The index tree has a maximum size limit of 3 nodes . . . . .                                                                                                                                                                           | 76 |
| 5.8  | Reconstructing profile from data on disk. . . . .                                                                                                                                                                                                                                                                 | 78 |
| 5.9  | A timeline of a run of lammps on Hazel Hen. In the timeline depicting the number of completed instructions, the first segment completes more than $8.5 \times 10^8$ instructions. However, the y-axis is cut at $7 \times 10^7$ to show the variation among the rest of the segments. . . . .                     | 80 |
| 5.10 | A histogram of lammps showing the duration and the number of instructions completed in the segments. The histogram of completed instructions is skewed towards lower values, that is the bars are concentrates on the left side, as the first segment completes more than $8.5 \times 10^8$ instructions. . . . . | 81 |
| 5.11 | A boxplot showing the spread of the identified clusters. The y-axis is logarithmic in scale. Clusters with less than 5 segments are not shown. . . . .                                                                                                                                                            | 81 |
| 5.12 | The number of segments in the identified clusters. Clusters with less than 5 segments are not shown. . . . .                                                                                                                                                                                                      | 82 |
| 5.13 | Timeline of Cluster 46. Some segments have longer execution time and are considered to be impacted by external interference. The dashed red line shows the threshold beyond which segments are considered interfered. . . . .                                                                                     | 83 |
| 5.14 | Histograms of Cluster 46 of lammps, showing the duration and the number of instructions completed in the segments. The dashed red line shows the threshold beyond which segments are considered interfered. . . . .                                                                                               | 84 |

---

|      |                                                                                                                                                                                                                                                                         |     |
|------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----|
| 5.15 | Comparing the identified clusters of segments of two runs of Sweep3D. Run 2 took longer to finish compared to run 1. Cluster 14 of run 2 has an outlier which took significantly longer to finish. . . . .                                                              | 85  |
| 5.16 | Overhead of profiling with LWM <sup>2</sup> . . . . .                                                                                                                                                                                                                   | 88  |
| 5.17 | Multiple runs of <code>tera_tf</code> on JUQUEEN, with measured and estimated interference classified as low, medium, or high. Runs are sorted by execution time in descending order. . . . .                                                                           | 89  |
| 5.18 | Logistic function and the highly interfered run probabilities, when the function is applied on the <code>tera_tf</code> runs. . . . .                                                                                                                                   | 90  |
| 5.19 | JUQUEEN: Prediction accuracy as difference of soft classification probability and percentage-point difference between measured and estimated interference. . . . .                                                                                                      | 94  |
| 5.20 | Hazel Hen: Prediction accuracy as difference of soft classification probability and percentage-point difference between measured and estimated interference. . . . .                                                                                                    | 95  |
| 6.1  | A Boxfish visualization of a five-dimensional torus network [84] ©2014 IEEE. . . . .                                                                                                                                                                                    | 99  |
| 6.2  | TorusViz <sup>ND</sup> visualizes a torus network by arranging all the nodes in a circle [22] ©2014 IEEE. The ordering of the nodes can either be based on the sequential curve or the Hilbert curve. Higher traffic on the links is depicted by thicker lines. . . . . | 100 |

---

## List of Tables

---

|     |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |    |
|-----|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----|
| 3.1 | Specifications of the file systems on TSUBAME2.5 used in our experiments. .                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  | 41 |
| 3.2 | Write bandwidth observed in an experimental run on Lustre when the probe open-write-close was executed against three different signal patterns at a chunk size of 1 MiB. . . . .                                                                                                                                                                                                                                                                                                                                                                                                             | 42 |
| 3.3 | Applications and the access pattern they represent including the chunk size.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 | 50 |
| 5.1 | The estimated impact of interference on segments in Cluster 46. . . . .                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      | 84 |
| 5.2 | Comparing two runs of Sweep3D on Hazel Hen. Run 1 is interfered and has a runtime dilation of 33 seconds. . . . .                                                                                                                                                                                                                                                                                                                                                                                                                                                                            | 84 |
| 5.3 | The estimated impact of interference for Cluster 14 of two runs of Sweep3D. Run 2 is impacted to a higher degree. The median of the cluster remain stable and is similar for both the runs. . . . .                                                                                                                                                                                                                                                                                                                                                                                          | 86 |
| 5.4 | Collective call rate of bulk-synchronous SPEC MPI2007 benchmarks on JUQUEEN and Hazel Hen systems. . . . .                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   | 87 |
| 5.5 | Measured ( $M$ ) and estimated ( $E$ ) interference for runs of <code>tera_tf</code> as percentages of their runtime. The difference between the two percentages is also listed (as $\delta$ ). Only runs 1-15 are shown. The runs are also classified to indicate <i>high</i> (H), <i>medium</i> (M), and <i>low</i> (L) degree of interference. Even though the estimated interference for runs 12 and 13 is only about 1% lower, they are misclassified, demonstrating that such a <i>hard</i> classification is not a suitable metric for evaluating the accuracy of our method. . . . . | 90 |
| 5.6 | The $F_1$ score of <code>tera_tf</code> runs. Runs 12 and 13 are misclassified (despite about only 1% error in the estimated interference), which reduces the $F_1$ score of classifying <i>medium</i> and <i>low</i> runs. . . . .                                                                                                                                                                                                                                                                                                                                                          | 91 |
| 5.7 | Measured ( $M$ ) and estimated ( $E$ ) probability of interference of <code>tera_tf</code> runs, as well as their difference ( $\delta$ ). Only runs 1-15 are shown. For runs 12 and 13, the different in probability of interference is low. . . . .                                                                                                                                                                                                                                                                                                                                        | 91 |



---

# 1 Introduction

---

Supercomputers performance has increased tremendously in the past few decades. In the Top500 list, which maintains a record of the worlds most powerful supercomputers, the maximum achieved computation performance has increased from 59.7 gigaflop/s in 1993<sup>1</sup> to 415.5 petaflop/s in 2020<sup>2</sup>. This sustained increase in performance can be partly attributed to Moore's Law, which famously predicted the number of transistors on a processor to double every two years [88]. Together with advancement in compiler technology and instruction-level parallelism, computers have seen significant increase in performance over the past few decades [45]. Supercomputers have also benefitted from this trend, experiencing an exponential growth rate of 1.9 until 2008 [122]. When single-thread performance started to stagnate, supercomputers growth rate also dropped, but to a lower degree (to 1.5), as it exploited multiple processing units per node, as shown in Figure 1.1.

## 1.1 Supercomputer architecture

Another significant factor in the growth of supercomputer performance is its distributed design. A typical supercomputer consists of hundreds of thousands of compute nodes connected through a communication interconnect, as shown in Figure 1.2. Applications span many compute nodes and exchange messages to synchronize and communicate. With such a distributed design, supercomputers have achieved higher performance by scaling to a large number of compute nodes. At the same time, while the design has remained distributed, the increasing scale has been a catalyst in evolving the architecture of a supercomputer.

### 1.1.1 Memory architecture

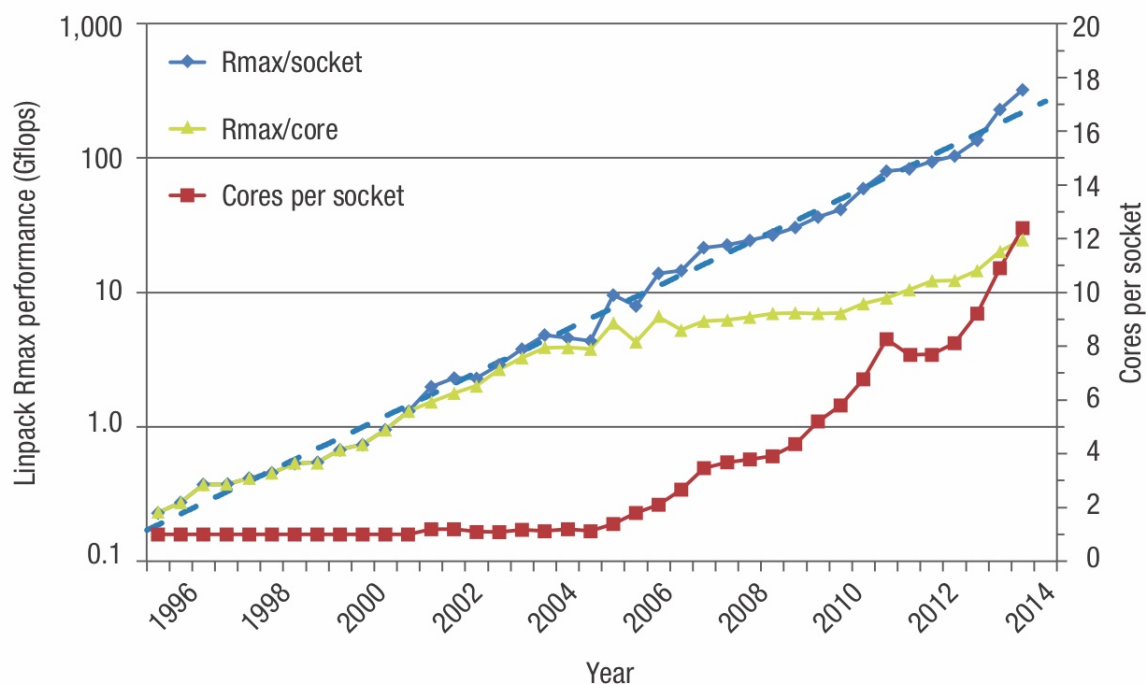
A supercomputer is a distributed system, consisting of up to hundreds of thousands of compute nodes. Each compute node has its own local memory, and processes executing on different

---

<sup>1</sup><https://top500.org/lists/top500/1993/06/>

<sup>2</sup><https://top500.org/lists/top500/2020/06/>





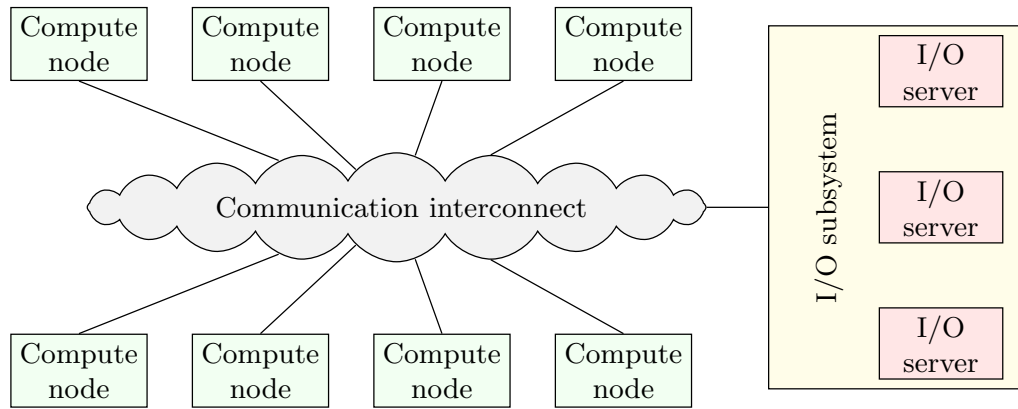
**Figure 1.1:** Increase in core count per socket has largely offset the stagnation in single-thread performance [122] © 2015 IEEE.

nodes communicate via the communication interconnect. A compute node itself consists of multiple sockets and processors. It has a NUMA architecture [45], that is each processor has its own directly attached cache and memory, as shown in Figure 1.3. A processor interconnect provides access to memory local to other processors on the node. Cache coherency protocol synchronizes the cached data.

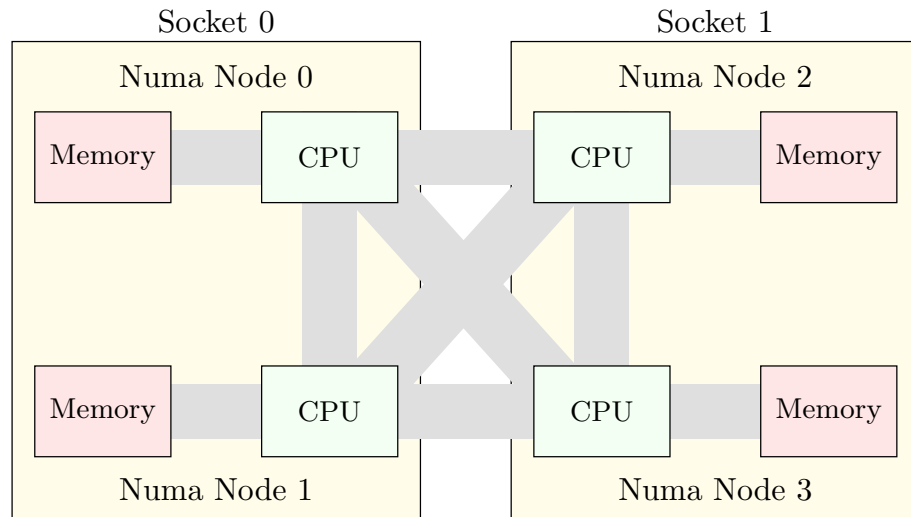
### 1.1.2 Communication interconnect

The communication interconnect binds all the components of a supercomputer into a coherent system. As the scale of the supercomputer has increased, so has the importance of the fabric connecting them. Applications are spanning increasingly higher number of compute nodes and have become sensitive to the performance of the communication interconnect. Coupled with financial considerations, supercomputers have been designed with innovative network architectures to achieve higher performance at lower cost. We describe three such architectures here.

A fat-tree is a hierarchical network topology [77], consisting of a tree like arrangement of switches, as shown in Figure 1.4. The *edge* switches are directly connected to the compute nodes, while each higher level of switches connect lower level of switches. The network can



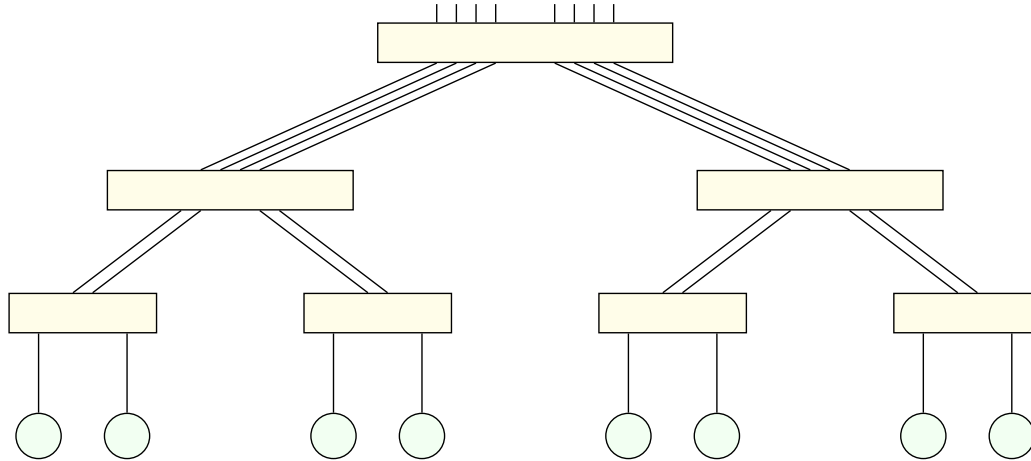
**Figure 1.2:** The architecture of a typical supercomputer.



**Figure 1.3:** The architecture of a compute node.

be setup in a *blocking* or *non-blocking* configuration. Switches in a non-blocking configuration have the same number of uplinks as downlinks, that is the number of network links coming from the lower level is equal to the number of links going to the higher level. In this configuration, every pair of compute nodes can communicate without blocking as there exists a unique network route between them. On the other hand, a blocking network has less uplinks than downlinks, meaning pairs of communicating nodes may share network links. When messages from a compute node are routed through a link under use, the messages are queued and are temporarily blocked until the link is free.

A torus network is another common network topology [27]. In the topology, there are no dedicated network switches, rather the nodes are directly connected to each other, as shown in Figure 1.5. The nodes at the edges are also connected to each other via a loop-back link, forming a torus, shown by dashed lines in the figure. The network can be configured as a



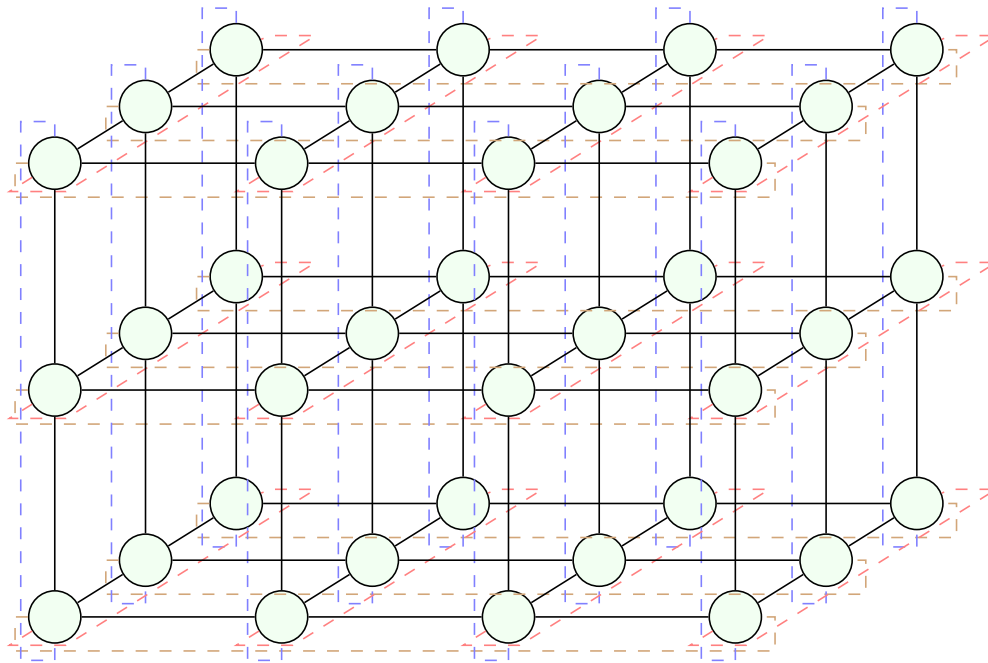
**Figure 1.4:** A non-blocking fat-tree topology.

three-dimensional [10], five-dimensional [21], and even six-dimensional [5] interconnect. Such a network architecture provides high bandwidth at low latency among neighboring compute nodes, and allows for scalability at lower cost.

Supercomputers that have a large number of compute nodes also commonly deploy a dragonfly network [57, 67]. It is a two-level hierarchical topology, resembling a highly connected network at each level, as shown in Figure 1.6. The lower level consists of groups of compute nodes that are connected using high-radix routers. The high-radix routers in a group also use a few ports to connect to routers in other groups, forming the higher level topology. When communicating, compute nodes may share network routes. There may also exist multiple routes among them and the network uses adaptive routing to avoid hotspots. The network performance of a dragonfly network depends upon how processes are mapped to nodes, the routing strategy, and the network traffic from other applications, causing application performance variation [37, 134, 137]. However it is deployed on some of the largest supercomputers as it also provides high global bandwidth as well as short routes among compute nodes.

### 1.1.3 I/O subsystem

Applications running on a supercomputer read and write a large amount of data [120]. For example, VPIC (a plasma physics code), when executed on 120,000 cores generates 30TB of data [15]. With increasing scale of applications, this trend is also increasing [76], with flagship projects predicting datasets of 100 petabytes or more [79]. To accommodate the high demand, supercomputers typically employ a central, parallel file system called the I/O subsystem. It is a shared resource, connected to all compute nodes through the communication interconnect.



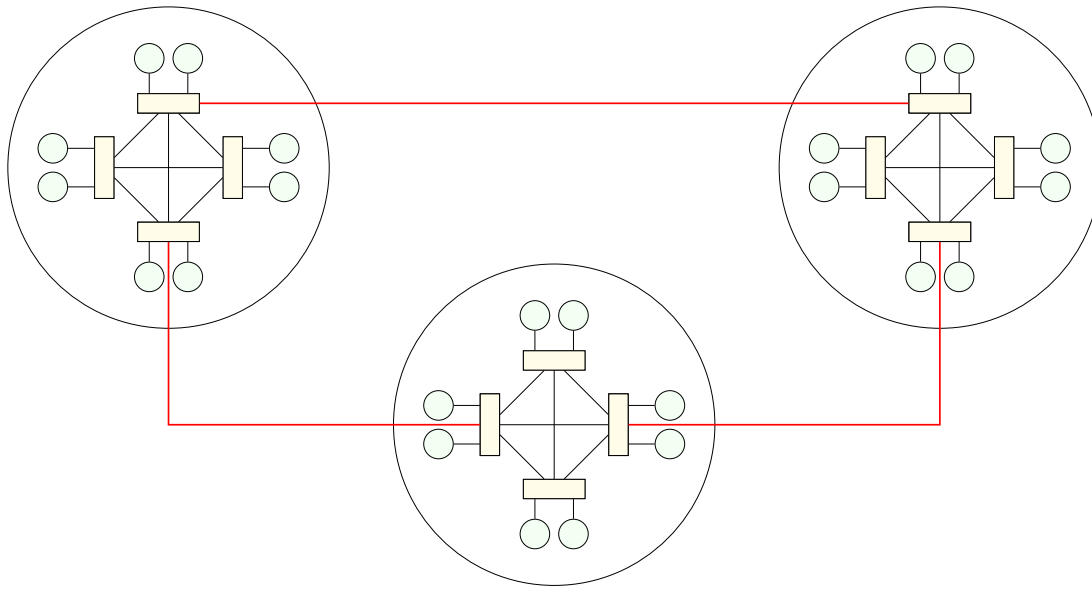
**Figure 1.5:** A three-dimensional torus network with a  $(4 \times 3 \times 3)$  network topology. The loop back links are shown as dashed lines.

When an application running on the supercomputer manipulates data, the I/O request is sent from the compute node to the I/O subsystem. Figure 1.7 shows the architecture of a typical I/O subsystem.

The I/O subsystem consists of multiple servers. Each server houses several RAID devices to store the data. On some supercomputers, the I/O subsystem also has dedicated nodes. In such a setup, a file access request from the compute nodes is routed to the I/O nodes, which relay them further to the I/O servers. On some modern systems, an I/O forwarding layer, called burst buffer, is used between I/O nodes and RAID storage devices [9, 78]. This layer employs NVMe drives with high read and write bandwidth, and acts as a cache. The layer can absorb bursts of write activities from applications and write them to the RAID devices as a continuous lower intensity stream. At the same time, it can also serve the data back to the application when servicing read operations. The I/O subsystem is further described in Chapter 3.

## 1.2 Variation in application performance

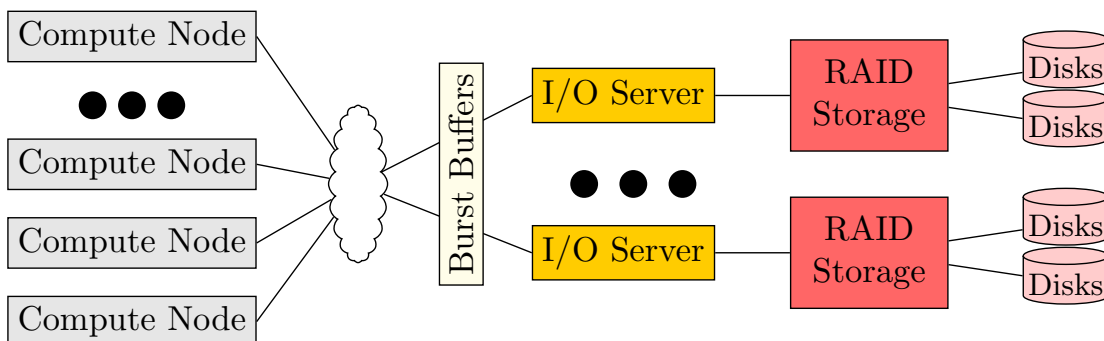
A supercomputer is a multi-tenant system, hosting multiple applications at the same time. The applications themselves consist of multiple processes, ranging from a few tens to hundreds of



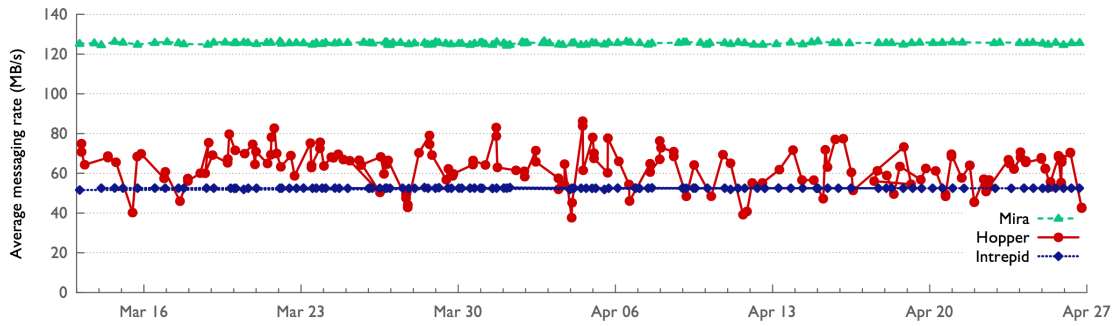
**Figure 1.6:** A dragonfly network topology consisting of three groups. Each group has eight nodes connected via four routers.

thousands in number. If such an application is executed multiple times, its execution time can vary. On some systems, such as the IBM Blue Gene systems, the variation is small, while on other systems, such as the Cray XE systems, the variation can be significant [8].

Figure 1.8 shows the average messaging rate in MB/s for a laser-plasma interaction code on three different systems over a period of three months [8]. The rate can be used to gauge the variation in execution time of the code. When the application takes longer to execute, the average messaging rate decreases, and vice versa. In the figure, Mira and Interpid (IBM Blue Gene systems) have a consistent average messaging rate, indicating low run-to-run variation. On the other hand, Hopper (Cray XE6) shows high variation in average messaging rate,



**Figure 1.7:** Architecture of a typical I/O subsystem.



**Figure 1.8:** Average message passing rate for a laser-plasma interaction application on IBM Blue Gene/Q (Mira), Cray XE6 (Hopper) and IBM Blue Gene/P (Interpid) systems [8]  
© 2013 IEEE.

indicating significant run-to-run variation. Similarly, numerous other studies have identified execution time variation on supercomputers [6, 7, 73, 87, 137, 138].

The execution time variation can be attributed to several factors. Of them, three major factors are the following:

### 1.2.1 Operating system jitter

Typically, on supercomputers, an application exclusively occupies the compute nodes. It is possible that this trend might change in the future, especially in the case of nodes that have accelerators attached to them, where a compute node might be shared between an accelerator-utilizing application and a none-accelerator-utilizing application. Nonetheless, the current established approach is to assign compute nodes to a single application exclusively. Even in such a configuration, the compute node resources are still shared between the application and the operating system. The operating system mostly remains idle but whenever it executes some daemons or utilities, or performs scheduling activities, it utilizes the compute node resources, degrading the performance of the application running on the compute node. This effect is called operating system jitter. Historically, operating system jitter has been a major source of execution-time variation, causing applications to slow down by as much as 50% [104]. Several studies have therefore investigated operating system jitter and have identified its sources [28, 129, 132], its impact on communication [6, 50], on multicore systems [111, 112], and at extreme-scale [7]. However, recent work has shown that modern operating systems have reduced their noise footprint significantly [87].

---

## 1.2.2 Process-to-node mapping

An application running on a supercomputer spans multiple processes. When the application executes, its processes are mapped to the compute nodes of the supercomputer. Depending upon the network architecture, the process-to-node mapping can have an impact on application execution time. For example, in fat-trees, if application processes are mapped to compute nodes that are attached to the same edge switch, communication among them will have low latency. On the other hand, if they are mapped to compute nodes attached to different edge switches, communication messages will get routed through multiple switches, adding latency and causing variation in performance. The variation can be especially significant in blocking fat-trees. Similarly, on dragonfly networks, communication among compute nodes belonging to the same group has lower latency compared to communication among nodes residing in different groups.

On torus networks, the process-to-node mapping can have a significant performance impact. As there are no dedicated network switches, messages from a node hop over intermediate nodes to reach their destination. If the distance between the communicating nodes is large, messages have to hop through many nodes, adding latency and variation. Some systems, like Blue Gene systems, address this issue by always allocating an exclusive compact network section to an application. Other systems, like Cray systems, allow application processes to scatter over the network, increasing the possibility of processes being mapped far apart.

## 1.2.3 Inter-application interference

The process-to-node mapping is not only a source of execution-time variation itself, but it also exposes an application to inter-application interference. The communication fabric is a shared resource. When multiple applications try to access it simultaneously, they create contention, causing inter-application interference and variation in application performance. The higher the number of shared network resources between communicating nodes, the higher the likelihood of interference. Recent studies have found inter-application interference to cause significant variation in application performance. Hopper (a Cray XE machine), whose performance data is listed in Figure 1.8, uses a three-dimensional torus network. The high variation in performance, as seen in the figure, is primarily caused by simultaneously running applications [8]. Similarly on dragonfly networks, applications share network routes and are susceptible to interference. Several studies have found inter-application interference to be significant in dragonfly networks. The degree of interference is affected by application communication patterns [134], process-to-node mapping [137], and adaptive routing algorithms [57].

---

Another shared resource on a supercomputer is the I/O subsystem, as described in Section 1.1.3. When multiple applications access the I/O subsystem simultaneously, they cause contention on resources and degrade performance, in some cases by more than a factor of 4 [119]. At the same time, studying I/O interference and identifying sources of variation is challenging [130]. Nonetheless, as I/O interference can cause significant interference, it has been a subject of several studies. They have analyzed I/O performance at a system level [17, 18], compared it across systems [82], identified characteristics that effect the magnitude of interference [73, 138], and proposed techniques to reduce the interference [33].

## 1.3 Motivation

We have shown that execution time variation on supercomputers can be significant and that it has three major sources. With increasing size of HPC systems and applications, inter-application interference has become a significant source of performance variation [87]. At the same time, there are many aspects that are still still unexplored. Especially the seemingly random manner in which it degrades the performance of an application, leading to several challenges.

### 1.3.1 Estimating application execution time

Estimating application execution time under the external influence is error prone. As the intensity of interference cannot be predicted, the resulting unpredictability of execution time can cause challenges when scheduling applications.

HPC systems employ a batch system to schedule and execute applications. Expected execution time is one of the parameters required to schedule a job. When the job exceeds the time, it is aborted. As inter-application interference can randomly extend the execution time of applications, users tend to provide a higher than expected execution time when submitting jobs. However, this practice misleads the job scheduler into making inefficient decisions, that can potentially lower system utilization. Similarly, several situations arise when users want to benchmark their application. Run-to-run variation means that users have to execute their application multiple times to have confidence in their measurements, which consumes a lot of precious core hours.



---

### 1.3.2 Analyzing application performance

The situation becomes more complicated when measuring and analyzing the performance of an application. Reliable analysis requires a performance measurement as close as possible to the *intrinsic* behavior of an application, that is the natural behavior of the application which is clean from outside influences. In the presence of seemingly random external interference, a performance analyst has no reliable method of gauging the impact of outside effects on a measurement. A common approach would then be for the analyst to take several measurements and then pick the fastest or the mean or median run [49]. Even then, the analyst cannot be sure as all the measurements can be interfered.

### 1.3.3 Wasting system resources

On the other hand, degraded application performance means wasting system resources. Applications running on a supercomputer utilize shared resources like the communication interconnect and the I/O subsystem on demand, yet they occupy the compute resources throughout their execution. When an application is interfered, its execution time increases, causing it to occupy compute resources for longer than needed, wasting system resources. Especially applications with high process count can significantly waste resources if they are exposed to interference.

## 1.4 Thesis contributions

Based on the mentioned challenges, identifying and understanding effects of inter-application interference has become an important topic. At the same time, it is a vast area of study. There are many complex sources of interference and significantly analyzing their effects is challenging. This dissertation studies certain aspects of the problem and contributes the following:

### 1.4.1 Identifying and evaluating inter-application interference

The first contribution is a method to identify inter-application interference among concurrently running HPC applications [115], which commonly arises when applications contend for access to shared resources such as the file system or the network. The method divides the application runtime into fine-grained time slices whose boundaries are synchronized across the entire system. Mapping performance data related to shared resources onto these time slices, it is possible to establish the simultaneity of their usage across jobs, which can be indicative of

---

inter-application interference. Experiments show that such interference effects, for which the developer is usually not to blame, can degrade application performance significantly. This dissertation further uses the method to rate the interference potential of common file-write patterns [73, 113]. A particular challenge that arises on high-dimensional torus networks is to give users a legible visual feedback for identifying contention and interference. We address this challenge by presenting a visualization technique that projects the torus networks onto a 2D display using simultaneous views [128].

### **1.4.2 Quantifying inter-application interference**

The second contribution is an approach to estimate the impact of sporadic and high-impact external interference on the runtime of bulk-synchronous MPI applications [114]. The approach identifies independent execution segments in an application run and establishes similarity by clustering them using their computation, communication, and I/O characteristics. Outliers in these clusters are segments exposed to external interference. An evaluation with several realistic benchmarks shows that the approach can estimate the impact of external interference already based on a single run.

## **1.5 Thesis structure**

This dissertation is structured as follows. Chapter 2 introduces our first contribution of a method to identifying inter-application interference. Chapter 3 then uses the method to study in depth the effect of file access pattern under concurrent I/O, identifying an interference trend between simultaneously writing small and large files. Chapter 4 briefly describes our method to visualize high-dimensional torus networks. Chapter 5 presents our second contribution of estimating the impact of interference on an application runtime. We survey related work in Chapter 6. The dissertation is concluded with a summary and outlook in Chapter 7.

## **1.6 Statement of Originality**

The work presented in this dissertation has been conducted under the supervision of Prof. Dr. Felix Wolf, head of the Parallel Programming chair, Department of Computer Science, TU Darmstadt. In its later stages, it has also been supported by Prof. Dr. Matthias Müller, Director of IT Center, RWTH Aachen. Furthermore, this dissertation has profited from the master thesis of Chih-Song Kuo conducted under my supervision [72]. The master thesis work contributed

---


to two publications [73] (where Chih-Song Kuo is the first author) and [113] (where I am the first author). This dissertation has also benefitted from the bachelor thesis of Lucas Theisen [127] carried out under my supervision, which resulted in one publication [128]. In the following, I describe the source of the material of each chapter.

Chapter 2 is based on my publication **Capturing inter-application interference on clusters** [115]. Under the supervision of Prof. Wolf, I was the primary contributor of the paper. The paper was a joint contribution with Prof. Voevodin’s group at Moscow State University. They provided a prototypical deployment of our LWM<sup>2</sup> tool on their GraphIt system, which is described in Section 2.1.3. Among others, Section 2.1 and 2.2 are quoted verbatim from the paper.

Chapter 3 is based on my journal publication **How File-access Patterns Influence the Degree of I/O Interference between Cluster Applications** [113]. I am the first author of this paper. It is an extension of the conference paper **How file access patterns influence interference among cluster applications** [73], which was derived from the master thesis work of Chih-Song Kuo [72] carried out under my supervision. He is the first author of the conference paper. The master thesis was a joint collaboration with Prof. Matsuoka’s group at Tokyo Institute of Technology. Among others, Section 3.3.4 is quoted verbatim from the paper **How File-access Patterns Influence the Degree of I/O Interference between Cluster Applications** [113]. With regards to research contribution, the work was continuation of our method to identify inter-application interference, presented in Chapter 2. I proposed the idea of evaluating the influence of file access patterns on I/O interference. I also developed the micro-benchmark used in the study and gathered initial data to support the claim. Chih-Song Kuo picked up from this point and investigated the different scenarios described in the chapter. He also was the main collaborator with Prof. Matsuoka’s group, who were indispensable for this study. They provided the computing resources to carry out all our experiments on the TSUMABE2.0 computer, and also helped Chih-Song in monitoring the backend servers of the I/O subsystem. Section 3.5 describes my contributions in detail.

Chapter 4 is based on my co-authored paper **Down to Earth – How to Visualize Traffic on High-dimensional Torus Networks** [128]. The work presented in the paper is described in the bachelor thesis of Lucas Theisen conducted under my supervision [127]. The paper presents two approaches to visualization. I only discuss the elements of the first approach in the thesis and reuse its figures, as I proposed the idea and which Lucas later implemented. The second approach was contributed by Lucas and is not discussed in this dissertation. Please refer to Section 4.5 for details.

Chapter 5 is based on my publication **Estimating the impact of external interference on application performance** [114]. Under the supervision of Prof. Wolf and support of Prof. Müller, I was the primary contributor of the paper. Among others, large portions of Section 5.1



---

and 5.2 are quoted verbatim from the paper.



---

## 2 Identifying inter-application interference

---

Supercomputers are typically setup as a cluster of computing devices, connected via a communication interconnect. The complete system is very rarely occupied by a single application, rather multiple applications execute on it simultaneously. A scheduler and a batch system is used to submit applications to the supercomputer. When the resources required by an application becomes available, and other scheduling constraints are met, an application is assigned compute nodes and starts executing on them. This way, a HPC cluster is shared, both in time and space.

It is common these days to assign compute nodes exclusively to an application. From the start of its execution till its end, compute nodes resources are occupied by the application. While other resources, such as the communication interconnect, which binds the compute nodes together, and the I/O subsystem, which handles the file access requests of applications, are utilized on demand. As these resources are shared among the running application, simultaneous access on these resources leads to contention, degrading performance of applications and causing inter-application interference.

In spite of this situation, users are exposed to only the behavior of their executed applications. In such a single-application view, the only way to consider interference from outside sources is to take multiple measurements and then quantify variation. Even in this approach, the nature of interference remains opaque. An intuitive way to account for external effects is to observe the behavior of all running applications by profiling them, and correlate their performance. While techniques exists to silently create profiles of all executing jobs on an HPC cluster [38, 69], the indicators needed to establish simultaneity of events, a prerequisite to any correlation analysis, are still missing.

This chapter presents a new performance-analysis approach that can be used to identify inter-application interference. As in earlier techniques, every job running on the system is profiled. However, the novelty of the approach is to divide the running of the system into small time slices for which application performance data is recorded separately. The slices

---

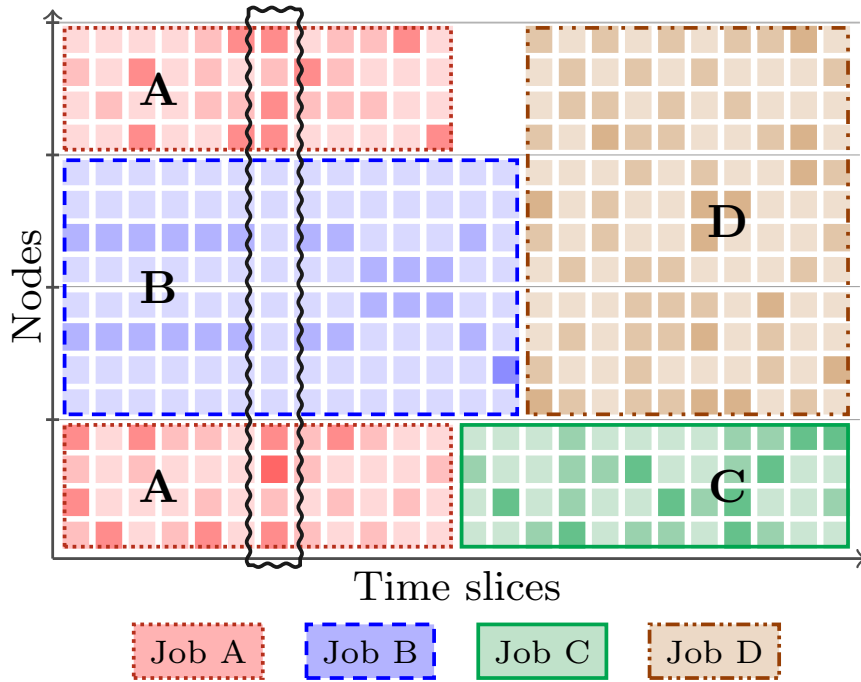
are thin enough so that even smaller performance events such as sudden peaks of file I/O in one application can be correlated with temporary performance degradation observed for another. The slice boundaries are synchronized across the whole system to ensure that the slices of different jobs running at the same time can be precisely mapped onto each other. Using the method, we evaluate different shared resources with respect to their interference potential and demonstrate the serious influence this phenomenon can have.

The rest of this chapter describes our method, evaluates the interference potential of different shared resources, and the influence this phenomenon can have.

## 2.1 Approach

To capture application interference, we model the operation of a cluster system as a time-space grid, as illustrated in Figure 2.1. The space dimension represents the hardware, which is divided into a set of disjoint nodes. Each node may run one or more processes. The time dimension represents the runtime of the system, which is divided into a set of disjoint time slices. Discrete time slices are needed to establish an approximate notion of simultaneity among performance phenomena. Assuming static allocation of hardware resources, each job on the system occupies one or more rectangular areas that all start at the same time and end at the same time. Performance data is collected at the granularity of processes and time slices. Thus, for each time slice and process, we record a vector of performance metrics, resulting in as many metric vectors per node as there are processes on the node during the time slice.

Now we can correlate performance data in two different ways: First, we can check whether we can see differences between different nodes that remain stable across an extended period of time. Although outside the scope of this dissertation, this information can be used to detect hardware anomalies that do not constitute outright failure but lead to some form of performance degradation applications cannot be blamed for. Examples include network links whose throughput is reduced due to cabling problems, a node throttled due to some error condition, or a smaller amount of per-node memory. Second, and this is central to our approach, we can compare the performance of an application during a given time slice with the performance background, that is, the performance of applications running at the same time on relevant parts of the system. Since the possibility of interference depends on the topological characteristics of the system, which may restrict the sharing of resources to certain zones such as the same subtree of a fat-tree network, we also precisely record the node each process is running on.



**Figure 2.1:** Operation of a cluster system as a time-space grid. The picture shows four nodes, each running four processes (denoted by colored squares) at a time. During the interval represented by the x-axis the system runs four jobs (A-D). The brightness of each square indicates the intensity of a performance metric. Synchronized time-slice boundaries allow the correlation of performance phenomena across jobs. In the highlighted time slice, high intensity in job A coincides with low intensity in job B.

### 2.1.1 Requirements

To implement the above approach, we need to collect performance data from every job running on the system. For this purpose, we implemented a system-wide profiler called *LWM*<sup>2</sup> (lightweight monitoring module) that is supposed to be dynamically linked to each application at job start—with the option of being disabled if needed. Of course, such mandatory performance screening has multiple benefits, including the early detection of application performance bottlenecks and guidance in choosing more advanced tools to conduct more sophisticated diagnostics. While our design takes such aspects into account, the primary focus of the narrative remains the detection of application interference. Our monitoring module satisfies the following requirements:

1. Collection of basic performance metrics. To be able to exploit topological information and to avoid communication and synchronization at larger scale, metrics have to be collected separately for each process.
2. No modification of the executable and minimum assumptions about its runtime system



---

and its parallel programming model(s)

3. Negligible overhead in terms of both time and space. We define this as  $\leq 1\%$  runtime penalty and  $\leq 5$  MB of required buffer space per process and day
4. Attribution of performance metrics related to the usage of shared resources to globally synchronized time slices

Whereas the first three requirements are common to silent profilers in general, the last one is new and challenging to fulfill without incurring too much overhead.

### 2.1.2 Design

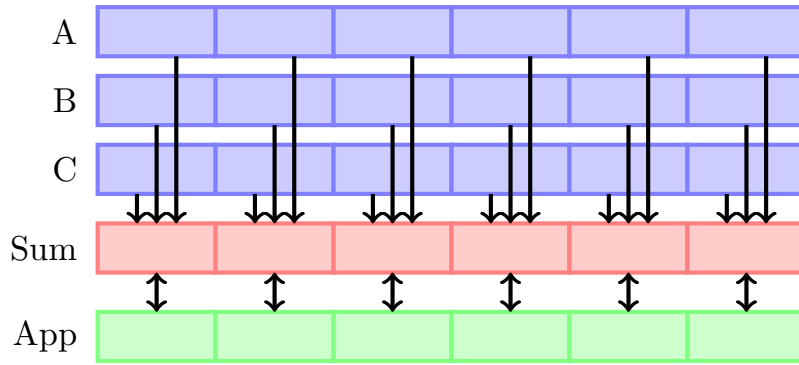
The monitoring module features a modular architecture for customization and further extendibility, offering compile time customization of profiling capabilities with support for further configuration of the compiled capability through environment variables. Capabilities include profiling of MPI applications, pthreads-based multithreaded applications, and CUDA applications through CUPTI [99], along with the interception of POSIX file I/O calls. Finally, sequential performance is measured through hardware counters using PAPI [91].

#### Instrumentation

To keep runtime dilation low, LWM<sup>2</sup> combines direct instrumentation via interposition wrappers with sampling. Interposition wrappers enclose all calls to MPI and POSIX file I/O functions while CUPTI callback functions are used for CUDA calls. Different from prior hybrid profiling approaches [123], the direct instrumentation is only used to count function calls and to intercept data-transfer parameters such as the number and size of messages, the data volumes read from or written to disk, or the amount of data transferred between hosts and accelerators. We explicitly refrain from direct time measurements in wrappers, as their cost can accumulate in application that frequently call any of the wrapped functions. Instead we earmark the wrapped routine currently being executed so that we can correctly attribute samples to them without having to examine the stack. Hardware counters are recorded when crossing time-slice boundaries.

#### Time slices

In addition to creating a job digest at the end that summarizes the whole execution in terms of performance indicators such as the time spent in MPI calls and that provides guidance in choosing more sophisticated diagnostic tools for further analyses, LWM<sup>2</sup> creates separate



**Figure 2.2:** Comparison of applications performance with the performance background. The performance background of an application is the sum of the metric values collected for all applications (A, B, C) running at the same time.

profiles for each time slice, which allow insights into the changing performance dynamics of the application.

As illustrated in Figure 2.1, the time slices are synchronized across the system so that it is possible to compare and aggregate performance data both across all processes of a parallel job and across all jobs running on the system at a given time. In essence, time slices offer a way to establish the simultaneity of performance incidents such as file-system access storms. While we left the length of the slices configurable, we chose a length of 4 s for our experiments. This is small enough to capture performance dynamics with reasonable granularity but still large enough in relation to the precision at which the system time is usually synchronized (in the order of 1 ms without global clock). As we explain below, we slice only the data collected in wrappers, which is why the sampling frequency does not have to be taken into account when defining the time slice length.

With performance data collected at the granularity of time slices, a user can now compare the performance dynamics of their own application with the performance background and identify correlations, as shown in Figure 2.2. When observing a performance drop for a specific interval, the user would search for peaks in the background that occurred during the same interval. The metric where this peak appears is then indicative of the source of interference. For example, an application experiences a slowdown during a specific time slice. If the performance background shows increased file I/O activity during this time slice then I/O interference emerges as a possible reason to be further investigated.

---

## Multithreaded applications

While the collection of time-sliced performance data is trivial for single-threaded applications, it is non-trivial for multithreaded ones. Specific challenges arise when trying to reconcile the desire to keep the memory footprint small with the requirements of thread safety and low runtime dilation. Nevertheless, when monitoring applications system wide multithreaded program cannot be ignored nor can they be treated like single-threaded ones. To make our solution most portable across a large variety of applications, we assume pthreads as the underlying threading library. With pthreads constituting the foundation of many higher-level threading libraries, we believe this choice not to be overly restrictive.

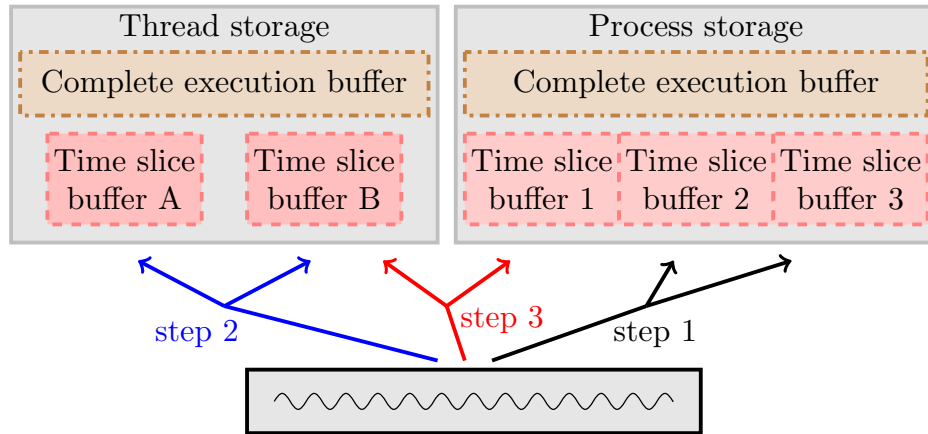
To record time-sliced profiles, LWM<sup>2</sup> requires linearly increasing buffer space. To keep this as small as possible even in view of long-running applications, we refrain from storing performance metrics individually for each thread over an extended period of time. Instead we aggregate the values collected by the threads of a process in their thread-local storage *in situ* at the end of each time slice. In this way, our memory footprint remains independent of the number of threads, as we have to store only one value per metric, process, and time slice. As a result, the total storage required for time-sliced profiles for a 24 hour execution is less than 4MB per process, for a 4 sec long time slice.

To efficiently aggregate data, each application thread maintains two thread-local buffers, one for even and one for odd time slices. A flag is used to indicate the active buffer. During a time slice, threads write to the active buffer. When a time slice ends, the flag is switched and the data in the inactive buffer is aggregated.

However, this is easier said than done. The flag to indicate active buffer requires a read-write lock. If the end of a time slice is signaled via a timer interrupt, the lock will have to be acquired in an interrupt handler, which is prohibited [100]. We therefore cannot use a timer interrupt to signal the end of a time slice.

**Heartbeat thread.** To solve this dilemma, we employ a so-called *heartbeat thread*, an auxiliary thread whose task is to collect the data from all application threads of the process whenever a time slice expires. The heartbeat thread is created during the initialization of LWM<sup>2</sup> and is only activated at time slice boundaries. It sleeps in between. While the heartbeat thread sleeps, each application thread updates its local metrics in the currently active buffer. When a time slice ends, the heartbeat thread wakes up and switches the active buffer of all threads. It then aggregates the values in the inactive buffers.

Let us assume the number of the current time slice is even. When the time slice ends, the heartbeat thread wakes up and allocates the process-wide summary buffer needed for the next time slice. It also turns the flag to odd after acquiring the write lock associated with it.



**Figure 2.3:** The steps taken by the heartbeat thread at the time slice boundary.

This is summarized in Figure 2.3. Only during this very short moment are application threads prevented from updating their local buffers. After the flag has been turned, the application threads use the odd buffer while the heartbeat thread aggregates the values of the previous (i.e., even) iteration into the process-wide time-slice buffer.

Since writing to the thread-local time-slice buffer requires the prior acquisition of a read lock to prevent a sudden update of the even-odd flag, the buffer cannot be accessed from interrupt handlers, which do not allow the acquisition of locks. Therefore, we only slice the information collected in wrappers via direct instrumentation, that is, the frequency and amount of data transfers. However, since interference is usually caused by data transfers to and from shared resources, this is enough to cover the most interesting cases of application interference. Plus, the statistical accuracy of sampling within the relatively narrow time-slice boundaries might be limited anyway. In addition to collecting data from each thread, the heartbeat thread also reads hardware counters on a per-process basis once the time-slice boundary has been reached.

**Thread-level concurrency.** As a silent profiler, LWM<sup>2</sup> was designed to capture not only interference between applications but also performance issues within applications, which, of course, includes limited thread-level concurrency. Following our philosophy that all our performance metrics have to refer to the process as a whole instead of individual threads, we calculate the *effective thread count* of a process. Whenever LWM<sup>2</sup> takes a sample, each active thread is hit by an interrupt. Thus, the total number of samples is larger than what would be expected for serial code. The effective thread count is thus defined as the ratio between these two numbers and provides a good indicator of thread-level concurrency, for example, when compared to concurrency goals as specified in `OMP_NUM_THREADS`. Since it relies on sampling, the metric is calculated only for the execution as a whole, where we believe it is

---

also most useful.

### 2.1.3 Deployment

LWM<sup>2</sup> is designed to collect performance data of each application running on a system. However collecting performance data also requires helper services acting alongside LWM<sup>2</sup>. In a typical deployment, one of these helper services will be a database agent which reads the application profiles LWM<sup>2</sup> generates and stores them in a database serving as the central performance profile repository. A Web front-end would give end users access to the job digest and changing dynamics of their application, as captured in time slices, and—for comparison—to the performance background of the system at the time their application was running.

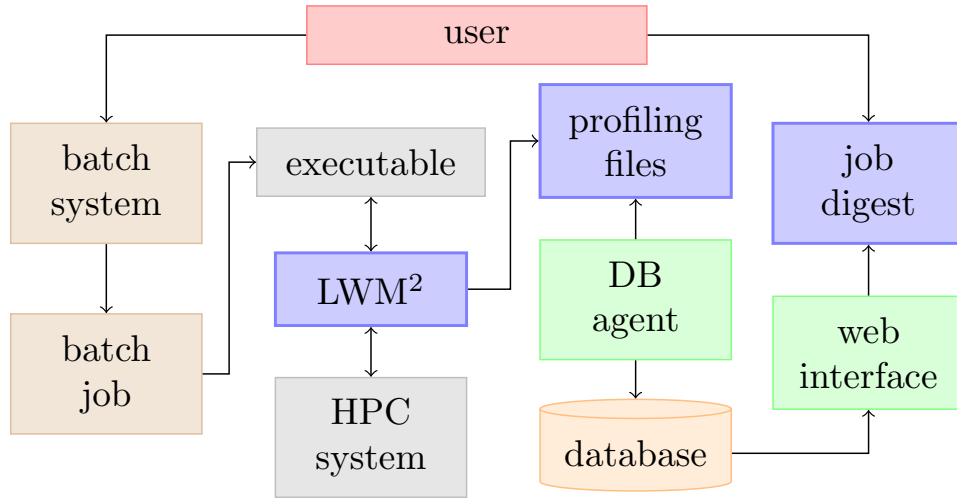
LWM<sup>2</sup> was deployed in a similar setup on the GraphIt system [55] at Moscow State University. GraphIT was 16 node HPC cluster, with each node consisting of two 6-core Intel Xeon X5650 CPUs and three NVidia “Fermi” Tesla M2050 GPUs. The nodes were connected through QDR Infiniband 4x interconnect. GraphIt has been decommissioned.

Figure 2.4 shows the setup on the GraphIt system. In the default batch system configuration, LWM<sup>2</sup> automatically profiled every executing application. When the application finished, LWM<sup>2</sup> wrote a job digest with performance information to the terminal. The information included:

1. The MPI collective call rate and bytes communicated
2. The number of point-to-point calls and bytes sent/received
3. The number of MPI and POSIX I/O operations, and bytes read/written
4. The number of floating points operations per second (when available)
5. The effective number of active thread count

For all the metrics, minimum, average, and maximum values were calculated and written in the job digest.

LWM<sup>2</sup> also wrote the full set of metrics to a predefined destination. The database agent monitored the location and pushed any new profiles to the database. Users could then access the profiling information through a web interface.



**Figure 2.4:** Depicting the functionality of the integrated setup deployed on the GraphIT system at MSU. Every batch job would get screened by LWM<sup>2</sup>. The resulting profile files were pushed into a database, which was accessed by a web frontend to generate a job digest for users.

## 2.2 Evaluation

Our evaluation pursues several objectives: First, we confirm the low runtime overhead of LWM<sup>2</sup>. Second, we demonstrate how it can be used to identify application interference. At the same time, we analyze the interference potential of typical shared resources.

All our tests were performed on two machines at the Jülich Supercomputing Centre: JUROPA was a Linux cluster consisting of 2208 compute nodes, each equipped with two Intel Xeon X5570 (Nehalem-EP) quad-core processors running at 2.93 GHz [63]. The nodes were connected through an Infiniband QDR network with non-blocking fat-tree topology. All our file I/O was performed on a Lustre file system with four meta-data servers (MDS) of type Bull NovaScale R423-E2 (two Nehalem-EP quad-core & two Westmere-EP, 6-core) and eight object storage (OST) servers of type Bull NovaScale R423-E2 (Westmere-EP, 6-core) attached to the same Infiniband network. JUROPA was decommissioned and dismantled on June 24, 2016. JUDGE was a Linux GPU cluster. The node we used in this study featured two Intel Xeon X5650 (Westmere) 6-core processor running at 2.66 GHz plus two Nvidia Tesla M2050 Fermi GPUs [61]. JUDGE was decommissioned on November 30, 2015.

---

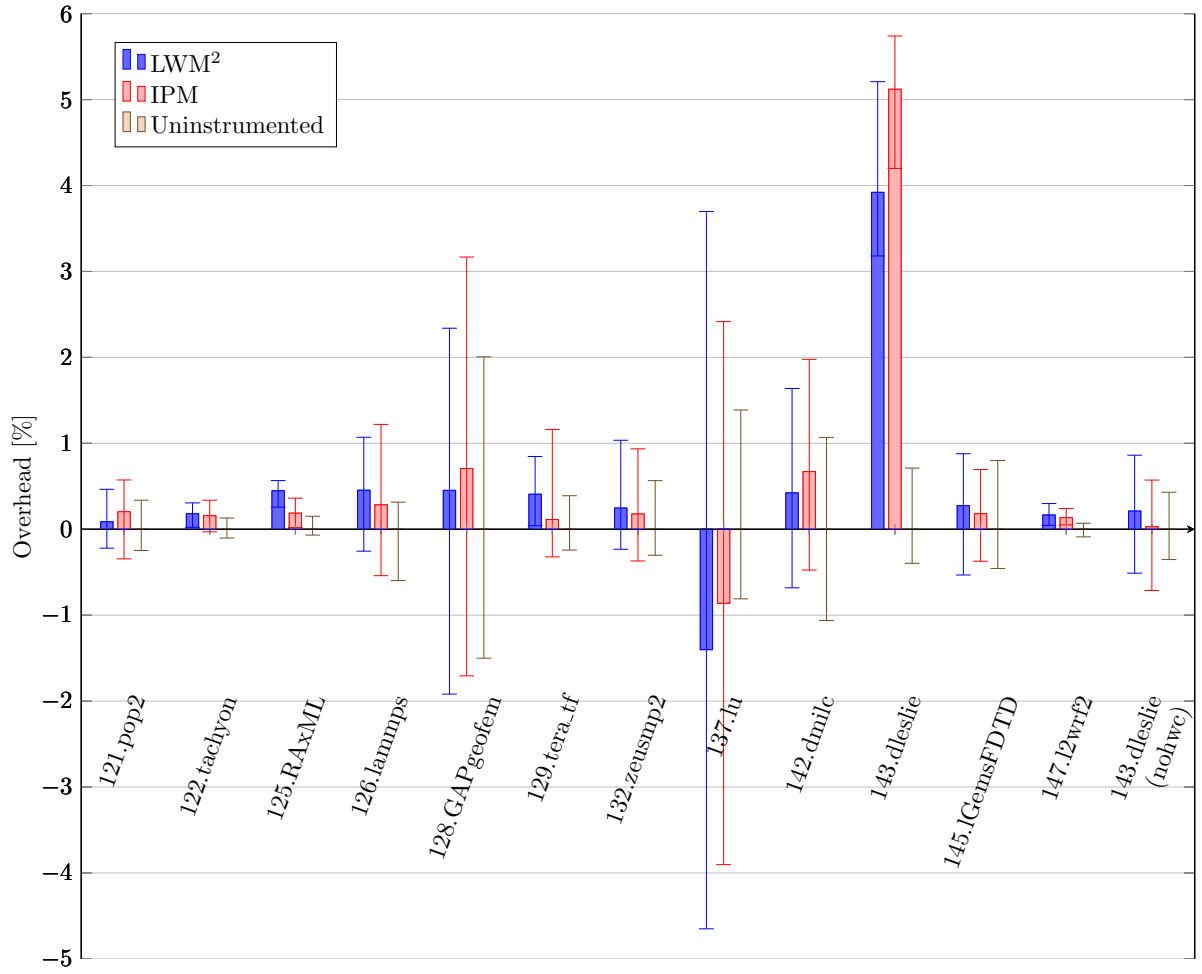
## 2.2.1 Overhead

Since silent profilers such as LWM<sup>2</sup> are supposed to be operated in a compulsory mode, requiring at least an explicit opt-out to be disabled, low overhead is essential for user acceptance. The overhead of a profiler falls in to categories—space and time. We collect currently 22 metric values á 8 bytes per time slice (i.e., every 4 s) and process, resulting in a memory footprint of less than 4 MB for 24 hours of execution. The runtime dilation was verified using applications from two benchmark suites: SPEC MPI2007 [90] for single-threaded MPI programs and the NAS Parallel Benchmark [131] for hybrid MPI/OpenMP programs. The profiling overhead was compared with IPM [38], a silent profiling module already used in production.

Measuring small overhead on a production system exposed to run-to-run variation is extremely challenging. To minimize the effects of this variation, we took a number of steps. Since application start up is usually very sensitive to variation, we compared only the times between initialization and finalization of MPI. We ran each benchmark 90 times, 30 times without profiling, 30 times with IPM, and 30 times with LWM<sup>2</sup>. To even out system load variations, we alternated between the three types of runs in a round-robin fashion. We removed outliers using the modified Z-score method [54] such that we can locate the mean value in the range of the runs with a confidence of 99%. The filtered set was used to calculate the mean, maximum, and minimum values.

### Single-threaded MPI

For our experiments, we ran all benchmarks from the SPEC MPI2007 suite for which a large reference data set was available on 32 nodes of JUROPA with 256 processes. The results, which quantify the overheads of LWM<sup>2</sup> in comparison with IPM, are shown in Figure 2.5. In all but one case, the overhead of LWM<sup>2</sup> is less than 1 %. The negative overhead of 137.lu is surprising, although not totally outside our earlier experiences with overhead measurements. Only 143.dleslie exhibits anomalously high overhead. In both cases, IPM mirrors the behavior of LWM<sup>2</sup>, which suggests that the source of anomaly lies outside the two profiling tools. Further investigation of 143.dleslie revealed the cause to be the dynamic loading of the shared PAPI library, a behavior observed only for this specific Fortran code and Linux kernel version. The last run in the figure shows the overhead for 143.dleslie without PAPI, which was less than 1%.



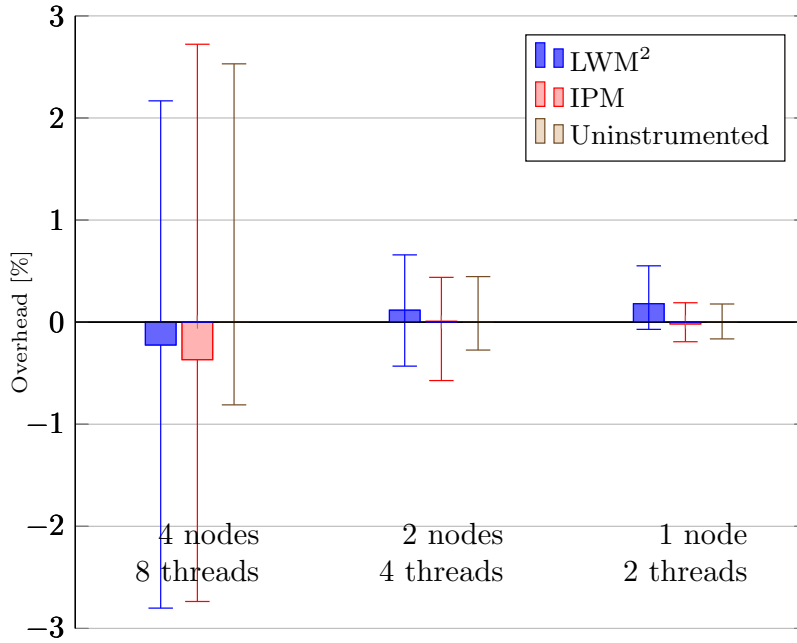
**Figure 2.5:** Runtime overhead of LWM<sup>2</sup> vs. uninstrumented execution for single-threaded MPI (SPEC MPI2007), also in comparison to IPM. The high and low marks indicate the spread after the removal of outliers.

## Hybrid MPI

We used the BT-MZ benchmark, which uses OpenMP to exploit shared-memory parallelism, from the NAS Parallel Benchmark suite to estimate the profiling overhead for hybrid applications with multiple threads. The benchmark was executed in three different configurations, keeping the number of processes constant while changing the number of threads. Figure 2.6 shows that the overhead of LWM<sup>2</sup> is less than 1%, with negative overhead for 8 threads. The anomaly can most likely be attributed to the high variance in execution times and overhead, also shown by IPM and the uninstrumented run.

The two sets of experiments confirm the low overhead of LWM<sup>2</sup>, which is similar to IPM, despite generating time-sliced profiles. Overall, the overhead is much less than the natural





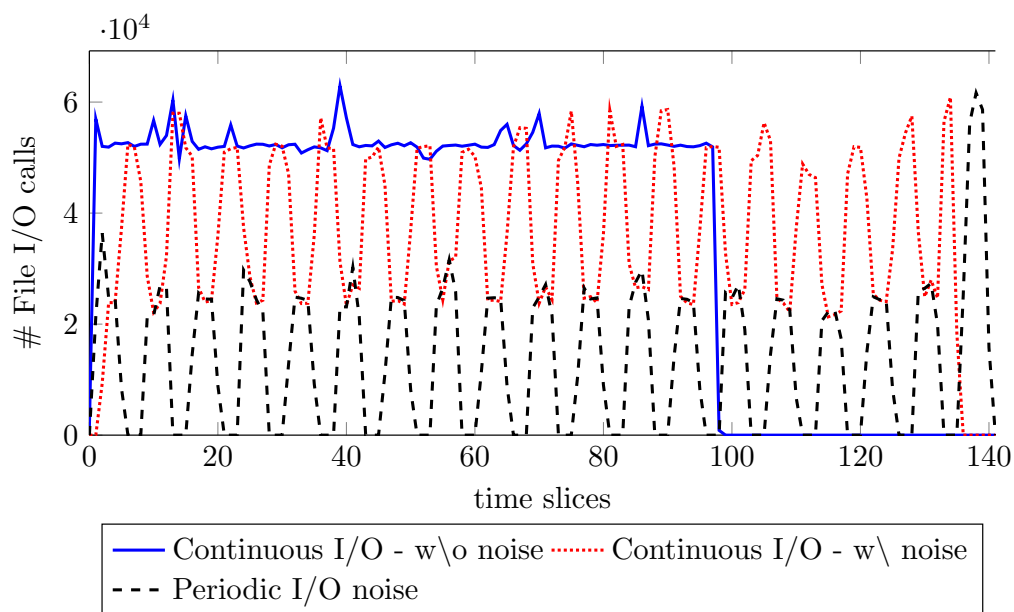
**Figure 2.6:** Runtime overhead of LWM<sup>2</sup> vs. uninstrumented execution for Hybrid MPI (NAS), also in comparison to IPM. The high and low marks indicate the spread after the removal of outliers.

run-to-run variation of the system.

## 2.2.2 Interference

In spite of running in space-sharing mode, applications on cluster systems usually share some resources that they cannot own exclusively. Contention for such shared resources may cause inter-application interference, one component of the overall runtime jitter. In a set of experiments, we measured the jitter resulting from this interference for different types of shared resources.

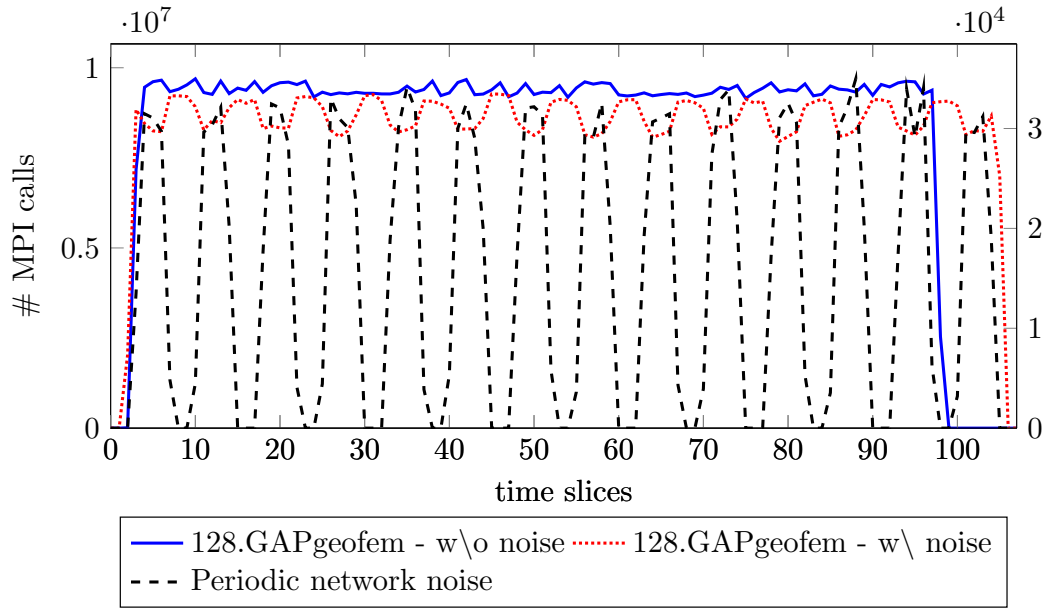
Concurrently executing applications usually share at least the file system and depending on its topology parts of the network. Sharing parts of the network means sharing a subset of the links and switches. Unless the file system is reached over a dedicated network, file I/O also implies network traffic that may interfere with network traffic between compute nodes. While some systems allocate nodes exclusively to a single job, others may even allow the sharing of nodes. There, node sharing is motivated by the desire to balance the need for classic CPUs and GPUs among a highly diverse workload. If nodes are shared then contention may occur with respect to the network interface, the memory, and, depending on the architecture, the accelerator bus. We assume a sensible node-sharing scheme in which applications run at



**Figure 2.7:** Inter-application interference when sharing I/O resources: continuous I/O benchmark vs. periodic I/O noise.

least on separate sockets with their physically attached memory, hence already minimizing memory-subsystem interference. We target each of these shared resources in experiments to measure their impact as a source of interference.

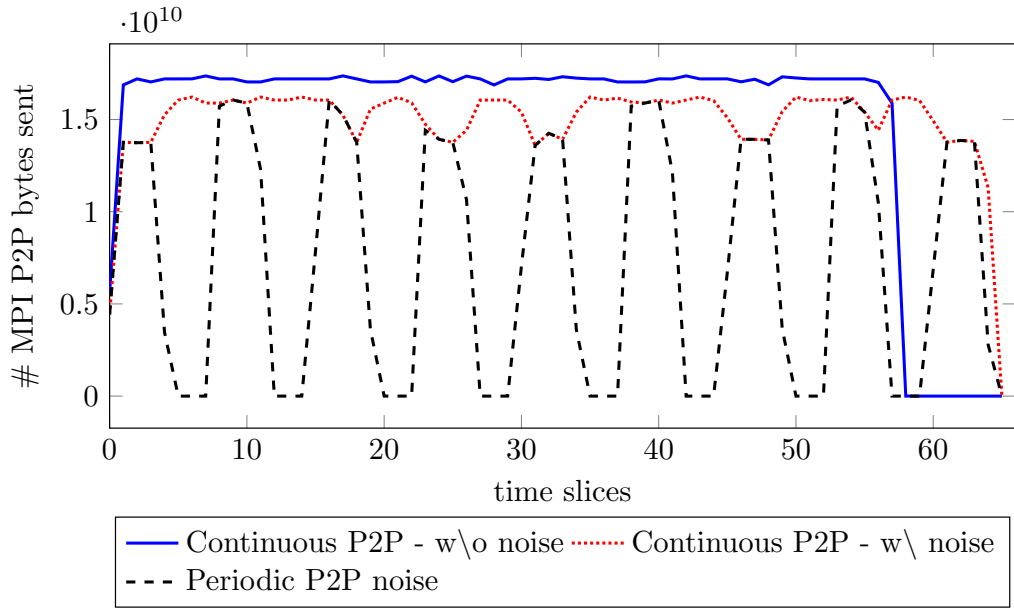
In production environments, cluster applications experience jitter from many sources. Extracting and measuring the contribution of contention for a single resource to inter-application interference is therefore challenging. To do exactly this, we created a set of benchmarks, one for each resource type, that inject noise according to a distinct periodic pattern of resource usage. If the benchmark interferes with another simultaneously executing application that makes use of the same resource, it will impose the inverted pattern on the time-sliced profiles of the application. That is, whenever the noise exhibits a peak, the metrics of the application suffer a dent. Note that because each metric value represents a rate (per time slice), the slowdown affects all metrics. Thus, we identify interference through inspection of time-sliced profiles. Moreover, we quantify the interference effect by comparing the execution time under noise with the time we see without introducing noise. In this study, LWM<sup>2</sup> was not mandatory for all jobs on the system, we could not profile the complete workload—just the jobs we submitted ourselves. However, the artificial noise pattern is distinct enough to make coincidence extremely unlikely. In a production deployment on a full cluster, where the source of interference is unknown, one would search for the source following the inverse strategy, trying to find a metric in the overall performance background that shows a peak where an application we are interested in suffered a dent.



**Figure 2.8:** Inter-application interference when sharing network resources: 128.GAPgeofem vs. periodic network noise.

## I/O subsystem

On JUROPA, file I/O operations may share the communication network and I/O nodes. At the same time, the network is also available to messages exchanged between compute nodes. To study the interference potential of file I/O operations, we ran two I/O intensive benchmarks side-by-side, one with a continuous I/O pattern, and one with a periodic I/O pattern. We ran each of the benchmarks with 256 processes on 32 nodes of JUROPA. In the first benchmark, which we call the probe, each process continuously opened a file, wrote a  $100 \times 100$  integer matrix in consecutive write operations to it, one per element, and closed it again. Each process used its own file. It was created in the beginning and then re-used in each iteration. The second benchmark, the noise, did the same, only that the stream of matrix writes was periodically interrupted by phases of inactivity. The benchmarks performed predominantly file I/O. Hence, any resulting interference pattern can be attributed to this I/O with high confidence. Figure 2.7 shows the number of write operations per time slice, accumulated across all processes, as an indicator of performance. The imprint the noise left on the probe as a sign of interference is clearly discernible, especially when compared to the execution without noise. The interference reduced the application's write performance significantly, with a drop of 50% at the points of interference and an overall execution time prolonged by 35%. The experiment was repeated several times and the same behavior was observed each time. Exposing 128.GAPgeofem from the SPEC MPI2007 suite, a finite-element code that performs I/O on a regular basis, to our artificial noise resulted every time in an abort with a



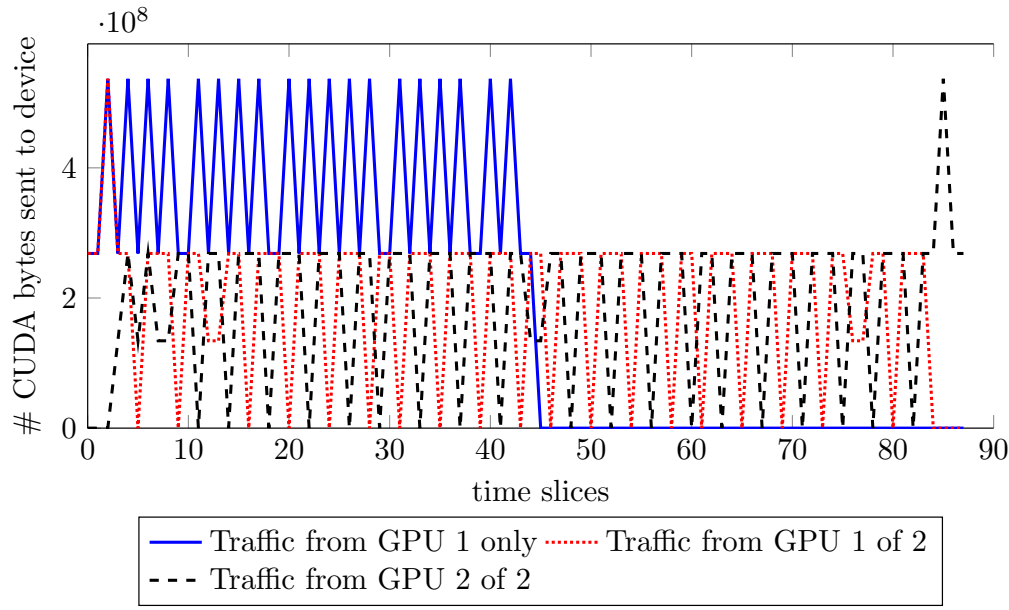
**Figure 2.9:** Inter-application interference when sharing node resources: communication interference during point-to-point communication.

“file not accessible” error, indicating that I/O subsystem interference can even be prohibitive in certain cases.

## Communication network

Although we experimented with a large variety of communication patterns and execution configurations, we were surprisingly unable to find any unequivocal evidence of significant interference at the level of message exchange between the compute nodes of JUROPA. This might have to do with the generous bandwidth and the well-balanced fat-tree topology of the network and does therefore not apply to other systems.

Nevertheless, in this study, what we found is a reproducible case of interference between 128.GAPgeofem, which performs both inter-process communication and file I/O in regular intervals, and a noise benchmark that alternates between periods of intensive global inter-process communication and periods of silence. Each program was executed on 128 nodes of JUROPA. This system has a two-layer fat-tree topology, where each leaf switch connects 24 compute nodes and the top level switches connect all the leaf switches. Utilizing 128 nodes guaranteed that the two programs used overlapping portions of the network. Figure 2.8 shows the number of collective calls per time slice with and without noise. Although not the only and not even the most relevant type of communication in this experiment, collective calls represent global synchronization points, which is why their number serves as



**Figure 2.10:** Inter-application interference when sharing GPGPU resources: two GPU benchmarks sharing one PCI Express bus.

a good indicator of the progress 128.GAPgeofem makes towards completion. It can be seen that during the network activity phase of the periodic noise benchmark, the performance of 128.GAPgeofem drops by 10%. This leads to an overall 5% longer execution time compared to a run without noise. The experiment was repeated several times and each time 7% to 10% degradation was observed during the active phase of the periodic noise benchmark. While this indicates that the interference occurs at the level of the network, it remains unclear whether the target of the interference was inter-process communication or file I/O. During the entire execution, each process of 128.GAPgeofem exchanges more than 1 GB of data in MPI point-to-point communication (and a smaller amount in collectives), while it writes only 0.5 MB of data to files. On the other hand, when exposing 128.GAPgeofem to network noise, we observed several failures due to the inaccessibility of the file system. Also in the light of our disability to demonstrate pure network interference, we are therefore inclined to attribute this phenomenon to cross-interference between network communication and file I/O.

## Node

To improve resource utilization, some accelerator-based HPC systems allow individual nodes to be shared between CPU- and GPU-oriented applications. One such system at the time of our study was Tsubame2 at Tokyo Institute of Technology, which permitted node sharing on its thin-nodes, consisting of two host processors and three GPUs [124]. Tsubame2 was

---

upgraded to Tsubame2.5 in the fall of 2013.

**Network interface.** Even if simultaneously executing applications do not share individual sockets, a common scheduling constraint in the interest of minimizing interference within the memory subsystem, the applications share at least the network interface of the node. In our experiments, we tried to investigate whether such sharing can lead to significant interference. Since none of our test systems allowed nodes to be shared between applications, we split the world communicator of a single benchmark to mimic two different applications. The communicator was split in such a way that the border between the two communicators divided nodes but assigned the sockets exclusively either to one or the other. Both communicators performed MPI point-to-point communication, the first one (i.e., the probe) in a continuous fashion, the second one (i.e., the noise) only periodically, that is, interrupted by phases of inactivity. Figure 2.9 shows the number of bytes sent per time slice for each communicator. For comparison, the probe communicator was also measured with the noise communicator disabled. We observed sporadic interference between the two communicators during execution. We ran the experiment several times and observed sporadic interference in each of the runs, albeit occurring at different points. On average, a run with ten bursts of noise lead to interference during four or five of these bursts. Thus, the average probability of interference for a single burst was between 0.4 and 0.5. A run of the benchmark with only collective calls did not produce any interference.

**PCI Express bus.** Modern heterogeneous clusters may feature more than one accelerator per node. If they are attached to the same PCI Express bus like on JUDGE then interference can occur when two GPU applications share a node even if they do not share GPUs. This is also true for a single application with multiple threads that use different GPUs independently. To test this hypothesis, we ran a simple benchmark on JUDGE, repeatedly multiplying matrices on a single GPU. For each multiplication, the benchmark, which was sequential on the host side, copied the matrix from the host memory to the device. Two instances of the benchmark were executed together on a shared node and compared with a single-instance run. Figure 2.10 shows the number of bytes transferred from host to device per time slice as an indicator of overall progress. It can be seen that for shared execution, the PCI Express bus performance of the application drops by up to 50%, resulting in 35% longer execution. To eliminate the host memory as the source of interference, we replaced the second GPU benchmark instance with another benchmark that accessed the host memory in a way similar to the GPU benchmark but without accessing the PCI bus. No performance degradation was observed in this case. This strongly points to the shared data bus as the bottleneck and source of performance degradation.

---

## 2.3 Summary

To analyze inter-application interference on clusters, we presented a novel profiling method based on the concept of globally synchronized time slices that allows the correlation of performance phenomena across jobs. We showed that it can be applied with only negligible overhead and is thus suitable for the silent profiling of entire workloads. By observing probe applications exposed to a pronounced periodic noise pattern, we analyzed the interference potential of several shared resources. We found file I/O to be a major catalyst of interference with I/O performance degraded by up to 50%. We further identified slower file I/O as a likely effect of concurrent inter-process communication. Finally, we found evidence of bottlenecks at the host channel adapter and the PCI Express bus that connects host and device on heterogeneous clusters when sharing nodes between applications. The latter can be easily addressed by investing in a separate bus.

---

## 3 Influence of file access patterns on I/O interference

---

To meet the growing resource requirements of HPC applications, the performance of HPC systems is continuously being increased. To this end, these systems frequently employ specialized network designs, such as dragonfly or torus networks. They also deploy a centralized, parallel file system to entertain bursty, parallel I/O from applications, which can be in tens to hundreds of terabytes [15]. These parallel file systems have a middle layer of I/O servers that connect the compute nodes of the HPC cluster to storage devices. When applications access a file, the request is routed through the communication interconnect to the I/O servers. Such decoupling of compute and storage allows for easy scaling of I/O resources, it also entails challenges. The parallel file system is a centralized resource, which means it gets shared by all running applications. When multiple applications access the file system simultaneously, it can lead to contention and can substantially degrade application performance. Applications that frequently access the file system or access large amount of data are especially susceptible to such contention, adding an element of variability to their performance [81].

HPC applications that access large amount of data or perform frequent file access requests are quite common. These include data-intensive codes such as the global cloud system resolving model GCRM [74], the cosmic microwave background analyzer MADCAP [12], and the Hardware Accelerated Cosmology Code (HACC) [44]. They generate massive amount of data during execution, resulting in large write requests. On the other hand, the Community Atmosphere Model (CAM) [29] of the Community Earth System Model (CESM) [53] and the continuum mechanics solver OpenFOAM [58] frequently checkpoint their state, resulting in small but recurring writes. Overall, very different classes of file-access patterns can be distinguished. Not only do these patterns access the file system in unique ways, but their sensitivity to interference from other applications that access the file system at the same time also varies widely. Likewise, they actively interfere with other I/O-intensive applications in different ways. All this makes access patterns an important factor for file-system contention. Our initial experiments with different access patterns revealed negligible interference in



---

the case of read-read and read-write contention. These results are also consistent with word-of-the-mouth understanding in the HPC community. Therefore, we concentrated our investigation on write-write contention and the most common access patterns involved.

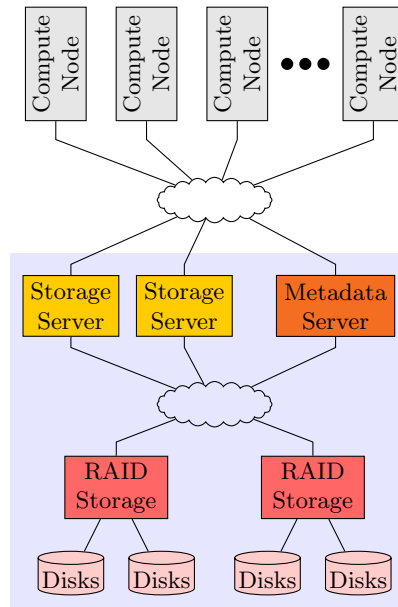
File system contention and the associated performance degradation are well known [126]. In this context, the influence of request size and process count was already studied from a single-application perspective [70], while process count has been identified as a factor of dominance when two applications compete for the file system [33]. Similarly, file-access patterns have been studied in various contexts [14, 26, 80, 116, 121]. The novelty of our research is that we study typical write access patterns found in HPC applications from the perspective of simultaneous access from multiple applications. To achieve this, we first developed a micro-benchmark capable of producing three distinct file-access patterns, mimicking those of real applications. The patterns simulate application checkpointing, out-of-core processing, and writing large output files. We explored the interference potential of these patterns by first running them in isolation and then running them concurrently against each other. We further evaluated the patterns in the form of realistic applications covering check-point-intensive and data-intensive access patterns. We not only observed different levels of interference between different patterns, but also identified some general rules such as writing large output files dominating checkpointing at smaller checkpoint sizes, with the trend being reversed for larger checkpoint sizes.

Taken together, our results bring us closer to effectively reducing interference by automatically recognizing applications with high interference potential, allowing their I/O to be separated in either space or time.

The remainder of this chapter is organized as follows. We first provide the necessary background information on parallel file systems, present our approach, including a taxonomy of file-access patterns, and explain our experiment design. We then present our results, ranging from micro-benchmark-only experiments to measurements with realistic applications. We end the chapter with a summary and also outline the contributors of this work.

## 3.1 Parallel File Systems

A typical file system found on modern HPC systems follows a client-server model, in the spirit of Network File System (NFS). It consists of I/O servers that are connected to RAID storage devices through a dedicated network. A message passing network connects the compute nodes to the I/O servers, as shown in Figure 3.1. When an application accesses a file, the request is routed to the I/O servers. The request is served by distributing it across storage devices depending upon how files are mapped to them. A single file can also be stripped



**Figure 3.1:** A typical Lustre configuration, with separate I/O servers for metadata and file storage.

across multiple storage devices. Such a configuration means simultaneous file accesses can be handled efficiently and in parallel. Examples of parallel file systems include Lustre, HDFS, PVFS, GPFS, PanFS, and FhGPS. We have used Lustre and GPFS in our experiments, and describe them below in detail.

### 3.1.1 Lustre

Lustre is one of the most commonly used parallel file system among the Top500 HPC systems [31]. It is commonly deployed as a central file system, where it separately stores metadata and actual files, as shown in Figure 3.1. Metadata resides on metadata servers (MDSs) while data is stored as objects on *object storage targets* (OSTs), attached to *object storage servers* (OSSs). Metadata includes the file-system structure, the layout of directories, and file attributes. When a file is accessed, its metadata is first accessed from the metadata server. The MDS access itself results in small seeks and reads or writes. The actual data manipulation then happens on OSS as potentially large reads or writes. Decoupling metadata and object storage server gives Lustre an opportunity to optimize each according to its common access pattern.

---

### 3.1.2 GPFS

GPFS stands for General Parallel File System. It is a proprietary parallel file system developed by IBM, and is commonly deployed on IBM systems, such as Summit<sup>1</sup> and Sierra<sup>2</sup>. It is also found on other HPC clusters. GPFS has a flexible architecture. It can be deployed in a shared-disk-cluster configuration where each compute node is responsible for a part of the file system. Another configuration, common on large HPC systems, is as a centralized file system with thousands of nodes. GPFS stores data files as well as their associated metadata on the same block-based devices called *network shared disks* (NSDs). GPFS also achieves high performance by striping data files across all disks in a storage pool.

## 3.2 Approach

HPC applications employ different I/O libraries and file formats to perform I/O operations. They also produce different process-to-file ratios. In our evaluation, we concentrate on single-file-per-process mode, as a major proportion of HPC applications still use POSIX-I/O or MPI-I/O in this scenario [17]. We also evaluate MPI-I/O in shared-file configuration. Given that MPI-I/O in single-file-per-process mode is similar to POSIX-I/O on our test systems (and also on many other systems [12]), we exercise only MPI-I/O in our micro-benchmarks.

### 3.2.1 File access patterns

HPC applications exhibit a variety of file access patterns, of which frequent checkpointing, file accesses for out-of-core processing, and writing of large output files are considered here. We implemented these three use cases as characteristic patterns in a micro-benchmark. We evaluated the patterns by running them against each other and against realistic applications, for a range of file sizes. The access patterns are listed in Figure 3.2.

**Open-write-close.** Our first pattern is called open-write-close (OWC), listed as Algorithm 1 in Figure 3.2. In this pattern, each process creates a new file, writes data to it and then closes it. In the next iteration, a new file is created again for data writing. This is a common pattern, used in applications, such as Flash [36], CESM [53] and OpenFOAM [58], for checkpointing. This access pattern creates high metadata traffic and can create bottleneck on systems with limited metadata resources.

---

<sup>1</sup><https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/>

<sup>2</sup><https://hpc.llnl.gov/hardware/platforms/sierra>

---

**Algorithm 1** Open-Write-Close

---

```
1: procedure OPEN_WRITE_CLOSE( $N, chunk\_size$ )
2:   for  $i \leftarrow 1, N$  do
3:     Open New File
4:     Write  $chunk\_size$ 
5:     Flush I/O
6:     Close File
7:   end for
8: end procedure
```

---

---

**Algorithm 2** Write-Seek

---

```
1: procedure WRITE_SEEK( $N, chunk\_size$ )
2:   Open File
3:   for  $i \leftarrow 1, N$  do
4:     Seek to file start
5:     Write  $chunk\_size$ 
6:     Flush I/O
7:   end for
8:   Close File
9: end procedure
```

---

---

**Algorithm 3** Aggregate-Write

---

```
1: procedure AGGREGATE_WRITE( $N, chunk\_size$ )
2:   Open File
3:   for  $i \leftarrow 1, N$  do
4:     Write  $chunk\_size$ 
5:   end for
6:   Close File
7: end procedure
```

---

**Figure 3.2:** Three I/O access patterns [72, 113].

**Write-seek.** Our second pattern is write-seek (WS), listed as Algorithm 2. In this pattern, each process creates a file at the beginning. Then, in each iteration, each process writes chunks of data to its file, and then seeks back to the beginning of the file. The file is closed at the end of the execution. Similar to open-write-close, this pattern also generates massive number of small file accesses. However, as the same file is reused and as only seek operation takes place between individual writes, this pattern generates less metadata traffic. The

---

write-seek pattern captures file accesses during out-of-core processing of HPC applications, such as in MADCAP [19]. Facing memory capacity pressure, HPC applications often have to resort to out-of-core processing. This means they write data they cannot hold in main memory temporarily to a file, and read it back once it needs to be processed. This results in a write-seek-read pattern. The pattern can have many different instantiations with respect to write size, seek size, and read size. As our goal is measuring write-write interference potential, for simplicity, we have reduced the pattern to a write followed by a complete seek.

**Aggregate-write.** Our third pattern is aggregate-write (AW), listed as Algorithm 3. In this pattern, each process creates a file at the start and then iteratively adds data to it in chunks, closing the file at the end. This pattern mimics writing of large output files in applications such as GCRM [74] and MADCAP [12]. This pattern has a small number of metadata operations but a large number of writes, leading to large files. Aggregate-write can overwhelm an I/O subsystem performance at scale.

Mitigating client-side I/O caching requires flushing I/O traffic after every write operation of the write-seek pattern. In the absence of flushing, small writes remain buffered in the Lustre client software and are overwritten by the writes in the next iteration. We have also found flushing of write buffers in real applications a common practice. Therefore, our addition of buffer flushes is not unusual. The need for flushes does not arise for large writes if they are larger than the OST buffer size. Moreover, aggregate-write is not affected by this issue as writes are initially buffered by the Lustre client software and eventually flushed to the file system. For consistency, our benchmark flushes writes for all chunk sizes on both Lustre and GPFS.

### 3.2.2 Capturing interference

We execute the patterns against each other and observe the drop in I/O throughput compared to an isolated run, to identify incidents of interference. In every benchmarking experiment, we have a pair of applications. We are interested in the throughput degradation of one application, which is called the *probe*. The degradation in throughput is quantified as *passive* interference. The other application, called the *signal*, is causing the degradation through *active* interference. We also execute the signal benchmark in a *periodic* fashion to study how the interference effects evolve with changing I/O access frequency. In the configuration, an active phase of I/O is followed by a silent phase with no I/O activity. The probe benchmark may suffer a drop in I/O throughput during the active phase, whose intensity is indicated by the depth of the dent.

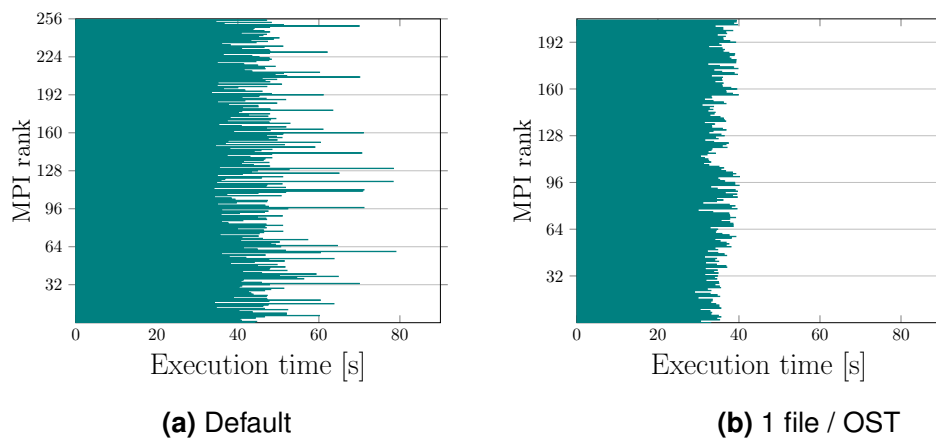
To measure how the I/O throughput of an application changes, we use the profiler

LWM<sup>2</sup> [115]. It is described in detail in Chapter 2. One aspect of LWM<sup>2</sup> important to our study is the ability to capture performance dynamics in *time slices*. The duration of the time slices can be configured. In our evaluation, a time slice of length 4 seconds was used. For the periodic version of the benchmark, a period length of 24 seconds was used, so that every period covers a few time slices.

Ideally, the I/O interference study should be conducted on a dedicated, fully reserved system to have a noise-free environment. However, this requires reserving an entire cluster which is prohibitively expensive. Conducting our experiments on a production cluster while using the mostly application-centric LWM<sup>2</sup> profiler, we had to address interferences from applications outside our experimental setup and irregularities in the behavior of the I/O servers.

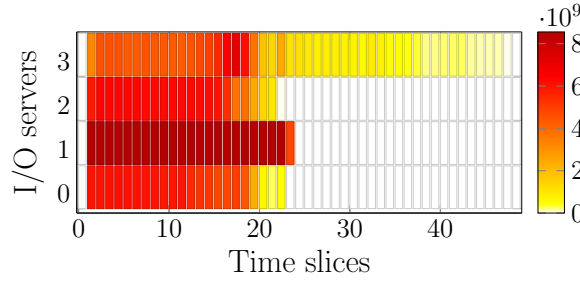
To address these two effects, we extended the LWM<sup>2</sup> profiler to also monitor the I/O server activities when an application is executing. For both GPFS and Lustre, we monitor the InfiniBand counters on the servers to estimate the I/O traffic. We also parse the statistics maintained by the Lustre client on each compute node, indicating the I/O traffic to/from the Lustre servers. Similar to application events, the server activities are also captured every time slice, giving us the ability to correlate them. To keep our evaluation runs as clean as possible, we discard the runs in which the load on the file-servers is 10% higher than the I/O traffic generated by our applications. Similarly, with server-side monitoring, we were able to learn more about the irregularities in the behavior of the I/O servers, and are now able to address them in our measurements.

### 3.2.3 Server-side imbalance



**Figure 3.3:** Mapping one file to one OST reduces the runtime imbalance among processes.

In some experiments, we observed substantial differences among the execution times



**Figure 3.4:** Write throughput of I/O servers. The performance of server 3 is degraded, leading to a longer execution time of I/O operations and hence the application.

of individual processes of an application that occurred sporadically on both file systems. In such cases, most of the processes would finish execution as expected, while the remaining ones would keep on performing I/O for a longer duration, up to twice as long in some cases, as shown in Figure 3.3a. Such observations are not uncommon and have been reported before [136]. In a closer investigation of the imbalance effect, we found the load imbalance among the file servers, as shown in Figure 3.4, to be a major factor. In our experiments, this happened in particular on Lustre, where files were randomly assigned to an OST in the one-file-per-process mode.

When many processes access the same OST, its performance drops, also affecting the performance of the associated I/O server. We verified this finding in a small experimental run by artificially enforcing an equal number of files per OST. Such a setup reduced the variance in execution time of processes by about 75%, as shown in Figure 3.3b. However, enforcing such a policy is not realistic, as it necessitates the number of application processes to be an integer multiple of the number of OSTs. On GPFS, the server-side imbalance was more pronounced for small files, but materialized to a lesser degree for large files as they were striped automatically across all *network-shared disks* (NSDs). Other factors can also contribute to the imbalance effect, such as the straggler phenomenon [136].

To address the effects of the server-side imbalance, while at the same time discern the effects of interference, in our runs, we evaluated the balanced part only, and did not consider the *tail-off* stage. As the server-side imbalance affects only the later part of a run, and the I/O volume written in it is a small portion of the total traffic, our approach is justifiable. In our experiments, the write volume in the imbalanced part of a run was less than 10%. Therefore, in our evaluation, we consider only the part of a run until any of the two running applications finished writing 90% of its total data volume. While our approach did not completely address the variance caused by the server-side imbalance, it significantly diminished the inaccuracy in a consistent way, while also conserving the interference effects.

---

| PFS    | Mount point | Metadata server | File server | storage devices per server | Bandwidth |
|--------|-------------|-----------------|-------------|----------------------------|-----------|
| GPFS   | /data0      | N/A             | 4           | 14                         | 20 GB/s   |
| Lustre | /work1      | 1               | 8           | 13                         | 50 GB/s   |

---

**Table 3.1:** Specifications of the file systems on TSUBAME2.5 used in our experiments.

## 3.3 Evaluation

In our experiments, we first executed our micro-benchmark in pairs to assess the interference effects of one pattern on another in its purest form. To confirm our findings, we then executed the micro-benchmarks against three realistic applications, OpenFOAM, MADbench2, and HACCIO, used for simulations of fluid dynamics, cosmic background radiation, and collisionless cosmic fluid creation, respectively. Finally, we also evaluated the observable interference effects between pairs of these applications.

### 3.3.1 Environment

We conducted our experiments on the TSUBAME2.5 supercomputer, which was deployed at Tokyo Institute of Technology, Japan at the time of the study [125]. The cluster, which was decommissioned in August of 2017, consisted of different types of nodes. The nodes we used made up the majority of the cluster. They consisted of two Intel Xeon X5670 Westmere-EP processors, with 6 cores and at a clock frequency of 2.93 GHz. They also had three NVIDIA Tesla K20X GPUs (GK110) attached to them. The cluster had a two-rail fat-tree InfiniBand 4X QDR interconnect, which was used for both network communication and file I/O.

TSUBAME2.5 was equipped with both a GPFS and a Lustre file system, attached at different mount points. The mount points were frequently updated. The configuration used in our experiments is as follows. GPFS was mounted on /data0 and was hosted on four NSD servers. Each server was connected to 14 NSDs (network-storage devices). Lustre was mounted on /work1 and was hosted on eight OSS (file servers). Each server was connected to 13 OSTs (RAID storage targets). One MDS (metadata server) handled all metadata requests, with one standby server. Only these two mount points were used in our experiments. On Lustre, the `qos_threshold_rr` parameter was set to 16%, which meant mostly a round robin selection of storage devices. The compute nodes were also equipped with 120 GB SSDs as scratch space. The I/O servers were connected to one of the two rails of the fat-tree network using two InfiniBand 4X QDR adapters. Table 3.1 shows a summary of the two file systems we used.



|            |                  | Signal           |            |                 |
|------------|------------------|------------------|------------|-----------------|
|            |                  | Open-write-close | Write-seek | Aggregate-write |
| Standalone | Bandwidth [GB/s] | 28.4             | 31.6       | 42.2            |
| Signal     | Bandwidth [GB/s] | 16.1             | 19.8       | 3.8             |
|            | Degradation [%]  | 43.31            | 35.76      | 9.95            |
| Probe      | Bandwidth [GB/s] | 16.3             | 16.8       | 9.6             |
|            | Degradation [%]  | 42.61            | 40.85      | 66.2            |

**Table 3.2:** Write bandwidth observed in an experimental run on Lustre when the probe open-write-close was executed against three different signal patterns at a chunk size of 1 MiB.

### 3.3.2 Experimental setup

Except for the experiments comparing patterns at different process counts, a single instance of a micro-benchmark or an application consisted of 256 processes. They were executed on 64 compute nodes. As the experiments were undertaken on a production system, we used the I/O server monitoring module of LWM<sup>2</sup> to exclude runs with more than 10% external noise. Every experiment was also repeated five times and we took the best-performing run, that is the run with the lowest degree of external interference.

We executed the three access patterns for chunk sizes ranging from 1 MiB to 256 MiB on a logarithmic scale. This means that for open-write-close and write-seek pattern, a file of a specific size was repeatedly written. For aggregate-write, the buffer size of each write operation had the specified size.

### 3.3.3 Micro-benchmarks

To evaluate the interference potential of the I/O access patterns, we execute them in pairs. We executed every pattern first against itself and then against the remaining two. As a result, there were six different experiments. However, every pattern was considered separately as a probe and as a signal, resulting in nine different interference scenarios (that is  $\{OWC, WS, AW\}^2$ ).

Table 3.2 shows the write throughput when an open-write-close probe is exposed to three different signal patterns. For both the probe and the signal patterns we show the isolated and the interfered throughput. We also quantify the severity of the interference effect in terms of the percentage degradation of the throughput  $T$ , defined as:

$$T = \frac{T_{isolated} - T_{interfered}}{T_{isolated}} \times 100$$

A high value of  $T$  indicates high degradation and severe interference caused by the signal pattern. As the focus of this study is the severity of the interference, in the remainder of the chapter we restrict ourselves to relative throughput degradation figures.

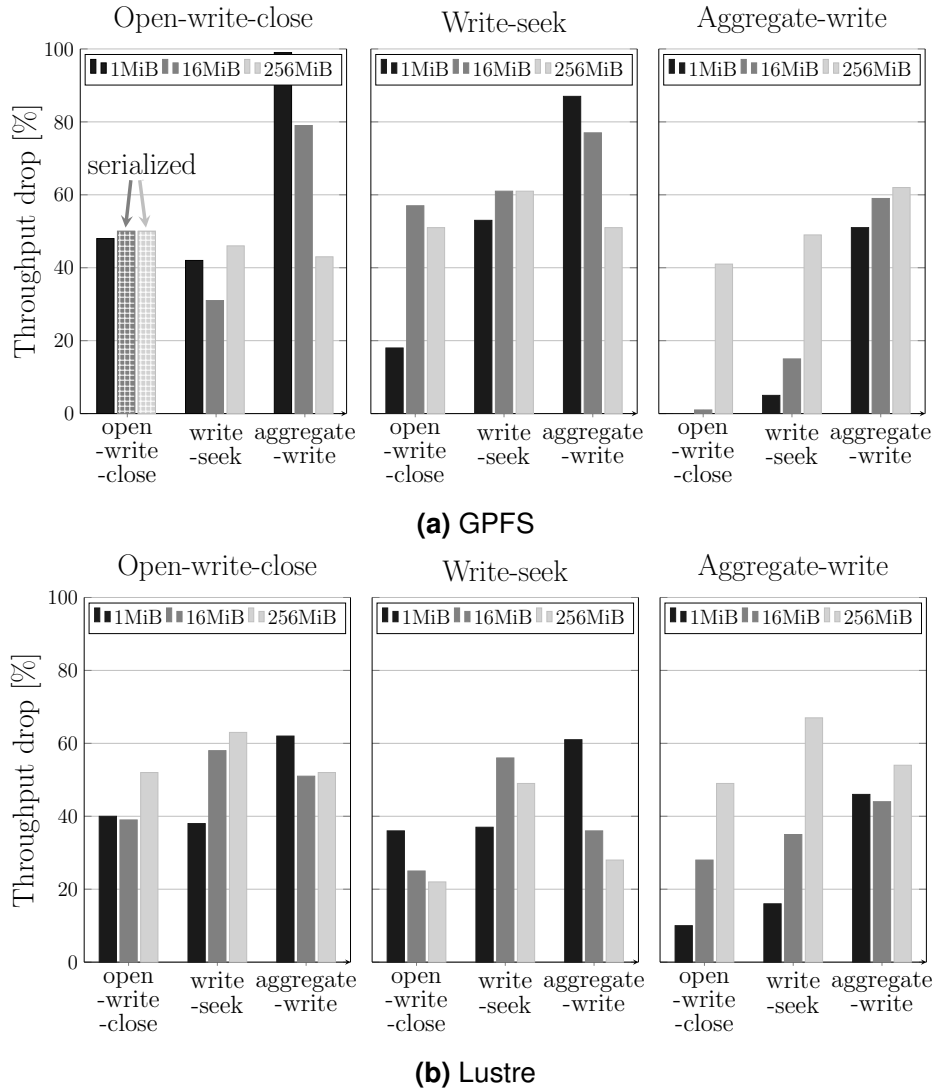
## Access patterns

We executed all the combinations of access patterns for chunk sizes of 1 MiB, 16 MiB, and 256 MiB on both Lustre and GPFS. Figure 3.5a shows the throughput drop for GPFS. For 1 MiB and 16 MiB chunk sizes, aggregate-write has a higher interference potential compared with the other two patterns. When aggregate-write is executed against the other two patterns, the latter's throughput drops by more than 80%, while the former's drops by a small amount. This shows that aggregate-write occupies most the I/O resources at 1 MiB and 16 MiB chunk sizes and dominates the other two patterns. At a chunk size of 256 MiB, open-write-close and write-seek are less affected by aggregate-write, with a throughput drop of about 40% to 50%. Aggregate-write itself is more affected by the other two patterns, indicating equal sharing of I/O resources. Open-write-close and write-seek have throughput drops of about 45% to 55% when executed against each other, which can be caused by equal distribution of I/O resources. Open-write-close, at chunk sizes of 16 MiB and 256 MiB, when executed against itself, becomes serialized, that is, one pattern of the pair executes first, almost completely degrading the second pattern during first's execution.

The same set of experiments was repeated on Lustre, as shown in Figure 3.5b. The interference trend for the 1 MiB, 16 MiB, and 256 MiB chunk sizes is similar to the trend on GPFS, but with different potential. Aggregate-write interferes the most at chunk size of 1 MiB, while itself being affected the least. However, the difference in throughput drop is not as severe as on GPFS. As the chunk size is increased to 16 MiB and 256 MiB, respectively, the interference potential of aggregate-write decreases, whereas that of open-write-close and write-seek increases. At a chunk size of 256 MiB, write-seek causes most of the throughput reduction, more than 60% for the other two patterns.

## Chunk size

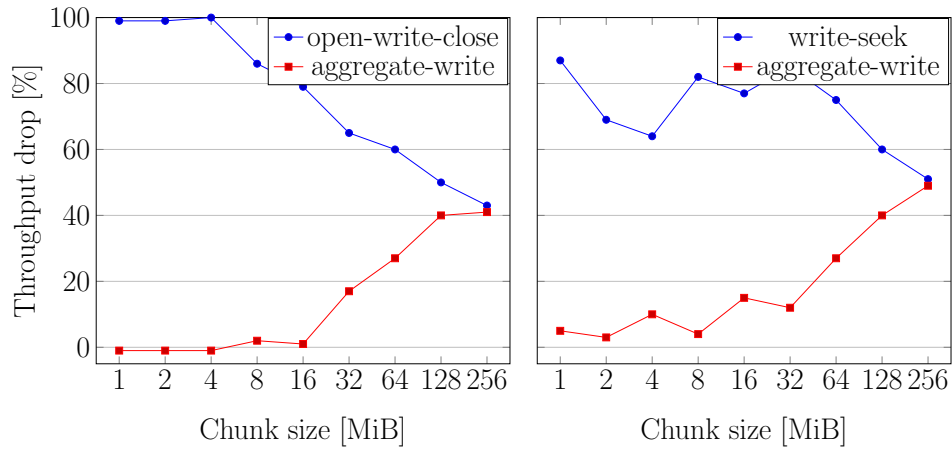
As the interference potential of the above patterns is affected to a large degree by the chunk sizes, we evaluate this more closely. We executed aggregate-write against open-write-close and write-seek for chunk sizes increasing from 1 MiB to 256 MiB logarithmically. The results



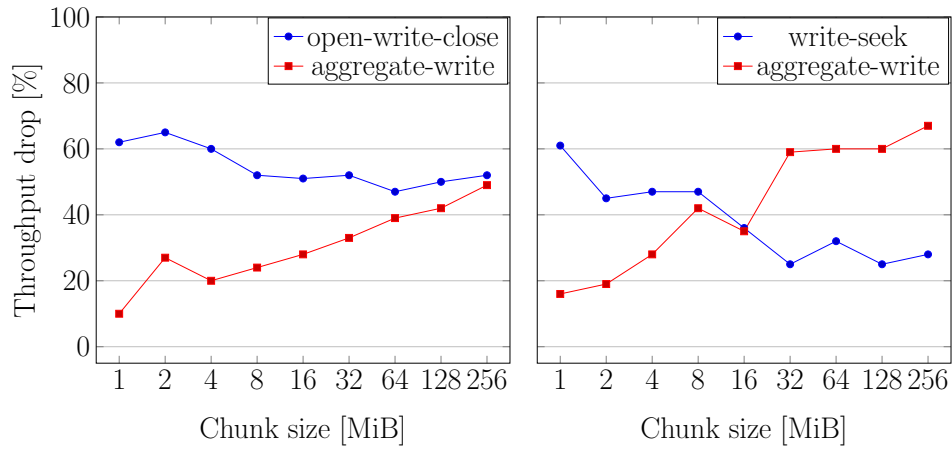
**Figure 3.5:** Drop in I/O throughput when the patterns are executed in pairs against each other. The pattern at the top shows the probe, while the x-axis shows the signal patterns. A lower bar means lower passive interference and higher throughput.

for GPFS are shown in Figure 3.6a. Open-write-close seems to share I/O resources with aggregate-write more evenly as the chunk size increases, with 256 MiB being the break-even point. Write-seek shows a similar trend but with the slope shifted to the right. Here the convergence begins when a chunk size of 32 MiB is reached. The interference trend is similar to open-write-close beyond the chunk size of 32 MiB, presumably because the I/O resources are shared more evenly. The two patterns break even at the last data point of 256 MiB.

We also investigated the effects of chunk size on Lustre. The results are summarized in Figure 3.6b. The trend of open-write-close on GPFS, where, at small chunk sizes, aggregate-



(a) GPFS



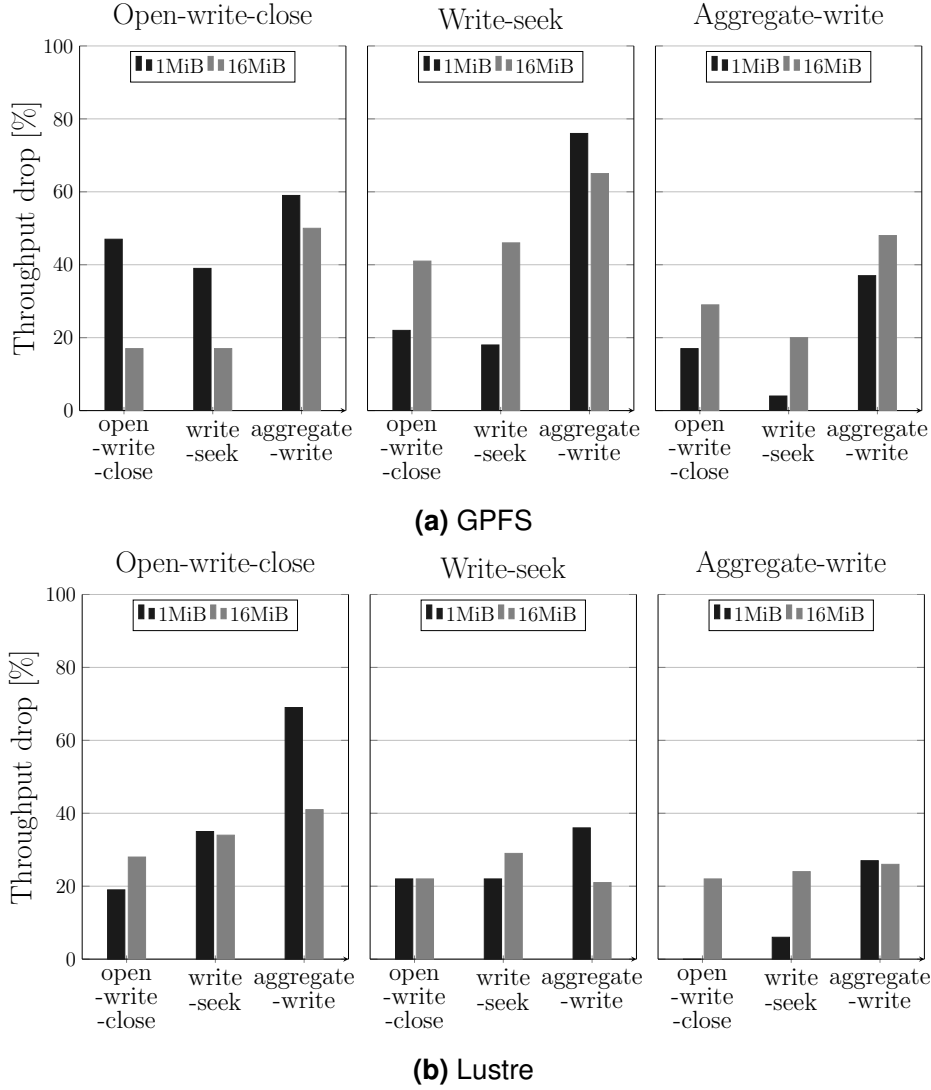
(b) Lustre

**Figure 3.6:** Effect of chunk size on throughput degradation.

write dominates open-write-close, reappears on Lustre. As the chunk size increases, open-write-close starts to perform better. The trend culminates at 256 MiB, where open-write-close and aggregate-write experience the same amount of throughput drop. In the case of write-seek, aggregate-write dominates at small chunk sizes. However, as the chunk size is increased, the trend is quickly reversed. Both patterns suffer the same amount of throughput degradation at 16 MiB, beyond which write-seek starts to dominate aggregate-write.

### High frequency vs. low frequency

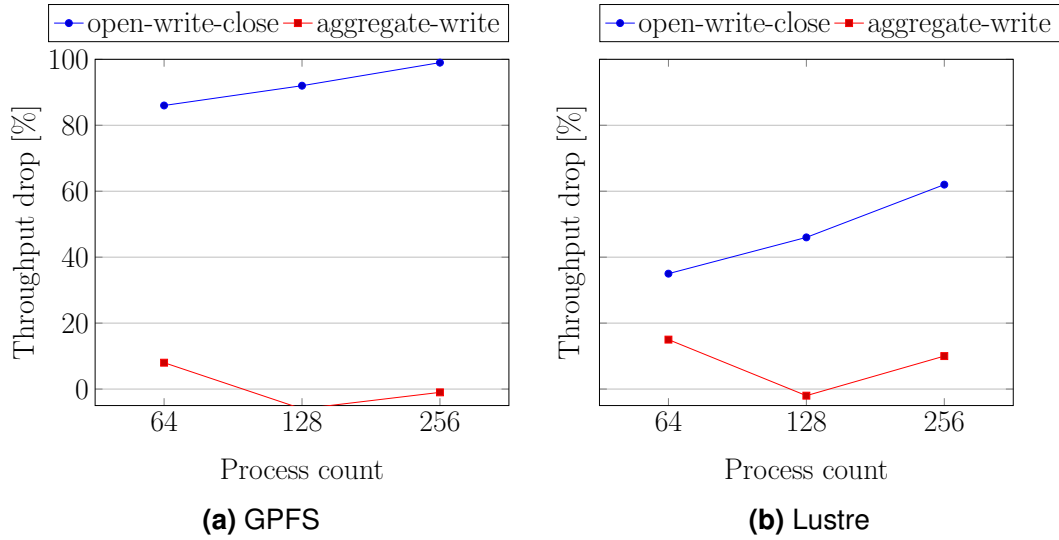
As the sensitivity to chunk size shows, the trend of interference among the patterns depends on their specific characteristics. To evaluate this further, we consider the file access frequency of a pattern. However, covering the whole breadth of possible write access frequencies is



**Figure 3.7:** Drop in I/O throughput when the patterns are executed in pairs against each other. The pattern at the top shows the probe, while the x-axis shows the signal patterns. A lower bar means lower passive interference and higher throughput. The signal pattern is executed in periodic mode.

prohibitively expensive. Instead, we expose the unaltered probe to a periodic signal, in which write activity phases alternate with computation busy-wait phases, mimicking bursty I/O. The period length of the signal was set to 24 seconds so that multiple consecutive time slices of LWM<sup>2</sup> fall in one period. The results of the experiments are shown in Figure 3.7.

Overall, the interference trend, both for GPFS and Lustre, is similar to what was observed for the sustained activity patterns, but with lower intensity. Aggregate-write still dominates open-write-close and write-seek at 1 MiB chunk size. Similarly, at the larger 16 MiB chunk



**Figure 3.8:** Effect of process count on the passive degradation produced by patterns on GPFS and Lustre.

size, aggregate-write causes lesser degradation while finding itself being victimized to a higher degree. On GPFS, at a chunk size of 16 MiB, open-write-close suffers less throughput degradation against itself and against write-seek. One of the reasons can be that at lower frequencies and at larger write chunk size, the metadata operations cease to be the I/O bottleneck, while, additionally, the low frequency prevents the write bandwidth of the system from being saturated. As a result, the performance of open-write-close degrades to a lesser degree.

### Process count

It has been previously observed that an application with higher process count dominantly occupies the I/O resources of a system when run against an application with lower process count [33]. However, does this relationship hold true if the two applications have different file access patterns? We investigated this by running open-write-close and aggregate-write against each other with a chunk size of 1 MiB. Because write-seek is similarly dominated by aggregate-write at this chunk size, we have concentrated our study on open-write-close. For each run, we executed the open-write-close pattern with 256 processes and the aggregate-write pattern with 64, 128 and 256 processes. The results of the experiments are shown in Figure 3.8.

The blue line shows the throughput degradation of open-write-close while the red line shows the throughput degradation of aggregate-write. For GPFS, we see in Figure 3.8a that open-write-close is degraded severely, even when aggregate-write occupies only one fourth

---

the space. As we increase the process count of aggregate-write, open-write-close degrades even more severely. Figure 3.8b shows the trend for Lustre, which is similar to that of GPFS, but with lesser degradation. Again, we see that open-write-close suffers higher degrees of degradation when run against aggregate-write, even when aggregate-write is one fourth the size. The throughput of open-write-close decreases even further as the process count of aggregate-write increases. From these experiments we can conclude that, when it comes to sharing of I/O resources between applications, the write-access pattern can play a bigger role than the application process count.

## Shared file

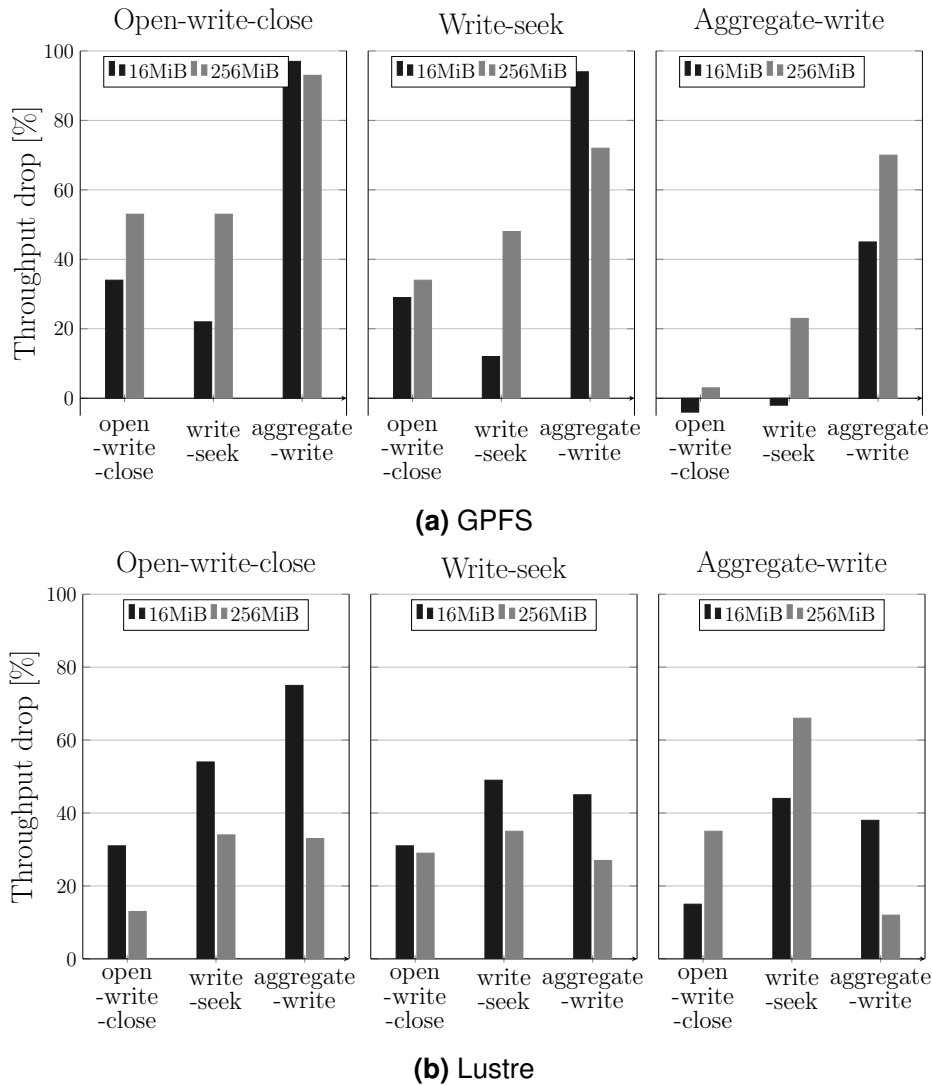
We also evaluated the increasingly common mode of shared files. We set up the sharing in such a way that each process was assigned a contiguous section of the file. The size of the section was the same as the chunk size for open-write-close and write-seek, while it was equal to the total write volume of each process for aggregate-write.

Figure 3.9a shows the interference potentials on GPFS. At a chunk size of 16 MiB, aggregate-write dominates the other two patterns significantly, while at 256 MiB, even though being still dominant, the drop in throughput it causes is less. These findings are comparable to the results with our single-file-per-process setup. However, in the pairwise execution, we found some cases where one pattern would completely dominate the other pattern, leading to serialization of the I/O traffic between the pairs. This serialization behavior was observed for both, patterns running against themselves and against other patterns. Figure 3.9b presents the results on Lustre, where we observed that aggregate-write dominates open-write-close at a chunk size of 16 MiB, while becoming itself slightly dominated by write-seek. At a chunk size of 256 MiB, open-write-close and write-seek are evenly interfered in all the runs but dominate aggregate-write. The behavior of aggregate-write is again consistent with the one-file-per-process case. Open-write-close, however, is less prone to interference at larger chunk sizes.

In the presented results, we have only touched a small set of possible options of writing shared files. Furthermore, shared files have their own set of characteristic access patterns. Therefore, a full coverage of shared files warrants a separate study.

## Discussion

We evaluated different factors and found that file access patterns and the chunk size of their I/O operation have a large impact on the interference potential of applications. At smaller chunk sizes, open-write-close and write-seek are dominated by aggregate-write, significantly



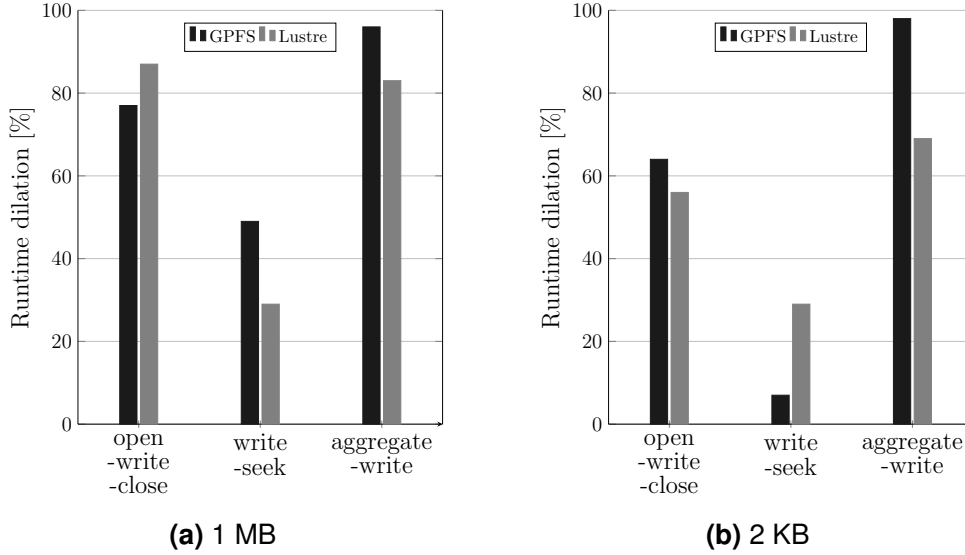
**Figure 3.9:** Drop in I/O throughput in MPI shared-file when the patterns are executed in pairs against each other. The pattern at the top shows the probe, while the x-axis shows the signal patterns. A lower bar means lower passive interference and higher throughput.

degrading the throughput of the former two while exerting little pressure on the latter. On the other hand, as chunk size increases, so does the interference potential of open-write-close and write-seek. At a certain chunk size, they both degrade as much as aggregate write, beyond which the trend may even become reversed. Open-write-close has less interference potential on Lustre compared to write-seek, while on GPFS they show similar degradation trends. The factors contributing to the interference trend are not clearly known. However, both the number file blocks written and the number of metadata operations performed by a pattern are determinant, and would explain the trend in Figure 3.6b. With the increase in chunk size, the number of blocks written by aggregate-write also increase, while the rate of



| Application | Access pattern   | Chunk size  |
|-------------|------------------|-------------|
| OpenFOAM    | open-write-close | a few bytes |
| MADBench2   | write-seek       | 74MB        |
| HACCIO      | aggregate-write  | 386MB       |

**Table 3.3:** Applications and the access pattern they represent including the chunk size.



**Figure 3.10:** Throughput degradation of OpenFOAM when run against the patterns.

metadata operations decreases.

### 3.3.4 Applications

After establishing an interference relationship among the access patterns through micro-benchmarks, we investigated the same effects using realistic applications. We consider three typical I/O-intensive HPC applications, OpenFOAM, MadBench2, and HACCIO. All three together provide one realistic use for each of the three access patterns, as summarized in Table 3.3.

#### OpenFOAM

OpenFOAM (Open source Field Operation And Manipulation) [58] is an open source computational-fluid-dynamics software. It is distributed by the OpenFOAM foundation. It is written in C++ and has a modular design. It provides parallel implementations of mathematical equation solvers and general physical models. It uses standard C++ file I/O

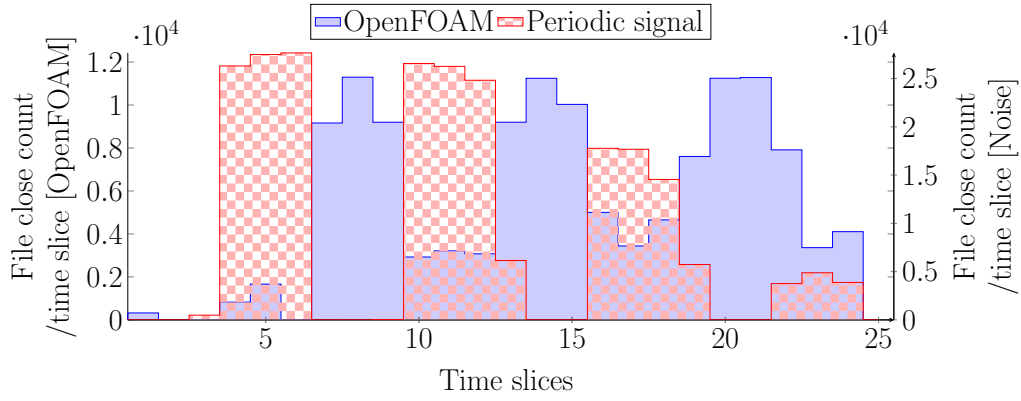
---

for checkpointing periodically. At a checkpointing event, each process writes a new file a few kilobytes in size, which makes the checkpointing behavior similar to our open-write-close pattern. As LWM<sup>2</sup> has limited support for C++ file I/O, we were able to record only the file-close count in our experiments. However, it is possible to substitute the throughput rate of OpenFOAM with the file-close rate. OpenFOAM frequently opens and closes files, up to 14000 in one time slice in our experiments. As the application itself had a few hundred processes, this means that most of the files closed in a time slice were also created in it. This makes it possible to substitute the throughput rate with the file-close rate. We complement this information by the dilation of the execution time, which is caused by drop in I/O throughput.

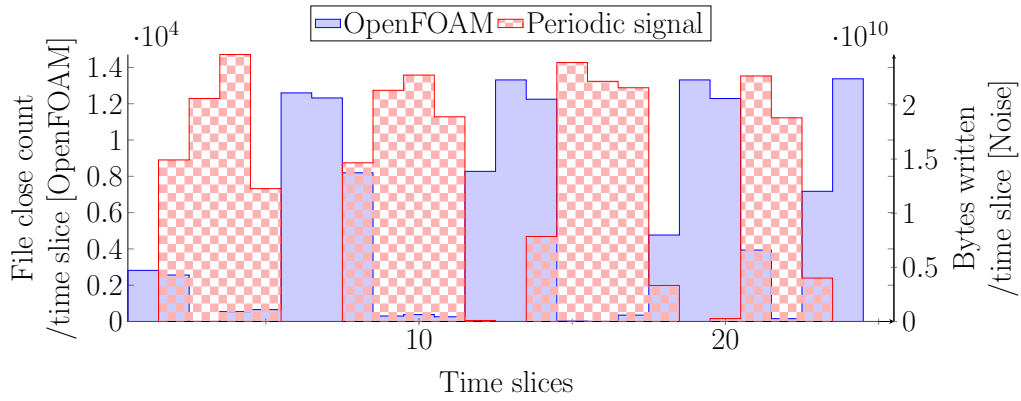
OpenFOAM is a library which is used by other applications as equation solver. In our setup, we used the version 2.3.0 of OpenFOAM and executed the cavity example from the official tutorial against our three micro-benchmark patterns, both on GPFS and Lustre. We set the chunk size of the patterns to 1 MiB, the smallest chunk size used in our micro-benchmark only experiments.

The I/O performance of OpenFOAM degraded when executed concurrently against all the three patterns, as shown in Figure 3.10a. As the I/O access pattern of OpenFOAM is similar to our open-write-close pattern, and it performs I/O at small chunk sizes, the significant interference potential of the aggregate-write patterns is immediately evident. It causes more than 80% drop on Lustre and 90% on GPFS. Unlike the micro-benchmark, OpenFOAM suffered also about 80% throughput drop against open-write-close. One reason for this behavior can be the unequal chunk size of the patterns and OpenFOAM. To verify this, we executed the patterns at 2 KiB chunk size. The results are shown in Figure 3.10b. Aggregate-write still dominates OpenFOAM, with a throughput drop of more than 90% for GPFS. Lustre on the other hand shows a slightly reduced drop of 70% in throughput. Open-write-close now degrades OpenFOAM's throughput by around 60%, similar to what the micro-benchmark allowed us to see. The interference of write-seek on Lustre remains at 30% for both chunk sizes. However, it declines from around 50% for 1 MiB to 10% for 2 KiB on GPFS. Overall, the interference trend is similar to that of our purely micro-benchmark-based observations.

To gain deeper understanding of the I/O interference dynamics during parallel execution, we executed open-write-close and aggregate-write in periodic mode against OpenFOAM. The periodic mode of the micro-benchmarks have been described in Section 3.3.3. We show the *time slice view* of OpenFOAM executing against open-write-close in Figure 3.11a and against aggregate-write in Figure 3.11b. During the active phase of the open-write-close pattern, the performance of OpenFOAM degrades by 60%-70%. On the other hand, the active phase of aggregate-write degrades the performance of OpenFOAM by up to 95%. This is quite evident in Figure 3.11b as the file-close rate of OpenFOAM drops significantly under interference. Comparing OpenFOAM to our open-write-close micro-benchmark, we see that it suffers in a similar way when exposed to aggregate-write, that is, its performance degrades significantly.



(a) OpenFOAM vs. open-write-close



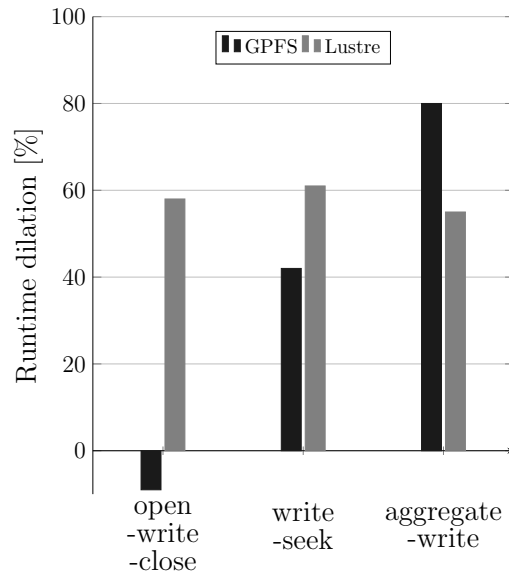
(b) OpenFOAM vs. aggregate-write

**Figure 3.11:** Execution of OpenFOAM against periodic occurrences of open-write-close and aggregate-write on GPFS. The time-slice view shows low throughput during the active phases of the micro-benchmarks.

## MadBench2

MADbench2 is derived from MADCAP, which is a cosmic microwave background radiation analysis software. It performs dense-linear-algebra calculation using ScaLAPACK [23]. The matrices required for its calculation normally do not fit in memory. Therefore, the matrices are stored on disk and read when needed. This means MadBench2 performs complex I/O operations in four phases. For our experiments, as the scope of our study is write-write contentions, we concentrate on the first phase, which has only writes and seeks. The other phases are either reads or a mixture of reads and writes. We henceforth use MadBench2 to refer to the build with the first phase only.

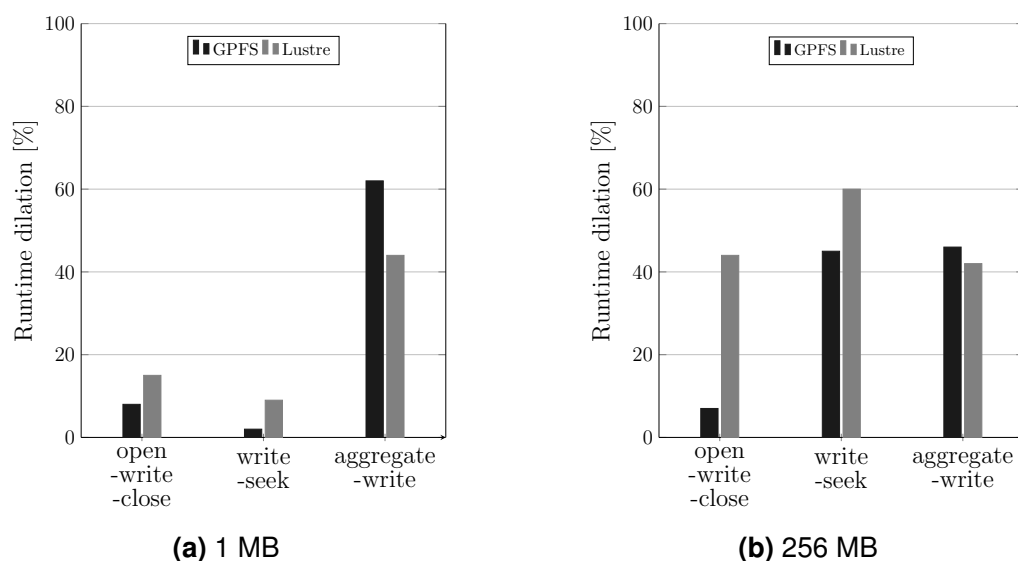
For the experiments, we setup MadBench2 to use POSIX I/O in the one-file-per-process mode. To maximize performance, we used the configuration recommended by Borill et



**Figure 3.12:** Throughput degradation of MadBench2 when run against different patterns.

al. [12], which is: WMOD=1, NPIX=50,000, NBIN=36, NGANGS=1, SBBLOCKSIZE=1, FBBLOCK- SIZE=128. Furthermore, MadBench2 is configured to run in I/O mode. In I/O mode, MadBench2 acts as a pure I/O benchmark, replacing computation with busy-wait cycles. With this configuration, and using 256 processes, MadBench2 writes 670 GB of data, with each process performing seeks with an offset of about 74 MB during execution. This makes the I/O behavior similar to the write-seek pattern with a chunk size of 74 MB. For this reason, we executed MadBench2 against the three patterns at a chunk size of 64 MB. The results are shown in Figure 3.12.

The throughput degradation on both file systems is quite different for MadBench2. On GPFS, aggregate-write generates the most interference, reducing the throughput by about 80%. Similarly, write-seek degrades the throughput of MadBench2 by about 40%. However, in the case of open-write-close, MadBench2’s runtime improves. For a chunk size of 64 MiB, the higher interference aggregate-write generates is consistent with our micro-benchmarks results. On Lustre, all the patterns generate similar interference levels, with aggregate-write degrading the throughput of MadBench2 slightly less compared to others. The reason is that, for write-seek, the interference trend already reverses at a chunk size of 64 MiB, as was shown in Figure 3.6b. Overall, the passive interference behavior of MadBench2 resembles that of our write-seek micro-benchmark.



**Figure 3.13:** Throughput degradation of HACCIO when run against different patterns.

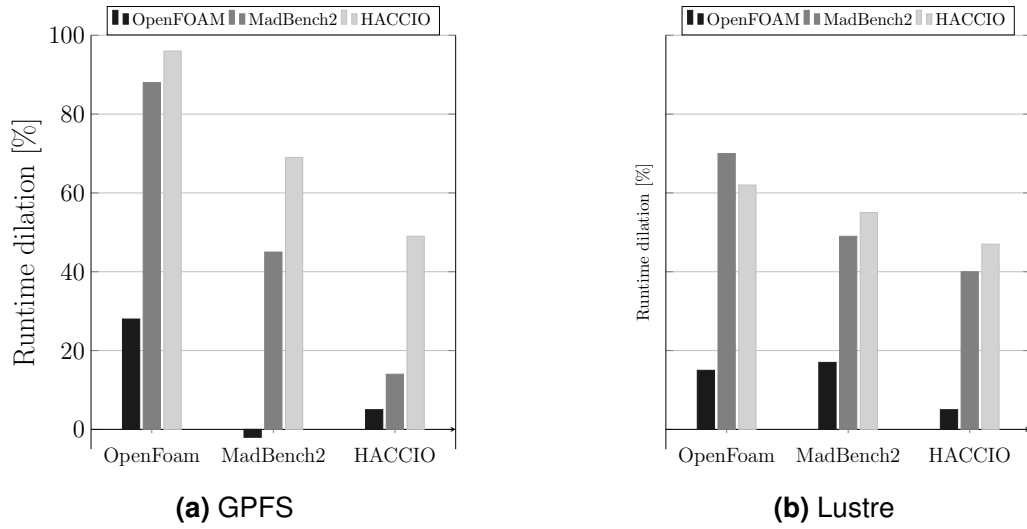
## HACCIO

HACCIO is an I/O benchmark code derived from a cosmology software framework called HACC (Hardware Accelerated Cosmology Code). HACC simulates the formation of collisionless fluids under the influence of gravity using N-body techniques. HACC has very high I/O demands, where a small simulation can write terabytes of data [44].

HACCIO writes large checkpoint files during its execution. It creates one file per process, and incrementally writes data to it. During checkpointing, the files are written, read back, and verified. As our work concentrates on write-write contention, we removed the read-back and verification part in our experiments. HACCIO can use different I/O modes during execution, including POSIX I/O, MPI with one file per process or MPI with one or more shared files.

In our experiments, we ran HACCIO with 256 processes and with POSIX I/O. During execution, each process wrote 3.6 GiB of data to its file, in chunks of 381 MiB. The I/O behavior can be equated to the aggregate-write pattern with a chunk size of 381 MiB. We ran HACCIO against the three patterns with chunk sizes of 1 MiB and 256 MiB, respectively. The results are shown in Figure 3.13.

With 1 MiB, on both GPFS and Lustre, open-write-close and write-seek degrade HACCIO's performance to a smaller degree than aggregate-write. This trend is consistent with our micro-benchmark results. In the case of aggregate-write, the degradation HACCIO suffers on GPFS is slightly higher than on Lustre (60% vs. 40%). In our micro-benchmark experiments for 1 MiB and 16 MiB, we also saw aggregate-write suffering a degradation of around 40% on Lustre and between 50% and 60% on GPFS. With 256 MiB on Lustre, the degradation caused



**Figure 3.14:** Throughput degradation when the applications are run against each other.

by write-seek grows to about 60%, while open-write-close and aggregate-write cause around 50% degradation. This is again similar to what has been observed with micro-benchmarks. Write-seek dominates aggregate-write at large chunk sizes. On GPFS, open-write-close degrades HACCIO by less than 10%. On the other hand, write-seek and aggregate-write cause about 50% degradation of the HACCIO write throughput. Overall, the trend is similar to our micro-benchmark-based observations.

### Application vs. application

After evaluating how micro-benchmarks with isolated access patterns interfere with realistic applications, we extend our investigation to interference between realistic applications. To achieve this, we ran OpenFOAM, MADBench2, and HACCIO first against themselves and later against each other, always using 256 processes per application. The results are shown in Figure 3.14. In the figure, the x-axis shows the probe applications whose runtime dilation is reported. For each probe application, we show a separate bar for each signal application that is causing degraded performance. In these figures, each application represents an access pattern, however, each one of them has a different chunk size and access frequency. Therefore, the interpretation our results requires consideration of pattern type, chunk size, and access frequency.

On GPFS, HACCIO generates the biggest interference of all, with OpenFOAM being degraded by more than 90% and MadBench2 by more than 60%. The values are similar to open-write-close against aggregate-write at a chunk size of 1 MiB and write-seek against aggregate-write at 256 MiB. MadBench2 degrades OpenFOAM by more than 80% and

---

HACCIO by more than 10%. Here, MadBench2's behavior diverges from write-seek, with high degradation for OpenFOAM and low degradation for HACCIO. A possible explanation for OpenFOAM against MadBench2 can be the large chunk-size difference, while for HACCIO it can be low access frequency, as was observed for periodic probe signals in Figure 3.7a. OpenFOAM against the other two applications generates a comparatively small throughput degradation. This is similar to our observation of open-write-close at small chunk sizes.

On Lustre, HACCIO degrades OpenFOAM by about 60%, while being degraded itself by less than 10%, similar to what was observed with micro-benchmarks. HACCIO degrades MadBench2 by about 55%, while being degraded itself by about 40%. This is again similar to micro-benchmark results, where for chunk sizes greater than 16 MiB, write-seek dominates aggregate-write. Looking at MadBench2 against OpenFOAM, we see that OpenFOAM's runtime is dilated by about 70%. This is because of the large chunk-size difference between OpenFOAM and MadBench2.

Considering the different access patterns, write chunk sizes, and access frequencies, the overall results are in line with our observations of synthetic micro-benchmarks.

## 3.4 Conclusion

In this study, we evaluated the interference potential of different I/O access patterns, and assessed factors such as the behavior of the patterns, their write chunk size, access frequency, process count, and sharing mode. Specifically, we found that file access pattern and its I/O chunk size are a significant factor in determining the interference potential of applications. At small chunk sizes, metadata-intensive applications can slow down significantly when executing concurrently with data-intensive applications, up to a factor of five in one of our experiments. However, the data-intensive applications are only slightly affected. Furthermore, the data-intensive application had higher interference potential even at smaller process count. However, with increasing chunk size, the interference trend is reversed.

Preventing I/O interference in a centralized and shared parallel file system is challenging. However, the impact of interference can be reduced by separating file accesses from different applications, either in space by isolating their I/O or in time by scheduling them separately. For this technique to work, it is important to automatically identify aggressive and sensitive patterns. Our technique can be leveraged to identify them and to dynamically separate them.

---

## 3.5 Contributions

The write-write contention study described in this chapter was carried out as a master thesis project by Chih-Song Kuo under my supervision [72], as also described in Section 1.6. The concept of inter-application interference, the method to identify it, and the profiling tool LWM<sup>2</sup> were already developed before his thesis, as described in Chapter 2. I had also identified the three write patterns used in this study. Each process in the open-write-close pattern initially accessed the same file in all iterations and was later changed to a new file for every iteration based on reviews of our conference paper [73]. I had also developed the micro-benchmark used in this study and had gathered initial experimental data for pattern vs. itself. Chih-Song carried the research from this point onward. His contributions are extending the LWM<sup>2</sup> profiler with server-side profiling and conducting all the experiments presented in this chapter, including the application-vs-application experiments. He also choose the appropriate I/O phase of MadBench2 for the experiments. He further identified the server-side imbalance phenomenon, and investigated the factors behind it. The postmortem approach to mitigate its effects is a joint contribution.

The master thesis was a joint collaboration with Prof. Matsuoka’s group at Tokyo Institute of Technology. They provided the computing resources for all our experiments on TSUB-AME2.0 supercomputer, and also helped Chih-Song develop the module to monitor the I/O servers.





---

## 4 Visualizing traffic on high-dimensional torus networks

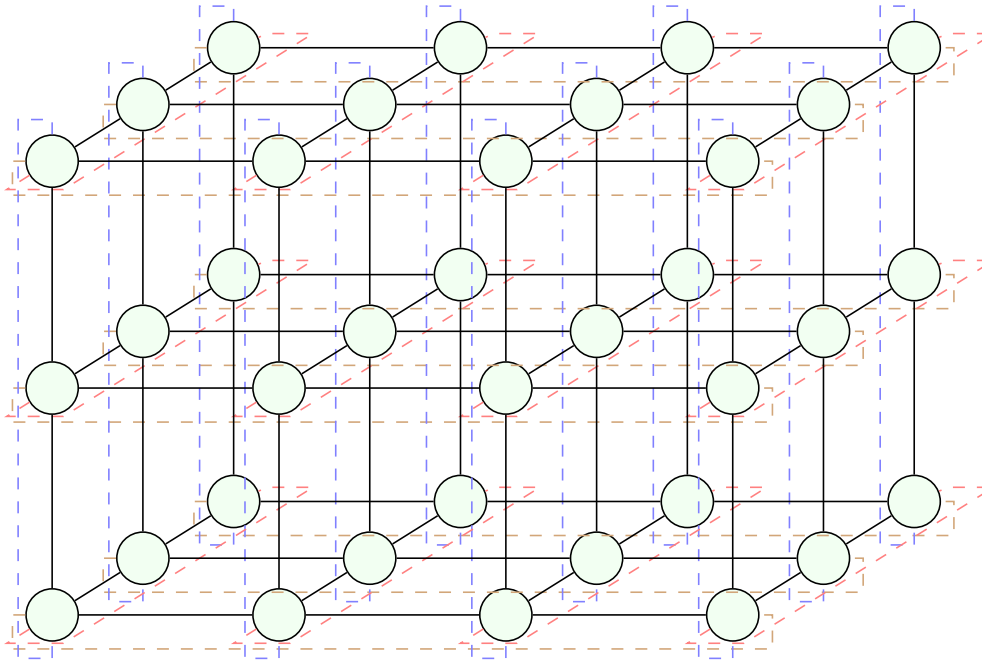
---

The communication interconnect is a shared resource on a HPC system. It binds all the compute nodes together, allowing applications running on them to exchange messages. In order to meet the increasing resource requirements of applications while also meeting financial constraints, innovative network architectures have been deployed on HPC systems. Examples include fat-tree [77], torus [27], and dragonfly [67] networks.

The torus network in particular has a peculiar architecture, as shown in Figure 4.1. It has a switch-less architecture, where the compute nodes are directly connected to each other, forming a three-dimensional [10], five-dimensional [21], and even six-dimensional [5] tori. Such a network architecture provides high bandwidth at low latency among neighboring compute nodes. Furthermore, as the network is scaled to a higher number of nodes, the number of required interconnects also scales linearly. In comparison, the number of required interconnects scale quadratically on an all-to-all network. As a result, torus networks provide scalability at low cost. However, it can also lead to inter-application interference.

Messages in a torus network are routed through intermediate nodes. When the communication nodes are far apart, the messages are exposed to traffic from intermediate nodes, which can lead to contention. Some systems, such as IBM Blue Gene systems, assign exclusive, compact network sections to applications. Users also have a degree of freedom in mapping application processes to compute nodes. Mapping communicating processes to neighboring nodes localizes network traffic and reduces contention [139].

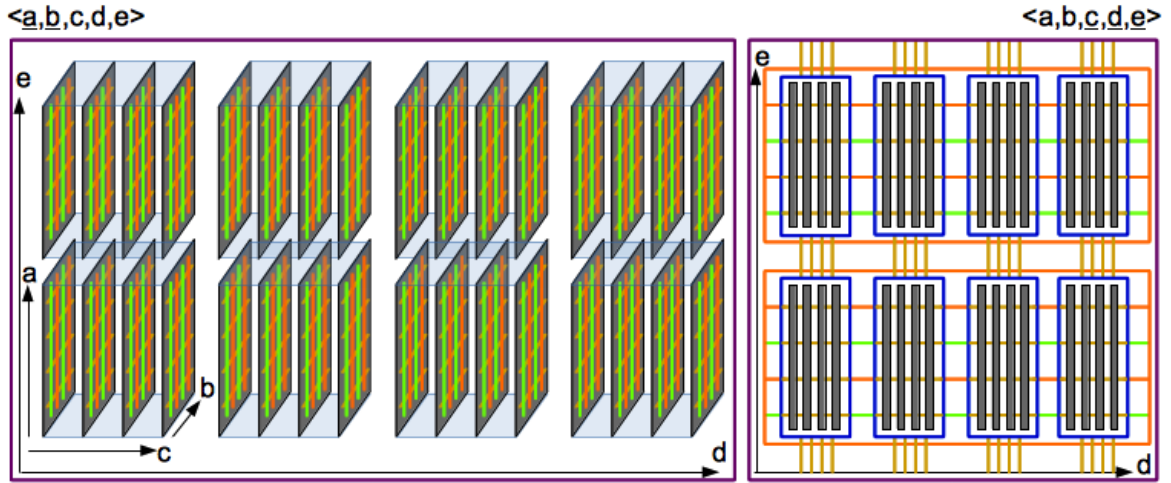
Other systems, such as Cray XE6 systems, do not enforce any restrictions on the process-to-node mapping. The nodes can be located close to each other or dispersed in the network, which can lead to contention and variation in performance. Bhatele et al. compared the performance of Mira and Intrepid (Blue Gene systems) with Hopper (Cray XE6) using a communication-heavy application [8]. They found the application to have consistent performance on the former two while it exhibited significant variation on the latter, caused mainly by inter-application interference.



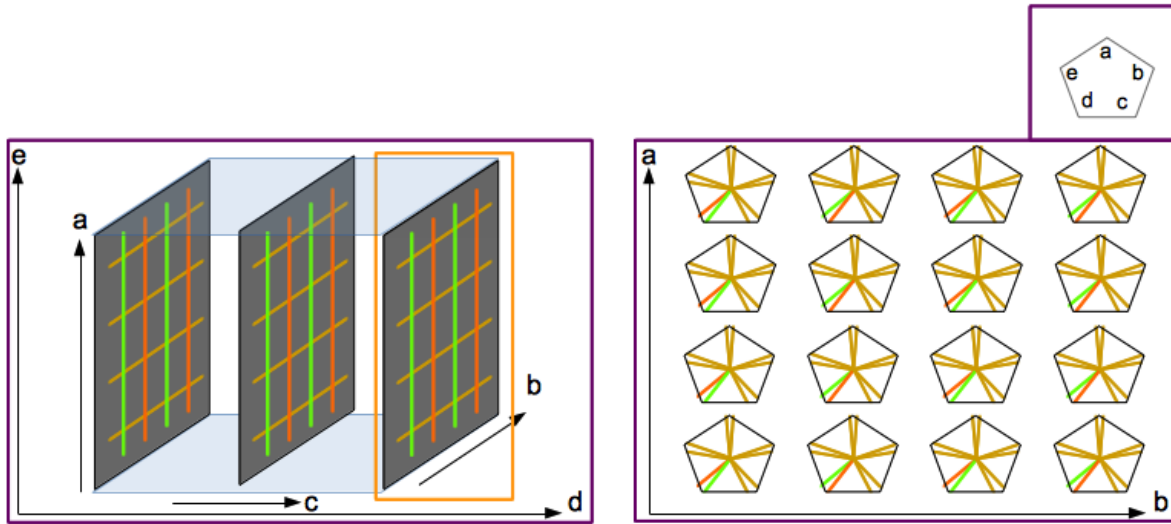
**Figure 4.1:** A three-dimensional torus network with a  $(4 \times 3 \times 3)$  network topology. The loop-back links are shown as dashed lines.

Identifying inter-application interference on these torus networks requires capturing communication traffic, analyzing it, and also providing visual feedback to the user. The visualization should be system-centric, that is in terms of compute nodes and their connecting network routes, making it possible to identify hotspots on the network and relate them back to applications. However, projecting a multi-dimensional torus network onto a two-dimensional display for an intelligible feedback is challenging. This is especially the case for systems with high-dimensional torus networks, such as the five-dimensional network of IBM Blue Gene/Q systems [21] and the six-dimensional network of Fujitsu's K computer [5]. We tackle this problem by proposing a visualization technique based on displaying simultaneous views to the user. Each view displays the same network section but uses a different visualization paradigm. The views also focus on different dimensions. They are interactive and synchronized with each other.

In the remainder of this chapter, we present our visualization technique demonstrating its use of simultaneous views. We then apply the technique on a synthetically generated application to identify hotspots. We summarize the chapter at the end and also outline the contributors of this work.



(a) Left: the detailed plane view, visualizing the traffic inside the  $(a \times b)$  planes in full detail. Right: the composite plane view summarizing the traffic in the remaining dimensions. The ordering of the dimensions is the same in the views.



(b) The zoomed-in view of the  $(a \times b)$  planes and the polygon view, displayed side-by-side. The polygon view shows node-level details of the selected plane, separately displaying the incoming and outgoing network traffic links.

**Figure 4.2:** An interactive method to visualize traffic on a high-dimensional torus.

## 4.1 Visualization

We use the example in Figure 4.2 to explain our approach. The figure depicts the mockup of an application on a  $(a \times b \times c \times d \times e)$  dimensional torus network.

---

### 4.1.1 Detailed plane view

In Figure 4.2a, the detailed plane view is shown on the left side. The view displays the network traffic along two dimensions as a series of planes, while the planes themselves are ordered based on the other dimensions. In the example, the planes visualize the traffic along the  $(a \times b)$  dimensions. The planes are horizontally ordered first along the  $c$  dimension and then along the  $d$  dimension. The vertically ordering is along the  $e$  dimension.

### 4.1.2 Composite plane view

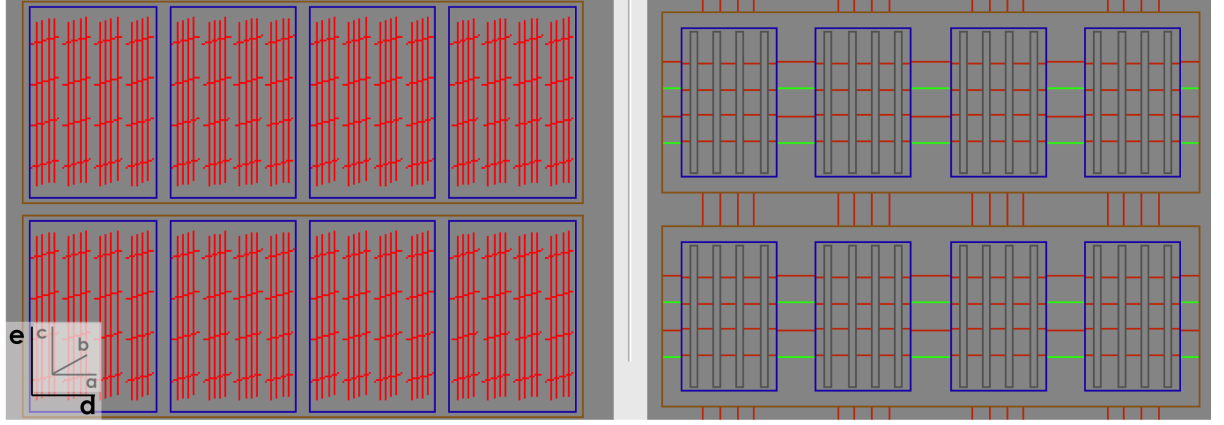
The composite plane view complements the detailed plane view, shown on the right side in Figure 4.2a. The blue rectangles in the figure represent the  $(a \times b \times c)$  cuboids. The black bars represent the  $(a \times b)$  planes of the detailed plane view, collapsed along the  $b$  dimension. As the length of the dimension  $a$  is four, there are four lines between the black bars. Each line represent the traffic between the planes along the  $c$  dimension. The traffic is aggregated along the  $b$  dimension, that is every line  $n$  represents the traffic between between nodes on plane  $(a \times b)$  where  $a = n$ .

The blue rectangles are ordered horizontally along the  $d$  dimension and vertically along the  $e$  dimension. The horizontal lines between them depict the traffic between the  $(a \times b)$  planes along the  $d$  dimension, while the vertical lines summarizes the traffic along the  $e$  dimension.

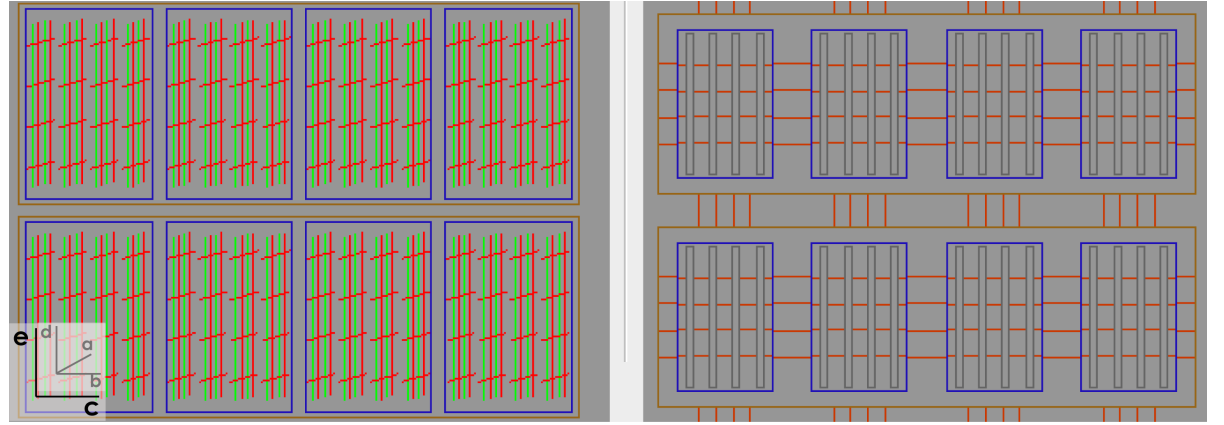
Taken together, the detailed and composite plane views display the traffic along all the dimensions. The views are synchronized, with the former detailing traffic along two dimensions while the latter summarizing it in the rest of the dimensions. Users can select the dimensions and their ordering in the two views.

### 4.1.3 Polygon view

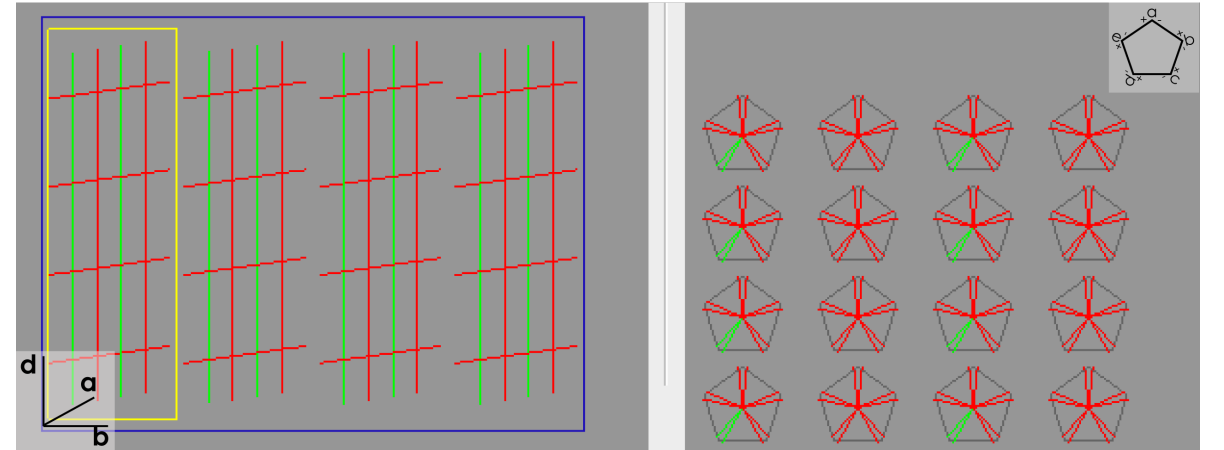
Users can magnify a plane by selecting it in the detailed plane view. In that case, the detailed plane view focuses on a single cuboid while the right side shows the polygon view. An example is shown in Figure 4.2b, where the  $(a \times b)$  planes are magnified. The right side of the figure depicts the polygon view, which shows node-level detail of the selected plane. Nodes are represented by polygons, with each corner representing one dimension of the network. The pair of lines originating from the center towards the corners display the incoming and outgoing traffic along that dimension.



(a) The detailed plane view showing detailed traffic inside the  $(b \times c)$  planes. The ordering of the dimensions is shown by the overlay.



(b) Changing the order of the dimensions to show the  $(a \times d)$  planes in the detail plane view.



(c) The polygon view, showing node-level details for the selected plane. The polygon view clearly shows alternating traffic along the  $d$ -dimension for even- and odd-ranked nodes.

**Figure 4.3:** Visualizing a synthetic example application occupying a  $(4 \times 4 \times 4 \times 4 \times 2)$  network section. The network traffic alternates along the  $d$  dimension for even- and odd-ranked nodes.

---

## 4.2 Example

We have developed a software prototype implementing our visualization technique. We present an example using a synthetically generated application. The application occupies a  $(4 \times 4 \times 4 \times 4 \times 2)$  section of a  $(a \times b \times c \times d \times e)$  dimensional torus network. Figure 4.3 visualizes the traffic of the example application.

In Figure 4.3a, the left side presents the detailed plane view, showing the traffic inside the  $(b \times c)$  planes. The links on the edges visualize the loop-back links. The planes are horizontally ordered, first along the  $a$  dimension inside the blue rectangles and then along the  $d$  dimension inside the brown rectangle. Dimension  $e$  is used for vertical ordering.

An overlay depicts the ordering of the dimensions. The nested dimensions (dimension  $b$ ,  $c$ , and  $d$  in our example) represent the planes and their ordering inside the blue rectangles, while the outer dimensions represent the ordering of the blue rectangles.

The composite view is shown on the right side of the figure. In the view, the ordering of the blue rectangles remain the same. The composite view summarizes the traffic along the  $c$ ,  $d$ , and  $e$  dimensions. In the view, we see uneven traffic along the  $d$  dimension.

We investigate the uneven traffic by changing the ordering of the dimensions to focus on  $(a \times d)$  planes, as shown in Figure 4.3b. In the detailed plane view, we see that the traffic along the  $d$  dimension alternates between even- and odd-ranked nodes. We can further zoom in on a specific plane by selecting it in the detailed plane view, as depicted in Figure 4.3c. The left side now shows the four  $(a \times d)$  planes of the  $(a \times b \times d)$  cuboid, while the right side shows the polygon view of the selected plane. In the view, the nodes are horizontally ordered along the  $a$  dimension and vertically along the  $d$  dimension. Each node is represented by a pentagon, and the pairs of lines show the traffic along each dimension. A overlay at the top right corner shows the dimension associated with each corner. The polygon view clearly shows alternating traffic along the  $d$  dimension for even- and odd-ranked nodes.

## 4.3 Interference identification

The technique presented in this chapter gives users a visual feedback about the traffic on torus networks. It can be used to identify hotspots and inter-application interference in the network. By distinguishing the applications of the processes that are executing on the compute nodes, it is possible to identify whether an application is surrounded by noisy neighbors and whether it is vulnerable to external interference. However, the degree of interference not only depends upon the communication characteristics of the applications, that is the frequency, volume, and type (point-to-point or collective) of the communication, but also on the how the processes

---

of the applications surround each other in the network and the degree of traffic on the links between them. Providing an intuitive visual feedback simplifies understanding the latter. Coupled with our method of globally synchronized time slices (presented in Chapter 1), it will be possible to establish simultaneity among network events and identify inter-application interference.

## 4.4 Conclusion

We presented a visualization technique based on simultaneous views to display traffic on high-dimensional torus network. The views are interactively synchronized and allow users to zoom into different sections of the network. Using a synthetic example application, we demonstrated how our technique can be used to find irregular traffic patterns. Our technique will help users visually locate network hotspots and identify inter-application interference on high-dimensional torus networks.

## 4.5 Contributions

The work presented in this chapter was described in the bachelor thesis of Lucas Theisen [127] conducted under my supervision, as described in Section 1.6. This work was also published in the proceedings of the first workshop on Visual Performance Analytics (VPA) [128]. I proposed the idea of visualizing high-dimensional torus networks through simultaneous view that are interactively synchronized. I further proposed displaying the torus network as a series of planes in detailed plane view, along with a summary in a separate view. Lucas worked on the implementation, and also came up with the concrete ideas for the summary in the form of the composite view and also contributed the polygon view. He also came up with other visualization techniques, which are covered in his thesis [127] and also in the mentioned paper [128]. Those techniques are not discussed in this dissertation.





---

## 5 Estimating the impact of interference

---

On many HPC systems, the execution time of applications can vary considerably between runs. The source of the variation can either be the operating system or inter-application interference caused by contention on shared resources by simultaneously running application. On such systems, measuring and analyzing the performance of applications can be challenging. To get an accurate picture of the performance of an application and to correctly identify bottlenecks, a performance measurement has to be as close as possible to the intrinsic behavior of an application. On systems that exhibit high run-to-run variation in application execution time, a measurement captures both the performance characteristics of an application and also the variation induced by external factors. However, for accurate analysis, the former has to be separated from the latter.

In common HPC applications, the natural performance characteristics remain the same in multiple executions, while the variation induced from external sources changes from run to run. Therefore, one approach to obtain a measurement as close as possible to the intrinsic behavior of an application can be to take multiple measurements, and pick the fastest run or the mean or median run, if a certain degree of interference is considered natural.

However, not only does this method consume more resources, but it is also inherently unreliable as the fastest run might also be interfered. After all, the system load also changes along macroscopic time scales (e.g., daytime or season).

To help performance analysts decide how much they can “trust” their benchmarking results and whether they need to repeat measurements, in this chapter, we present a novel approach to estimate the impact of external interference on the execution time of a common class of MPI applications. As a distinctive feature, our method can deliver such an estimate with negligible overhead based on a single run. Moreover, it is agnostic to the source of interference. Instead, it exploits the properties of bulk-synchronous MPI applications that perform frequent global all-to-all operations. Such applications not only make up a significant portion of HPC workload (almost two-thirds of unique benchmarks in the SPEC MPI2007 suite V2.0 [90] fall in this category), they are also the most sensitive to external interference [3, 40, 48].

---

## 5.1 Approach

Most HPC applications are iterative in nature. After a brief initialization, they go through different phases that are repeated over and over. Similar phases have similar execution times unless a phase instance is struck by external interference. The stronger the impact, the greater the elongation of the execution time.

Figure 5.1a shows a trace snippet of a typical HPC application. The application performs several iterations, whose execution times are, however, not uniform. Figure 5.1b presents a histogram of the execution times of the segments, showing variation in them. The variation can be either intrinsic or extrinsic. Intrinsic variation is the natural behavior of the application, as some iteration of the application perform extra work, like checkpointing or additional computation, and therefore have longer execution times. Extrinsic variation is the caused by external influences, such as inter-application interference. As the variations are not distinguished in the figure, it is not possible to elicit the natural behavior of the application. Figure 5.1a identifies two classes of iterations, A and B, based on distinguished by their programmatic characteristics and visible as two peaks in Figure 5.1b. Figure 5.1c and 5.1d show the execution time histograms of these segments. The variation that remains in these two histograms is extrinsic and caused by noise such as interference from other jobs executing at the same time.

The key is therefore to divide the execution of an application into segments and classify them based on their intrinsic characteristics. Without external interference, the segments in each class will have the same execution time. In other words, any variation seen in the execution time of segments belonging to a class is caused by extrinsic sources.

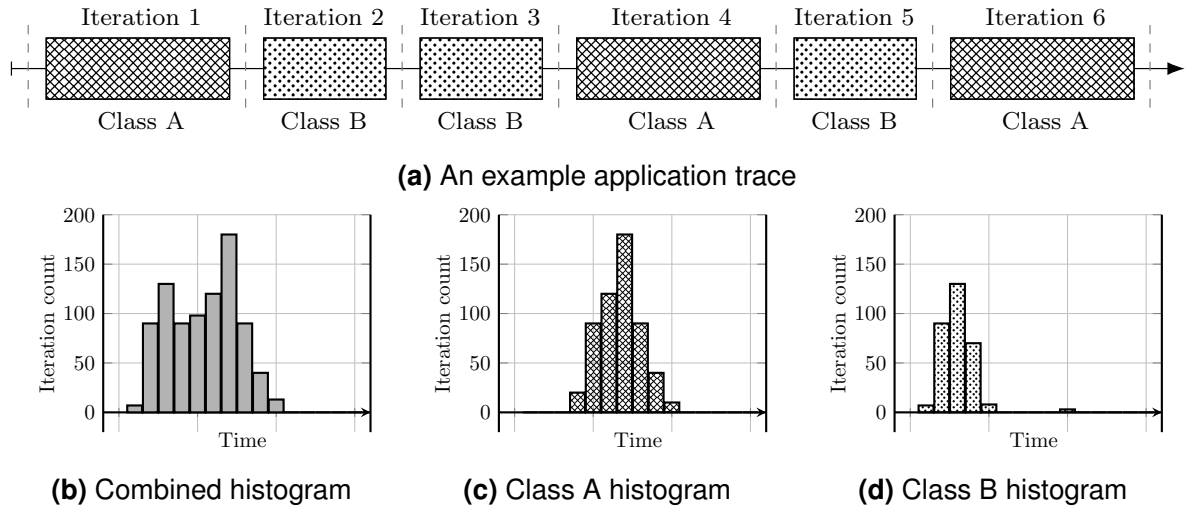
In our approach, we first identify different segments in an application and then classify them into different classes. As execution time of segments in a class can vary, we have found hardware and software counters that represent the computation, communication, and file I/O features suitable for classification of application execution segments.

### 5.1.1 Methodology

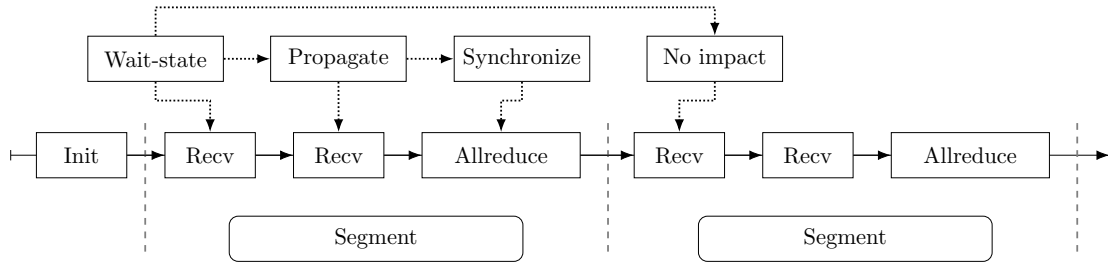
In this section, we present the concepts and methodology that form the basis of our approach.

#### Execution segments

To identify segments, we take advantage of the bulk-synchronous nature of many HPC applications, specifically we exploit periodic (blocking) all-to-all communication. Although this practically restricts our method to such applications, we claim that we can still cover



**Figure 5.1:** Application iterations and their histograms.



**Figure 5.2:** Execution segments and wait-state propagation.

major portions of today’s HPC workloads. After all, this is not an uncommon feature. For example, almost two-thirds of unique benchmarks in the SPEC MPI2007 suite fall into this category. At the same time, applications with frequent all-to-all communication suffer more than others from external interference because every delay of a process will likely induce waiting time in all others.

We use the global all-to-all communication operation as a boundary between execution segments. Segments identified this way might not exactly match the programmatically specified iterations of an application. Nonetheless, an all-to-all operation will appear at least once in the core loop of a bulk-synchronous application and will therefore identify repeated application execution segments. When multiple all-to-all operations appear in an iteration, the iteration will be split into multiple parts. Therefore, an all-to-all operation can be used to isolate repeating application execution segments.

Furthermore, such all-to-all operations constitute a global synchronization point among processes. While the MPI standard does not explicitly require an all-to-all synchronization, the nature of the operation leads to such a dependency [50]. As a result, the execution of a

---

bulk-synchronous application can be split into multiple segments, with each ending at an all-to-all operation. These segments will be independent of each other with respect to wait states that occur in response to external interference. That is, a wait state whose root cause lies within a segment will not propagate across a global synchronization point [11], as shown in Figure 5.2.

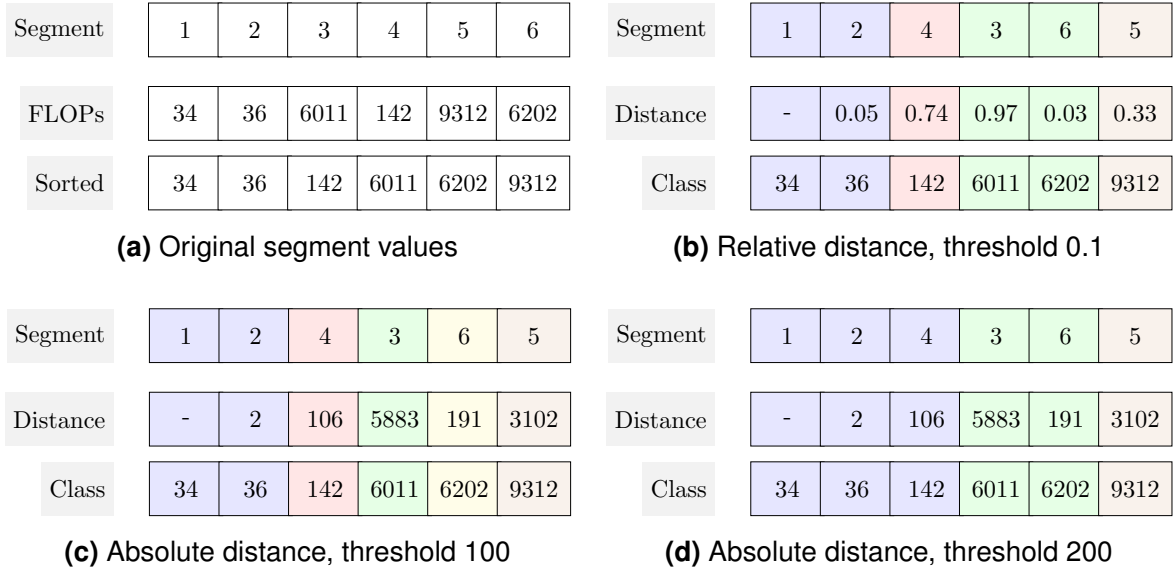
Due the above two reasons, application execution segments can be analyzed independently of each other, and at the same time provide a representation of repeated application segments. We therefore use them as a basic unit of our analysis.

For applications using non-blocking collectives, the wait operation of the collective call can be used as a boundary indicator. For non-bulk-synchronous applications, (similar to the approach of ScalaTrace [92]) recurring MPI calls can be used to identify repeating application areas that represent execution segments. However, in both cases, adjacent application segments may not be fully independent, with wait-states and interference-induced delays potentially propagating across segment boundaries. We therefore concentrate on bulk-synchronous applications with blocking collectives in this study and consider the remaining cases as future work.

## **Clustering segments based on computation features**

To classify segments, we first compare them in terms of their computation work. We use the floating-point instructions hardware counter to measure the amount of work done. When the floating-point counter is not available, such as on Intel Haswell processors [101], we use the total number of completed instructions as a proxy. To focus solely on the computation parts, we discard the communication and I/O operation parts of a segment when counting the number of instructions. While the captured values are still perturbed by OS jitter, we have found floating-point operations to be the most stable, whereas the total instruction completed count shows less than 1% variability.

We cluster segments based on the above mentioned counters to establish similarity among them. As the duration of segments in an application can vary widely, the possible range of feature values can be quite large. Furthermore, OS jitter and inaccuracies introduced when reading and storing hardware counters cause variation among hardware-counter values of similar segments [32]. Therefore, the most appropriate clustering algorithm for our task needs to handle a large range of values, and at the same time be tolerant to variations inside a cluster.



**Figure 5.3:** Clustering of segments based on relative and absolute thresholds.

## Common clustering algorithms

Common clustering algorithms such as K-means require the number of clusters to be known *a priori*. If such information is not available, such algorithms are executed for a range of cluster counts and an internal cluster criterion, such as the Calinski and Harabasz (CH) criterion [16], is applied to identify the most appropriate number of clusters. Even for a particular number of clusters, these clustering algorithms require several iterations to find the optimum centroids. These factors result in algorithms that, overall, are complex to implement and can take a significant amount of time for large numbers of data points.

Density-based algorithms such as DBScan [34] seem to present an alternative. They can identify the appropriate number of clusters in a single pass. Such algorithms use a distance threshold to split the data points into clusters. However, relying on a fixed distance for a large range of values results in either merging distinct clusters with lower values if the threshold is too large, or splitting a single cluster with a modest range of higher values into multiple clusters if the threshold is too small.

## Clustering with relative distance

To overcome these difficulties, we designed a simple clustering algorithm that can identify clusters in one-dimensional data even with a large value range in a single pass. The algorithm uses the same principles as density-based algorithms, but relies on *relative* distance between any two data points. The algorithm requires the data type to have a total order and a

---

threshold for the maximum relative distance between any two data points in a cluster. We define the relative distance between two points as subtracting one from the ratio between the the larger and the smaller distance of the two points from the origin. As the algorithm relies on relative distance, it can identify clusters with a modest degree of internal variance both at the lower and higher end of the value range.

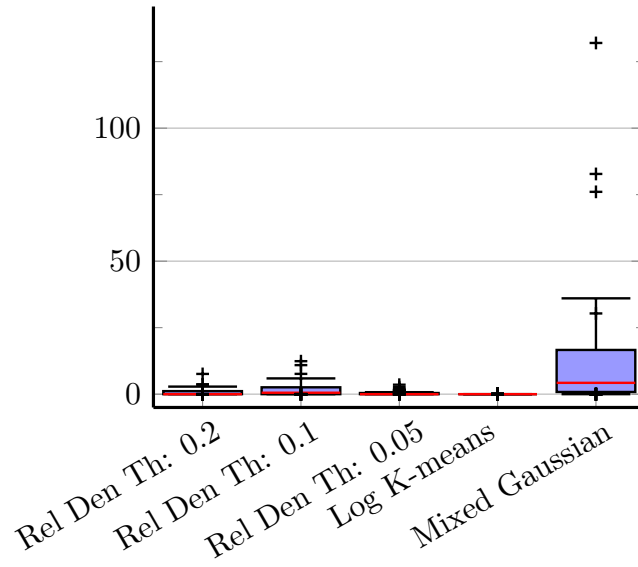
Figure 5.3a illustrates the challenge of clustering segments with a high value range and variation among similar segments. In the figure, we have the floating-point operations (FLOP) count of six segments which we wish to classify. For clarity, the figure also shows the FLOP count in an ascending order. To a human observer, it would seem natural to classify segments 1 and 2 together, and segments 3 and 6 together. Segments 4 and 5 would seem to form their own clusters. Now we classify them based on relative and absolute distance, as shown in Figures 5.3b to 5.3d.

In each figure, the bottom row shows the segment values in an ascending order. The segments are grouped into clusters, identified by different colors. The second from bottom row shows the distance (relative or absolute) between consecutive segments. In Figure 5.3c and 5.3d, we see that the absolute distance between segments 2 and 4 is 106 and it seems natural to classify them into separate clusters. On the other hand, segments 3 and 6 have a distance of 191 and should be clustered together. With absolute distances, it is not possible to having such a clustering. Figure 5.3c shows clustering based on absolute distance with a threshold of 100. In the figure, segments 2 and 4 are clustered separately, but at the same time, segments 3 and 6 are also separated. In Figure 5.3d, the threshold is increased to 200. Now segments 3 and 6 are not clustered together, but so are segments 2 and 4. On the other hand, relative distance can easily cluster the segments properly, grouping together segments 3 and 6, and separating segments 2 and 4, as shown in Figure 5.3b. Hence, for segments with a large value range and a modest degree of internal variation, relative distance can cluster segments intuitively.

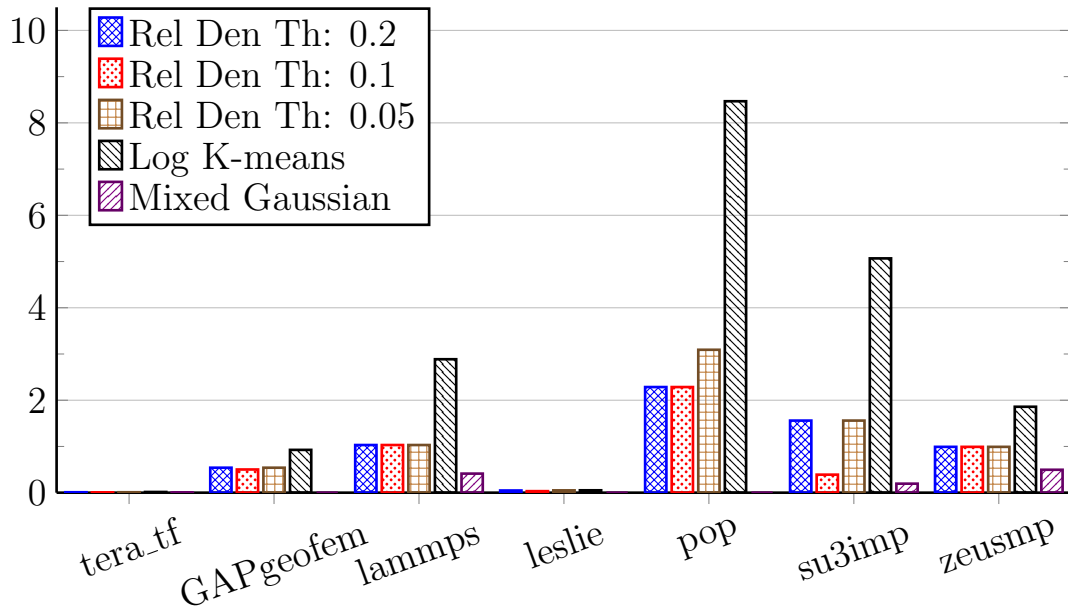
## Clustering algorithm

Our relative-distance-based algorithm first sorts all the values in an ascending order and then assigns the smallest element to the first cluster. After that, it iterates through the remaining sequence and, at each step, picks the value at position  $i$  from the sorted list that was assigned to a cluster in the previous step and determines the relative distance to the next value at  $i + 1$ . If the relative distance is less than the threshold, the value at  $i + 1$  is placed in the same cluster as the value at  $i$ . Otherwise, a new cluster is created for the value at  $i + 1$ .

We evaluated the relative density based clustering algorithm and compare it with the K-means algorithm and the Expectation-Maximization (EM) algorithm that assumes the



(a) Standard deviation as a percentage of mean, showing the spread of the clusters.



(b) Number of small clusters.

**Figure 5.4:** Evaluation of clustering algorithms using SPEC MPI2007 benchmark suite.

data to exhibit a mixed Gaussian distribution. For the relative-density-based algorithm, we used thresholds of 0.2, 0.1, and 0.5. For the K-means and the mixed Gaussian method, we generated clusters with total cluster counts of 2 - 40 and used the CH Index to pick the best fitting cluster count. The total instructions completed hardware counter was used to cluster the bulk-synchronous benchmarks of SPEC MPI2007 suite. The algorithms were



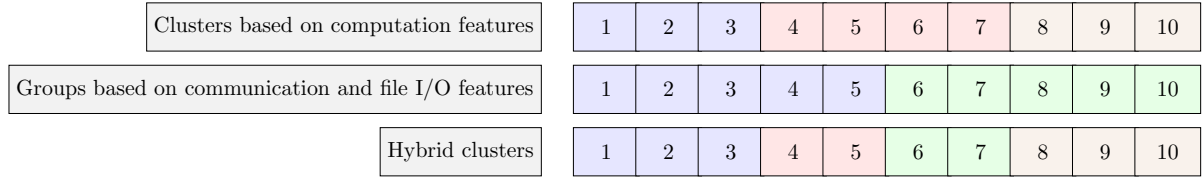
| Segments               | 1   | 2   | 3   | 4   | 5 |
|------------------------|-----|-----|-----|-----|---|
| File-I/O bytes read    | 25  | 10  | 10  | 25  | 1 |
| File-I/O bytes written | 120 | 90  | 90  | 120 | 0 |
| Point-to-point calls   | 35  | 30  | 30  | 35  | 0 |
| Point-to-point bytes   | 250 | 280 | 280 | 250 | 0 |
|                        | 1   | 4   |     |     |   |
| Groups                 | 2   | 3   |     |     |   |
|                        |     |     | 5   |     |   |

**Figure 5.5:** Grouping of segments based on communication and file I/O features.

evaluated by comparing the mean normalized standard deviation of the created clusters and by comparing the percentage segments that were not clustered. We consider clusters with less than five segments as unclustered. Figure 5.4 shows the results of the evaluation. The K-means algorithm created clusters with the least standard deviation, closely followed by the relative-density-based algorithm at thresholds of 0.05 and 0.1, while the mixed-Gaussian-based algorithm created clusters with very high standard deviation. On the other hand, the K-means algorithm did not cluster almost twice the number of segments compared to the relative-density-based algorithm, while the mixed-Gaussian-based algorithm had the least unclustered segments. Based on the two criteria, we found the relative-density-based algorithm at a threshold of 0.1 to provide the best results.

### Grouping segments based on communication and file I/O features

As communication and I/O features of a segment, we consider the number and the accumulated volume of communication and I/O operations, including the number of point-to-point send/receive calls broken down by their blocking semantics, the number of collective calls broken down by their number of senders vs. recipients, and the number of bytes sent or received through them. Similarly, as file-I/O features we capture the number of open/close operations, the number of read/write operations, and the accumulated number of bytes read or written. Since there is no clear relationship between these metrics and the execution time of a segment, we consider the corresponding values as nominal data, that is the execution time of segments cannot be ordered based on these metrics. For example, a segment may run longer than another segment, although its number of sends is smaller. At the same time, these metrics are fairly stable and usually not subject to any jitter. Thus, we consider all



**Figure 5.6:** Combining groups based on communication and file I/O features, and clusters based on computation features into hybrid clusters.

segments that share the same unique combination of communication and file I/O metrics a separate group. Figure 5.5 shows five segments, and their communication and file-I/O features. Segments 1 and 4 have the exact same features and form a group. Segments 2 and 3 are also grouped together, while segment 5 forms a separate group. In some applications, different processes can send/receive different number of bytes in a segment, which leads to many small cluster of segments. In such cases, either taking the median number of bytes among the processes, or completely discarding the bytes metric results in large clusters with small variation.

### Combining clusters and groups

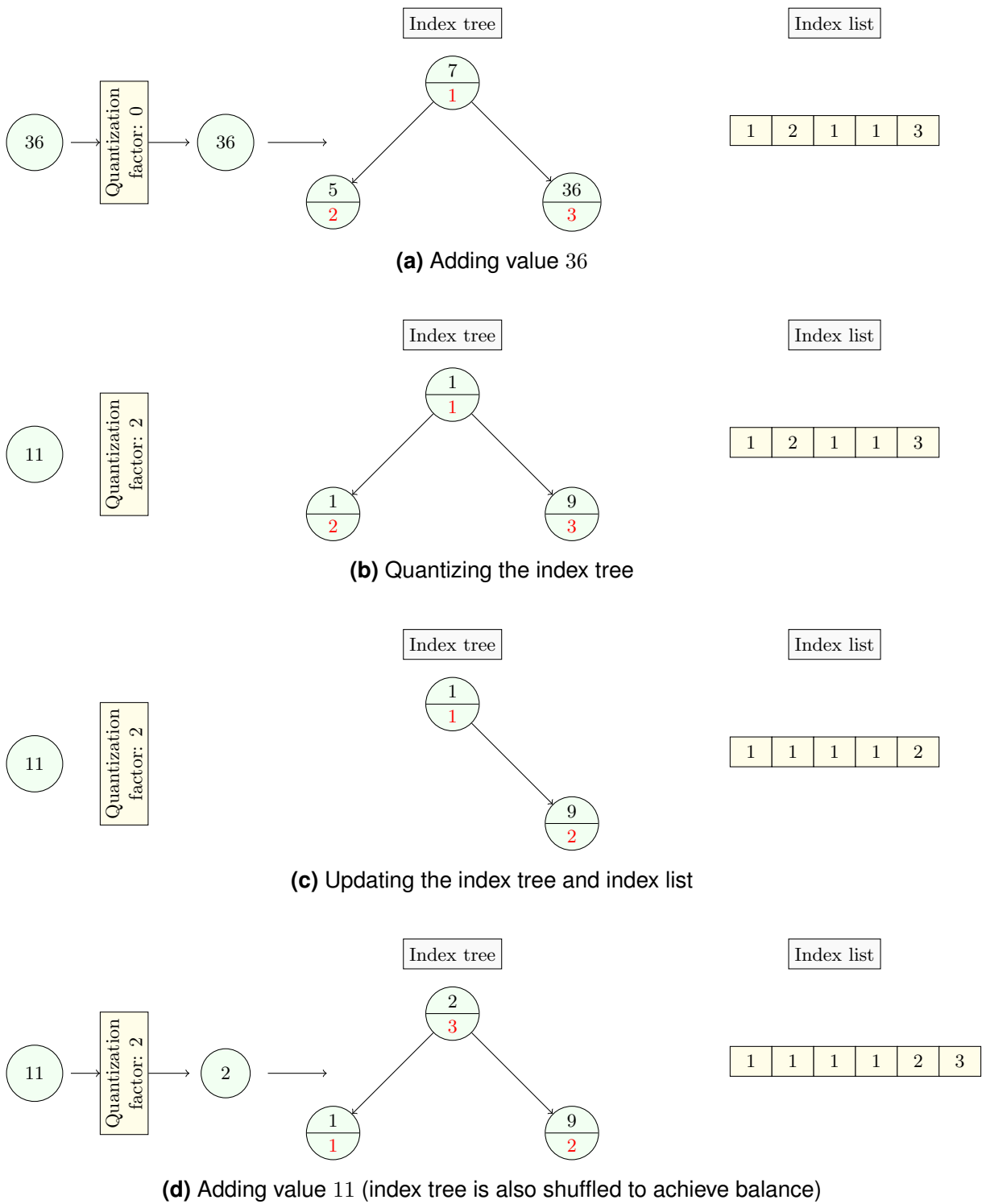
In our methodology, we first cluster segments based on their computation features, and separately group them based on their communication and file-I/O features. Later, we combine the clusters and groups into hybrid clusters, as shown in Figure 5.6. Segments belonging to the same computation cluster but to different communication/file-I/O group (or vice versa) are split up into different hybrid clusters. As a result, a hybrid cluster has segments that have similar computation, communication, and file-I/O features.

## 5.1.2 Profiling methodology

To capture application segments and their features, we use LWM<sup>2</sup> [115], a low-overhead profiler which is described in Chapter 2.

### Capturing segments and metrics

For this study, we extended LWM<sup>2</sup> to generate profiles at the granularity of the execution of application segments. The end of an all-to-all collective call is used as the end of a segment profile and the start of a new one. `MPI_Init` is considered as the start of the first segment, while `MPI_Finalize` is considered as the end of the last segment. When an application executes, LWM<sup>2</sup> records the previously described computation, communication, and file I/O



**Figure 5.7:** An example demonstrating data compression at runtime in LWM<sup>2</sup>. The index tree has a maximum size limit of 3 nodes

---

features. The metrics are recorded on a per-process basis. When a segment finishes, an in-memory profile of the segment is generated.

## Storing profiles in memory

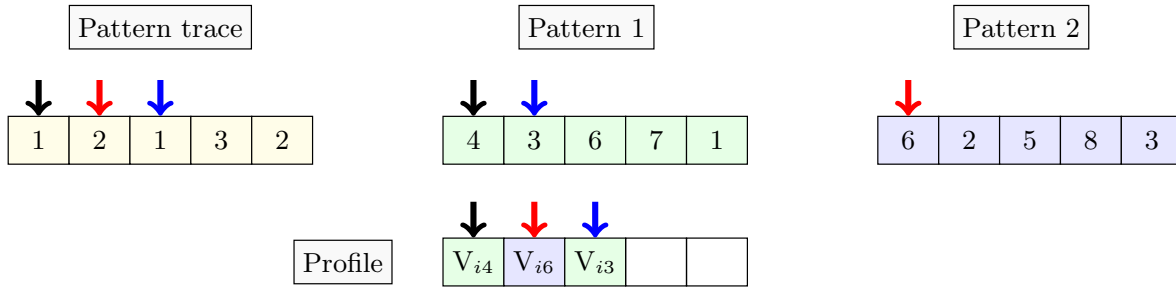
To reduce the memory footprint of the profiles, the individual feature values are stored in memory in the form of an index tree and an index table. The index tree is an auto-balancing AVL tree that records all the unique values of the feature. The index list stores the feature values by referencing nodes in the AVL tree. The index list also maintains the order in which the values were captured. It is important to maintain the order of arrival as it enables us to combine the values of individual features into a complete profile. When a new metric value is captured, it is added to the index tree. The node is assigned an index value, which is appended to the index list.

LWM<sup>2</sup> further reduces the memory footprint of the profiles by limiting the unique values in the index tree to 256. When the number of unique values exceeds the threshold, the values are quantized, that is, values with small difference are grouped together by dividing them by two, until there are at most 256 unique values. Similar values in the index tree are merged. The indexes assigned to the nodes are also updated, both in the index tree and in the index list. The number of division operations is also recorded.

Figure 5.7 presents an example of storing feature values. In the example, the maximum size of the index tree is limited to three. In the nodes of the tree, the feature value is shown in the upper part, while its index in the lower part. In Figure 5.7a, a feature value 36 is added. As there is no need for quantization at this point, the value 36 is added as is to the index tree and assigned an index of 3. The index 3 is also appended to the index list. In Figure 5.7b, a new value 11 is captured. As the index tree has reached its maximum size, the values in the tree are quantized until there are less than three unique values. After dividing the values twice by two, the number of unique values is reduced to two. In Figure 5.7c, nodes with similar values are merged in the index tree. The indices are also updated in the index tree and the index list. Now, the new value 11 can be added, as shown in Figure 5.7d. After quantization, the value is stored as two. Note, the index tree is also shuffled to achieve balance between its subtrees.

## Distinguishing patterns at runtime

Different segments in an application run can have widely different feature values. If the feature values of all the segments are stored together, because of quantization, variation among features values that are important to distinguish different patterns can be lost. To



**Figure 5.8:** Reconstructing profile from data on disk.

avoid such a scenario, at runtime, LWM<sup>2</sup> groups profiles based on their communication and file-I/O features. Other than the numbers of bytes communicated or read/written, LWM<sup>2</sup> uses the same feature set as described in Section 5.1.1 to create groups of segments on the fly. Feature values for each group are then stored separately. LWM<sup>2</sup> also records a separate trace that captures the order in which segments belonging to different groups execute at runtime.

### Merging per-process feature values

At the end of the execution, the per-process groups are merged into application-wide groups. The trace recording the order of the groups is also merged into an application-wide trace. Similarly, the index trees of individual features in the groups are also merged into application-wide index trees with a maximum of 512 unique values. If there are more than 512 unique values, they are merged by dividing the values by two until at most 512 unique bins remain, similar to the process described before. Node indices are also regenerated. Then, the index list is first updated on every process and later combined together into an application-wide index list. The combined data is written to disk. Interference is estimated postmortem.

### Reconstructing a profile from data on disk

In a profile, features of a segment are recorded and stored separately. The segments themselves are grouped separately based on their pattern. While such a structure reduces storage requirements, it complicates the recreation of a profile from data. We explain the reconstruction using the example in Figure 5.8. First, we read the pattern trace, which records the order of execution of different patterns. In the figure, the first entry in the trace is 1, indicating the first pattern. We therefore access the index list of pattern 1 and read its first entry. The value 4 in the list indicates value at that index in the index tree. We fetch the corresponding value (which is  $V_{i4}$  in our example) and load it into the profile. Similarly, the second entry in the pattern trace points to the second pattern. Following the same procedure, the value  $V_{i6}$  is loaded into the profile. When pattern 1 is read the second time in the pattern trace, we fetch

---

the value pointed to by the second entry in its index list (value  $V_{i3}$  in our example) and load it into the profile. This process is continues until the whole pattern trace file is read.

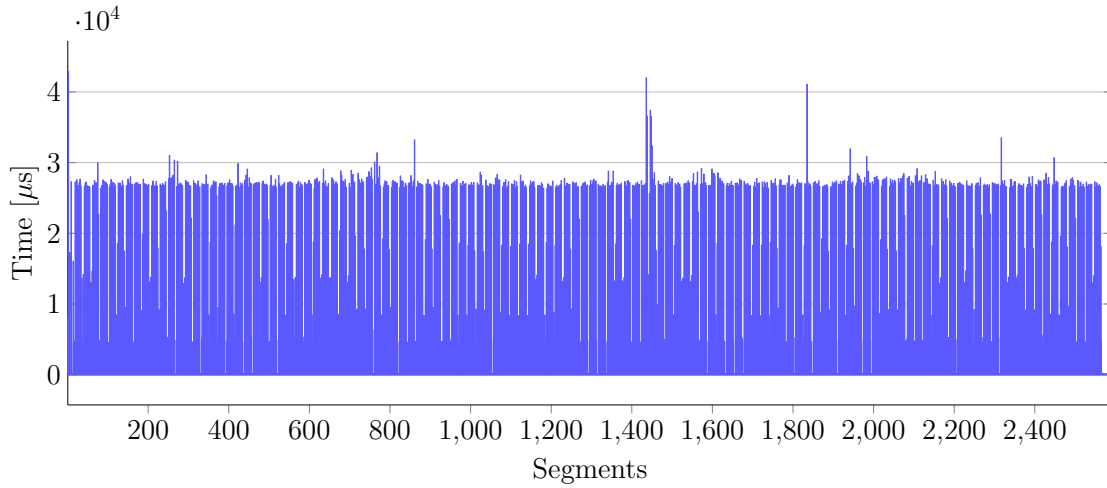
### 5.1.3 Estimating interference

We estimate the impact of interference based on the segmented profile of a single run. First, we cluster the segments according to their computation features, and group them according to their communication and file-I/O features. Afterwards, we combine the clusters and groups into hybrid clusters, as described earlier. The segments in each of the resulting clusters are assumed to exhibit similar behavioral characteristics and consume about the same intrinsic execution time.

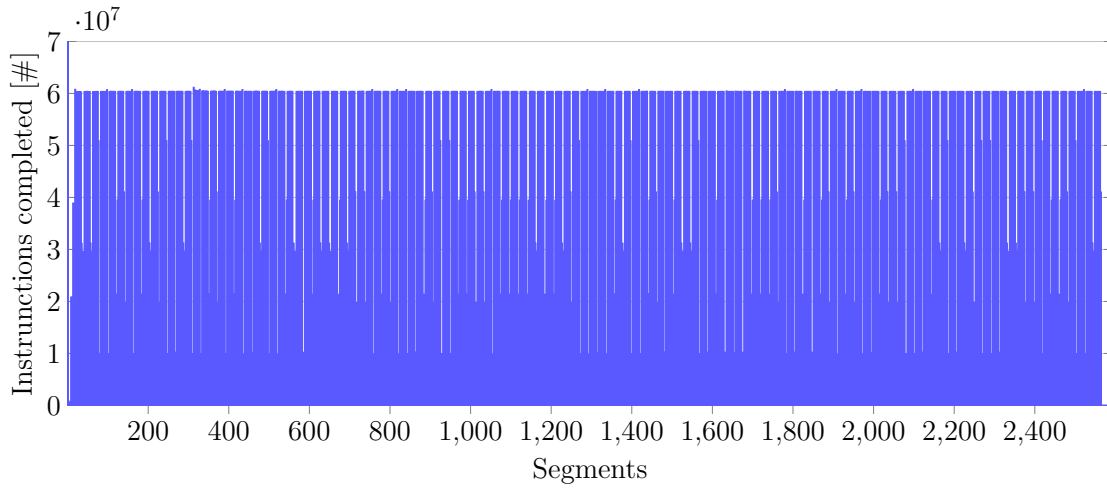
Any segment in a group that has a significantly higher execution time is considered to be affected by interference. More precisely, we classify a segment as interfered if its execution time is four MAD greater than the median of the group, with MAD (Median Absolute Deviation) being  $MAD = \text{median}(|X_i - \text{median}(X)|)$ . Median and MAD are known for their robustness to variability. The threshold of four MAD greater than median gives a confidence interval of more than 99.5%. The impact of interference on a segment is estimated as the portion of execution time of the segment in excess of the threshold. Adding the interference impact computed for all segments yields the interference impact for the entire program and is provided as a percentage of the (interfered) execution time.

#### Separating instantaneous interference from continuous interference

Execution time variation can arise from either high-frequency but usually low-impact interference such as certain types of OS jitter or from low-frequency but often high-impact interference such as sudden I/O contention. We call the former kind continuous interference and the latter kind instantaneous interference. Continuous interference affects almost all segments of a profile and as a result also affects the median in a group. In contrast, instantaneous interference only affects selected segments and the median remains largely unaffected. While both kinds of interference prolong execution time, instantaneous interference is more likely to create undesirable artifacts in performance measurements a performance analyst may wish to remove. In contrast, continuous interference is often seen as an unavoidable evil one has to live with on a given system. Our approach only reports instantaneous interference as it classifies segments as interfered based on median and MAD. Under continuous interference, the medians in segment profiles are also displaced which ensures that it leaves no imprint on our estimates.



(a) Duration of the segments

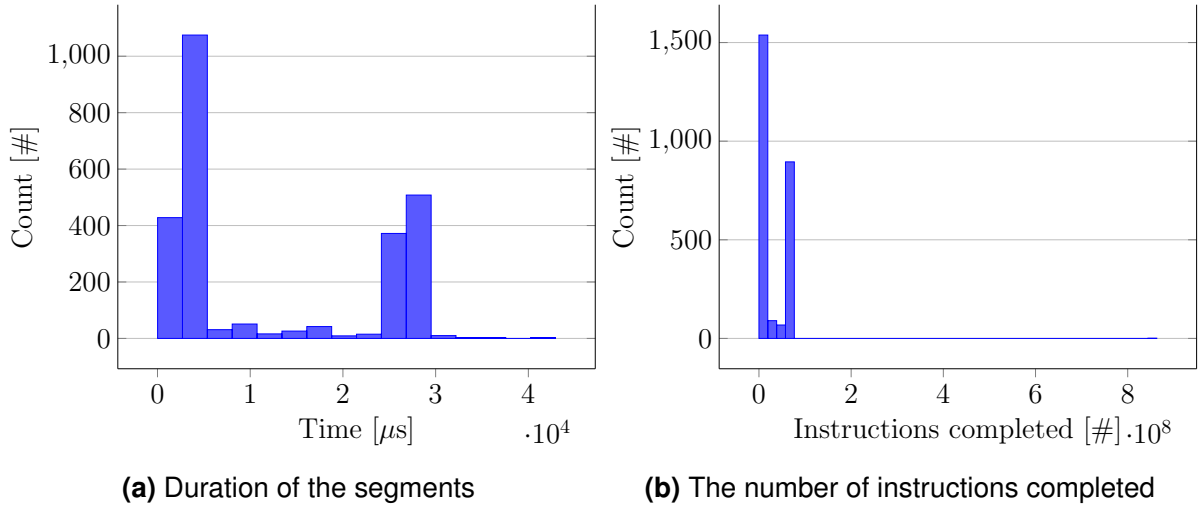


(b) The number of instructions completed in the segments

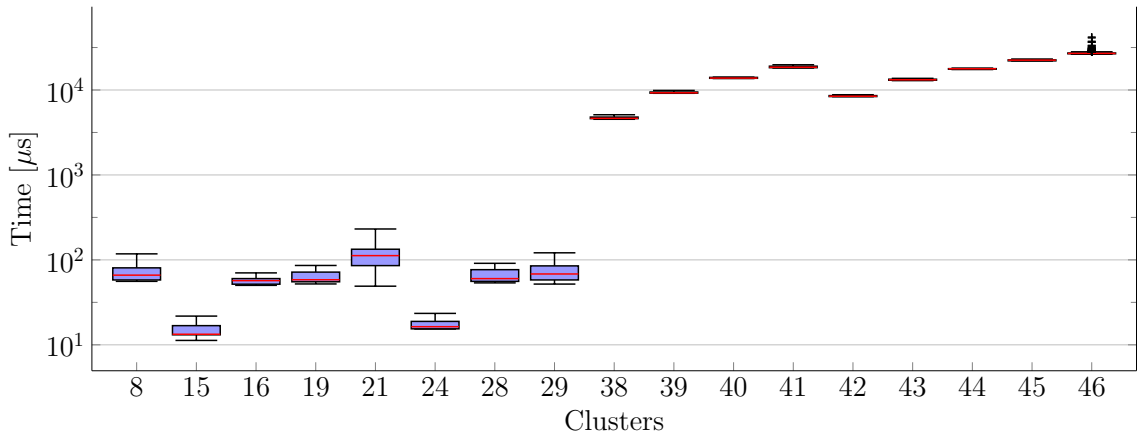
**Figure 5.9:** A timeline of a run of lammps on Hazel Hen. In the timeline depicting the number of completed instructions, the first segment completes more than  $8.5 \times 10^8$  instructions. However, the y-axis is cut at  $7 \times 10^7$  to show the variation among the rest of the segments.

#### 5.1.4 Interference estimation in depth

We take a deeper look at how interference is estimated in our method. We first analyze a run of lammps, a benchmark in the SPEC MPI2007 suite, and demonstrate how to extract the intrinsic behavior of an application. We then briefly compare two runs of Sweep3D to verify our estimated impact of external interference. Both of the applications were executed on Hazel Hen, a Cray XC40 machine, using 16 nodes, with 24 processes on each node.



**Figure 5.10:** A histogram of lammps showing the duration and the number of instructions completed in the segments. The histogram of completed instructions is skewed towards lower values, that is the bars are concentrates on the left side, as the first segment completes more than  $8.5 \times 10^8$  instructions.

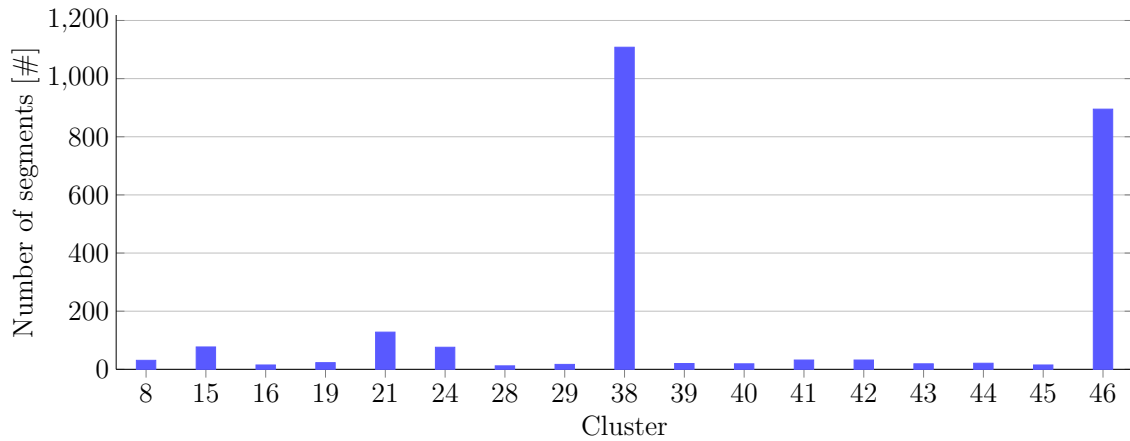


**Figure 5.11:** A boxplot showing the spread of the identified clusters. The y-axis is logarithmic in scale. Clusters with less than 5 segments are not shown.

### Extracting intrinsic behavior

Figure 5.9 presents the timeline of the run. The x-axis represents all the segments of the run, sorted in the order of their occurrence. In Figure 5.9a, the y-axis represents the duration of the segment in microseconds, while in Figure 5.9b, the y-axis represents the number of instructions completed in the segment. We do not consider communication and file I/O operations when counting the number of instructions completed. The figures show that the duration of the segments (as well as the numbers of completed instructions) vary significantly,





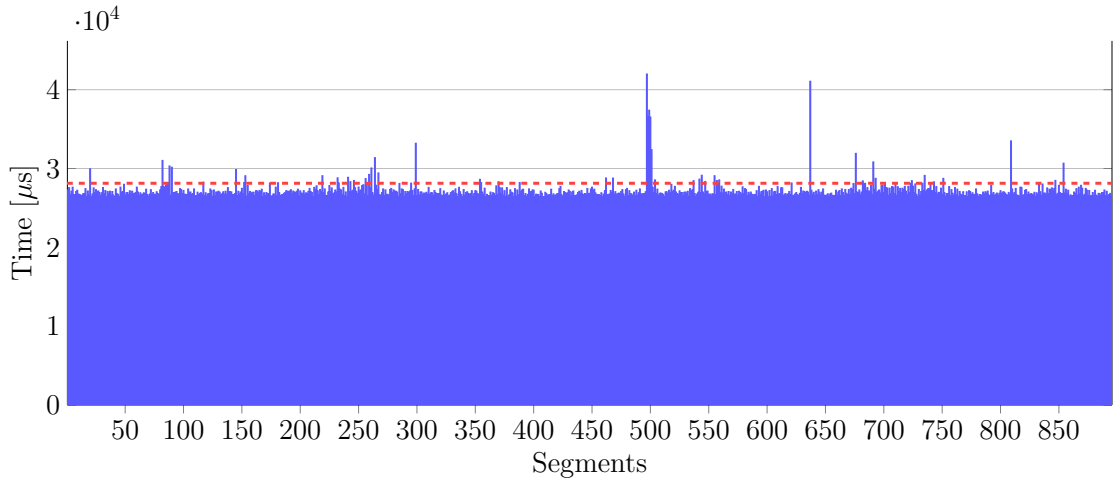
**Figure 5.12:** The number of segments in the identified clusters. Clusters with less than 5 segments are not shown.

making it challenging to identify any groups of similar segments.

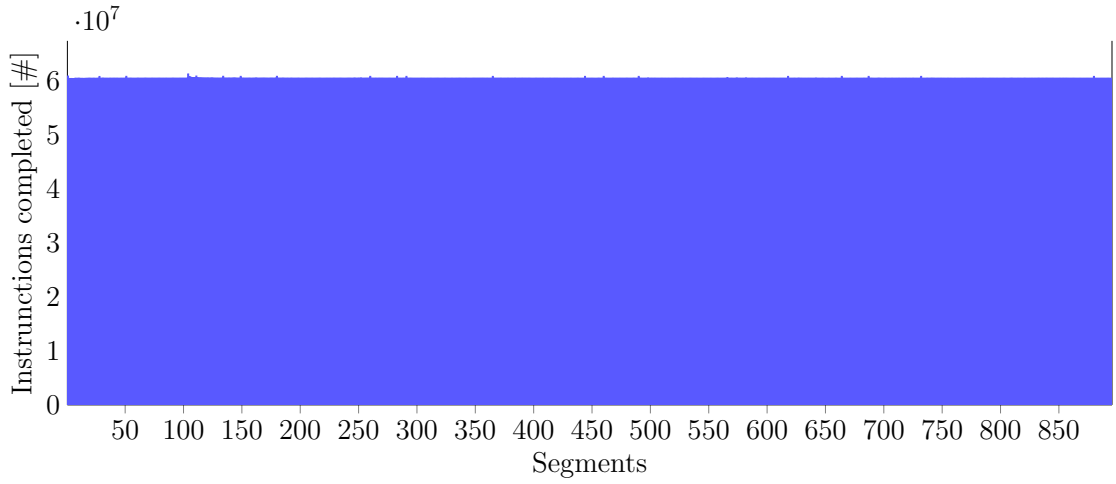
Figure 5.10 shows the histogram of the two metrics. The histogram of segments duration in Figure 5.10a has two peaks, indicating a bi-modal distribution of segments. At the same time, it is not possible to distinguish whether the segments in the bins other than the peak are interfered or that they are exhibiting their natural variation. The histogram of the number of instructions completed (shown in Figure 5.10b) also suggest a bi-modal distribution. However, there are a number of segments that fall in the bins between the two peaks, suggesting that those segments are not interfered and are exhibiting their intrinsic variation. The challenge now is to identify interfered segments in the presence of such intrinsic variation.

Figure 5.11 presents the clusters of segments identified by our approach. A boxplot is used to show the spread of the clusters. In the figure, the y-axis represents time in microseconds and is logarithmic in scale. Figure 5.12 plots the size of each cluster, that is the number of segments belonging to that cluster. Clusters with less than 5 segments are considered as noise and are not shown. The two figures identify two large clusters, 38 and 46, confirming the bi-modal distribution. They also identify other clusters will small number segments. These are the segments that have intrinsic variation in their behavior. The figures also reveal outliers in Cluster 46.

In Figure 5.13 and 5.14, we take a deeper look at Cluster 46. The timeline of the duration of the segments show that, except for a few, most of the segments have similar execution time. On the other hand, in the timeline of the number of instructions completed, all segments have similar count, even those that take longer time to execute. Based on this difference, we conclude with confidence that the elongated time of the few segments is caused by external interference.



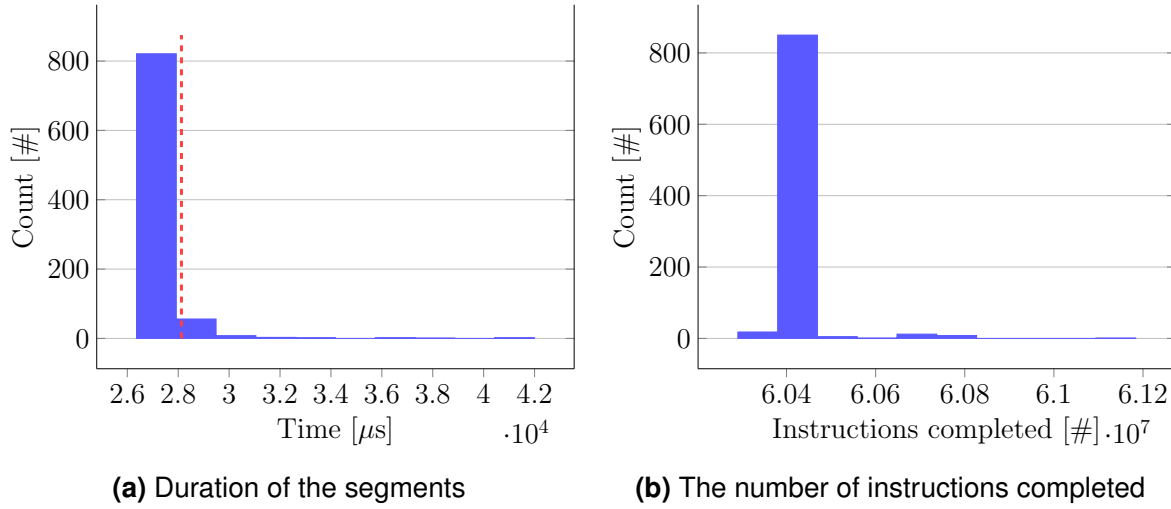
(a) Duration of segments belonging to cluster 46



(b) The number of instructions completed in segments belonging to cluster 46

**Figure 5.13:** Timeline of Cluster 46. Some segments have longer execution time and are considered to be impacted by external interference. The dashed red line shows the threshold beyond which segments are considered interfered.

We apply our method to estimate the impact of interference. Using the median and the MAD of the cluster, we derive the threshold beyond which a segment is considered interfered. It is represented by the dashed red line in Figure 5.13a and 5.14a. The impact of interference is estimated in Table 5.1. 58 segments out of 853 are interfered while the impact is 178 milliseconds.



**Figure 5.14:** Histograms of Cluster 46 of lammps, showing the duration and the number of instructions completed in the segments. The dashed red line shows the threshold beyond which segments are considered interfered.

| Median [ $\mu s$ ] | MAD [ $\mu s$ ] | Interference threshold [ $\mu s$ ] | # of segments above threshold | estimated interference [ $\mu s$ ] |
|--------------------|-----------------|------------------------------------|-------------------------------|------------------------------------|
| 26951              | 294.14          | 28127.48                           | 58                            | 178221                             |

**Table 5.1:** The estimated impact of interference on segments in Cluster 46.

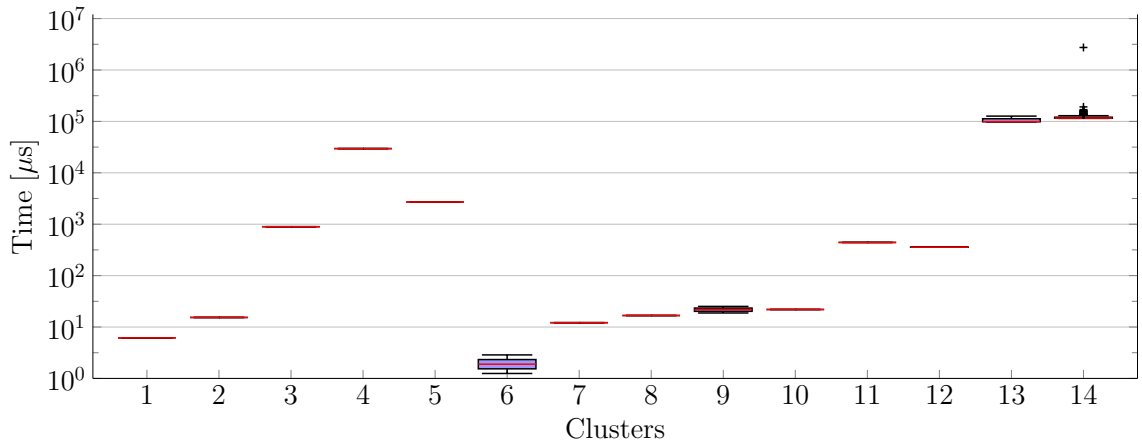
### Verifying our estimate of interference

We compare two runs of Sweep3D, listed in Table 5.2. Run 1 took 326 seconds to finish while run 2 took 293 seconds. As run 2 has shorter execution time, we take it as the baseline run, even though we cannot be sure whether the run itself was interfered or not. Taking run 2 as the baseline, run 1 has a measured interference of 33 seconds. Now we apply our method to estimate the impact of interference.

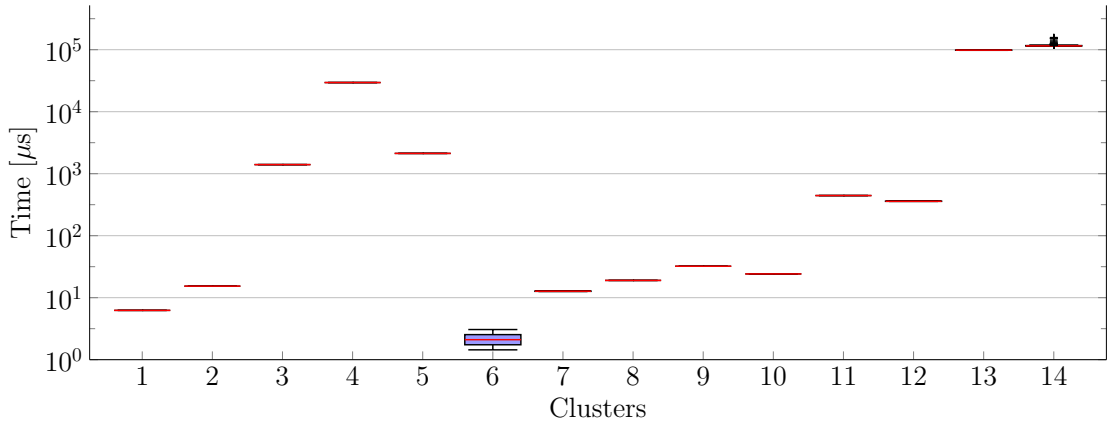
We first look at the identified clusters of segments of the first run, which are shown in

| Run | Exec time of run [s] | measured interference [s] |
|-----|----------------------|---------------------------|
| 1   | 326                  | 33                        |
| 2   | 293                  | zero                      |

**Table 5.2:** Comparing two runs of Sweep3D on Hazel Hen. Run 1 is interfered and has a runtime dilation of 33 seconds.



(a) Run 1



(b) Run 2

**Figure 5.15:** Comparing the identified clusters of segments of two runs of Sweep3D. Run 2 took longer to finish compared to run 1. Cluster 14 of run 2 has an outlier which took significantly longer to finish.

Figure 5.15a. Cluster 14 has an outlier segment which took significantly longer time to finish. As the identified clusters in a run are supposed to have similar execution time, our method considers this outlier segment as interfered. For confirmation, we look at the identified clusters in the second run in Figure 5.15b. All clusters are similar between the two runs, except for Cluster 14, which does not have the outlier segment in the second run. Its absence indicates that its deviation in the first run is not an intrinsic behavior but rather the result of external influence.

We estimate the impact of interference for both runs in Table 5.3. For simplicity, we focus on Cluster 14 only. It is one of the two large clusters in the run, its segments have the longest execution time, and it also has the large outlier segment in the first run, indicating that it should account for most of the deviation in execution time. We estimate the impact

---

---

| Run | Median<br>[s] | MAD<br>[s] | Interference<br>threshold [s] | # of segments<br>above threshold | estimated<br>interference [s] |
|-----|---------------|------------|-------------------------------|----------------------------------|-------------------------------|
| 1   | 1.16          | 0.012      | 1.21                          | 49                               | 35                            |
| 2   | 1.15          | 0.005      | 1.17                          | 45                               | 5                             |

---

**Table 5.3:** The estimated impact of interference for Cluster 14 of two runs of Sweep3D. Run 2 is impacted to a higher degree. The median of the cluster remain stable and is similar for both the runs.

of interference for the first run as 35 seconds, while for the second run as 5 seconds. The measured interference for run 1 is 33 seconds. As run 2 is taken as the baseline when calculating the measured interference, it is assumed that run 2 is interference free. The result show that our method is able to estimate the impact of interference, just from a single run, up to a high degree of accuracy. Our results also suggest that run 2 is also interfered to a small degree, which can be verified by taking more runs. Another aspect can be that a certain degree of interference is unavoidable on Hazel Hen.

## 5.2 Evaluation

To evaluate our approach, we use the following benchmarks:

1. Those seven codes from the SPEC MPI 2007 V2.0 suite that are bulk-synchronous according to our definition and that have a large data set available.
2. Sweep3D, a time-independent 3D neutron transport simulation.
3. HACC, an application that simulates the formation of collision-less fluids and whose regular checkpointing behavior makes it a popular I/O benchmark.

Table 5.4 lists the collective call rates of the benchmarks, that is the frequency at which the benchmarks perform the MPI collective operation. We test our method both in a controlled environment with artificially injected interference, and on a production system with real interference.

### 5.2.1 Experimental setup

Because of its low OS jitter, we chose JUQUEEN, an IBM Blue Gene/Q system, as our controlled environment [62]. JUQUEEN was decommissioned on May 24, 2018. It housed 28,672 compute nodes, with each consisting of a 16 core IBM PowerPC®A2 processor and 16 GB of

| Collective call rate (Hz) |           |        |        |        |      |         |         |      |
|---------------------------|-----------|--------|--------|--------|------|---------|---------|------|
| Tera_tf                   | GAPgeofem | Lammps | Leslie | POP    | MILC | Sweep3D | Zeusmp2 | HACC |
| JUQUEEN                   |           |        |        |        |      |         |         |      |
| 1.14                      | 305.52    | 64.31  | 1.71   | 42.27  | 0.54 | 0.30    | 0.11    | 0.45 |
| Hazel Hen                 |           |        |        |        |      |         |         |      |
| 15.25                     | 4848.50   | 71.73  | 8.70   | 339.65 | 1.66 | 15.0    | 0.98    | 32.4 |

**Table 5.4:** Collective call rate of bulk-synchronous SPEC MPI2007 benchmarks on JUQUEEN and Hazel Hen systems.

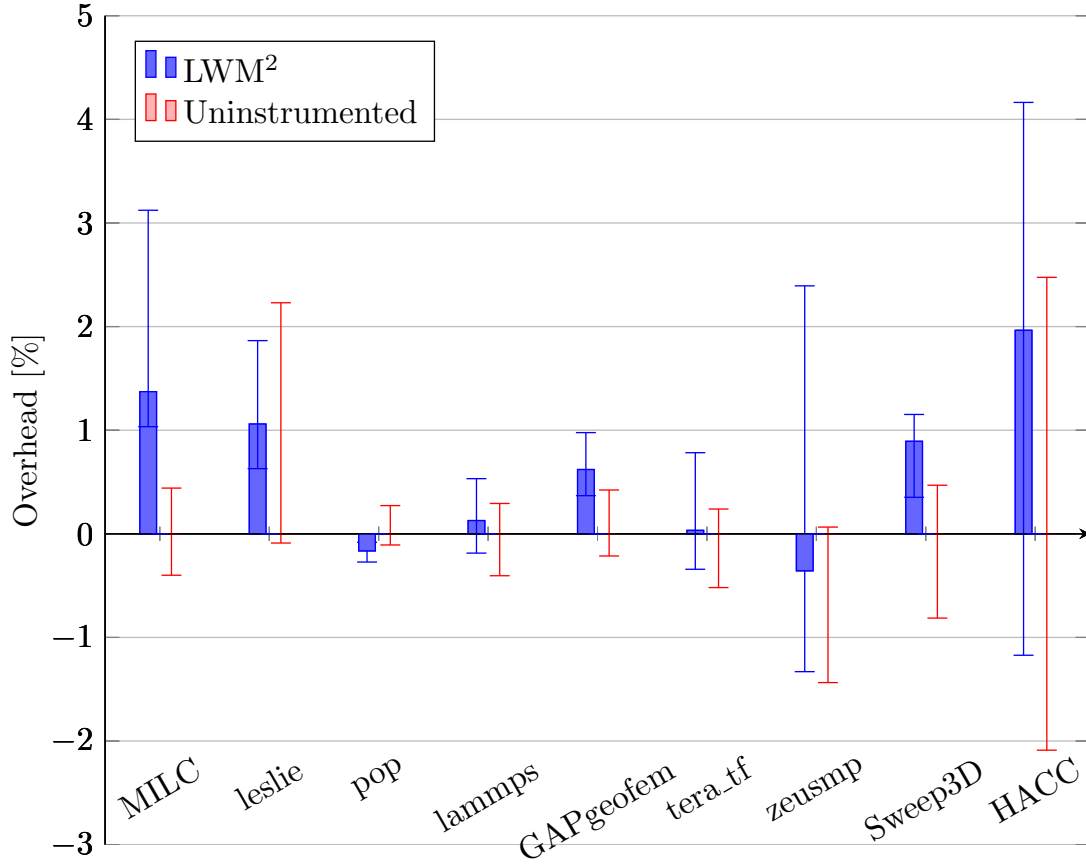
memory. It had a 5D Torus communication interconnect and minimized network interference by making node boards with 512 cores the smallest allocation unit. Since its GPFS file system was shared, JUQUEEN cannot be considered controlled for I/O intensive workloads though, which, however, among our benchmarks only affects HACC.

For our tests under production conditions, we use Hazel Hen, a Cray XC40 system. Hazel Hen was decommissioned on February 25, 2020. It consisted of 7712 compute nodes, each of them featuring two 12-core Intel Haswell E5-2680v3 processors and 128 GB of memory [47]. Applications running on Hazel Hen were known to experience significant run-to-run variation, majorly due to cache misses in the Aries chip under heavy network load from multiple applications [46].

## 5.2.2 Overhead

We first evaluate the profiling overhead of LWM<sup>2</sup>. It was less than 1% in our evaluation in Section 2.2.1. However, for our work in this chapter, LWM<sup>2</sup> has been substantially extended. Especially the direct recording of time in communication and file I/O operations, segment identification, pattern identification and grouping, and data compression using index trees and lists require many operations at runtime. We therefore evaluate the overhead of LWM<sup>2</sup> after the extensions. We use the same set of benchmarks as used for evaluating the accuracy of our interference estimation approach.

We executed the benchmarks on Hazel Hen, using 16 nodes with 24 processes per node. We executed each benchmark 9 times with LWM<sup>2</sup> and 9 times without profiling. We alternated between the two types to even out, as much as possible, the system load variation. The median execution time of the runs without profiling was used to find the minimum, median,



**Figure 5.16:** Overhead of profiling with LWM<sup>2</sup>.

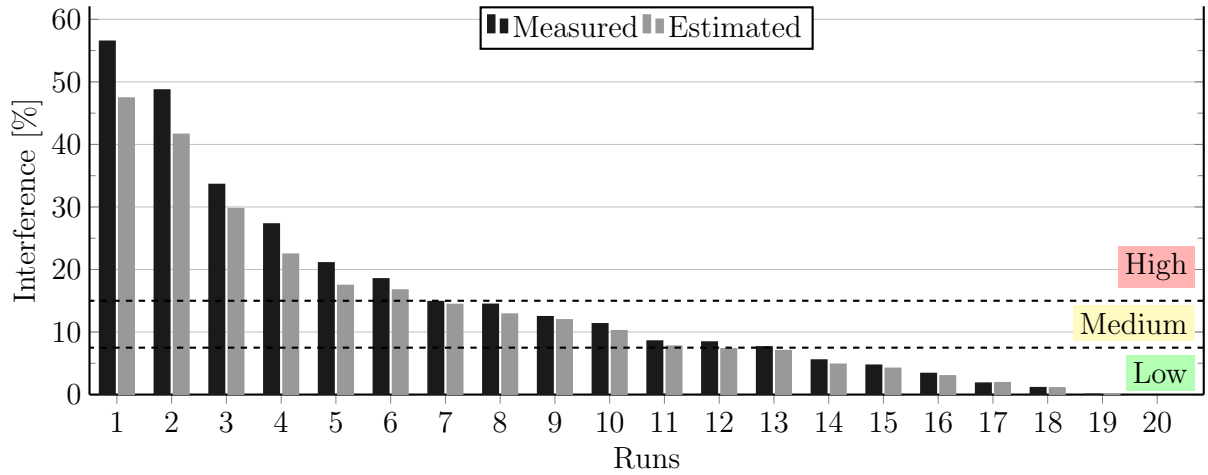
and maximum overhead of LWM<sup>2</sup>. The overhead is calculated as:

$$overhead = \left( \frac{T_{li}}{\tilde{T}_u} - 1 \right) \times 100$$

where  $T_l$  are the runs that were profiled with LWM<sup>2</sup>,  $T_{li}$  is the execution time of the run  $i$ ,  $T_u$  are the uninstrumented runs, and  $\tilde{T}_u$  is the median execution time without profiling.

Figure 5.16 presents the overhead for the set of benchmarks. The median overhead is represented by the bars, while the error bars represent the minimum and maximum overhead. For example, in the figure, MILC has a maximum overhead of around 3% and a minimum overhead of around 1%. The median overhead is around 1.5%. To highlight the variation in execution time of the non-profiled runs, we also draw their minimum and maximum execution times as a percentage of their median.

The figure shows that median overhead of LWM<sup>2</sup> remains below 2%. Some benchmarks show variation of  $\pm 2\%$ , however that is observed for both profiled and non-profiled runs. Such a degree of variation might be natural on Hazel Hen. The negative overhead of certain benchmarks can also be attributed to the natural run-to-run variation on Hazel Hen. Overall,



**Figure 5.17:** Multiple runs of `tera_tf` on JUQUEEN, with measured and estimated interference classified as low, medium, or high. Runs are sorted by execution time in descending order.

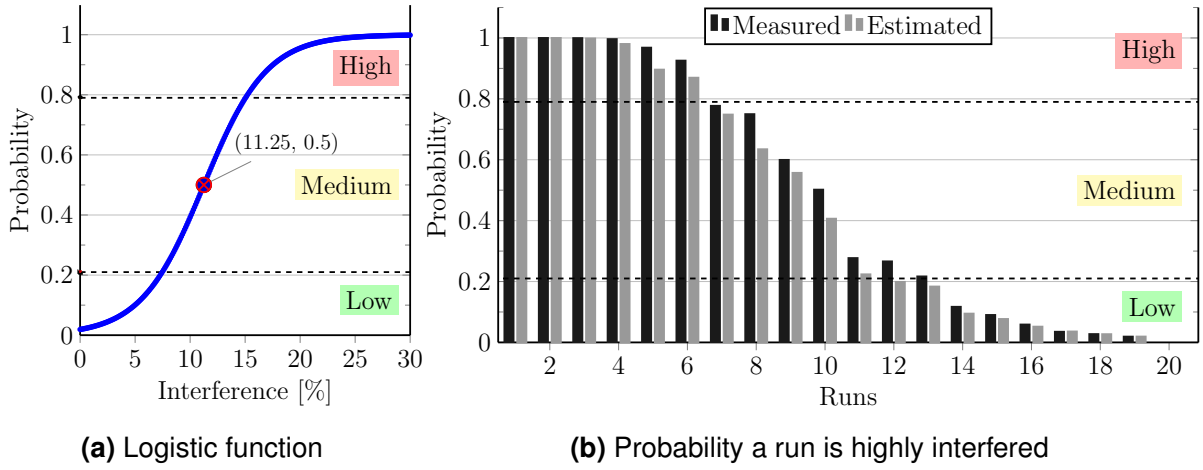
our measurements indicate that LWM<sup>2</sup> maintains low overhead.

### 5.2.3 Evaluation methodology

With the exception of the file system, our controlled environment is without any significant natural interference. This is why the runtime of a job is usually close to its intrinsic execution time, providing us with a ground truth for interference-free execution. To test our method, we inject artificial interference into application runs using a tool called `intM` (interference Modeler), which we have developed for this purpose. `intM` sits as an interposition wrapper between an application and the runtime, and mimics network and file I/O interference by introducing delays in function calls. `intM` supports interference injection in MPI communication and I/O functions, as well as in POSIX I/O. The interference added to the regular execution time follows a Gaussian distribution, with configurable mean and standard deviation. The probability of when an interference event strikes a communication or file I/O operation is also configurable.

Specifically, we inject gradually increasing interference into multiple runs of a benchmark. Figure 5.17 shows such runs for the SPEC MPI2007 benchmark `tera_tf` as an example. We compare the *estimated* with the *measured* impact of interference on each run. Measured interference is the execution-time difference between a run and the fastest run of the benchmark, in percent of the (interfered) runtime. Estimated interference is calculated individually for each run as percentage of its runtime using our approach without considering any other run. To clean the measured interference from effects of continuous interference and other





**Figure 5.18:** Logistic function and the highly interfered run probabilities, when the function is applied on the `tera_tf` runs.

|          | Runs               |      |      |      |      |      |      |      |      |      |     |     |     |     |     |
|----------|--------------------|------|------|------|------|------|------|------|------|------|-----|-----|-----|-----|-----|
|          | 1                  | 2    | 3    | 4    | 5    | 6    | 7    | 8    | 9    | 10   | 11  | 12  | 13  | 14  | 15  |
|          | Interference [%]   |      |      |      |      |      |      |      |      |      |     |     |     |     |     |
| <i>M</i> | 56.4               | 48.7 | 33.5 | 27.2 | 21   | 18.5 | 14.8 | 14.4 | 12.4 | 11.3 | 8.5 | 8.4 | 7.6 | 5.5 | 4.6 |
| <i>E</i> | 47.4               | 41.6 | 29.7 | 22.4 | 17.4 | 16.7 | 14.4 | 12.8 | 11.9 | 10.2 | 7.7 | 7.3 | 7   | 4.8 | 4.2 |
| $\delta$ | 9.1                | 7.1  | 3.9  | 4.8  | 3.6  | 1.8  | 0.5  | 1.6  | 0.5  | 1.1  | 0.8 | 1.1 | 0.6 | 0.7 | 0.5 |
|          | Run classification |      |      |      |      |      |      |      |      |      |     |     |     |     |     |
| <i>M</i> | H                  | H    | H    | H    | H    | H    | M    | M    | M    | M    | M   | M   | M   | L   | L   |
| <i>E</i> | H                  | H    | H    | H    | H    | H    | M    | M    | M    | M    | M   | L   | L   | L   | L   |

**Table 5.5:** Measured (*M*) and estimated (*E*) interference for runs of `tera_tf` as percentages of their runtime. The difference between the two percentages is also listed (as  $\delta$ ). Only runs 1-15 are shown. The runs are also classified to indicate *high* (H), *medium* (M), and *low* (L) degree of interference. Even though the estimated interference for runs 12 and 13 is only about 1% lower, they are misclassified, demonstrating that such a *hard* classification is not a suitable metric for evaluating the accuracy of our method.

influences that are largely constant across the entire duration of a run but may vary between runs, such as different process-to-node mappings, we reduce the measured interference by the amount of time the medians in segment profiles are displaced. We observe the median displacement during clustering, and attribute it to continuous interference.

From a performance analysis perspective, the impact of interference can be categorized

| Class  | True Positive | False Positive | False Negative | Precision | Recall | F1 Score |
|--------|---------------|----------------|----------------|-----------|--------|----------|
| High   | 6             | 0              | 0              | 1         | 1      | 1        |
| Medium | 5             | 0              | 2              | 1         | 0.71   | 0.83     |
| Low    | 6             | 2              | 0              | 0.75      | 1      | 0.86     |

**Table 5.6:** The  $F_1$  score of `tera_tf` runs. Runs 12 and 13 are misclassified (despite about only 1% error in the estimated interference), which reduces the  $F_1$  score of classifying *medium* and *low* runs.

| Runs                     |   |   |   |      |      |      |      |      |      |      |      |      |      |      |      |  |
|--------------------------|---|---|---|------|------|------|------|------|------|------|------|------|------|------|------|--|
|                          | 1 | 2 | 3 | 4    | 5    | 6    | 7    | 8    | 9    | 10   | 11   | 12   | 13   | 14   | 15   |  |
| Interference probability |   |   |   |      |      |      |      |      |      |      |      |      |      |      |      |  |
| $M$                      | 1 | 1 | 1 | 1    | 0.97 | 0.93 | 0.78 | 0.75 | 0.6  | 0.5  | 0.28 | 0.27 | 0.22 | 0.12 | 0.09 |  |
| $E$                      | 1 | 1 | 1 | 0.98 | 0.9  | 0.87 | 0.75 | 0.63 | 0.56 | 0.41 | 0.22 | 0.2  | 0.18 | 0.09 | 0.08 |  |
| $\delta$                 | 0 | 0 | 0 | 0.02 | 0.07 | 0.06 | 0.03 | 0.12 | 0.04 | 0.1  | 0.05 | 0.07 | 0.03 | 0.02 | 0.01 |  |

**Table 5.7:** Measured ( $M$ ) and estimated ( $E$ ) probability of interference of `tera_tf` runs, as well as their difference ( $\delta$ ). Only runs 1-15 are shown. For runs 12 and 13, the different in probability of interference is low.

into the classes low, medium, and high, as shown in Figure 5.17. A low-interference run is perturbed to a negligible degree and can be used for performance analysis, whereas a high-interference run is heavily perturbed and should be discarded. The medium category is between these extremes: It might be worthwhile to invest in a new performance measurement, while, at the same time, the run can be used to gauge performance at large. Using the analogy of a traffic light, low means green light for performance analysis, medium means yellow light, and high red light. We have set the threshold for low interference to below 7.5%, for high interference to above 15%, and classify everything in-between as medium.

To evaluate our approach, one method can be to categorize the *measured* and *estimated* interference into the above mentioned interference classes, and use methods like  $F_1$  score [106] to evaluate accuracy. While such categorization is useful to distinguish runs in practice, accuracy evaluation via *hard* classification into these three categories can run into pitfalls. In the following example, we demonstrate that *hard* classification is not suitable for evaluating the accuracy of our approach.

Table 5.5 lists the measured ( $M$ ) and estimated ( $E$ ) interference for the `tera_tf` runs as percentages of their execution time. Their percentage-point difference, that is the difference

between the percentage of measured and estimated interference, is also listed as  $\delta$ . The table also groups the runs into the classes *high* (H), *medium* (M), and *low* (L). Even though the differences for runs 12 and 13 are 1.1% and 0.6%, respectively, the measured and estimated interference fall into different categories. In Table 5.6, we calculate the  $F_1$  score for the runs.  $F_1$  score is calculated as:

$$F_1 = 2 \times \frac{p \times r}{p + r}$$

$$p = \text{precision} = \frac{TP}{(TP + FP)}$$

$$r = \text{recall} = \frac{TP}{(TP + FN)}$$

where  $TP$  is the number of *true positives*,  $FP$  the number of *false positives*, and  $FN$  the number of *false negatives*.

As runs 12 and 13 are misclassified, the *precision* and *recall* for classifying runs as *medium* and *low* is reduced, resulting in an  $F_1$  score of 0.83 and 0.86 respectively. As the percentage-point difference for the runs 12 and 13 is quite low, we conclude that the  $F_1$  score is not the right metric for our evaluation.

An alternative way of measuring accuracy can be using the percentage-point difference itself. Even though this method provides a good insight into the accuracy of the estimated interference, one downside of this approach is that for highly interfered runs, the percentage-point difference is not that critical as long as both agree on the judgment that the run is highly interfered. Runs 1 and 2 in Table 5.5 are such cases, where the percentage-point difference is high. Without the knowledge that both measured and estimated interference judge the runs as highly interfered, one would assume low accuracy for the estimation method. Therefore, to present a complete picture, we compliment the percentage-point difference with the following method.

Based on the intuition that the impact of interference is a measure of the suitability of a run for performance analysis, we convert the interference magnitude into probability of a run being interfered. A run with high probability value is considered unsuitable for analysis. For this purpose, we use a logistic function as a *soft* classifier to convert the magnitude of interference into interference probability. Using soft classification, the probability that a run previously categorized as low is actually highly interfered should be close to zero, while for runs categorized as high it should be near one. Similarly, at the mid-point of the medium category, the probability should be exactly 0.5. Figure 5.18a shows a logistic function that we have designed for this purpose, while Figure 5.18b shows the corresponding probabilities for the *tera\_tf* runs.

Table 5.7 shows the interference probabilities for the *tera\_tf* runs determined using the

---

logistic function. We see that for runs 1 and 2, the difference in probabilities is zero, as both methods judge the runs as highly interfered. Similarly, the difference in probabilities for runs 12 and 13 is small, as the percentage-points difference between measured and estimated interferences is also small.

Formally, a logistic function is an “S” shaped function that maps values from  $(-\infty, \infty)$  onto  $(0, L)$ . It is defined as

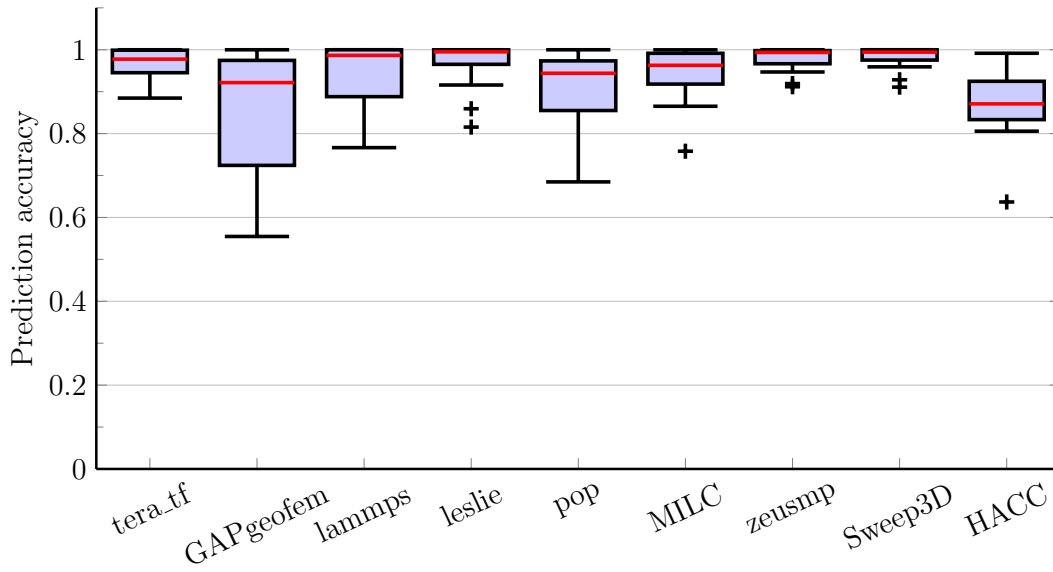
$$f(x) = \frac{L}{1 + \exp^{-k(x-x_0)}}$$

where  $k$  is the steepness,  $x_0$  is the inflection point, and  $L$  is the maximum. As explained before, we define the inflection point,  $x_0$ , to be 11.25, the mid-point of the medium class. Similarly, setting the maximum value,  $L$ , to 1, and steepness,  $k$  to 0.35, the probabilities at interference magnitudes of 7.5%, 11.25%, and 15% are 0.21, 0.5, and 0.79, respectively.

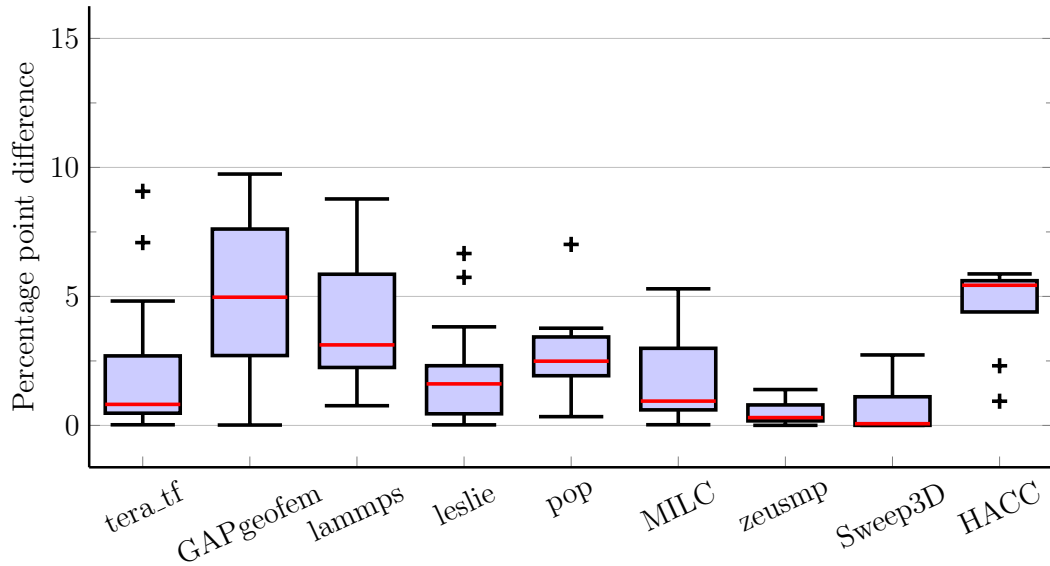
Using this logistic function, we derive probabilities for measured and estimated interference for each run of a benchmark. The difference between the two probabilities is the inaccuracy of interference prediction, and its compliment is the accuracy. We determine the accuracy of our approach for all the runs of each benchmark and draw the results as boxplots (Figure 5.20). As the logistic function in Figure 5.18a shows, an accuracy of less than 0.5 means a significant deviation between measured and estimated interferences. To also give a more direct impression of the results, we complement probability differences with boxplots of the percentage-point difference between measured and estimated interference.

## 5.2.4 Results

On JUQUEEN, our controlled environment, each benchmark was executed at least 15 times with a gradually increasing amount of artificial interference injected. Figure 5.17 shows the series for `tera_tf` as an example. The interference was adjusted in such a way that multiple runs were produced for each interference class. We executed each benchmark on 256 nodes, with 4 processes running on each node. Figure 5.19b presents on the left how accurately we predict the interference probabilities and on the right the percentage-point difference between measured and estimated interference. Except for GAPgeofem, the median accuracy for all the benchmarks on JUQUEEN is above 0.9. Similarly, for most benchmarks, the minimum accuracy is above 0.8. This shows that in most cases, estimated and measured interference leads to the same conclusion. That the accuracy of our predictions for certain runs of GAPgeofem was low can be attributed to its high collective-call rate of around 300 Hz, as shown in Table 5.4. At such a high frequency, large numbers of small execution segments are created, easily leading to measurement artifacts that disturb our analysis.



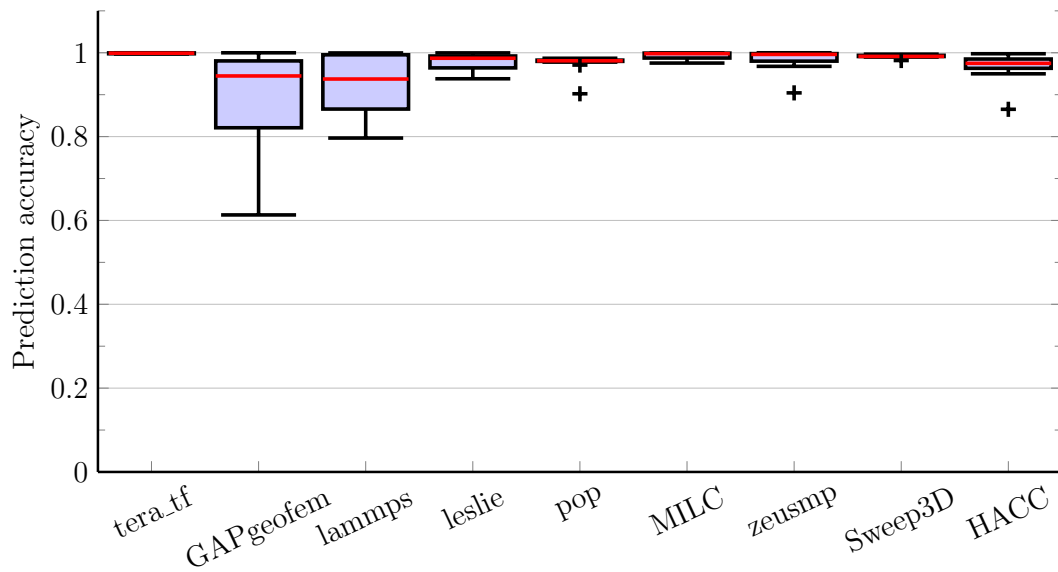
(a) Prediction accuracy



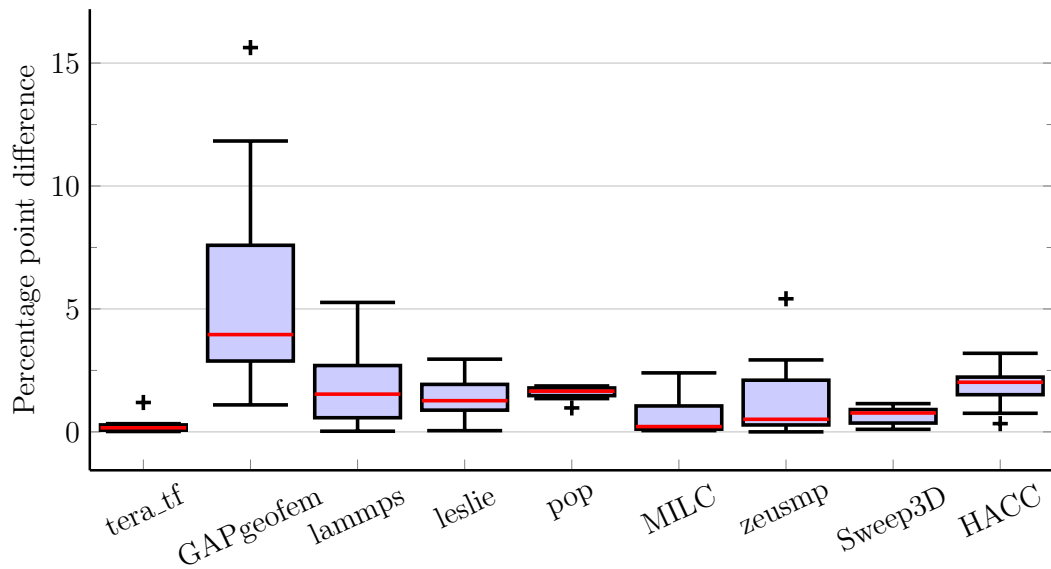
(b) Percentage-point difference

**Figure 5.19:** JUQUEEN: Prediction accuracy as difference of soft classification probability and percentage-point difference between measured and estimated interference.

On Hazel Hen, our production environment, we executed the benchmarks using 16 nodes, with 24 processes on each node. Each benchmark was executed 12 times. Due to the relative small scale of the runs and the sporadic nature of interference, many benchmarks were affected by interference to a smaller degree. Nonetheless, highly interfered runs were encountered and were accurately classified. On the left, Figure 5.20b shows the prediction accuracy of



(a) Prediction accuracy



(b) Percentage-point difference

**Figure 5.20:** Hazel Hen: Prediction accuracy as difference of soft classification probability and percentage-point difference between measured and estimated interference.

benchmark runs, complemented by the percentage-point difference between measured and estimated interference on the right. The figures show that, except for GAPgeofem, the impact of interference was estimated with a high degree of accuracy. GAPgeofem shows again low accuracy, which may again be attributable to its high collective-call frequency. Since the call frequency is measurable, we believe that it would be generally possible to warn the user of

---

possible inaccuracies in such rare cases. Finding an appropriate threshold, however, is left to future work.

## 5.3 Conclusion

We presented a method to estimate the impact of interference on bulk-synchronous applications from a single run. Our method first splits the execution of an application into segments that can be analyzed separately. For each segment, we then identify communication, computation, and file I/O features sufficient to represent its intrinsic performance characteristics. The segments are then clustered based on these characteristics. Any outliers in the clusters are caused by interferences. Impact of the interference can then be estimated from the magnitude of displacement of the outliers from the median. Our evaluation shows that our method can estimate the impact of interference with high accuracy based on a single run. Our tool chain now provides a warning light to performance analysts that tells them when they need to rerun their experiments because the data they have just collected was subject to interference. It can also be integrated with other performance-analysis tools using the P<sup>n</sup>MPI interface [110].

---

## 6 Related work

---

This dissertation touches different aspects of the broader topic of inter-application interference, including performance analysis tools, run-to-run variation and its sources, and identifying and estimating the impact of interference. In this chapter, we present some of the related work on these topics.

### 6.1 Performance analysis tools

A number of tools have been developed to capture and analyze the performance of HPC applications. They capture application events and create a profile [2, 17, 24, 38, 69, 118] or a trace [13, 20, 41, 95, 105] or both [68, 118]. Some of them [41, 95] are portable but require application recompilation to profile user functions and OpenMP regions, while others [20] take advantage of dynamic loading or patching but rely on a specific OpenMP runtime. In our literature survey, we found IPM [38], PerfSuite [69], Darshan [17], Autoperf [24], and HPCToolkit [2] to be without these limitations. Similar to LWM<sup>2</sup>, they have low runtime overhead and can profile applications without modification.

Most of the performance analysis tools are developed to capture the performance of a single application. LWM<sup>2</sup> is designed for a system-wide deployment on an HPC cluster. IPM [38], Darshan [17], and Autoperf [24] have also been developed with the same principle.

#### 6.1.1 Automatic screening of applications

IPM, which stands for Integrated Performance Measurement framework, is a lightweight profiler deployed on NERSC machines [38]. It automatically screens applications executing on the machines, and captures performance metrics of communication, threading, and file-I/O events. When an application finishes execution, IPM writes a textual job summary. The level of detail in the summary is configurable. The basic version of the summary aggregates the time spent in MPI, OpenMP, and file-I/O operations as percentages, while the full version breaks down the time by individual functions. Separate profiles are also generated for



---

application regions explicitly marked with `MPI_Pcontrol`. Besides the textual summary, the performance profiles are stored in a separate database, which can be used for evaluating performance at the system level.

Autoperf is a lightweight profiling tool developed to screen applications on IBM Blue Gene/Q systems [24]. It is deployed on systems, Mira and Cetus, at the Argonne National Laboratory. Autoperf has been developed as a monitor for MPI usage on a system and only collects the amount of time spent and the number of bytes communicated in individual MPI operations.

Darshan is another lightweight profiler that automatically screens applications executing on a production HPC system [17]. It is deployed on Interpid at the Argonne National Laboratory. Darshan only instruments file I/O and supports MPI I/O, POSIX I/O, HDF5, and Parallel netCDF interfaces. It captures several metrics, including the number of I/O operations, the amount of time spent in them, the number of bytes read or written, and the data alignment.

While IPM, Autoperf, and Darshan share many of the characteristics of our inter-application interference identification approach presented in Chapter 2, they are focused on screening individual applications. They lack the semantics to establish simultaneity among performance events from different applications. Our approach, on the other hand, is fundamentally geared towards correlating performance across applications.

### 6.1.2 Segmented profiles

The OpenMP profiler ompP supports incremental profiling [39], a concept similar to our time-slice approach, based on either a timer, a hardware counter, or a user-specified event. Yet, it still profiles applications in isolation and does not support the synchronized generation of incremental profiles across multiple applications. As a consequence, inter-application interference cannot be identified, as incremental profiles cannot be compared across a system. LWM<sup>2</sup>, on the other hand, is geared towards identifying such interferences based on globally synchronized time slices [115].

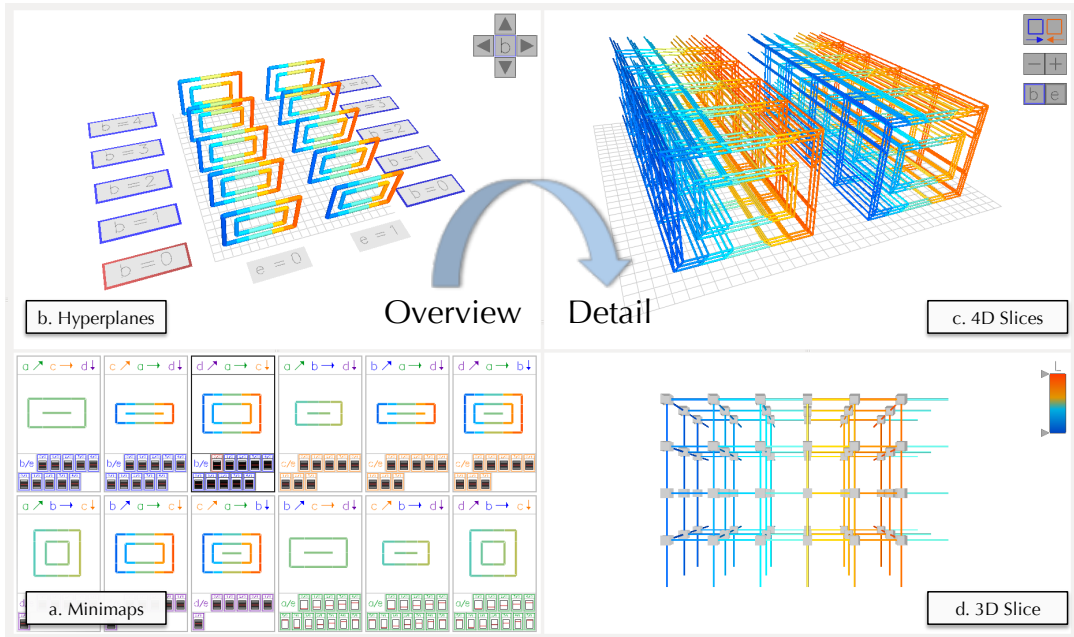
### 6.1.3 System monitoring

There are also numerous tools for system monitoring, such as Ganglia [83], Nagios [96], VGXP [64], and LDMS [4]. These tools monitor at the system-level and are not aware of the application perspective. LDMS (Lightweight Distributed Metric Service), which is deployed at Sandia National Lab, collects system-level information at a subsecond time interval. It can also combine the information from the scheduler to give an application-level view of system

utilization. However, it is also not aware of application-level metrics. Our work with LWM<sup>2</sup> can compliment the information captured by these tools.

## 6.2 Visualizing multi-dimensional data

There are general methods that can be used to visualize multi-dimensional data. The methods include independent parallel coordinates [85] and multi-pixel bar graph [66]. However, in them, the connectivity relationship between individual data points is lost.

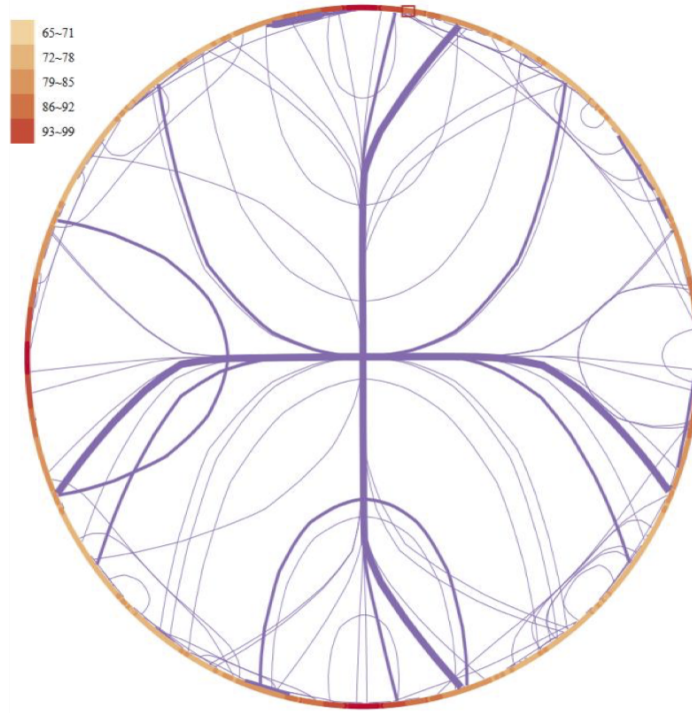


**Figure 6.1:** A Boxfish visualization of a five-dimensional torus network [84] ©2014 IEEE.

In the HPC context, Schulz et al. presented the HAC model that combined metrics from application, hardware, and communication domains [109]. By projecting the data among the domains and correlating them, users can gain valuable insights. However, this method is able to visualize only a three-dimensional torus. Similarly, CUBE4 visualizes a multi-dimensional torus as a set of cubes, but is limited to node level metrics [52]. Our method on the other hand is focused on displaying link-level metrics.

Boxfish projects a 3D torus network onto a 2D plane, by looking at the cube from the middle, forming concentric circles [75]. This makes it possible to visualize network traffic while maintaining structural integrity. An extension to the tool supports visualizing 5D torus by projecting hyperplanes onto multiple views [84], as shown in Figure 6.1. The minimaps view in the bottom left corner project three dimensions of the network as concentric circles. As only three dimensions are displayed, the minimaps are generated for every combination

of three dimensions. The hyperplanes view in the top left corner reveals further details about the selected minimap by visualizing the complete set of projections. The 4D slices view complements the hyperplane view by showing individual links of the hyperplanes. Finally, the 3D slice view displays a three-dimensional subtorus. Similar to our approach, Boxfish displays multiple simultaneous views at different level of details. However, it uses a single visualization approach, which they apply to summarize data at different levels. Our approach proposes multiple ways of summarizing data in different views.



**Figure 6.2:** TorusViz<sup>ND</sup> visualizes a torus network by arranging all the nodes in a circle [22] ©2014 IEEE. The ordering of the nodes can either be based on the sequential curve or the Hilbert curve. Higher traffic on the links is depicted by thicker lines.

TorusViz<sup>ND</sup> untangles the nodes of a high-dimensional torus into a string and uses two numbering schemes, the sequential curve or the Hilbert curve, to order the nodes [22]. It then uses hierarchical edge bundling [51] to display the network connections among the nodes. Figure 6.2 shows a sample visualization. The nodes are ordered based on the Hilbert curve. Higher traffic on the links is depicted by thicker lines. The visualization method of TorusViz<sup>ND</sup> is complimentary to our approach.

Similar to the dual view technique [97], we propose a method that uses multiple simultaneous views to display different aspects of the same network section. Our method integrates zooming and panning [89], but across multiple synchronized views. Our simultaneous views form a hierarchy that project the same network section at different levels of details.

---

## 6.3 Run-to-run variation

Applications executing on HPC systems experience run-to-run variation. The cause of variation can be, among others, activities of operating system and contention on shared resources. Some of related work on the topic is presented below.

### 6.3.1 Operating system

Until recently, the operating system was considered as the major source of variation, causing degradation by as much as 50% [104], and has, therefore, been studied extensively. Studies have looked at the sources of operating system jitter and found that timer interrupts and scheduling of daemons are major contributors [28, 111, 129]. Furthermore, the unsynchronized nature of these events is also a contributing factor [7, 35], and that these events also effect the network performance of HPC applications [50]. However, modern operating systems have reduced their noise footprint [87]. At the same time, some HPC vendors have developed their own lightweight, low-noise operating system, such as IBM Compute Node Kernel (CNK) [43] and Cray Compute Node Linux (CNL) [65]. The work presented in this dissertation is focused on inter-application interference, however the method described in Chapter 5 is applicable to any low-frequency, high-impact interference.

### 6.3.2 Inter-application interference

Besides the activities of the operating system, inter-application interference is another major source of run-to-run variation, which arises due to contention on shared resources. While several studies have investigated contention on shared memory subsystem [30, 56, 59, 93, 94, 103], this dissertation is focused on interference caused by the communication interconnect and the I/O subsystem.

#### Network interference

The runtime variation caused by network interference is a challenging problem [119], nonetheless it has been subject of many studies. Hoelfer et al. studied the impact of network noise on large-scale parallel applications and found it to degrade communication performance by a factor of two [48]. They further identified the degree of impact to be effected by network type, routing schemes, and node allocation policy. Jokanovic et al. estimated a 10% loss in system performance caused by application contention on slim fat trees [60]. Patki et al. investigated several factors that influence performance reproducibility, including job placement

---

in a fat-tree topology [102]. They found that tasks assigned to a random configuration of routers resulted in higher run-to-run variation compared to a compact assignment.

Similarly, Bhatele et al. observed significant performance variation on Hopper and identified contention from neighboring jobs as the most significant factor [8]. Hopper had a 3D torus network topology and did not enforce mapping of application processes near to each other. As a result, contention was especially high when application processes were mapped further apart and there were communication-heavy applications in between.

Dragonfly is another common network topology deployed on large-scale HPC clusters [67]. It uses adaptive routing to avoid hotspots in the network. Yang et al. investigated the effect of the job placement strategy on dragonfly networks and found that random placement maximizes network utilization but degrades the performance of jobs with less intensive communication [137]. On the other hand, contiguous placement alleviates this problem but creates hotspots in the network. Wang et al. also investigated several factors, such as job placement, routing policy, communication pattern and external interference, and concluded that communication characteristics aware job placement results in the best performance on dragonfly networks [134]. Savoie et al. used Quality of Service (QoS) metrics at the network level to reduce network interference. Their simulations show an improvement of up to 27% in individual application performance, with only 5% degradation for competing applications [108]. These studies are orthogonal to our work and serve as a motivation.

## **File I/O interference**

Interference from contention on the I/O subsystem is another significant factor causing inter-application interference. However, isolating the cause and effect of performance events of parallel I/O is challenging [130]. Nonetheless, studies have investigated I/O interference by either monitoring variation in performance of real applications or using benchmarks, such as IOR [117]. Lofstead et al. periodically measured the performance of the IOR benchmark on multiple systems and found that even moderate sharing can cause I/O performance to vary by up to 43% [81]. They also propose an adaptive I/O approach to mitigate interference. Luu et al. studied six years of I/O logs from three different systems [82]. They identified that a few applications dominated the I/O traffic on a system. Furthermore, for most of those applications, a median job achieved less than 1% of the maximum I/O throughput of the system. Xie et al. evaluated different factors affecting the data ingestion rate of a Lustre file system [136]. For concurrent writes by multiple applications, they found significant variance in aggregate bandwidth, by more than a factor of three. In each of their experiments, some applications (which changed from one run to another) would experience low bandwidth. They found that the target storage device was not at fault for the variation. Yildiz et al. identified different factors influencing I/O interference, including backend storage device,

---

access pattern, data distribution policy, and request size [138]. They conclude that I/O interference arises from interplay among several components in the I/O subsystem. These studies are orthogonal to our work. We focus on write-write contention and identify file access pattern as a significant factor [73, 113].

### 6.3.3 Other sources

There are other factors that also contribute to the run-to-run variation of execution time of applications. Rountree et al. studied the impact of power capping on performance of applications [107]. They found that without any power cap, processors occupied by an application have different power consumption. When a power cap is applied, its effects the performance of the processors differently and causes performance variation. Gholkar et al. also observed up to 50% variation in instruction per cycles of different processors under the same power cap [42].

Similarly, Bilge et al. found the Turbo Boost feature in processors to cause performance variation [1]. Using a math kernel, their experiments on three different leadership systems showed a variation of 8% to 16%. These studies show that other factors can also contribute to run-to-run variation and serve as a future work for our interference estimation method.

## 6.4 Estimating the impact of inter-application interference

Most of the studies on estimating the impact of external interference on applications have been carried out by simulating various scenarios. The studies include estimating effects of: system noise on MPI communication [50], optimum job placement and routing strategy on dragonfly network [134, 137], and operating system jitter [133]. Similarly, Mondragon et al. studied inter-application interference using Extreme Value Theory (EVT) [87]. They created a stochastic model to extrapolate the performance of bulk-synchronous applications under the influence of external interference, including the operating system, file I/O, and communication interconnect. Their model successfully predicted the performance of applications under external influences. In our work, we proposed and implemented a method to estimate the impact of interference using a single application run [114].

While most studies investigate a single source of run-to-run variation, Chunduri et al. evaluated multiple sources on a Xeon Phi based Cray XC system and provided mitigating solutions where possible [25]. They found the operating system to cause variation in performance on either the first or the last core. Similarly, they found the MPI collective performance to vary depending upon the system load, job size, and process-to-node mapping. Furthermore, they

---

found the tile feature of the system, where two cores on a processor share the same L2 cache, causes higher run-to-run variation as the memory footprint of an application increases. Our interference estimating method can capture any low frequency, high impact interference.

## 6.5 I/O access patterns and benchmarking

To study I/O interference in detail in Chapter 3, we first identified three typical write access patterns and then evaluated their interference potential. Several earlier studies have contributed to different aspects of this topic. Miller et al. found I/O to be bursty and cyclic [86]. They also distinguished three access patterns, namely required I/O, checkpointing, and data staging as the most common I/O types. These patterns roughly correspond to our aggregate-write, open-write-close, and write-seek patterns, respectively. However, they were studied to optimize I/O from a single-application perspective, while we look at their interference potential when executed concurrently. Byna et al. classified file access patterns to generate I/O-access signatures of applications [14]. These signatures were then used to improve data prefetching. Shan et al. created a parameterized I/O benchmark called IOR that can mimic the file access pattern of realistic applications [116]. Lofstead et al. found six common read patterns in the analysis part of simulation software [80]. The read patterns were used to compare end-to-end performance of logically contiguous and log-based files. Congiu et al. manually analyzed the I/O behavior of applications to identify their patterns [26]. A framework, transparent to the application, then translated the knowledge of these patterns into hints to the parallel file system. In our work, we evaluate the interference potential of I/O access patterns when executed concurrently.

Similarly, as part of the SIO initiative, Smirni et al. classified I/O patterns according to their spatial and temporal features [120, 121]. Nieuwejaar et al. classified file accesses with respect to access size, file size, access frequency, sequentiality, etc. in the CHARISMA project [98]. These studies are orthogonal to our work and part of the broader field of file-access characterization.

I/O performance has been the subject of several studies, looking at the performance from a single application perspective [12], from the file-system perspective [71], and from the overall system perspective [76, 140]. Furthermore, Uselton et al. statistically analyzed the I/O behavior of HPC applications and identified a bottleneck in the read-ahead window of the Lustre file system [130].

In our work, we analyze how file writes of concurrently running jobs interfere and determine factors that influence the magnitude of interference. [73, 113]. While application process count is already known as one of the factors [33], we consider process count in



---

the context of access patterns and examine the influence of further parameters such as write-chunk size and access frequency on write performance.

SIOX records I/O accesses at each level of the the I/O stack, identifies access patterns, and characterizing the I/O subsystem [135, 141] with the objective of pinpointing I/O bottlenecks. Our work contributes insights into write performance variation as a result of access patterns and request sizes.





---

## 7 Summary

---

In this dissertation, we have presented methods to identify and quantify inter-application interference. In Chapter 1, we introduced the run-to-run variation in execution time observed on HPC clusters and identified the major contributing factors. We also showed that inter-application interference is a significant source of the variation. We motivated the topic by presenting the design of a supercomputer and by describing how applications executing on it are susceptible to inter-application interference. Finally, we laid out the contribution of the dissertation.

We then presented our first contribution, a novel approach to correlate the performance behavior of applications running side by side. To accomplish this, we divided the application runtime into fine-grained time slices whose boundaries are synchronized across the entire system. Mapping performance data related to shared resources onto these time slices, we were able to establish the simultaneity of their usage across jobs, which is indicative of inter-application interference. We applied our approach on a production system and identified inter-application interference when simultaneously accessing the communication interconnect, the I/O subsystem and the shared GPU attached to a node. Our experiments showed that our approach can identify interference among applications.

Later, we focused on file I/O and used our method to analyze the influence of file-access patterns on the degree of interference. As it is by experience most intrusive, we focused our attention on write-write contention. We observed considerable differences among the interference potentials of several typical write patterns. In particular, we found that if one parallel program writes large output files while another one writes small checkpointing files, then the latter is slowed down when the checkpointing files are small enough. The trend is reversed and the former is slowed down when the checkpointing files are large. Moreover, already applications with few processes writing large output files can significantly hinder applications with many processes from checkpointing small files. Such effects can seriously impact the runtime of real applications—up to a factor of five in one instance. Overall, we found out that there is an interference trend between simultaneously writing small and large files.

At the end, we presented our second contribution, a novel approach to estimate the

---

impact of sporadic and high-impact interference on bulk-synchronous MPI applications. We introduced the concept of application execution segments that are independent of each other in terms of wait states. Using features considered sufficient to capture intrinsic performance characteristics, we grouped the segments that should have similar execution time. Any deviation in the execution time in the group is considered as the result of external interference. Evaluating in a controlled environment and on a production system, we demonstrated that our methodology can estimate the impact of external interference from a single run.

Inter-application interference is a prevalent issue in HPC clusters, causing run-to-run variation and wastage of system resources. The methods proposed and implemented in this dissertation can help determine the interference at the system level, helping identify underutilization of system resources. We also used our interference identification method to find a trend between simultaneously writing small and large files. Finally, our approach to estimating the impact of interference on individual applications gives valuable insight to users and performance analysts. The contributions of this work will help uncover inter-application interference and enable users make informed decisions.

## 7.1 Outlook

Run-to-run variation and inter-application interference is becoming prevalent. This dissertation presents methods to identify them at the system level and at the application level. However, there is still much work left.

At the application level, our method to estimate the impact of external interference is focused on bulk-synchronous applications with blocking semantics. The method can be extended to cover non-blocking collectives. Such collectives have a non-blocking call, followed by a wait call that ensures that the operation has completed. Our method can be adapted by consider the wait call as segment boundaries. Furthermore, an even more generic model that captures repeating patterns in an application to identify execution segments should be investigated. The methodology should also be extended to handle performance variation caused by power saving features, such as reducing CPU frequency. Finally, the methodology can be incorporated into performance analysis tools to identify regions of interference. This will help performance analysts discard parts of measurements that are not reliable, leading to analysis with higher confidence.

For investigating I/O interference, we focused on three typical write patterns. The patterns can be extended to cover more generic patterns, such as writes followed by reads and seeks of different sizes. Similarly, other I/O interfaces, such as HDF5 should also be evaluated.

As inter-application interference is a system-level phenomenon, we consider it natural to

---

identify it and mitigate it at the system level. In this regard, we consider the contributions in Chapter 2 to be of critical importance. While there are systems that either screen every application on a system (Autoperf [24], IPM [38]), capture the I/O behavior (Darshan [17]), or monitor system resource usage (LDMS [4]), what is needed is a holistic approach to capture performance metrics at every level and in a way to establish simultaneity. Therefore, the implementation in Chapter 2 should be extended in the future with system-level metrics and deployed on a production grade system. Furthermore, the gathered information should be analyzed to identify application behaviors and establish interference trends (such as the trend between simultaneously writing small and large files, described in Chapter 3). Finally, the trends can be translated into scheduler decisions and adaptive routing of network and I/O traffic to reduce and mitigate inter-application interference.

---

# Bibliography

---

- [1] Bilge Acun, Phil Miller, and Laxmikant V. Kale. “Variation Among Processors Under Turbo Boost in HPC Systems”. In: *Proceedings of the International Conference on Supercomputing (ICS’16)*. New York, NY, USA: Association for Computing Machinery, 2016, pp. 1–12.
- [2] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. “HPCToolkit: Tools for performance analysis of optimized parallel programs”. In: *Concurrency and Computation: Practice and Experience* 22.6 (2010), pp. 685–701.
- [3] Saurabh Agarwal, Rahul Garg, and Nisheeth K. Vishnoi. “The Impact of Noise on the Scaling of Collectives: A Theoretical Approach”. In: *High Performance Computing – HiPC 2005*. Ed. by David A. Bader, Manish Parashar, Varadarajan Sridhar, and Viktor K. Prasanna. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 280–289.
- [4] A. Agelastos, B. Allan, J. Brandt, P. Cassella, J. Enos, J. Fullop, A. Gentile, S. Monk, N. Naksinehaboon, J. Ogden, M. Rajan, M. Showerman, J. Stevenson, N. Taerat, and T. Tucker. “The Lightweight Distributed Metric Service: A Scalable Infrastructure for Continuous Monitoring of Large Scale Computing Systems and Applications”. In: *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC ’14)*. Nov. 2014, pp. 154–165.
- [5] Y. Ajima, S. Sumimoto, and T. Shimizu. “Tofu: A 6D Mesh/Torus Interconnect for Exascale Computers”. In: *Computer* 42.11 (Nov. 2009), pp. 36–40.
- [6] P. Beckman, K. Iskra, K. Yoshii, and S. Coghlan. “The Influence of Operating Systems on the Performance of Collective Operations at Extreme Scale”. In: *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER)*. Sept. 2006, pp. 1–12.
- [7] Pete Beckman, Kamil Iskra, Kazutomo Yoshii, Susan Coghlan, and Aroon Nataraj. “Benchmarking the effects of operating system interference on extreme-scale parallel machines”. In: *Cluster Computing* 11.1 (Mar. 2008), pp. 3–16.

- 
- [8] Abhinav Bhatele, Kathryn Mohror, Steven H. Langer, and Katherine E. Isaacs. “There goes the neighborhood: performance degradation due to nearby jobs”. In: *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC ’13)*. IEEE Computer Society, 2013.
- [9] Wahid Bhimji, Debbie Bard, Melissa Romanus, David Paul, Andrey Ovsyannikov, Brian Friesen, Matt Bryson, Joaquin Correa, Glenn K. Lockwood, Vakho Tsulaia, Suren Byna, Steve Farrell, Doga Gursoy, Chris Daley, Vince Beckner, Brian Van Straalen, David Trebotich, Craig Tull, Gunther H. Weber, Nicholas J. Wright, Katie Antypas, and none Prabhat. *Accelerating science with the NERSC burst buffer early user program*. Tech. rep. Lawrence Berkeley National Laboratory; National Energy Research Scientific Computing Center, 2016.
- [10] M. Blumrich, D. Chen, P. Coteus, A. Gara, M. Giampapa, P. Heidelberger, S. Singh, B. Steinmacher-burow, T. Takken, and P. Vranas. “Design and analysis of the BlueGene/L torus interconnection network”. In: *IBM Research Report* (2003).
- [11] David Böhme, Markus Geimer, Felix Wolf, and Lukas Arnold. “Identifying the root causes of wait states in large-scale parallel applications”. In: *Proceedings of the 39th International Conference on Parallel Processing (ICPP)*. San Diego, CA, USA: IEEE Computer Society, Sept. 2010, pp. 90–100.
- [12] Julian Borrill, L. Oliker, J. Shalf, and Hongzhang Shan. “Investigation of leading HPC I/O performance using a scientific-application derived benchmark”. In: *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC ’07)*. IEEE Computer Society, 2007, pp. 1–12.
- [13] BSC. *Extræ instrumentation package*. Accessed: 01.12.2019. <https://tools.bsc.es/extrae>. Dec. 2019.
- [14] Surendra Byna, Yong Chen, Xian-He Sun, Rajeev Thakur, and William Gropp. “Parallel I/O Prefetching Using MPI File Caching and I/O Signatures”. In: *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC ’08)*. Piscataway, NJ, USA: IEEE Computer Society, 2008, 44:1–44:12.
- [15] Surendra Byna, Jerry Chou, Oliver Rübel, Prabhat, Homa Karimabadi, William S. Daughton, Vadim Roytershteyn, E. Wes Bethel, Mark Howison, Ke-Jou Hsu, Kuan-Wu Lin, Arie Shoshani, Andrew Uselton, and Kesheng Wu. “Parallel I/O, Analysis, and Visualization of a Trillion Particle Simulation”. In: *Proceedings of the ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis (SC’12)*. Washington, DC, USA: IEEE Computer Society Press, 2012.
- [16] Tadeusz Caliński and Jerzy Harabasz. “A dendrite method for cluster analysis”. In: *Communications in Statistics-theory and Methods* 3.1 (1974), pp. 1–27.

- 
- [17] 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”. In: *Proceedings of the IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST)*. May 2011, pp. 1–14.
- [18] P. Carns, R. Latham, R. Ross, K. Iskra, S. Lang, and K. Riley. “24/7 Characterization of petascale I/O workloads”. In: *Proceedings of the IEEE International Conference on Cluster Computing and Workshops (CLUSTER)*. 2009, pp. 1–10.
- [19] Jonathan Carter, Julian Borrill, and Leonid Oliker. “Performance Characteristics of a Cosmology Package on Leading HPC Architectures”. In: *High Performance Computing - HiPC 2004*. Ed. by Luc Bougé and Viktor K. Prasanna. Vol. 3296. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2005, pp. 176–188.
- [20] Jordi Caubet, Judit Gimenez, Jesus Labarta, Luiz DeRose, and Jeffrey Vetter. “A Dynamic Tracing Mechanism for Performance Analysis of OpenMP Applications”. In: *OpenMP Shared Memory Parallel Programming*. Vol. 2104. Lecture Notes in Computer Science. Springer, 2001, pp. 53–67.
- [21] Dong Chen, Noel A. Easley, Philip Heidelberger, Robert M. Senger, Yutaka Sugawara, Sameer Kumar, Valentina Salapura, David L. Satterfield, Burkhard Steinmacher-Burow, and Jeffrey J. Parker. “The IBM Blue Gene/Q interconnection network and message unit”. In: *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC ’11)*. New York, NY, USA: ACM, 2011, 26:1–26:10.
- [22] S. Cheng, P. De, S. H. -. Jiang, and K. Mueller. “TorusVis<sup>ND</sup>: Unraveling High-Dimensional Torus Networks for Network Traffic Visualizations”. In: *Proceedings of the First Workshop on Visual Performance Analysis (VPA’14)*. New Orleans, LA, USA, 2014, pp. 9–16.
- [23] Jaeyoung Choi, Jack J Dongarra, Roldan Pozo, and David W Walker. “ScaLAPACK: a scalable linear algebra library for distributed memory concurrent computers”. In: *Proceedings of the IEEE Fourth Symposium on the Frontiers of Massively Parallel Computation*. 1992, pp. 120–127.
- [24] S. Chunduri, S. Parker, P. Balaji, K. Harms, and K. Kumaran. “Characterization of MPI Usage on a Production Supercomputer”. In: *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC ’18)*. 2018, pp. 386–400.
- [25] Sudheer Chunduri, Kevin Harms, Scott Parker, Vitali Morozov, Samuel Oshin, Naveen Cherukuri, and Kalyan Kumaran. “Run-to-run Variability on Xeon Phi Based Cray XC Systems”. In: *Proceedings of the ACM/IEEE International Conference for High*

- 
- Performance Computing, Networking, Storage and Analysis (SC '17)*. New York, NY, USA: ACM, 2017, pp. 1–13.
- [26] G. Congiu, M. Grawinkel, F. Padua, J. Morse, T. Süß, and A. Brinkmann. “MERCURY: A Transparent Guided I/O Framework for High Performance I/O Stacks”. In: *Proceedings of the 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP'17)*. Mar. 2017, pp. 46–53.
- [27] W. J. Dally. “Performance analysis of k-ary n-cube interconnection networks”. In: *IEEE Transactions on Computers* 39.6 (June 1990), pp. 775–785.
- [28] P. De, R. Kothari, and V. Mann. “Identifying sources of Operating System Jitter through fine-grained kernel instrumentation”. In: *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER)*. 2007, pp. 331–340.
- [29] John M. Dennis, Jim Edwards, Ray Loy, Robert Jacob, Arthur A. Mirin, Anthony P. Craig, and Mariana Vertenstein. “An application-level parallel I/O library for Earth system models”. In: *International Journal of High Performance Computing Applications* 26.1 (2012), pp. 43–53.
- [30] T. Dey, Wei Wang, J.W. Davidson, and M.L. Soffa. “Characterizing multi-threaded applications based on shared-resource contention”. In: *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2011, pp. 76–86.
- [31] David A. Dillow, Douglas Fuller, Feiyi Wang, H. Sarp Oral, Zhe Zhang, Jason J Hill, and Galen M Shipman. *Lessons Learned in Deploying the Worlds Largest Scale Lustre File System*. Tech. rep. Oak Ridge National Laboratory (ORNL); Center for Computational Sciences, 2010.
- [32] J. Dongarra, K. London, S. Moore, P. Mucci, D. Terpstra, Haihang You, and Min Zhou. “Experiences and lessons learned with a portable interface to hardware performance counters”. In: *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*. Apr. 2003, pp. 1–6.
- [33] Matthieu Dorier, Gabriel Antoniu, Robert Ross, Dries Kimpe, and Shadi Ibrahim. “CALCioM: Mitigating I/O Interference in HPC Systems through Cross-Application Coordination”. In: *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*. Phoenix, United States, May 2014.
- [34] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. “A Density-based Algorithm for Discovering Clusters a Density-based Algorithm for Discovering Clusters in Large Spatial Databases with Noise”. In: *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD'96)*. AAAI Press, 1996, pp. 226–231.



- 
- [35] Kurt B. Ferreira, Patrick Bridges, and Ron Brightwell. “Characterizing application sensitivity to OS interference using kernel-level noise injection”. In: *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '08)*. Austin, TX, USA: IEEE Computer Society, Nov. 2008.
- [36] B. Fryxell, K. Olson, P. Ricker, F. X. Timmes, M. Zingale, D. Q. Lamb, P. MacNeice, R. Rosner, J. W. Truran, and H. Tufo. “FLASH: An Adaptive Mesh Hydrodynamics Code for Modeling Astrophysical Thermonuclear Flashes”. In: *The Astrophysical Journal Supplement Series* 131.1 (2000), pp. 273–334.
- [37] P. Fuentes, E. Vallejo, C. Camarero, R. Beivide, and M. Valero. “Throughput Unfairness in Dragonfly Networks under Realistic Traffic Patterns”. In: *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER)*. Sept. 2015, pp. 801–808.
- [38] K. Fuerlinger, N. J. Wright, and D. Skinner. “Effective Performance Measurement at Petascale Using IPM”. In: *Proceedings of the 16th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*. Dec. 2010, pp. 373–380.
- [39] Karl F rlinger, Michael Gerndt, and Jack Dongarra. “On Using Incremental Profiling for the Performance Analysis of Shared Memory Parallel Applications”. In: *Proceedings of the International European Conference on Parallel and Distributed Computing (Euro-Par)*. Vol. 4641. Lecture Notes in Computer Science. Springer, 2007, pp. 62–71.
- [40] Rahul Garg and Pradipta De. “Impact of Noise on Scaling of Collectives: An Empirical Evaluation”. In: *Proceedings of 13th International Conference on High Performance Computing (HiPC 2006)*. Ed. by Yves Robert, Manish Parashar, Ramamurthy Badrinath, and Viktor K. Prasanna. Bangalore, India, Dec. 2006, pp. 460–471.
- [41] Markus Geimer, Felix Wolf, Brian J. N. Wylie, Erika  brah m, Daniel Becker, and Bernd Mohr. “The Scalasca performance toolset architecture”. In: *Concurrency and Computation: Practice and Experience* 22.6 (Apr. 2010), pp. 702–719.
- [42] N. Gholkar, F. Mueller, and B. Rountree. “Power tuning HPC jobs on power-constrained systems”. In: *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques (PACT)*. 2016, pp. 179–190.
- [43] M. Giampapa, T. Gooding, T. Inglett, and R. W. Wisniewski. “Experiences with a Lightweight Supercomputer Kernel: Lessons Learned from Blue Gene’s CNK”. In: *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '10)*. 2010, pp. 1–10.
- [44] Hal Finkel. *Cosmic Structure Probes of the Dark Universe (Porting and Tuning HACC on Mira)*. <https://www.alcf.anl.gov/files/darkuniverseesptechreportwrapped.pdf>. [Online, accessed: 11-Aug-2014]. 2014.

- 
- [45] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. 5th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.
- [46] HLRS. *Communication on Cray XC40 Aries network*. Accessed: 01.05.2017. [wickie.hlrs.de/platforms/index.php/Communication\\_on\\_Cray\\_XC40\\_Aries\\_network](http://wickie.hlrs.de/platforms/index.php/Communication_on_Cray_XC40_Aries_network). May 2017.
- [47] HLRS. *Cray XC40 - HLRS Platforms*. Accessed: 01.12.2019. [https://kb.hlrs.de/platforms/index.php/Cray\\_XC40](https://kb.hlrs.de/platforms/index.php/Cray_XC40). Dec. 2019.
- [48] T. Hoeﬂer, T. Schneider, and A. Lumsdaine. “The impact of network noise at large-scale communication performance”. In: *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. May 2009, pp. 1–8.
- [49] Torsten Hoeﬂer and Roberto Belli. “Scientific Benchmarking of Parallel Computing Systems, Twelve ways to tell the masses when reporting performance results”. In: *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC’15)*. 2015.
- [50] Torsten Hoeﬂer, Timo Schneider, and Andrew Lumsdaine. “Characterizing the Influence of System Noise on Large-Scale Applications by Simulation”. In: *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC ’10)*. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11.
- [51] D. Holten. “Hierarchical Edge Bundles: Visualization of Adjacency Relations in Hierarchical Data”. In: *IEEE Transactions on Visualization and Computer Graphics* 12.5 (Sept. 2006), pp. 741–748.
- [52] VI-HPS. *CUBE 4 Visualization tool*. Accessed: 06.08.2014. <http://www.vi-hps.org/projects/score-p/#cube4>. Aug. 2014.
- [53] James W. Hurrell, M. M. Holland, P. R. Gent, S. Ghan, Jennifer E. Kay, P. J. Kushner, J.-F. Lamarque, W. G. Large, D. Lawrence, K. Lindsay, W. H. Lipscomb, M. C. Long, N. Mahowald, D. R. Marsh, R. B. Neale, P. Rasch, S. Vavrus, M. Vertenstein, D. Bader, W. D. Collins, J. J. Hack, J. Kiehl, and S. Marshall. “The Community Earth System Model: A Framework for Collaborative Research”. In: *Bulletin of the American Meteorological Society* 94.9 (2013), pp. 1339–1360.
- [54] B. Iglewicz and D.C. Hoaglin. *How to detect and handle outliers*. Vol. 16. ASQC Quality Press (Milwaukee, Wis.), 1993.
- [55] issuu. *Moscow State University High Performance Computing by Peter Bryzgalov*. Accessed: 29.12.2019. [https://issuu.com/pyotr777/docs/msu\\_hpc](https://issuu.com/pyotr777/docs/msu_hpc). Dec. 2019.

- 
- [56] Ravi Iyer, Li Zhao, Fei Guo, Ramesh Illikkal, Srihari Makineni, Don Newell, Yan Solihin, Lisa Hsu, and Steve Reinhardt. “QoS policies and architecture for cache/memory in CMP platforms”. In: *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. ACM, 2007, pp. 25–36.
- [57] Nikhil Jain, Abhinav Bhatele, Xiang Ni, Nicholas J. Wright, and Laxmikant V. Kale. “Maximizing Throughput on a Dragonfly Network”. In: *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC ’14)*. Piscataway, NJ, USA: IEEE Press, 2014, pp. 336–347.
- [58] Hrvoje Jasak, Aleksandar Jemcov, and Zeljko Tukovic. “OpenFOAM: A C++ library for complex physics simulations”. In: *Proceedings of the International workshop on coupled methods in numerical dynamics*. 2007, pp. 1–20.
- [59] Xiaomin Jia, Jiang Jiang, Tianlei Zhao, Shubo Qi, and Minxuan Zhang. “Towards Online Application Cache Behaviors Identification in CMPs”. In: *Proceedings of the 12th IEEE International Conference on High Performance Computing and Communications (HPCC)*. 2010.
- [60] A. Jokanovic, G. Rodriguez, J. C. Sancho, and J. Labarta. “Impact of Inter-application Contention in Current and Future HPC Systems”. In: *Proceedings of the IEEE Symposium on High Performance Interconnects*. Aug. 2010, pp. 15–24.
- [61] JSC. *JUDGE system in Jülich*. Accessed: 01.12.2019. [https://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JUDGE/JUDGE\\_node.html](https://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JUDGE/JUDGE_node.html). Dec. 2019.
- [62] JSC. *JUQUEEN system in Jülich*. Accessed: 01.12.2019. <https://www.fz-juelich.de/ias/jsc/juqueen>. Dec. 2019.
- [63] JSC. *JUropa system in Jülich*. Accessed: 01.12.2019. [www.fz-juelich.de/jsc/juropa](http://www.fz-juelich.de/jsc/juropa). Dec. 2019.
- [64] Y. Kamoshida and K. Taura. “Scalable Data Gathering for Real-Time Monitoring Systems on Distributed Computing”. In: *Proceedings of the 8th IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*. May 2008, pp. 425–432.
- [65] Larry Kaplan and Jim Harrell. “Cray Compute Node Linux”. In: *Operating Systems for Supercomputers and High Performance Computing*. Ed. by Balazs Gerofi, Yutaka Ishikawa, Rolf Riesen, and Robert W. Wisniewski. Singapore: Springer Singapore, 2019, pp. 99–120.
- [66] Daniel Keim, Ming Hao, Umesh Dayal, Meichun Hsu, and Julain Ladisch. “Pixel Bar Charts: A New Technique for Visualizing Large Multi-Attribute Data Sets Without Aggregation”. In: *Proceedings of the IEEE Symposium on Information Visualization 2001 (INFOVIS’01)*. Washington, DC, USA: IEEE Computer Society, 2001, pp. 113–126.

- 
- [67] J. Kim, W. J. Dally, S. Scott, and D. Abts. “Technology-Driven, Highly-Scalable Dragonfly Topology”. In: *Proceedings of the International Symposium on Computer Architecture*. June 2008, pp. 77–88.
- [68] Andreas Knüpfer, Christian Rössel, Dieteran Mey, Scott Biersdorff, Kai Diethelm, Dominic Eschweiler, Markus Geimer, Michael Gerndt, Daniel Lorenz, Allen Malony, WolfgangE. Nagel, Yury Oleyunik, Peter Philippen, Pavel Saviankou, Dirk Schmidl, Sameer Shende, Ronny Tschüter, Michael Wagner, Bert Wesarg, and Felix Wolf. “Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir”. In: *Tools for High Performance Computing 2011*. Springer, 2012, pp. 79–91.
- [69] Rick Kufrin. “PerfSuite: An Accessible, Open Source Performance Analysis Environment for Linux”. In: *6th International Conference on Linux Clusters: The HPC Revolution*. Chapel Hill, NC, USA, 2005.
- [70] J. M. Kunkel and T. Ludwig. “Performance Evaluation of the PVFS2 Architecture”. In: *Proceedings of the 15th EUROMICRO International Conference on Parallel, Distributed and Network-Based Processing (PDP’07)*. Feb. 2007, pp. 509–516.
- [71] Julian M. Kunkel and Thomas Ludwig. “Bottleneck Detection in Parallel File Systems with Trace-Based Performance Monitoring”. In: *Proceedings of the International European Conference on Parallel and Distributed Computing (Euro-Par)*. Ed. by Emilio Luque, Tomàs Margalef, and Domingo Benítez. Vol. 5168. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2008, pp. 212–221.
- [72] Chih-Song Kuo. “I/O Subsystem as a Source of Inter-Application Interference on Supercomputers”. MA thesis. German Research School for Simulation Sciences, 2014.
- [73] Chih-Song Kuo, Aamer Shah, Akihiro Nomura, Satoshi Matsouka, and Felix Wolf. “How file access patterns influence interference among cluster applications”. In: *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER)*. 2014, pp. 1–8.
- [74] Ernest Orlando Lawrence Berkeley National Laboratory. *Global Cloud Resolving Model Simulations*, Ernest Orlando Lawrence Berkeley National Laboratory. <http://vis.lbl.gov/Vignettes/Incite19>. [Online, accessed: 11-Aug-2014]. 2014.
- [75] A. G. Landge, J. A. Levine, A. Bhatele, K. E. Isaacs, T. Gamblin, M. Schulz, S. H. Langer, P.-T. Bremer, and V. Pascucci. “Visualizing Network Traffic to Understand the Performance of Massively Parallel Simulations”. In: *IEEE Transactions on Visualization and Computer Graphics* 18.12 (2012), pp. 2467–2476.

- 
- [76] Samuel Lang, Philip Carns, Robert Latham, Robert Ross, Kevin Harms, and William Allcock. “I/O performance challenges at leadership scale”. In: *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC ’09)*. New York, NY, USA: IEEE Computer Society, 2009, 40:1–40:12.
- [77] C. E. Leiserson. “Fat-trees: Universal networks for hardware-efficient supercomputing”. In: *IEEE Transactions on Computers* C-34.10 (Oct. 1985), pp. 892–901.
- [78] Ning Liu, J. Cope, P. Carns, C. Carothers, R. Ross, G. Grider, A. Crume, and C. Maltzahn. “On the role of burst buffers in leadership-class storage systems”. In: *Proceedings of the IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*. 2012, pp. 1–11.
- [79] Glenn K Lockwood, Damian Hazen, Quincey Koziol, RS Canon, Katie Antypas, Jan Balewski, Nicholas Balthaser, Wahid Bhimji, James Botts, Jeff Broughton, Tina L. Butler, Gregory F. Butler, Ravi Cheema, Christopher Daley, Tina Declerck, Lisa Gerhardt, Wayne E. Hurlbert, Kristy A. Kallback-Rose, Stephen Leak, Jason Lee, Rei Lee, Jialin Liu, Kirill Lozinskiy, David Paul, Prabhat, Cory Snaveley, Jay Srinivasan, Tavia Stone Gibbins, and Nicholas J. Wright. *Storage 2020: a vision for the future of HPC storage*. Tech. rep. Lawrence Berkeley National Laboratory; National Energy Research Scientific Computing Center, 2017.
- [80] Jay Lofstead, Milo Polte, Garth Gibson, Scott Klasky, Karsten Schwan, Ron Oldfield, Matthew Wolf, and Qing Liu. “Six Degrees of Scientific Data: Reading Patterns for Extreme Scale Science IO”. In: *Proceedings of the 20th International Symposium on High Performance Distributed Computing (HPDC’11)*. New York, NY, USA: ACM, 2011, pp. 49–60.
- [81] Jay Lofstead, Fang Zheng, Qing Liu, Scott Klasky, Ron Oldfield, Todd Kordenbrock, Karsten Schwan, and Matthew Wolf. “Managing Variability in the IO Performance of Petascale Storage Systems”. In: *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC ’10)*. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–12.
- [82] Huong Luu, Marianne Winslett, William Gropp, Robert Ross, Philip Carns, Kevin Harms, Mr Prabhat, Suren Byna, and Yushu Yao. “A Multiplatform Study of I/O Behavior on Petascale Supercomputers”. In: *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*. HPDC ’15. New York, NY, USA: ACM, 2015, pp. 33–44.
- [83] Matthew L. Massie, Brent N. Chun, and David E. Culler. “The ganglia distributed monitoring system: design, implementation, and experience”. In: *Parallel Computing* 30.7 (2004), pp. 817–840.

- 
- [84] C. M. McCarthy, K. E. Isaacs, A. Bhatele, P. Bremer, and B. Hamann. “Visualizing the Five-dimensional Torus Network of the IBM Blue Gene/Q”. In: *Proceedings of the First Workshop on Visual Performance Analysis (VPA’14)*. New Orleans, LA, USA, 2014, pp. 24–27.
- [85] K. T. McDonnell and K. Mueller. “Illustrative Parallel Coordinates”. In: *Proceedings of the 10th Joint Eurographics / IEEE - VGTC Conference on Visualization (EuroVis’08)*. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2008, pp. 1031–1038.
- [86] Ethan L. Miller and Randy H. Katz. “Input/Output Behavior of Supercomputing Applications”. In: *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC ’91)*. Albuquerque, New Mexico, USA: IEEE Computer Society, 1991, pp. 567–576. ISBN: 0-89791-459-7.
- [87] O. H. Mondragon, P. G. Bridges, S. Levy, K. B. Ferreira, and P. Widener. “Understanding Performance Interference in Next-Generation HPC Systems”. In: *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC ’16)*. Nov. 2016, pp. 384–395.
- [88] Gordon E. Moore. “Cramming more components onto integrated circuits”. In: *Electronics* (Apr. 1965), pp. 114–117.
- [89] Tomer Moscovich, Fanny Chevalier, Nathalie Henry, Emmanuel Pietriga, and Jean-Daniel Fekete. “Topology-aware Navigation in Large Networks”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI ’09)*. New York, NY, USA: ACM, 2009, pp. 2319–2328.
- [90] Spec MPI. *Standard Performance Evaluation Corporation. (2007) SPEC MPI2007 Benchmark Suite*. Accessed: 01.01.2013. [www.spec.org/mpi2007/](http://www.spec.org/mpi2007/). Jan. 2013.
- [91] Philip J. Mucci, Shirley Browne, Christine Deane, and George Ho. “PAPI: A Portable Interface to Hardware Performance Counters”. In: *Proceedings of the Department of Defense HPCMP Users Group Conference*. 1999, pp. 7–10.
- [92] Frank Mueller, Xing Wu, Martin Schulz, BronisR. de Supinski, and Todd Gamblin. “ScalaTrace: Tracing, Analysis and Modeling of HPC Codes at Scale”. In: *Applied Parallel and Scientific Computing*. Ed. by Kristján Jónasson. Vol. 7134. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 410–418.
- [93] Sai Prashanth Muralidhara, Lavanya Subramanian, Onur Mutlu, Mahmut Kandemir, and Thomas Moscibroda. “Reducing memory interference in multicore systems via application-aware memory channel partitioning”. In: *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44)*. ACM, 2011, pp. 374–385.



- 
- [94] Onur Mutlu and Thomas Moscibroda. “Parallelism-Aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems”. In: *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA)*. IEEE Computer Society, 2008, pp. 63–74.
- [95] W. E. Nagel, A. Arnold, M. Weber, H. C. Hoppe, and K. Solchenbach. “VAMPIR: Visualization and Analysis of MPI Resources”. In: *Supercomputer 12.1* (Jan. 1996), pp. 69–80.
- [96] Nagios. *Nagios - The industry standard in IT infrastructure monitoring*. Accessed: 01.12.2019. <https://www.nagios.org>. [Online, accessed: 11-Aug-2014]. Dec. 2019.
- [97] Galileo Mark Namata, Brian Staats, Lise Getoor, and Ben Shneiderman. “A Dual-view Approach to Interactive Network Visualization”. In: *Proceedings of the Sixteenth ACM Conference on Conference on Information and Knowledge Management (CIKM '07)*. New York, NY, USA: ACM, 2007, pp. 939–942.
- [98] N. Nieuwejaar, D. Kotz, A. Purakayastha, C.S. Ellis, and M.L. Best. “File-access characteristics of parallel scientific workloads”. In: *IEEE Transactions on Parallel and Distributed Systems* 7.10 (Oct. 1996), pp. 1075–1089.
- [99] NVIDIA. *CUDA Toolkit Documentation: CUPTI*. Accessed: 09.01.2013. [docs.nvidia.com/cuda/cupti](https://docs.nvidia.com/cuda/cupti). Jan. 2013.
- [100] Man pages. *signal-safety(7) - Linux manual pages*. Accessed: 01.12.2019. <http://man7.org/linux/man-pages/man7/signal-safety.7.html>. Dec. 2019.
- [101] PAPI. *PAPITopics:SandyFlops - PAPI Docs*. Accessed: 01.12.2019. <http://icl.cs.utk.edu/projects/papi/wiki/PAPITopics:SandyFlops>. Dec. 2019.
- [102] Tapasya Patki, Jayaraman J. Thiagarajan, Alexis Ayala, and Tanzima Z. Islam. “Performance Optimality or Reproducibility: That is the Question”. In: *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '19)*. New York, NY, USA: Association for Computing Machinery, 2019.
- [103] C.M. Patrick, N. Voshell, and M. Kandemir. “Minimizing interference through application mapping in multi-level buffer caches”. In: *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2011, pp. 44–55.
- [104] F. Petrini, D.K. Kerbyson, and Scott Pakin. “The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q”. In: *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '03)*. 2003, pp. 55–55.

- 
- [105] Vincent Pillet, Jesús Labarta, Toni Cortes, and Sergi Girona. “PARAVER: A Tool to Visualize and Analyze Parallel Code”. In: *Proceedings of WoTUG-18: Transputer and Occam Developments*. Vol. 44. Apr. 1995, pp. 17–31.
- [106] C. J. van Rijsbergen. *Information Retrieval*. 2nd. Newton, MA, USA: Butterworth-Heinemann, 1979.
- [107] B. Rountree, D. H. Ahn, B. R. de Supinski, D. K. Lowenthal, and M. Schulz. “Beyond DVFS: A First Look at Performance under a Hardware-Enforced Power Bound”. In: *Proceedings of the IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum*. 2012, pp. 947–953.
- [108] L. Savoie, D. K. Lowenthal, B. R. de Supinski, K. Mohror, and N. Jain. “Mitigating Inter-Job Interference via Process-Level Quality-of-Service”. In: *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER)*. Sept. 2019, pp. 1–5.
- [109] M. Schulz, J. A. Levine, P.-T. Bremer, T. Gamblin, and V. Pascucci. “Interpreting Performance Data across Intuitive Domains”. In: *Proceedings of the International Conference on Parallel Processing (ICPP)*. Sept. 2011, pp. 206–215.
- [110] Martin Schulz and Bronis R. de Supinski. “P<sup>N</sup>MPI Tools: A Whole Lot Greater Than the Sum of Their Parts”. In: *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC ’07)*. Reno, NV, USA: ACM, 2007, 30:1–30:10.
- [111] S. Seelam, L. Fong, J. Lewars, J. Divirgilio, B.F. Veale, and K. Gildea. “Characterization of System Services and Their Performance Impact in Multi-core Nodes”. In: *Proceedings of the IEEE International Parallel Distributed Processing Symposium (IPDPS)*. Anchorage, AK, USA, 2011, pp. 104–117.
- [112] S. Seelam, L. Fong, A. Tantawi, J. Lewars, J. Divirgilio, and K. Gildea. “Extreme scale computing: Modeling the impact of system noise in multicore clustered systems”. In: *Proceedings of the IEEE International Symposium on Parallel Distributed Processing (IPDPS)*. Atlanta, GA, USA, 2010.
- [113] Aamer Shah, Chih-Song Kuo, Akihiro Nomura, Satoshi Matsuoka, and Felix Wolf. “How File-access Patterns Influence the Degree of I/O Interference between Cluster Applications”. In: *Supercomputing Frontiers and Innovations* 6.2 (2019).
- [114] Aamer Shah, Matthias Müller, and Felix Wolf. “Estimating the Impact of External Interference on Application Performance”. In: *Proceedings of the 24th International European Conference on Parallel and Distributed Computing (Euro-Par)*. Ed. by Marco Aldinucci, Luca Padovani, and Massimo Torquati. Torino, Italy: Springer International Publishing, 2018, pp. 46–58.



- 
- [115] Aamer Shah, Felix Wolf, Sergey Zhumatiy, and Vladimir Voevodin. “Capturing inter-application interference on clusters”. In: *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER)*. 2013, pp. 1–5.
- [116] Hongzhang Shan, Katie Antypas, and John Shalf. “Characterizing and predicting the I/O performance of HPC applications using a parameterized synthetic benchmark”. In: *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '08)*. Piscataway, NJ, USA: IEEE Computer Society, 2008, 42:1–42:12.
- [117] Hongzhang Shan and John Shalf. “Using IOR to Analyze the I/O performance for HPC Platforms”. In: *Proceedings of the Cray User Group Conference (CUG'07)*. 2007.
- [118] Sameer S. Shende and Allen D. Malony. “The TAU Parallel Performance System”. In: *International Journal of High Performance Computing Applications* 20.2 (2006), pp. 287–311.
- [119] D. Skinner and W. Kramer. “Understanding the causes of performance variability in HPC workloads”. In: *Proceedings of the IEEE International Workload Characterization Symposium*. Oct. 2005, pp. 137–149.
- [120] E. Smirni, R.A. Aydt, A. Chien, and D.A. Reed. “I/O requirements of scientific applications: an evolutionary view”. In: *Proceedings of 5th IEEE International Symposium on High Performance Distributed Computing (HPDC'96)*. Aug. 1996, pp. 49–59.
- [121] Evgenia Smirni and Daniel A. Reed. “Workload characterization of input/output intensive parallel applications”. In: *Computer Performance Evaluation Modelling Techniques and Tools*. Ed. by Raymond Marie, Brigitte Plateau, Maria Calzarossa, and Gerardo Rubino. Vol. 1245. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1997, pp. 169–180.
- [122] E. Strohmaier, H. W. Meuer, J. Dongarra, and H. D. Simon. “The TOP500 List and Progress in High-Performance Computing”. In: *Computer* 48.11 (Nov. 2015), pp. 42–49.
- [123] Zoltán Szebenyi, Todd Gamblin, Martin Schulz, Bronis R. de Supinski, Felix Wolf, and Brian J. N. Wylie. “Reconciling Sampling and Direct Instrumentation for Unintrusive Call-Path Profiling of MPI Programs”. In: *Proceedings of the 25th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*. Anchorage, AK, USA: IEEE Computer Society, May 2011, pp. 637–648.
- [124] Tokyo Tech. *Tsubame2 | [GSIC] | Tokyo Institute of Technology | Global Scientific Information and Computing Center*. Accessed: 01.12.2019. <https://www.gsic.titech.ac.jp/en/tsubame2>. Dec. 2019.

- 
- [125] Tokyo Tech. *What is TSUBAME? | [GSIC] | Tokyo Institute of Technology | Global Scientific Information and Computing Center*. Accessed: 01.12.2019. <https://www.gsic.titech.ac.jp/en/tsubame>. Dec. 2019.
- [126] The National Institute for Computational Sciences. *I/O and Lustre Usage*. <https://www.nics.tennessee.edu/computing-resources/file-systems/io-lustre-tips>. [Online, accessed: 11-Aug-2014]. 2014.
- [127] Lucas Theisen. “VisTorus - A Tool to Visualize Traffic on High-Dimensional Torus Interconnects”. MA thesis. German Research School for Simulation Sciences, 2014.
- [128] Lucas Theisen, Aamer Shah, and Felix Wolf. “Down to Earth – How to Visualize Traffic on High-dimensional Torus Networks”. In: *Proceedings of the First Workshop on Visual Performance Analysis (VPA'14)*. New Orleans, LA, USA, Nov. 2014, pp. 1–6.
- [129] Dan Tsafir, Yoav Etsion, Dror G. Feitelson, and Scott Kirkpatrick. “System noise, OS clock ticks, and fine-grained parallel applications”. In: *Proceedings of the 19th International Conference on Supercomputing (ICS)*. New York, NY, USA: ACM, 2005, pp. 303–312.
- [130] A. Uselton, M. Howison, N.J. Wright, David Skinner, N. Keen, J. Shalf, K.L. Karavanic, and L. Oliker. “Parallel I/O performance: From events to ensembles”. In: *Proceedings of the IEEE International Symposium on Parallel Distributed Processing (IPDPS)*. 2010, pp. 1–11.
- [131] Rob F. van der Wijngaart and Haoqiang Jin. “The NAS Parallel Benchmarks, Multi-Zone Versions”. In: NAS-03-010 (June 2003).
- [132] E. Vicente and R. Matias Jr. “Exploratory Study on the Linux OS Jitter”. In: *Brazilian Symposium on Computing System Engineering (SBESC)*. Nov. 2012, pp. 19–24.
- [133] E. Vicente and R. Matias. “Modeling and simulating the effects of OS Jitter”. In: *Winter Simulations Conference (WSC)*. Dec. 2013, pp. 2151–2162.
- [134] X. Wang, M. Mubarak, X. Yang, R. B. Ross, and Z. Lan. “Trade-Off Study of Localizing Communication and Balancing Network Traffic on a Dragonfly System”. In: *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. May 2018, pp. 1113–1122.
- [135] Marc C. Wiedemann, Julian M. Kunkel, Michaela Zimmer, Thomas Ludwig, Michael Resch, Thomas Bönisch, Xuan Wang, Andriy Chut, Alvaro Aguilera, Wolfgang E. Nagel, Michael Kluge, and Holger Mickler. “Towards I/O analysis of HPC systems and a generic architecture to collect access patterns”. In: *Computer Science - Research and Development* 28.2-3 (2013), pp. 241–251.

- 
- [136] Bing Xie, Jeffrey Chase, David Dillow, Oleg Drokin, Scott Klasky, Sarp Oral, and Norbert Podhorszki. “Characterizing Output Bottlenecks in a Supercomputer”. In: *Proceedings of the ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis (SC ’12)*. Los Alamitos, CA, USA: IEEE Computer Society, 2012, 8:1–8:11.
- [137] X. Yang, J. Jenkins, M. Mubarak, R. B. Ross, and Z. Lan. “Watch Out for the Bully! Job Interference Study on Dragonfly Network”. In: *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC ’16)*. Nov. 2016, pp. 750–760.
- [138] O. Yildiz, M. Dorier, S. Ibrahim, R. Ross, and G. Antoniu. “On the Root Causes of Cross-Application I/O Interference in HPC Storage Systems”. In: *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. May 2016, pp. 750–759.
- [139] Hao Yu, I-Hsin Chung, and Jose Moreira. “Topology Mapping for Blue Gene/L Supercomputer”. In: *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC ’06)*. New York, NY, USA: ACM, 2006.
- [140] Weikuan Yu, J.S. Vetter, and H.S. Oral. “Performance characterization and optimization of parallel I/O on the Cray XT”. In: *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*. 2008, pp. 1–11.
- [141] Michaela Zimmer, Julian Martin Kunkel, and Thomas Ludwig. “Towards Self-optimization in HPC I/O”. In: *Supercomputing*. Ed. by Julian Martin Kunkel, Thomas Ludwig, and Hans Werner Meuer. Vol. 7905. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 422–434.