

# New cross-layer techniques for multi-criteria scheduling in large-scale systems

by

Alberto Cascajo García

A dissertation submitted by in partial fulfillment of the  
requirements for the degree of Doctor of Philosophy in

Computer Science and Technology

Universidad Carlos III de Madrid

Advisor(s):

Jesús Carretero Pérez

David Expósito Singh

Tutor:

Jesús Carretero Pérez

December, 2021

This thesis is distributed under license “Creative Commons **Atribution - Non Commercial - Non Derivatives**”.



Dedicado a mi hija, Mía, que con su nacimiento ha conseguido convertir 2020 y 2021 en los años más especiales que podría imaginar (teniendo en cuenta la situación mundial).

Gracias por tu llegada y por crecer dándonos tantas alegrías como horas de sueño nos quitas.

Te quiero.

## ACKNOWLEDGEMENTS

Gracias. Gracias a todas aquellas personas que han contribuido al desarrollo de esta tesis, tanto de forma material como en forma de apoyo, ánimo, etc.

De forma más profunda, necesito dar las gracias a una persona muy especial: *Irene*. Al término de esta tesis sumamos casi 13 años juntos, casi 5 años de convivencia, más de un año siendo papá y mamá, y desde el primer día has estado apoyando todo lo que he querido hacer. Has sacrificado tiempo, horas de sueño, energía y ánimo para que pudiese dedicarlo a mi trabajo y culminar esta fase. No tengo ninguna duda de que gracias a ti, esto ha sido más fácil. Sólo lamento no haber estado al 100% los primeros meses de nuestra pequeña. Las tareas no terminaban y las noches se me quedaban cortas, pero por suerte todo se calmó y hemos podido vivir juntos lo más maravilloso del mundo. Espero poder devolverte, de alguna manera, todos esos esfuerzos que has hecho.

También quiero dar las gracias a mis dos directores: *Jesús y David*. De verdad siento que habéis sido un gran apoyo, unos guías magníficos, y no sólo en el ámbito de la tesis, sino también en los aspectos personales. Desde el inicio me he sentido muy arropado y bien asesorado. Puedo decir con total seguridad que sois dos referentes en cuanto a trabajo, esfuerzo y dedicación, y los mejores directores que podría haber tenido.

De manera más general quiero dar las gracias a toda mi familia al completo: papá y mamá, los primeros por derecho. Siempre habéis tratado de darme todo, y para mí, a día de hoy, “todo” consiste en alcanzar el máximo grado académico, en haber logrado crear una familia, y el haber labrado el camino para definir mi futuro. Vuestra contribución a todo esto ha sido clave. Por supuesto, no me olvido de los demás: *Nerea*, que nos has cuidado y mimado, y además te has ocupado de quienes te necesitaban en esos momentos tan críticos (y aprovecho para decirte un enorme GRACIAS por ello también); *Yayes* que siempre estáis ahí para todos, pase lo que pase; *Iván*, gracias a tu apoyo, consejo y las ganas que me has transmitido de apuntar alto, has ayudado a que mire siempre hacia delante. *Tata*, gracias por estar también siempre ahí para aconsejarme, ayudar si era necesario, ser un apoyo tras la paternidad, etc.; Finalmente, primicos, sobris y allegados (si, va también por vosotros *Rubén y Dani*), gracias por contribuir al buen ambiente y alegría que vivimos. Sin las risas, el buen humor, el meternos unos con otros, etc., la vida sería sería mucho más aburrida. Tengo la suerte de que nuestra familia está muy unida, nos vemos a menudo, nos contamos las cosas, ayudáis a aliviar el estrés y nos damos consejos pensando en lo mejor para las personas. Gracias a todos los fines de semana que pasamos juntos, nos distraemos e incluso discutimos, nuestro estado de ánimo y nuestras ganas de seguir adelante, todos juntos, se hacen más fuertes. Y siento que todos me habéis ayudado, de alguna manera, a terminar todas las tareas que engloban esta tesis. Gracias.

También quiero agradecer los ánimos y las risas que me han transmitido otras tantas personas, tanto compañeros de despacho, de universidad, compañeros moteros, y a los



Gañanucos, porque sin esas salidas, sin esos chistes y sin esas discusiones, los días habrían sido mucho más largos y mi mente quizá se habría atascado en varias ocasiones.

Y por supuesto, a aquellos que están siguiendo este largo camino les deseo todo el ánimo del mundo y, si necesitan algo, saben dónde encontrarme. Esta experiencia me ha ayudado a ver el nivel de estrés que pueden alcanzar las personas en este campo tan competitivo, y unas palabras, ánimos, una lluvia de ideas, o cualquier cosa que se salga de lo normal, por absurdo que parezca, pueden ayudar mucho mentalmente. ¡Contad conmigo!

*“La programación es una carrera entre los desarrolladores, que intentan construir mayores y mejores programas a prueba de idiotas, y el Universo, que intenta producir mayores y mejores idiotas. Por ahora va ganando el Universo” - Edward Tufte*

*“Daría todo lo que sé, por la mitad de todo lo que no sé.” - Descartes*

*“No te tomes la vida demasiado en serio, jamás saldrás vivo de ella.” - Anónimo*

## PUBLISHED AND SUBMITTED CONTENT

- A. Cascajo, D. E. Singh, J. Carretero [**main author**]. “Performance-aware scheduling of parallel applications on non-dedicated clusters”. Published in: *Electronics* 2019, 8(9), 982; <https://doi.org/10.3390/electronics8090982>. Ref: [1]
  - This publication is partly included in this Thesis in Chapters 3, 4, 5 and 7.
- A. Cascajo, D. E. Singh, J. Carretero [**main author**]. “Framework escalable para monitorización y planificación de aplicaciones paralelas”. Presented and published in: *Actas Jornadas Sarteco 2019*. ISBN: 978-84-09-12127-4. Ref: [2]
  - This publication is partly included in this Thesis in Chapters 3 and 5.
- A. Cascajo, D. E. Singh, J. Carretero [**main author**]. “LIMITLESS - Light-weight MonItoring Tool for Large-Scale Systems”. Published and presented in: *29th International Conference on Parallel, Distributed and Network-Based Processing (PDP) 2021*. Ref: [3]
  - This publication is partly included in this Thesis in Chapters 4, 5 and 7.
- A. Cascajo, D. E. Singh, J. Carretero [**main author**]. “LIMITLESS - Light-weight MonItoring Tool for Large-Scale Systems”. Submitted for approval: *MICROPROCESSORS AND MICROSYSTEMS (MICPRO) 2021*.
  - This publication is partly included in this Thesis in Chapters 4, 5 and 7.
- A. Bustos, A. Cascajo, A. J. Rubio-Montero, J. Navarro, J. A. Moríñigo, D. E. Singh, R. Mayo-García, J. Carretero [**main author**]. “Energy Consumption Studies of WRF Executions with the LIMITLESS Monitor”. Accepted: *Latin America High Performance Computing Conference (CARLA) 2021*. .
- A. Cascajo, D. E. Singh, J. Carretero [**main author**]. “LIMITLESS - Planificación basada en monitorización”. Presented and published in: *Jornadas Sarteco 2021*.
  - This publication is partly included in this Thesis in Chapters 5 and 7.
- A. Bustos, A. Cascajo, A. J. Rubio-Montero, J. Navarro, J. A. Moríñigo, D. E. Singh, R. Mayo-García, J. Carretero [**Co-author**]. “Estudio de consumo energético de las simulaciones climáticas con WRF usando LIMITLESS”. Published in: *Jornadas Sarteco 2021*.
  - This publication is partly included in this Thesis in Chapter 3.

- A. Cascajo, G. Gomez, J. Escudero, P. J. García, D. E. Singh, F. Algaro, J. Carretero, F. J. Quiles [**main author**]. “Improving Congestion Control through Fine-Grain Monitoring of InfiniBand Networks”. Submitted for approval in: *International Parallel and Distributed Processing Symposium (IPDPS) 2022*.
  - This publication is partly included in this Thesis in Chapters 3, 4 and 7.

## OTHER RESEARCH MERITS

- A. Cascajo, D. E. Singh, J. Carretero [**main author**]. “Adaptive scheduling of HPC applications using malleability and dynamic migration”. Presented and published in: *14th Scheduling for Large Scale Systems Workshop (2019)*. Ref. [\[4\]](#)
  - This publication is partly included in this Thesis in Chapters 3, 5 and 7.
- A. Cascajo, D. E. Singh, J. Carretero [**main author**]. “LIMITLESS - Planificación de grano fino basada en monitorización de los dispositivos en tiempo cuasi-real”. Presented in: *Workshop CABAHLA 2021* .
  - This publication is partly included in this Thesis in Chapters 3.

Tesis Doctoral

# New cross-layer techniques for multi-criteria scheduling in large-scale systems.

AUTOR: Alberto Cascajo García

DIRECTOR: Jesús Carretero Pérez

CODIRECTOR: David Expósito Singh

Firmas del Tribunal Calificador

Nombre y apellidos

Firma

Presidente:

Secretario:

Vocal:

En Leganés, a

de

del 202

## DECLARATION

I, Alberto Cascajo García, declare that this PhD. thesis titled New cross-layer techniques for multi-criteria scheduling in large-scale systems and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this PhD. thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this PhD. thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the PhD. thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.



This PhD dissertation has been partially supported by the Spanish Ministry of Science and Innovation under an FPI fellowship associated to a National Project with reference TIN2016-79637-P (from July 1, 2018 to October 10, 2021).

## ABSTRACT

The global ecosystem of information technology (IT) is in transition to a new generation of applications that require more and more intensive data acquisition, processing and storage systems. As a result of that change towards data intensive computing, there is a growing overlap between high performance computing (HPC) and Big Data techniques in applications, since many HPC applications produce large volumes of data, and Big Data needs HPC capabilities.

The hypothesis of this PhD. thesis is that the potential interoperability and convergence of the HPC and Big Data systems are crucial for the future, being essential the unification of both paradigms to address a broad spectrum of research domains. For this reason, the main objective of this Phd. thesis is purposing and developing a monitoring system to allow the HPC and Big Data convergence, thanks to giving information about behaviors of applications in a system which execute both kind of them, giving information to improve scalability, data locality, and to allow adaptability to large scale computers. To achieve this goal, this work is focused on the design of resource monitoring and discovery to exploit parallelism at all levels. These collected data are disseminated to facilitate global improvements at the whole system, and, thus, avoid mismatches between layers. The result is a two-level monitoring framework (both at node and application level) with a low computational load, scalable, and that can communicate with different modules thanks to an API provided for this purpose. All data collected is disseminated to facilitate the implementation of improvements globally throughout the system, and thus avoid mismatches between layers, which combined with the techniques applied to deal with fault tolerance, makes the system robust and with high availability.

On the other hand, the developed framework includes a task scheduler capable of managing the launch of applications, their migration between nodes, as well as the possibility of dynamically increasing or decreasing the number of processes. All these thanks to the cooperation with other modules that are integrated into LIMITLESS, and whose objective is to optimize the execution of a stack of applications based on multi-criteria policies. This scheduling mode is called *coarse-grain scheduling based on monitoring*.

For better performance and in order to further reduce the overhead during the monitorization, different optimizations have been applied at different levels to try to reduce communications between components, while trying to avoid the loss of information. To achieve this objective, data filtering techniques, Machine Learning (ML) algorithms, and Neural Networks (NN) have been used.

In order to improve the scheduling process and to design new multi-criteria scheduling policies, the monitoring information has been combined with other ML algorithms to identify (through classification algorithms) the applications and their execution phases, doing offline profiling. Thanks to this feature, LIMITLESS can detect which phase is exe-



cutting an application and tries to share the computational resources with other applications that are compatible (there is no performance degradation between them when both are running at the same time). This feature is called *fine-grain scheduling*, and can reduce the makespan of the use cases while makes efficient use of the computational resources that other applications do not use.

## RESUMEN

El ecosistema global de las tecnologías de la información (IT) se encuentra en transición a una nueva generación de aplicaciones que requieren sistemas de adquisición de datos, procesamiento y almacenamiento cada vez más intensivo. Como resultado de ese cambio hacia la computación intensiva de datos, existe una superposición, cada vez mayor, entre la computación de alto rendimiento (HPC) y las técnicas Big Data en las aplicaciones, pues muchas aplicaciones HPC producen grandes volúmenes de datos, y Big Data necesita capacidades HPC.

La hipótesis de esta tesis es que hay un gran potencial en la interoperabilidad y convergencia de los sistemas HPC y Big Data, siendo crucial para el futuro tratar una unificación de ambos para hacer frente a un amplio espectro de problemas de investigación. Por lo tanto, el objetivo principal de esta tesis es la propuesta y desarrollo de un sistema de monitorización que facilite la convergencia de los paradigmas HPC y Big Data gracias a la provisión de datos sobre el comportamiento de las aplicaciones en un entorno en el que se pueden ejecutar aplicaciones de ambos mundos, ofreciendo información útil para mejorar la escalabilidad, la explotación de la localidad de datos y la adaptabilidad en los computadores de gran escala. Para lograr este objetivo, el foco se ha centrado en el diseño de mecanismos de monitorización y localización de recursos para explotar el paralelismo en todos los niveles de la pila del software. El resultado es un framework de monitorización en dos niveles (tanto a nivel de nodo como de aplicación) con una baja carga computacional, escalable, y que se puede comunicar con distintos módulos gracias a una API proporcionada para tal objetivo. Todos datos recolectados se difunden para facilitar la realización de mejoras de manera global en todo el sistema, y así evitar desajustes entre capas, lo que combinado con las técnicas aplicadas para lidiar con la tolerancia a fallos, hace que el sistema sea robusto y con una alta disponibilidad.

Por otro lado, el framework desarrollado incluye un planificador de tareas capaz de gestionar el lanzamiento de aplicaciones, la migración de las mismas entre nodos, además de la posibilidad de incrementar o disminuir su número de procesos de forma dinámica. Todo ello gracias a la cooperación con otros módulos que se integran en LIMITLESS, y cuyo objetivo es optimizar la ejecución de una pila de aplicaciones en base a políticas multicriterio. Esta funcionalidad se llama *planificación de grano grueso*.

Para un mejor desempeño y con el objetivo de reducir más aún la carga durante la ejecución, se han aplicado distintas optimizaciones en distintos niveles para tratar de reducir las comunicaciones entre componentes, a la vez que se trata de evitar la pérdida de información. Para lograr este objetivo se ha hecho uso de técnicas de filtrado de datos, algoritmos de Machine Learning (ML), y Redes Neuronales (NN).

Finalmente, para obtener mejores resultados en la planificación de aplicaciones y para diseñar nuevas políticas de planificación multi-criterio, los datos de monitorización

recolectados han sido combinados con nuevos algoritmos de ML para identificar (por medio de algoritmos de clasificación) aplicaciones y sus fases de ejecución. Todo ello realizando tareas de profiling offline. Gracias a estas técnicas, LIMITLESS puede detectar en qué fase de su ejecución se encuentra una determinada aplicación e intentar compartir los recursos de computacionales con otras aplicaciones que sean compatibles (no se produce una degradación del rendimiento entre ellas cuando ambas se ejecutan a la vez en el mismo nodo). Esta funcionalidad se llama *planificación de grano fino* y puede reducir el tiempo total de ejecución de la pila de aplicaciones en los casos de uso porque realiza un uso más eficiente de los recursos de las máquinas.

## CONTENTS

1. INTRODUCTION. . . . .	1
1.1. Definitions and Scope . . . . .	1
1.2. Motivation . . . . .	3
1.3. Objectives. . . . .	5
1.4. Research methodology. . . . .	6
1.5. Structure and content . . . . .	6
2. STATE-OF-THE-ART . . . . .	8
2.1. Monitoring systems . . . . .	9
2.2. Scheduling algorithms . . . . .	14
2.3. Modeling, securing and predicting based on monitoring data . . . . .	20
2.4. Summary . . . . .	26
3. SYSTEM ARCHITECTURE AND DESIGN . . . . .	27
3.1. LIMITLESS System monitor . . . . .	27
3.2. Smart Analytic Component . . . . .	32
3.3. Application scheduler . . . . .	33
3.4. InfiniBand support . . . . .	34
3.5. Cooperation with third-party components . . . . .	34
3.5.1. FlexMPI. . . . .	35
3.5.2. CLARISSE . . . . .	37
3.5.3. ElasticSearch and Kibana . . . . .	37
3.6. Summary . . . . .	39
4. FRAMEWORK LOGIC AND ALGORITHMS . . . . .	41
4.1. Performance counter collection . . . . .	41
4.2. Monitoring system . . . . .	43
4.2.1. Communication between components . . . . .	47
4.2.2. Monitoring policies . . . . .	48
4.2.3. Fault tolerance . . . . .	49

4.2.4. Limitless Daemon Server. . . . .	50
4.3. Communication API and Visualization tools . . . . .	52
4.4. Improving control congestion for InfiniBand networks. . . . .	57
4.5. Summary . . . . .	59
5. ANALYTIC COMPONENT . . . . .	60
5.1. Smart analytic support . . . . .	60
5.2. Multi-criteria scheduling . . . . .	63
5.2.1. Scheduling based on monitoring: <i>Coarse-grain scheduling</i> . . . . .	63
5.2.2. Scheduling based on monitoring: <i>Fine-grained scheduling</i> . . . . .	66
5.3. Summary . . . . .	69
6. THEORETICAL MODELING. . . . .	70
6.1. System monitor model. . . . .	70
6.1.1. Communication model. . . . .	73
6.1.2. Server workload . . . . .	77
6.2. In-node threshold filter. . . . .	78
6.3. Fault tolerance . . . . .	79
6.4. Simulation model . . . . .	80
6.4.1. Simulation model validation . . . . .	83
6.5. Communication cost calculator . . . . .	84
6.6. In-node threshold calculation . . . . .	89
6.7. Summary . . . . .	92
7. EVALUATION AND RESULTS. . . . .	93
7.1. Monitoring overhead. . . . .	93
7.2. LIMITLESS monitoring tool vs Collectd . . . . .	95
7.3. Scheduling based on monitoring . . . . .	97
7.3.1. Coarse-grain scheduling based on monitoring information. . . . .	98
7.3.2. Fine-grained scheduling based on monitoring . . . . .	108
7.4. Optimizations. . . . .	115
7.4.1. In-node analysis optimization. . . . .	116
7.4.2. In-transit analysis optimization . . . . .	117

7.5. Improving control congestion for InfiniBand networks. . . . .	131
7.5.1. Platform and experiments description . . . . .	131
7.5.2. Results . . . . .	132
7.6. Summary . . . . .	137
8. CONCLUSION AND FUTURE WORK . . . . .	139
8.1. Contribution . . . . .	141
8.2. Future work. . . . .	142
BIBLIOGRAPHY. . . . .	143

## LIST OF FIGURES

1.1	Evolution of the internet traffic, data centers workload and energy usage for the last ten years [7]. The Y-axis represents the growth factor. . . . .	3
1.2	Up-to-date graph of Moore's Law based on a figure by Kurzweil [Steve Jurvetson, 10 Dec. 2016] [8]. . . . .	4
1.3	Flow diagram of the methodology used during the PhD. thesis. . . . .	7
2.1	Scheduling in HPC - Exclusive scheduling policy. . . . .	16
2.2	Scheduling in HPC - Shared scheduling policy when an application uses all cores from a node due to the availability of free computational resources. . . . .	16
2.3	Scheduling in HPC - Shared scheduling policy when an application uses the unused cores from various nodes due to their free computational resources. . . . .	17
2.4	This taxonomy represents different actions to do with monitoring information: displaying the incoming metrics to view the current state of the system, generating application models for application profiling, creating a list of trusted applications and send notifications if new application is detected, and predicting future system and application states to improve the scheduling task. . . . .	20
3.1	General overview of the LIMITLESS architecture and interrelation with other components (Scheduler, CLARISSE and FlexMPI. . . . .	28
3.2	LIMITLESS - Deployment with fault tolerance mechanisms enabled in the first branch of the topology. . . . .	31
3.3	Relation between ES, LA and the application scheduler. . . . .	32
3.4	Explanation of how and MPI application is executed in FlexMPI environment. . . . .	36
3.5	ELK stack vs $ES + K + LIMITLESS$ architecture. . . . .	38
3.6	Analytic component - Integration examples. . . . .	40
4.1	Top command - Example output. . . . .	42
4.2	Systems monitor architecture. . . . .	44
4.3	LIMITLESS - Deployment with fault tolerance mechanisms enabled in the first branch of the topology. . . . .	49
4.4	LDS architecture. . . . .	50

4.5	fLASHINg - Visualizing general performance metrics for all nodes registered and monitored by LIMITLESS. . . . .	54
4.6	fLASHINg - Visualizing the CPU heatmap from the nodes registered and monitored by LIMITLESS. . . . .	55
4.7	Kibana - Visualizing general state of the cluster. Each point represents an aggregation of nodes (bigger sizes represent more nodes, and smaller, less nodes). . . . .	55
4.8	Kibana - Visualizing the general performance metrics for a certain node with IP 10.0.40.19 in a certain moment. . . . .	56
4.9	Kibana - Visualizing the Kibana dashboard which analyzes the data and provides functions to exploit the information (due to new plugins). . . . .	56
4.10	Representation of the pipeline from when the data is collected until instructions are sent to the congestion control. . . . .	58
5.1	Systems monitor and scheduling architecture for coarse-grain scheduling. . . . .	64
5.3	Fine-grain scheduling result - Execution phases where both applications can share a node. . . . .	68
5.2	CPU execution patterns of two random applications that will share a node. . . . .	68
6.1	Relation between framework components and the sets defined in the theoretical model in Definitions 1 and 2. . . . .	71
6.2	Relation between framework components and the sets defined in Definitions 3. . . . .	72
6.3	Description of the communication overheads between the framework components. It corresponds to the overheads defined in the set of Definitions 13, 14 and 15. . . . .	76
6.4	OMNET++. Example with a rack of 20 LDM nodes and one LDS. The devices in the middle are switches that allow the topology representation. . . . .	81
6.5	Screenshot of the SIMCAN Scenario Creator tool. . . . .	82
6.6	LIMITLESS Deployment Checker - Example output of the execution of <i>cmd line 1</i> without TMR. . . . .	86
6.7	LIMITLESS Deployment Checker - Example output of the execution of <i>cmd line 2</i> with TMR in mode 1 (no redundancy). . . . .	87
6.8	LIMITLESS - TMR connection in the worst case: each component will send the same information to three next components in the framework. . . . .	87



6.9	LIMITLESS Deployment Checker - Example output of the execution of <i>cmd line 3</i> the with TMR enabled and errors due to consuming the network bandwidth and the maximum nodes that an aggregator or a server can manage. . . . .	89
6.10	Example - Execution of the command-line <i>cmd line 4</i> based on a monitoring log collected in a real cluster. . . . .	91
6.11	Application model - Original Memory and CPU performance of Jacobi algorithm execution. . . . .	91
6.12	Application model - Memory and CPU performance simulated with value 1% in in-node threshold. . . . .	92
7.1	LIMITLESS - CPU usage collected during the execution of Jacobi. . . . .	97
7.2	LIMITLESS - MEM usage collected during the execution of Jacobi. . . . .	98
7.3	Collectd - CPU usage obtained during the execution of Jacobi. . . . .	98
7.4	Collectd - Memory usage obtained during the execution of Jacobi. . . . .	99
7.5	Schenario A - Coarse-grain scheduling evaluation. Each bar of each application shows the execution time per 100 iterations. Applications with striped bars are applications that create interference. Applications with two bars exhibit change in the execution time due to interferences. . . . .	102
7.6	Scenario A - Gantt diagram of the execution when the nodes can be shared, but the interference detection is not active. The length of diagram corresponds to the makespan, with a value of 14.889 seconds. . . . .	103
7.7	Scenario A - System load using the coarse-grain scheduling policy, including shared nodes and interference detection. . . . .	104
7.8	Scenario A - System load using shared nodes, but disabling CLARISSE interference detection. . . . .	105
7.9	Scenario A - System load using a typical exclusive policy without shared nodes and interference detection. . . . .	106
7.10	Scenario B - Coarse-grain scheduling evaluation. Each bar of each application shows the execution time per 100 iterations. Applications with striped bars with are applications that create interference. Applications with two bars exhibit change in the execution time due to interferences. . . . .	107
7.11	Scenario B - Gantt diagram of the execution when the nodes can be shared, but the interference detection is not active. The length of diagram corresponds to the makespan, with a value of 13.256 seconds. . . . .	108
7.12	Distribution of the number of infections with COVID-19 during EpiGraph simulation for use cases 1 and 2. . . . .	109

7.13	EpiGraph use case 1 - Communication during the execution which shows two waves. . . . .	110
7.14	EpiGraph use case 2 - Communication during the execution which shows one wave. . . . .	110
7.15	Use case 1 - Relation between EpiGraph and MG when they are executed in the same node at the same time (two waves). . . . .	112
7.16	Use case 2 - Relation between EpiGraph and MG when they are executed in the same node at the same time (one wave). . . . .	113
7.17	Use case 1 (Node 1) result combining EpiGraph and MG when LIMITLESS detects compatible phases between both applications. . . . .	113
7.18	Use case 2 (Node 1) result combining EpiGraph and MG when LIMITLESS detects compatible phases between both applications. . . . .	114
7.19	Use case 3 (Node 1) result combining EpiGraph and MG when LIMITLESS detects compatible phases between both applications. . . . .	115
7.20	Use case 4 (Node 1) result combining EpiGraph and MG when LIMITLESS detects compatible phases between both applications. . . . .	115
7.21	Synthetic benchmark with three different CPU phases to evaluate the in-node analysis impact. . . . .	117
7.22	A comparison between the number of monitoring packets sent with and without in-node analysis optimization, including two tolerance values. . .	118
7.23	In-node analysis A - CPU monitored during 24h . . . . .	118
7.24	In-node analysis A - CPU monitored using 10% tolerance. . . . .	119
7.25	In-node analysis B - Memory monitored during 24h . . . . .	119
7.26	In-node analysis B - Memory monitored using 10% tolerance. . . . .	120
7.27	In-node analysis A - Error as difference of CPU performance with/without tolerance, setting this value un 10% (original figure in Figure 7.24). . . .	120
7.28	LIMITLESS model generation versus top counters. Jacobi algorithm use case. . . . .	122
7.29	Free - Jacobi memory pattern. The total memory consumption is represented as a percentage of use. . . . .	122
7.30	LIMITLESS model generation versus top counters. Scalar Penta-diagonal solver use case. . . . .	123
7.31	LIMITLESS model generation versus top counters. Integer Sort use case. .	123
7.32	LIMITLESS model generation versus top counters. BT-IO use case. . . .	124

7.33	LIMITLESS - BTIO read/write operations over time. The Y-axis represents the percentage of read/writes with respect to the total. . . . .	124
7.34	LIMITLESS model generation versus top counters. Epigraph use case. . .	125
7.35	LIMITLESS - Epigraph communication usage. The Y-axis represents the communication usage in Kbps. . . . .	125
7.36	Performance metrics for Jacobi method executed in an exclusive node. . .	127
7.37	Performance metrics for BT-IO benchmark from NAS Parallel Benchmarks.	127
7.38	Use case 1 - CPU use of Jacobi method executed in an exclusive node. . .	128
7.39	Use case 2 - CPU use of two Jacobi instances executing in the same node. There is I/O interference between them. . . . .	129
7.40	Use case 3 - CPU use of BTIO benchmark from NAS Parallel Benchmarks.	129
7.41	36-node KNS topology built on the cluster. . . . .	132
7.42	<i>portXmitData</i> in node 30. . . . .	133
7.43	<i>portXmitWait</i> in node 30. . . . .	134
7.44	<i>portXmitData</i> in node 37. . . . .	134
7.45	<i>portXmitWait</i> in node 37. . . . .	135
7.46	P2PBW+Sync. No congestion (Average). . . . .	136
7.47	P2PBW+Sync. No congestion (99%-tile). . . . .	136
7.48	P2PBW+Sync with congestion (Average). . . . .	137
7.49	P2PBW+Sync with congestion (99%-tile). . . . .	137

## LIST OF TABLES

2.1	Key properties of some monitoring platforms. . . . .	13
2.2	Machine Learning algorithms studied and tested. . . . .	25
3.1	Parameters that can be collected in a monitored node. . . . .	30
3.2	Parameters collected in InfiniBand networks. . . . .	30
3.3	Parameters collected with IPMITOOL. . . . .	30
3.4	Performance counters collected from IBA subnet manager. . . . .	35
3.5	Information provided by FlexMPI to the framework for each iteration and for each process. . . . .	36
4.1	Information provided by <i>rstatd</i> process. . . . .	43
4.2	Management functions available in LIMITLESS API . . . . .	53
4.3	Query functions available in LIMITLESS API . . . . .	54
6.1	Maximum number of packages allowed by CL . . . . .	76
6.2	Simulation vs Real - Experimentation under different conditions in both simulated and real environments. Every use case simulates one hour. . .	83
6.3	LIMITLESS - Deployment Checker parameters. . . . .	84
7.1	Summary - LIMITLESS monitor overhead under different sampling inter- vals. . . . .	94
7.2	Monitoring overhead with the minimum sampling interval in a compute- node with Intel(R) Xeon(R) Silver 4214 CPU with 12 real cores, and 24 virtual cores (Hyperthreading). . . . .	95
7.3	Monitoring overhead with the minimum sampling interval in a compute- node with Intel(R) Xeon(R) Gold 6138 CPU with 20 real cores, and 40 virtual cores (Hyperthreading). . . . .	95
7.4	LIMITLESS monitor overhead in a local PC. . . . .	96
7.5	Summary - Collectd overhead under different sampling intervals in a local PC. . . . .	96
7.6	Use cases characteristics for the evaluation. . . . .	99

7.7	Example of a workflow that generates interference. T1, T2, T3 are the execution time per 10 iterations. Overhead represents the overhead of the migration process measured in seconds. . . . .	101
7.8	Summary - Comparison between results obtained in Scenario A and Scenario B with all of the policies. . . . .	108
7.9	Accuracy of the different classification algorithms using patterns of 60 and 100 seconds of EpiGraph. . . . .	111
7.10	Execution summary of the use cases. Accumulated CPU time for each use case and scheduling. . . . .	116
7.11	Applications modeled by LIMITLESS. . . . .	121
7.12	Predictors accuracy at node level monitoring - Accuracy expressed as a percentage of correct predictions. . . . .	130
7.13	Predictors accuracy at application-level monitoring - Accuracy expressed as a percentage of correct predictions. . . . .	130
7.14	Percentage of network traffic saved of all the prediction algorithms, including CPU, IO, Memory and Network collected metrics. . . . .	131
7.15	Congestion control configuration. . . . .	133

## ABBREVIATIONS

<b>LIMITLESS</b>	<b>L</b> ight-weight <b>M</b> onItoring <b>T</b> ool for <b>L</b> arge <b>E</b> <b>S</b> cale <b>S</b> ystems
<b>GPGPU</b>	<b>G</b> eneral <b>P</b> urpose <b>G</b> raphic <b>P</b> rocessing <b>U</b> nit
<b>HPC</b>	<b>H</b> igh <b>P</b> erformance <b>C</b> omputing
<b>IT</b>	<b>I</b> nformation <b>T</b> echnologies
<b>I/O</b>	<b>I</b> nput / <b>O</b> utput
<b>LAPACK</b>	<b>L</b> inear <b>A</b> lgebra <b>P</b> ACKage
<b>MPI</b>	<b>M</b> essage <b>P</b> assing <b>I</b> nterface
<b>OpenMP</b>	<b>O</b> pen <b>M</b> ulti- <b>P</b> rocessing
<b>SaaS</b>	<b>S</b> oftware <b>a</b> s <b>a</b> <b>S</b> ervice
<b>UC3M</b>	<b>U</b> niversity <b>C</b> arlos <b>III</b> of <b>M</b> adrid
<b>NN</b>	<b>N</b> eural <b>N</b> etwork
<b>ML</b>	<b>M</b> achine <b>L</b> earning
<b>IP</b>	<b>I</b> nternet <b>P</b> rotocol
<b>PAPI</b>	<b>P</b> erformance <b>A</b> pplication <b>P</b> rogramming <b>I</b> nterface
<b>LSDS</b>	<b>L</b> arge- <b>S</b> cale <b>D</b> istributed <b>S</b> ystems
<b>FLOPS</b>	<b>F</b> loating <b>P</b> oint <b>O</b> perations per <b>S</b> econd
<b>LSTM</b>	<b>L</b> ong <b>S</b> hort- <b>T</b> erm <b>M</b> emory
<b>ANN</b>	<b>A</b> rtificial <b>N</b> eural <b>N</b> etwork
<b>NN</b>	<b>N</b> earest <b>N</b> eighbour
<b>MS</b>	<b>M</b> onitoring <b>S</b> ystems
<b>MILP</b>	<b>M</b> ixed <b>I</b> nteger <b>L</b> inear <b>P</b> rogramming
<b>DVFS</b>	<b>D</b> ynamic <b>V</b> oltage <b>F</b> requency <b>S</b> caling
<b>CNN</b>	<b>C</b> onvolutional <b>N</b> eural <b>N</b> etwork
<b>CCR</b>	<b>C</b> ross- <b>C</b> luster <b>R</b> eplication
<b>IBA</b>	<b>I</b> nfini <b>B</b> and <b>A</b> rchitecture
<b>WD</b>	<b>W</b> atch <b>D</b> og
<b>TMR</b>	<b>T</b> riple <b>M</b> odular <b>R</b> edundancy
<b>API</b>	<b>A</b> pplication <b>P</b> rogramming <b>I</b> nterface
<b>LDM</b>	<b>L</b> IMITLESS <b>D</b> aemon <b>M</b> onitor
<b>LDA</b>	<b>L</b> IMITLESS <b>D</b> aemon <b>A</b> ggregator
<b>LDS</b>	<b>L</b> IMITLESS <b>D</b> aemon <b>S</b> erver
<b>ES</b>	<b>E</b> lastic <b>S</b> earch
<b>LA</b>	<b>L</b> IMITLESS <b>A</b> nalytics
<b>fFLASHING</b>	<b>L</b> ARGE <b>S</b> cale <b>H</b> eatmap <b>v</b> isualizer with <b>N</b> odejs
<b>KNS</b>	<b>K</b> eysto <b>N</b> enes <b>S</b>
<b>AdaBoost</b>	<b>A</b> daptive <b>B</b> oosting
<b>ANBC</b>	<b>A</b> daptive <b>N</b> aive <b>B</b> ayes <b>C</b> lassifier
<b>BAG</b>	<b>B</b> oost <b>r</b> ap <b>A</b> Ggregating
<b>DT</b>	<b>D</b> ecision <b>T</b> ree
<b>DTW</b>	<b>D</b> ynamic <b>T</b> ime <b>W</b> arping
<b>GMM</b>	<b>G</b> aussian <b>M</b> ixture <b>M</b> odel <b>C</b> lassifier

<b>HMM</b>	<b>H</b> idden <b>M</b> arkov <b>M</b> odels
<b>SVM</b>	<b>S</b> upport <b>V</b> ector <b>M</b> achine
<b>MAD</b>	<b>M</b> anagement <b>D</b> atagrams
<b>FECN</b>	<b>F</b> orward <b>E</b> xplicit <b>C</b> ongestion <b>N</b> otification
<b>BECN</b>	<b>B</b> ackward <b>E</b> xplicit <b>C</b> ongestion <b>N</b> otification
<b>CC</b>	<b>C</b> ongestion <b>C</b> ontrol
<b>IBA</b>	<b>I</b> nfin <b>B</b> and <b>A</b> rchitecture
<b>GPCNeT</b>	<b>G</b> lobal <b>P</b> erformance and <b>C</b> ongestion <b>N</b> etwork <b>T</b> est

## 1. INTRODUCTION

### 1.1. Definitions and Scope

HPC and Big Data are, traditionally, designed to solve different problems, and that is the reason why they are built in a different way. At first, the typical HPC infrastructure has the computation subsystems and the storage decoupled, using a distributed filesystem to store the data (e.g. GPFS, Lustre). However, Big Data frameworks mix the computation and the storage in the same node [5].

HPC and Big Data tools and developers are separated, basically because HPC has been oriented towards computationally intensive problems and are tightly coupled, while Big Data has been designed towards data analysis in high scalable and loosely coupled applications. Between both worlds or paradigms, there are different degrees of HPC/Big Data applications that need intensive computation, storage capability and data management. Those applications could be executed in both platforms, but none of them is completely ideal to be run (due to their design) in both development framework, mainly for the scalability requirements, performance, resource efficiency, etc.

Data intensive applications require a unification of the, traditionally, separated HPC and Big Data paradigms in order to allow developers and researchers understand better the results, as well as perform the experiments in an efficient way. As the simulations are every time more complex, the input, intermediate and output data sets grow noticeably, and there are new HPC problems that are data-intensive [6]. Moreover, as has been mentioned before, there are a really close relationship between data-intensive applications and the necessity of decouple the data and the computation.

For everything described before, researchers are agreeing with the idea that there is a growing need of those frameworks that deal with problems where HPC performance and the Big Data flexibility converge. This interest is justified, mainly, by the fact that storage sharing can help infrastructures to integrate themselves quickly.

Nowadays, in the field of Information Technology (IT) there are two main ways to operate: Cloud Computing and Cluster Computing. We can say that Cloud Computing is based on many areas of computer science such as HPC, Grid Computing, Virtualization and Utility Computing, and Cluster Computing is the way that has been applied with technical or scientific applications to run more complex tasks like forecast simulations, which needs more power of computation than general purpose applications. Both ways



have a similar behavior if we focus our attention in how they solve their problems: they distribute their work/tasks in many sub-tasks to run them separately and simultaneously in a computer group, to run the whole tasks more rapidly.

Moreover, the architectures for Cloud Computing are different than for Cluster Computing because they have different objectives and abstraction layers. Cloud Computing has become a paradigm for delivering services over the Internet, and Cluster Computing has become a paradigm for researching fields. Monitoring tools for clouds and clusters is an important task for both providers and clients for many reasons: capacity, resource and performance planning and management. In this scenario, monitoring activities must be fine-grained and accurate to operate rapidly and efficiently these platforms, and to manage their complexity.

In order to guarantee the performance required by the applications (and services in cloud environments), administrators and developers have to (1) quantify the capacity and the resources (CPU, memory, storage, etc), and (2) calculate the estimated workload. However, there are two ways to face the develop of a monitor for cloud and cluster computing based on what we want to analyze: high-level monitoring is related to information on the status of virtual platform, and low-level monitoring is related to the status of the physical infrastructure of the whole cloud/cluster. In our case, we want to understand what is happening actually in terms of performance, how users use the platform resources, and it is necessary to obtain this information in soft real time. Analyzing this information, we could infer system states for (i) resource provisioning, (ii) resource planning, (iii) detection of system failures, (iv) and other management actions like power off unused machines to save energy.

To manage such infrastructures in an efficient way, the monitoring tool must support real or soft real-time operation and must scale up to thousands of heterogeneous nodes, I/O sub-systems and network topologies. If these requirements are not achieved, system managers won't have a global view of their systems.

If the monitoring tool can reach all those requirements and do it in an efficient way, it will be able to deal with the next generation of clusters and data centers. Besides, it could be possible to obtain important information about the system and applications (due to the amount of data collected), providing it in real-time. That information is important, not only for the user but also for the scheduler. The scheduler could use the information to design new policies based on the current state of the machines and the available resources at every moment. It means that the current policies can be improved and new ones can be designed (for example, allocating applications to nodes with free computational resources).

The scheduler can take advantage of the collected information to allocate the applica-

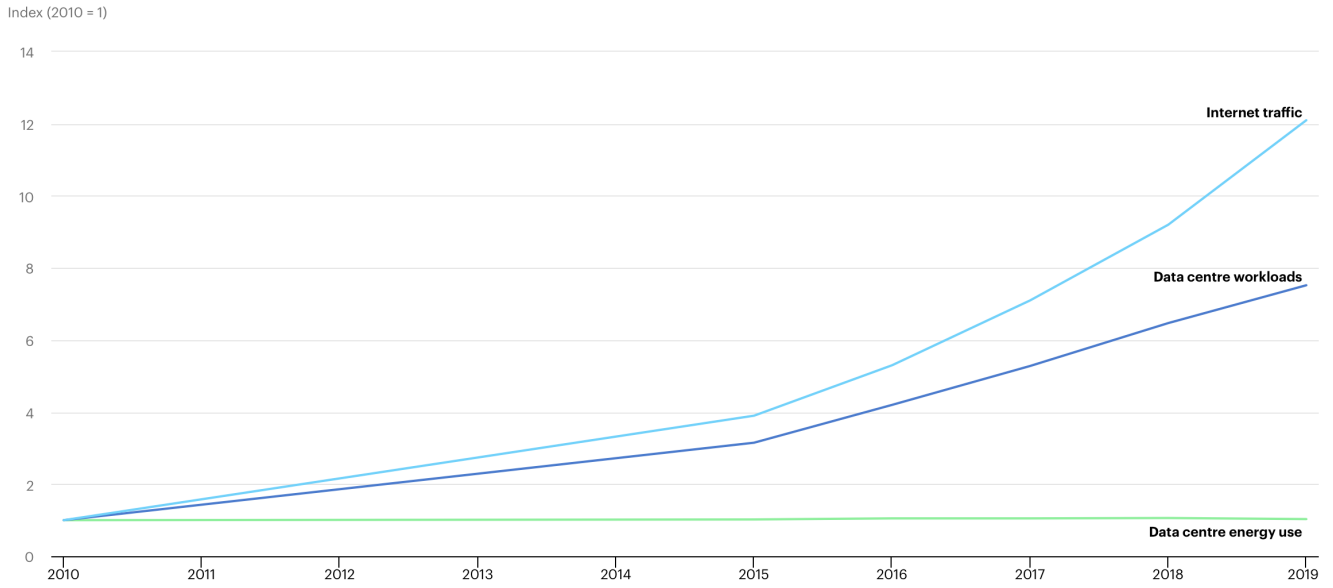


Figure 1.1: Evolution of the internet traffic, data centers workload and energy usage for the last ten years [7]. The Y-axis represents the growth factor.

tions efficiently, relating the profiles of the applications to the available resources on each machine. If the scheduler receives the application's performance metrics, it can get the profile and behavior of each one. Then, the scheduler can select better nodes to execute one application or execute it in a shared node with other applications that do not produce performance degradation (interference). This *interference* between applications means that, when two (or more) applications are executed in the same compute-node, one (or more) of them suffers a performance degradation due to de contest to use the available resources. Hence, detecting this interference is a requirement to design new scheduling policies for shared nodes.

## 1.2. Motivation

The growth of the Internet caused an explosion of traffic and data in the world since the beginning of twenty-first century. As Figure 1.1 shows, the last ten years the traffic and the data have increased in factors of 12x and 8x respectively. In the past, the traditional way to store the data consist of a database server, which is a computer with a lot of storage devices connected, but currently, this idea is unused. The amount of data, its variety, the speed with which the information is produced, etc., require to distribute the process across a big infrastructure designed for this purpose. Moreover, if we think about Big Data, the whole information has to be collected, processed, stored, and analyzed in Large-Scale Distributed

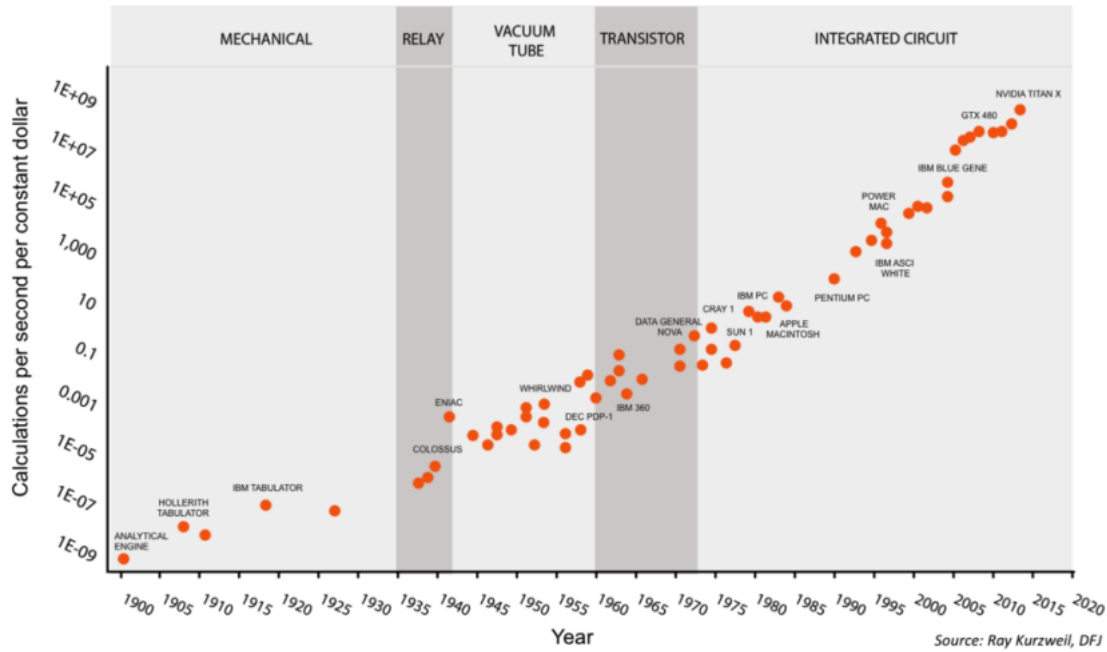


Figure 1.2: Up-to-date graph of Moore's Law based on a figure by Kurzweil [Steve Jurvetson, 10 Dec. 2016] [8].

Systems (LSDS) in a distributed way.

On the other hand, there is an increment of the computing power (which can be seen in Figure 1.2). We are on the road to the future generation of HPC machines: *exascale computing*.

The last years, these two separated paradigms (HPC and Big Data) are in a process of convergence. The Big Data community wants to use Machine Learning and Deep Learning algorithms, that are computation intensive algorithms, from HPC. However, HPC users (who are focused on computation) are seeing how the computation platforms have to deal with large amounts of data (which is generated by the executing applications), and how that are becoming a bottleneck of the storage infrastructure.

This convergence is the origin of the idea behind this PhD. thesis: *how to know what is happening in the infrastructure and how to improve the performance of the applications providing new scheduling techniques based on the holistic view of the platform*. If we can provide solutions for those concepts, users and system administrators could improve infrastructures and applications to accommodate future hybrid architectures. This accommodation should imply little effort for reducing the makespan of their tasks. Besides, one alternative to do it consists of scheduling applications concurrently in the same node.

### 1.3. Objectives

The hypothesis that origins this PhD. thesis, as it has been shown before, is based on the increasing demand for more and more computational resources to face the growth of Internet traffic, the amount of new data produced each day, and the tendency to provide *Software as a Service* (SaaS), instead of desktop applications (or installed applications). For these reasons, the main objective of this PhD. thesis is to develop a system monitor for large-scale systems in combination with desirable features, which provide intelligence to the system and allow improving application scheduling, as well as the understanding of what happens in the system (both at node and application-level) continuously. In order to face these challenges, the following objectives have been defined:

- **O1. Explore new techniques and abstractions to allow HPC applications the exploiting of the parallelism, locality, elasticity and adaptability of LSDS.** The idea is to design a monitoring tool capable of making decisions to help applications run efficiently, identifying where the processes that use the same data are executed, allocating the best nodes based on the application's profile, etc.
- **O2. Design a monitoring tool to be able to run in heterogeneous and homogeneous machines.** The idea is to research how to provide the performance information regardless of the system's features.
- **O3. Explore different techniques to collect information both at node-level and application-level.** A complete monitoring tool for HPC and Big Data should include different methods to read the performance counters, disable those that are not detected, and provide as many metrics as possible.
- **O4. Design new scheduling policies based on multiple criteria.** The idea consists of improving the current scheduling policies to provide new strategies based on using Machine Learning and Neural Network algorithms. With these methods, the framework should be able to identify applications, predict their future performance, and improve the scheduling task based on the data collected, the detection of interference between applications, and using shared nodes.
- **O5. Explore how to bring machine learning and neural network support to the proposed framework.** The plan is to use Machine Learning and Neural Networks to exploit the information. The framework should be able to recognize applications, predict their performance, detect interference between their executions, schedule better the executions, and design topology-aware deployments of the monitor efficiently.

These objectives are the main goals that this PhD. thesis wants to reach based on the initial hypothesis.

#### **1.4. Research methodology**

The methodology used to develop this PhD. thesis is an empirical and iterative methodology, which consists of the following steps. Besides, the process is repeated as many times as necessary depending on the achievement of the objectives previously defined. The methodology is the following one:

1. Study of the state-of-the-art regarding high-performance techniques for monitoring and scheduling applications in large-scale systems, as well as their application in different infrastructures (HPC, BigData, Grids, etc).
2. Analysis of requirements for the strengths and weaknesses found in the previous study in order to design a proposal that covers the weaknesses detected in the state-of-the-art.
3. Propose and develop a framework model that is capable of monitoring and providing support to schedule applications in large-scale distributed systems. Besides, define new components and features that have not been covered in other similar models.
4. Perform a comprehensive evaluation of the algorithms and tools proposed in point 3, as well as a comparison with other similar works identified in the state-of-the-art. The objective is to offer results that can lead to the integration of the proposals in other systems.

#### **1.5. Structure and content**

The rest of the document is divided into the following chapters:

- Chapter 2, *State-of-the-art*, reviews the state-of-the-art monitoring and scheduling large scale distributed systems. It also includes HPC techniques that can be applied to improve scalability and resilience, and other smart techniques to provide intelligence and visualization features.
- Chapter 3, *System architecture and design*, presents the developed monitoring framework and shows its components: a resource collector that is executed in the

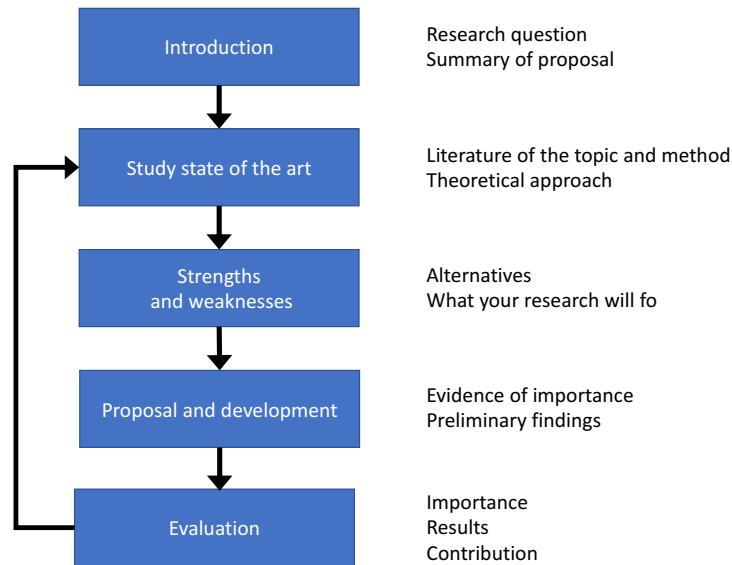


Figure 1.3: Flow diagram of the methodology used during the PhD. thesis.

compute nodes, components that provide scalability retransmitting the information, a central server that gathers the whole data collected, an analytic component that executes smart algorithms for allocating the applications (it is done by using new scheduling policies), and other integrated components.

- Chapter 4, *Framework logic and implementation details*, presents how each component has been implemented and optimized, including its functions and algorithms.
- Chapter 5, *Analytic component*, shows details about how the framework is able to identify applications, predict performance and schedule applications dynamically based on the information collected.
- Chapter 6, *Mathematical model*, shows the mathematical representation of the LIMITLESS system monitor, which explains how to calculate the scalability, and how to design different topologies.
- Chapter 7, *Evaluation and results*, shows a detailed evaluation of each component, its performance, the impact of the optimizations, and the results obtained due to the analytic component scheduling applications.
- Chapter 8, *Conclusion and future work*, summarizes the results and the contributions of this work, as well as the future work.

## 2. STATE-OF-THE-ART

In order to put this monitoring and scheduling work in its proper context, I give a brief overview of some approaches that are representative for the areas of monitoring systems and application scheduling in large scale systems.

Several studies have attempted to analyze and define the basic concepts related to large-scale distributed systems (LSDS) monitoring in general, and cloud monitoring in particular. This is because developers think that monitoring in cloud systems is harder than in large-scale systems, mainly because monitoring virtual resources is a mayor limitation in the cloud context. But monitoring high-performance computing (HPC) large-scale platforms is a difficult task, which becomes more and more challenging as the complexity and scale of the infrastructure increases. Moreover, managing these kinds of systems is also a complex task, and, if some components are located in different security domains, have different configurations or operating systems, it becomes even more difficult.

Before focusing on the background of this PhD. thesis, it is necessary to distinguish between the different types of monitoring applications in the HPC/Big Data scope. There are two main groups depending on the monitorization level: node-level and application-level monitoring systems. The first group collects the metrics from the hardware components, providing a clear view of how a node (or groups of nodes) is at a certain moment: node-level performance metrics. This information is useful when the user needs to know how many resources are available in order to launch more applications, to increase/reduce the number of processes in execution, or to determine if a node is still running after a failure sign, for example. The second group is in charge of collecting the performance counters from applications. This information does not provide a general view of the machine on its own, but it can be used to check the correct execution of the application, if it is possible to change the number of resources that it uses, etc. For instance, SLURM, which is a system that provides job scheduling and cluster managing, provides methods to allocate resources and monitor job executions, but this task can be improved giving more specific data from the system and the applications.

LIMITLESS arises from the idea of providing a simple, quickly, and general view of all machines in a cluster, that can be used to improve other features related to performance: migrating applications, increasing or reducing the number of processes, scheduling applications based in user-defined policies, etc. For those reasons, the proposed framework will try to combine the best of both levels of monitoring: software and hardware levels.

## 2.1. Monitoring systems

Monitoring information in distributed systems (DS) has been addressed through many approaches and tools from a technological point of view. Exascale requires new monitoring techniques, such as sub-optimal period scheduling [9] and strong usage of HPC system statistics [10], to improve the system utilization and reducing the overhead of these tools. Thus, the effort on developing monitoring tools has been continuous in HPC systems. Ganglia [11] is one of these tools, which is one of the most used monitoring tools for HPC systems, such as grids or heterogeneous clusters. However, Ganglia is not designed to be used for monitoring virtual resources, which is the main challenge in cloud environments. If the user wants to use Ganglia in a cloud environment, he needs to combine it with other tools to reach this goal (e.g., sFlow [12]). It uses carefully studied data structures and algorithms to achieve low overheads and high concurrency levels [13]. It can scale to manage clusters with 2000 nodes.

Nagios [14][15] is another example, and it consists of an open-source solution for monitoring compute-nodes and services in a closed network. It is a well-known framework used to gather monitoring information in large-scale systems, but this approach does not work in virtual environments, due to the dynamism associated with these systems, and, typically, the main collected information is related to the availability of the components. Moreover, one of the main problems using Nagios is its lack of scalability, so, in very large-scale systems as these systems want to manage, this solution is inadequate to deal with the monitoring task in the next level of scalability.

The same behavior can be observed in two other well-known systems such as GridICE [16] and MonALISA [17]. Some researchers think that this lack of functionality is a limitation [18] but these frameworks were made for monitor large-scale systems (at a host level), not for cloud environments (with virtual resources). That is why these tools provide good results on grid platforms, but not in cloud platforms. GridICE is a monitoring service architecture that provides an entire framework to manage and monitor large mainframe-style systems (for example, components that collect data and processors of those data). MonALISA is a set of autonomous subsystems (registered as dynamic services), that are able to cooperate in monitoring tasks in large-scale distributed applications, and to be used by other services or clients that require such information.

The TIMaCS project [19] is a hierarchical and scalable monitoring and management framework for very large computing systems. It reduces the complexity of managing these kinds of systems by including efficient tools for scalable low-level system monitoring. At the same time, it increases the quality of the information delivered to the system adminis-



trator because it incorporates data analysis to the original data. It allows the administrator to perform preventive actions or check the state of the whole system. One thing that is not deeply described is how the authors decided what information is “important”, what is the overhead of the low-level monitoring component, and the methodology used to analyze the information and filter it. In this PhD. thesis, LIMITLESS tries to offer the user the whole possible information and then, delegate to him/her the task of selecting what information is considered important.

Zenoss [20] is an open-source enterprise solution that allows management, monitoring and reporting on large-scale systems. Its main objective is to provide features like availability and performance monitoring, event management and user/alert management. This tool fulfills the goal of the monitoring paradigm because it makes easier decision-making. However, related to the fault tolerance, it needs to be configured to prevent the different single points of failure (e.g., number of nodes per collector, which is also directly related with the levels of redundancy). In some cases, there are problems because a collector could fail, and the system status could be wrongly monitored. That is why resilience is important. Moreover, at the same time as systems increase their complexity (number of nodes, devices, etc.), Zenoss must be restarted to add the newest platform information to its configuration.

Some researchers at Los Alamos National Laboratory (LANL) [21] propose some features that next-generation monitoring systems must have. They have developed their design and implementation in response to the scale and complexity of the new Trinity [22] deployment, but this model can be adapted to monitoring other clusters. The monitoring objectives can be summarized in: (1) the Monitoring System (MS) must cover many different kind of components such as nodes, file systems, networks, etc.; (2) MS must provide a large quantity of monitoring information at scale, like management, memory, storage, etc. (it is important to enable the trend analysis that allows us understanding the performance of the components in different situations); (3) MS must analyze the information and capture meaningful data to show the current state of a cluster. In addition, MS requires a simple but flexible design, because it must be capable of handling big amounts of data from many different sources to reach a good scalability. Moreover, the design must be also modular because, in order to monitor all kind of large-scale systems, the MS must allow integration, reconfiguration, and removal heterogeneous components quickly and dynamically.

Collected [23] is an application that runs as a daemon, which collects system and application performance metrics, also providing mechanisms to store the collected metrics in different ways. It is developed in C for performance and portability, it provides effective

networking features and is extensible (it allows the integration with other components). Its main limitation is that monitoring function is so far limited to simple threshold checking, and the developers recommend using Collectd in combination with Nagios to show more details related to machine states, times since power on, workloads, etc.

DIMON [24] is a monitoring tool designed to provide information from decentralized edge-computing networks. In this kind of clusters, end-users can actively collaborate in the self-provision of network services. Actually, monitoring these systems is really hard because the network and the number of devices constantly change. Due to the partitions of the network and the changes in the monitoring servers, there are frequent failures in the monitoring system, which produce a lack of information from different parts of the network. The main information provided by the monitor is related to the number of devices and the network state. From the performance point of view, there is no information about the overhead in the CPU in each device. DIMON also provides general information about the network traffic overhead between different phases during monitoring. However, there is not a certain value to know the impact of the monitorization in the devices.

The Supermon architecture is presented in [25]. It describes a flexible framework for cluster monitoring. It is an old proposal that describes how to monitor Linux machines based on the system call `sysctl()`. This work is useful because it introduces how the tool gets the performance metrics from three sources: `/proc`, `rstat` and `sysctl()`. The first one is a special filesystem used by the Linux kernel to store its information. The second one is a routine that gathers statistics from the Linux kernel. Finally, the third one performs different operations in the Linux kernel (e.g., visualizing performance metrics). The comparison between those three methods shows that the third one is faster than the others, but there is a limitation: the amount and richness information provided. However, the results show that it is possible to monitor a system with reduced sampling rates with an acceptable overhead, showing patterns that would not be seen on coarse-grained monitors. Supermon uses a data protocol based on symbolic expressions and can be executed in individual nodes or in entire clusters. This reason allows the author to state that this framework is scalable and can run in heterogeneous clusters, but there are no proofs of executions in LSDS.

*Distributed Modular Monitoring system* (DiMMon) [26] is another framework for monitoring distributed systems. Its main characteristics include: the collected metrics can be sent in different ways (depending on the purposes), the parameters of the nodes can be dynamically updated (reconfiguration), and it has the capability of calculating performance metrics of an individual job while data is being collected. The main issue is how the architecture has been described, where each monitored node is connected to others, and all of them send and receive messages between them to store and share the information.

Finally, another important feature of DiMMon is that each node can be different, with different tasks and metrics, improving the adaptability of the system to the applications and the users.

The next related work is not a *monitoring tool per se*. In [27] the authors present XDMoD, a complex and complete visualization tool that includes a component to transmit collected information from the nodes to the application domain. Then, the application provides charts and reports per job that are interesting for administrators and users. However, it needs a component to monitor the machines and collect the information. The key of this project is that it provides (due to external monitoring components) views and reports of the collected information. After a poll, users coincide that those views were interesting.

In the same way as the last described work, Graphite [28] is a processor and visualizer of the incoming metrics, which should have a specific format. The creators describe it as an open-source time-series metrics monitoring system. However, the application does not monitor any system, but only obtains the monitoring information from external components. The point of this application is the combination with other tools to provide a complete monitoring framework: a metrics collector, a data transmitter, and a visualizer.

CloudWatch [29] is a private tool developed by Amazon for DevOps. It collects monitoring data in the form of logs, metrics, and events, providing a unified view of AWS resources, applications, and services running on the servers. Thus, it provides a global view of the available resources, applications and services that are running. Besides, it can be used to detect anomalies and to define alarms (or events). And this work is also useful because many of these features are required by administrators and users. Note that the anomalies have different grades of interest, and the users need to define their own advices, alarms and management rules.

HP OpenCall reinforces the idea of creating a monitoring framework (a set of tools that provide performance information in different platforms) instead of a monitoring application. It includes a set of more than 50 software tools that cover the majority of the administration issues in IT areas. The project started 30 years ago and then, HP has included different tools to extend the offered functionalities. Its objective is to simplify the performance management and the availability of the systems. However, the user interfaces are based on Unix and the graphics do not seem to be smooth enough (many users discard its utilization because of its UI). Moreover, for a given action (e.g., plot the time series, set some parameters, post-process the information, etc.) it is necessary to access a certain application because each one runs independently (instead of running in a coordinated way).

Finally, other commercial tools are used in large-scale environments. One of the most famous tools is IBM Tivoli Monitoring [30] which is a set of service components to

monitor the performance and availability of distributed operating systems and applications, focusing on the physical resources (it can be used in clouds in combination with Tivoli Monitoring for Virtual Environment tool [31]).

Table 2.1: Key properties of some monitoring platforms.

Platform	Scalability	Elasticity	Adaptability	Extensibility	Resilience	Availability	Reporting	Deployment	App. profiling	Dynamic reconfiguration	Event detection	Smart analytics	Scheduling
Ganglia [11]		✓	✓	✓	✓	✓	✓				✓		
Nagios [32], [33]		✓		✓	✓	✓	✓				✓		
MonaLisa [17]	✓	✓	✓		✓	✓	✓	✓		✓			
GridICE [34]	✓	✓	✓	✓			✓	✓					
TIMaCS [35]	✓	✓	✓			✓	✓				✓		
Zenoss [20]			✓	✓	✓	✓	✓				✓		
Collectd [23]		✓				✓	✓						
Collectl [36]							✓						
Supermon [25]	✓					✓	✓						
DiMMon [26]							✓			✓			
XDMoD [27]		✓	✓		✓	✓	✓		✓		✓		
CloudWatch [29]	✓		✓		✓	✓	✓		✓		✓		
HP OpenCall [30]		✓	✓		✓	✓	✓					✓ <sup>1</sup>	
IBM Tivoli [31]	✓		✓		✓	✓	✓						

Table 2.1 shows a summary of the studied monitoring tools and their main features that are interesting in HPC environments. As it can be seen, it shows information about sixteen monitors and/or monitoring frameworks, and the main identified features are described below. The scalability refers to the ability to accommodate larger loads just by adding resources or making hardware more powerful. Elasticity is the ability to dynamically fit the resources needed to cope with high loads. Adaptability is the ability of a process to continue its execution when changes that can affect its functions are made in the system. Extensibility indicates the ability of a software to extend its functionality (for example, include support for new components) and the effort that the integration with the new components requires. Resilience is the ability of a software to absorb the negative impact of a potential problem, while continuing to provide its service. Availability is the probability that a system can continue its execution in the event of a failure. Reporting is the feature that provides useful and summarized information to the user, in a report, in a

<sup>1</sup> Among all of its applications, there are some post-processors that can be identified as “analytic” but it is not included.

file or whatever other method, automatically.

There are other useful features depending on the monitoring objective. The feature called *deployment* includes an easy way to deploy the monitor over the whole cluster without spending time doing manual configurations or coordinating the system components. Application profiling refers to the capability of providing application models and their characteristics from their executions. Dynamic reconfiguration refers to the capability of the monitor to update its parameters without restarting (it means that there is no need to stop the monitoring service, make the changes and deploy it again). Sometimes, administrators do not need monitoring tools that collect all data every time interval. For this reason, it is interesting the concept *event-driven monitoring*. It refers to a monitoring mode that only detects and sends notifications when an event occurs (events that should be configured by the monitor administrator), and these events can be, for example, thresholds defined for certain performance metrics, specific applications in execution, etc.

## 2.2. Scheduling algorithms

This PhD. thesis tries to face, as well as the large-scale cluster monitoring, the problem of optimizing the overall throughput of a set of applications executed on a cluster. The optimization is done by means of taking advantage of the monitoring information to (1) design new scheduling policies and (2) provide dynamic multi-criteria scheduling. This multi-criteria scheduling refers to new application scheduling algorithms that optimize more than one variable (e.g., makespan and communications. In this case, the algorithm will try to minimize the total execution time, while allocate applications that perform a lot of communication in nodes that are topologically far each other). To have a clear view of the state of the art on application scheduling, a study of the existing research works has been done.

This PhD. thesis will also present LIMITLESS, a monitoring and scheduling framework that shows closer coordination between the monitor and scheduler. Both components are coordinated and work together to improve the scheduling process based on the monitoring performed at system and application levels. The objective is to determine if there is performance degradation when more than one application is sharing multicore processors during HPC workloads executions [37], and if those executions produce performance hotspots. Finally, to manage these potential cases of performance degradation and hotspots, a new scheduling policy has to be defined to mitigate the effect of the contention produced by that shared-nodes [38], and trying to reduce poor scaling conditions.

Schedulers are focused on optimize three main variables: fairness, utilization and dy-

namicity (FUD). However, generally, scheduling consists of, taking a queue of applications and a list of computational resources, executing all the applications using all the available resources to complete the execution as early as possible (in other words, to reduce the makespan), taking into account the scheduling strategies specified by the administrator (scheduling policies and, in some platform, the user hints). The way to find the optimal solution with the existing number of resources and application executions submitted by the users depend on the software used (for example Slurm [39]), but the procedure is conceptually the same. Figures 2.1, 2.2 and 2.3 show an overview of how the scheduling process is carried out. The first one schedules applications based on an exclusive policy (Each application will run on a single node unless the number of processes is greater than the cores of the compute node. In that case, more than one node will be used). The next two figures show scheduling based on a shared policy, where applications can share nodes to leverage the computational resources that one of them does not use.

When a set of applications has to be scheduled, the first step starts when the users submit their applications and the related resources required by the applications (arrow 1 in figures). Then, the scheduler puts those applications in queues and evaluates the available resources, nodes or cores, depending on the policy used (bracket 2 in figures). Finally, when the scheduler has identified the nodes where an application can be run, it deploys the application based on the requested and available resources on each node (arrows 3 in figures). In Figure 2.1, an application can be run in one node, and the scheduling policy is exclusive. So that, the scheduler searches free nodes and runs the application on them. Figures 2.2 and 2.3 show the same use case, but in the first figure, there is one node that has available all the computational resources that the application needs. So that, the scheduler runs the application on that node. However, in the second figure, the first three nodes have enough resources to run the application, and the scheduler distributes the application processes between those nodes. In this case, the resources are used more efficiently, although there is a risk that the applications may interfere (there is interference between two applications if one of them, or both, suffers performance degradation when they are sharing a node) with each other, generating interference and degrading their performance (for example, two numerical memory-intensive applications which heavily use the memory hierarchy will generate interference at cache level).

In [40] the authors present a new proposal for a static scheduling policy for  $n$  periodic applications, which consists of periodic tasks on time-critical completion time. This solution uses the information about the applications once they have been executed. However, in future executions of the applications, if there is any variation in the parameters or configurations, the scheduling result could not be optimal. The algorithm used to find the optimal solution is based on an improved Mixed Integer Linear Programming (MILP). The

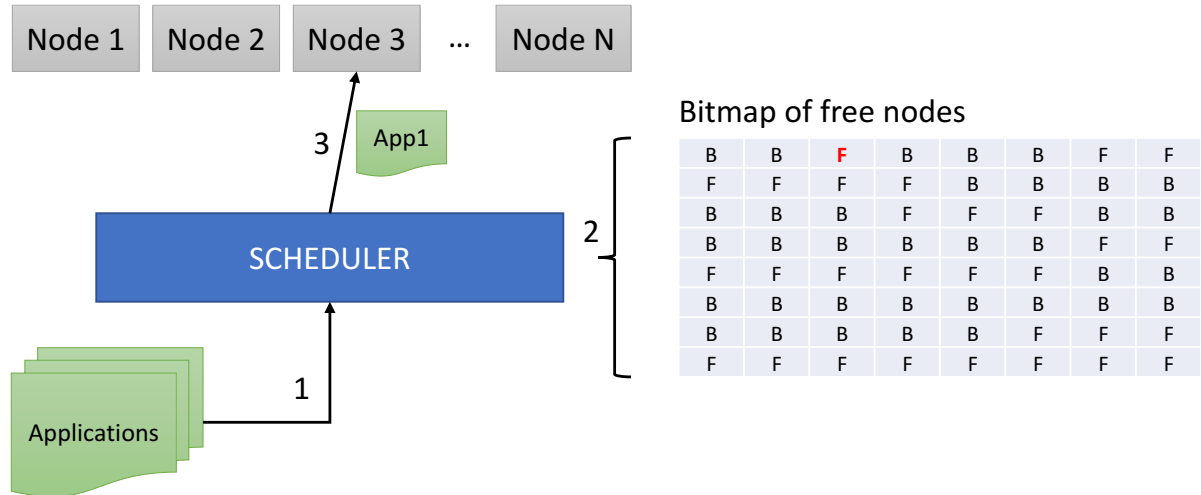


Figure 2.1: Scheduling in HPC - Exclusive scheduling policy.

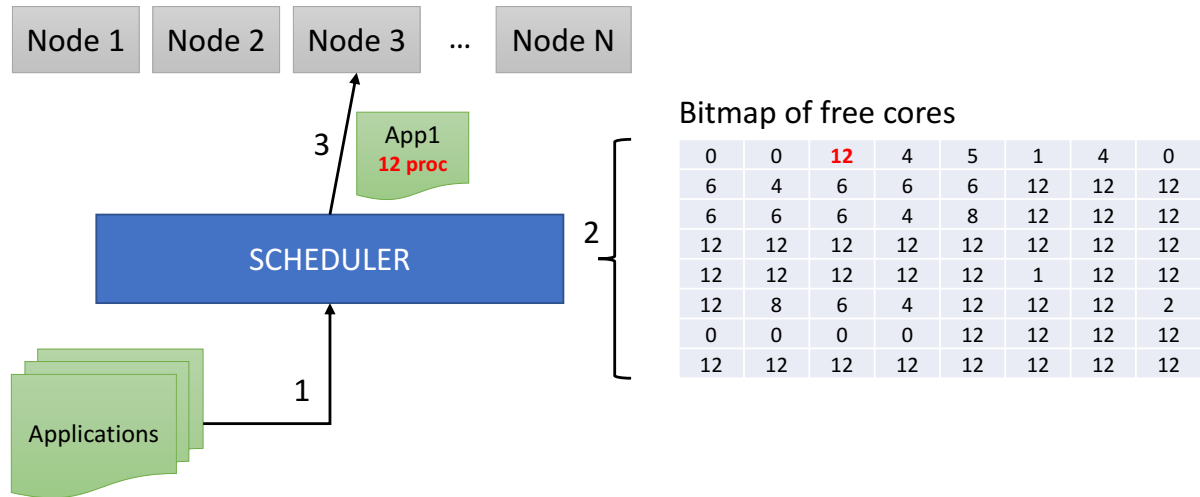


Figure 2.2: Scheduling in HPC - Shared scheduling policy when an application uses all cores from a node due to the availability of free computational resources.

scheduling policies designed in LIMITLESS differs because it performs real-time scheduling based on monitoring, and our framework can schedule both periodic and non-periodic tasks. However, it is possible that the solution won't be optimal because of the lack of complete application information.

In the same way, authors in [41] present a related work on energy-aware scheduling based on genetic algorithms. They describe this solution as *Hybrid Genetic Algorithm (HGA)* because they mix a classic genetic algorithm with a stochastic evolution algorithm to take into account the DVFS information in the scheduling process. Sometimes, a certain optimization problem is not polynomial or linear, which means that it cannot be exactly solved. For this kind of problem, approximation algorithms must be used. Using Machine



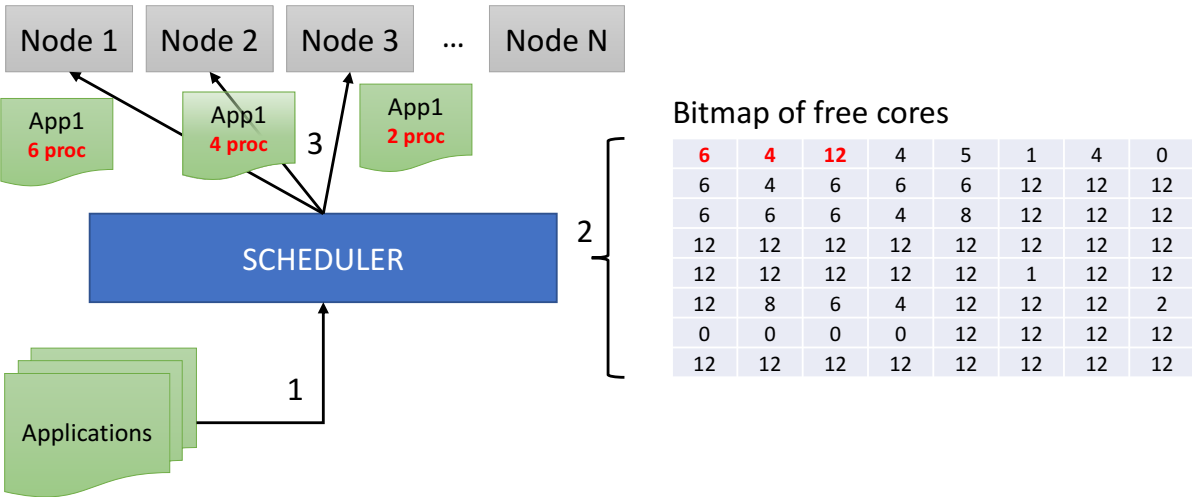


Figure 2.3: Scheduling in HPC - Shared scheduling policy when an application uses the unused cores from various nodes due to their free computational resources.

Learning and Artificial Intelligence algorithms is one of the main hot topics for solving optimization problems.

In [42] the authors exploit modern many-core processors with parallel computing techniques based on threads such as OpenMP and Pthreads. They describe a new methodology to reduce the penalty of the NUMA effect in specific HPC computational operations (linear algebra HPC operations), using dynamic load-balancing. This work has been considered relevant because this solution could be integrated with other monitoring and scheduling frameworks, providing workload balance between the different parallel applications. Integrating this related work could improve the scheduling process in standard application schedulers.

Following this researching line, a solution based on a distributed systems monitoring combined with prediction is proposed in [43]. The main interesting feature of this work is that this solution is able to determine what is the best set of compute-nodes to execute a certain application, taking this decision based on the information provided by a monitoring tool and the prediction performance algorithms. Hence, the scheduling is based on predicting the immediate future state of the cluster, and distribute the applications in the nodes that have the predicted better performance.

In [44] authors proposed a co-scheduling (which means that the solution will work in coordination with the original application scheduler) inside the Linux kernel for bulk synchronous parallel applications. However, this proposal is not based on monitoring computational resources but on integrating them. A similar work that includes monitoring information in co-scheduling is proposed in [45]. It uses the CPU and memory performance metrics to improve energy efficiency and the overall throughput of a supercomputer when



intensive applications are running in the same node. Another work in this field is [46], which uses monitoring information to provide profiling of HPC applications to take it into account for co-scheduling.

Sedighi et al. describe their proposal to improve the FUD process in [47]. Their solution is based on the assumption that a scheduler should optimize three different parameters: fairness, utilization and dynamicity (FUD). This work presents how to balance scheduling parameters in shared HPC platforms. However, there are other alternatives that propose hybrid scheduling algorithms based on monitoring heterogeneous and large-scale clusters [48], [49].

Other alternatives are Tetris [50] and LoTES [51], which consider the CPU, memory, network bandwidth and I/O performance to improve the cluster efficiency and reduce the execution time of the application stack: makespan. Tetris uses its own shortest-running-time-first algorithm to trade-off cluster efficiency for speeding up individual jobs, and the results are obtained in a 250-node cluster. The second solution uses combinatorial optimization techniques to get the optimal scheduling by matching the best dispatch of jobs between machines based on profiling, and then a linear programming model is used to maximize the system capacity.

Currently, *malleability* is one of the main topics in HPC and Cloud computing, but it is more interesting in virtualized environments (because moving virtual machines is more difficult than processes). A framework that provides elasticity (malleability) for existing MPI applications in these environments is proposed in [52]. This framework also includes performance monitoring for resources. In this case, MPI jobs can share compute-nodes, but If interference between the jobs is detected between jobs, one of them is terminated and a new program is restarted on a different number of instances. This work contributes including malleability in the scheduler, which allows it to dynamically manage the resources that the executing applications are using, redistributing those resources between the applications based on the scheduling policy decisions. This work is similar to one of the LIMITLESS features. In the case of LIMITLESS, it does not kill the jobs. The malleability implemented blocks the execution of a process and restarts it in a new job created in other node (migration). This process requires waiting for a certain time, depending on the cluster characteristics and the problem size. Note that there is a timesaving in LIMITLESS because the jobs are not killed but paused.

In [53] authors describe another proposal focused on clouds environments to provide elasticity for high-performance applications. Its differentiation factor consists of providing elasticity for high-performance applications avoiding the necessity of modifying the original source code. This is an interesting proposal because the authors combine it with

policies to increase the resources for the oldest running applications. The throughput increases reaching ranges of 26%.

Authors in [54] [55] explain how their solution, which is a control layer over MPI library, reduces the energy consumption by moving or re-sizing the applications. This solution implies that some syscalls used in the applications should be changed for new function calls. The provided API allows the applications to dynamically increase or reduce the number of processes. It means that one application is able to reduce its processes in one node and create more in other nodes. Moving applications consists of removing all the processes of an application in one node and creating new processes in another node. The relationship between the number of processes and the energy consumption is important because this work allows designing new scheduling policies based on application migration to reduce energy consumption.

Scheduling applications in sharing-node environments has an important problem: the interference between applications that are running in the same node. In [56] the authors propose a methodology to provide scheduling in HPC clusters that allow sharing nodes, based on detecting contention between jobs when they are executed at the same time on the same compute-node. The point is that they use a virtual HPC cluster, where jobs are executed using virtual machines that can be moved between different nodes. The problem is that the performance penalization is high, generating performance degradation in the jobs (in comparison with the time spent by the jobs without migration). Besides, the scheduler does not take into account the contention effects in the decision process (the interference-related time could be lower than the migration time), and the migration has to be done in certain execution phases (instead of at the moment when the interference is detected).

Another proposal related to the malleability feature is presented in [57], which includes a mechanism to increase and decrease the parallel processes in execution. To do it, the system migrates the task, performs dynamic load balancing, and uses shared memory and checkpoint-restart. However, the described scheduler doesn't include interference-aware features because the authors assume that each MPI process has dedicated resources.

The idea of combining the scheduler with the monitoring information in real-time has not been considered in many works, including if the inclusion of malleability and autonomous decision-making is taken into account. HPC monitoring and analytics for intelligence resource allocation is a very old field [58]. There is also studies about the dynamic reconfiguration to avoid contention. However, these solutions have not been extensively applied. It means that a new solution which includes these ideas can fill this gap by providing a full monitoring and scheduling framework that can, autonomously,

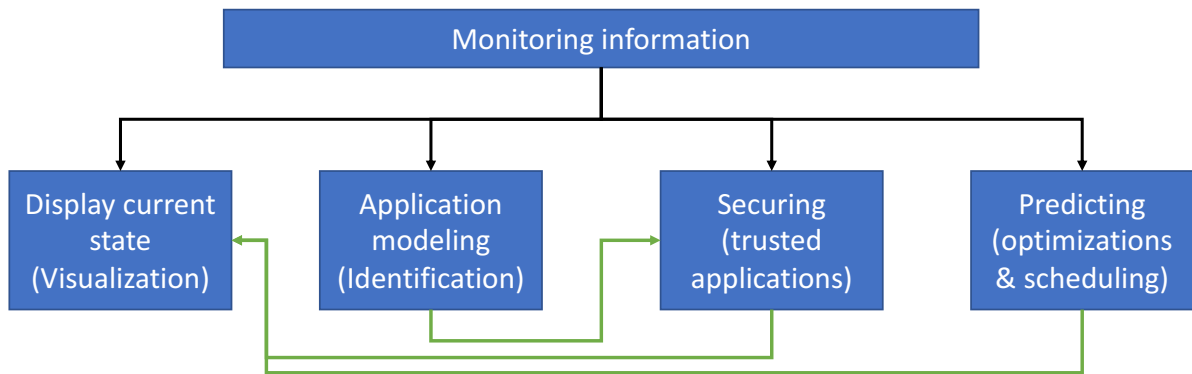


Figure 2.4: This taxonomy represents different actions to do with monitoring information: displaying the incoming metrics to view the current state of the system, generating application models for application profiling, creating a list of trusted applications and send notifications if new application is detected, and predicting future system and application states to improve the scheduling task.

manage the system with better throughput than static scheduling approaches.

### 2.3. Modeling, securing and predicting based on monitoring data

One way to increase the performance of the application (and maximize the system usage) consists of providing details about the execution of those applications. Figure 2.4 shows a brief taxonomy of different actions that can be performed with monitoring data. The first use is to visualize the information collected to provide a global view of the current (or historical) state of the system. The second application is to use the information to generate models of the applications that have been running, which involves the application identification and profiling. The application models can be used to recognize new executions of the same application (including when there are minimal variations), and to improve the application scheduling due to the possibility of executing at the same time, and in the same node, applications with different profiles (assuming that there will not be interference between them). The third use is related to security, which uses the list of applications identified by the scheduler to create a list of trusted applications. The idea is to send notifications to the administrators when unknown applications are running. This does not mean that the detected applications are malware, but it is another security layer. This security layer will be more accurate as new applications are added to the list of known applications). Finally, the monitoring information can be used, in combination with new algorithms and techniques, to provide predictions about the future states of the system, which can be used to optimize the communications, or to improve the scheduling process

algorithms. This is carried out taking into account, not only the current state and available resources, but the future state. Some related works about these topics are presented below.

Many applications have different phases during their executions. It means that each application shows a performance that depends on the time. Due to this, in [59], the authors have focused on how to provide dynamic application profiling based on the information collected by an application monitor. Typically, these kinds of monitoring tools exhibit the same hierarchical architecture and methodology to obtain the system performance metrics: collecting performance counters at a node level and, then, sending the data to centralize the whole information, either in a database, a file, or discarded it. Once the information is centralized, different algorithms are used to exploit the data to generate knowledge about the application behavior (such as profiles, models, predictions, etc.). Currently, there is another step in that methodology that consists of using the application profile to improve the application scheduling.

Another monitoring framework for applications (tasks and jobs) is presented by Rohl et al. in [60]. The name of this framework is LIKWID. Its architecture is similar to another monitor described in [61] because the system is hierarchically organized with the same objectives and with the same components: there is a monitoring tool (Diamond), InfluxDB as database (which is non-SQL), and Grafana, which is in charge of visualize the performance data. However, the difference is that this solution is oriented to small and medium-sized clusters because authors don't provide tests in large-scale systems.

Another example of application scheduling based on system monitoring is [27], where the authors focus on the post-processing of the Slurm log file, with the objective of extracting certain information and generating reports for the users. From the user's point of view, this solution provides important information about their apps. However, the scheduling process is based on the resources allocated for each application, instead of the real application performance. In contrast, LIMITLESS includes monitoring in two levels and provides dynamic scheduling (instead of post-process the data to take future decisions).

In the same area but including new algorithms, there are related works focused on combining monitoring with machine-learning techniques for different purposes. The following examples describe the different related works, starting with Yu et al. [62], which presents an article that explains how to combine network monitoring with prediction algorithms to design cross-layer security algorithms for intrusion detection. The proposal of Rashti et al. [63] tries to reduce the energy consumption in a certain network (based on wireless sensors) using prediction models. Their idea consists of reducing the power consumption of the sensors that will be inactive, and those sensors are the results of a

prediction algorithm trained with the historical data. In the next related work [64], Tang et al. propose the utilization of the Support-Vector-Machine (SVM) algorithm to predict the future state of the bridges that are monitored. Xiaobing et al. [65] provide an evaluation that shows the benefits of real data exploitation. They made a study of a gas tunnel, and the predictions prevent gas leaks. It is possible due to predicting based on the monitoring data and using machine learning algorithms. This work also includes a detailed methodology description and an exhaustive evaluation of the prediction accuracy. In the same way as the last work, [66] presents a combination of monitoring and predicting algorithms, but adapted to wired network transmission line sag. In this case, the predictions try to advance what will be the movement of those studied wired lines. Finally, in [67] the authors presented a method to apply the machine learning Nearest Neighbor algorithm [68] to make predictions using an unprocessed dataset so that the algorithm could be adapted to any kind of monitoring data, potentially providing good predictions.

In the follow we provide a overview of different prediction techniques based on machine learning. Note that the field of many of the works is not related to this PhD. thesis, but the ideas provided in these works can be extended to application and system modelling. In [69] the authors use Long Short-Term Memory (LSTM) networks to predict stock market prices. It is an artificial recurrent neural network for deep learning applications. With that algorithm, this work describes a prediction model and a series of experiments that obtain up to 55.9% of accuracy.

In [70] the authors point out the importance of glucose control for diabetes management. Recently, deep learning has been applied in healthcare and medical research, and this work presents a deep learning model that can forecasting glucose levels in simulated and real patient cases. This proposal is designed to run on Android devices. However, it is important to know that the authors have compared the performance of their application in a smartphone with a similar application in a laptop, reducing the execution time for similar results in a 100 factor.

Another related work in medical research is [71], that uses neural networks to predict the tumor category. In this case, an Artificial Neural Network (ANN) model has been developed based on the Multilayer Perceptron Topology and uses the medical data to classify the tumor. In this case, the ANN has been trained using a real dataset from the Institute of Oncology of Ljubljana, Yugoslavia, and the model is able to predict the tumor category with 76.67% of accuracy.

The authors in [72] present a novel method for predicting the evolution of the students in online courses. The idea is to collect all possible data from the user when it is studying (student-lecture, video-watching, clickstreams, etc.) to provide that to a time series neural

network. The objective is to predict how is the evolution for each student to adapt the teaching-models as early as possible. With this proposal, the evaluation on two datasets shows positive results, detecting better the course baseline (compared to the calculation on the past years) by more than 60% in the first dataset, and more than 15% in the other.

In [73] authors state that various computational models have been proposed to predict protein-protein interactions (PPIs) automatically. However, they claim that the problem is still far from being solved. Lab experiments are expensive and are limited by certain experimental protocols. It means that simulations or predictions are needed, and that is why the authors proposed a new model based on neural networks (NN) called *Ensemble Deep Neural Networks (EnsDNN)*. This model predicts PPIs based on different representations of amino acid sequences. The evaluation of the model has been done with six PPIs, achieving an accuracy of 95.29%, sensitivity of 95.12%, and precision of 95.45% on predicting the interactions. In this case they have to use a NN ensemble because each NN recognizes one type of interaction. For this reason, to predict if the input data is correct or not, a combination of NN is needed.

Big Data techniques and algorithms have reached a big important in biomedical and healthcare communities. However, the analysis accuracy is reduced if the medial data is incomplete. In [74] the authors propose new machine learning algorithms for predicting chronic diseases. The novel of this proposal is that they use the medical data in combination with information about the historical medical characteristics of certain regions to improve the accuracy. The experimentation has been done over real-life data collected from China in 2013-2015. Moreover, to deal with the incomplete data, the authors use a latent factor model to reconstruct that missing information. The solution is based on a Convolutional Neural Network (CNN), and compared with other prediction algorithms, the accuracy is around 95%.

Prediction and forecasting can be used in many fields. The study in [75] describes the utilization of some machine learning algorithms to predict the drilling rate of penetration (ROP). The behavior of this variable is unique to specific geological conditions, so that, its application is not simple. Besides, other factors can produce variations in ROP (for example, the rotation rate, the bit type, etc.). Those characteristics transform the single variable prediction into a multi-variable problem. The authors evaluate different algorithms and provide results for all of them: “artificial neural networks (ANN) based on multi-layer perceptron (MLP), ANN with a radial basis function (RBF), support vector regression (SVR), and hybrid MLP optimized with a particle swarm algorithm (MLP-PSO)”. All the algorithms predicted ROP accurately, being MLP-PSO the best.

Many of current related works perform predictions over one variable. In a different way,

[76] proposes a multi-variable gray predictor model (GM). This multi-variable grey model, based on the dynamic background algorithm, improves the forecasting performance on a certain number of sequences. The novelty here is the defined matrix of the multi-variable grey model, that can predict well the minimum and maximum values of the future interval. This work is important because other single-variable algorithms could be adapted to take into account multiple variables following the steps described in the paper.

Other multi-variable machine learning algorithm has been described in [77], and the objective is to improve the energy efficiency of the GPU applications based on monitoring information. The algorithm manages the GPU DVFS and the number of active slices. Then, its multi-rate predictive control, based on its predictions, is able to reduce the energy consumption in GPU (up to 25%) in the experiments.

In the same way, [78] introduces a novel method of using Hill Climbing algorithm (HC) for optimizing a multi-variable problem and a very large search space. The problem of HC algorithm is that it cannot be applied to tune the objective-variable in the use case because it has three parameters to be tuned. Hence, the HC algorithm has been modified to include two dynamic variables to make the prediction search faster. This work is interesting because the collected information is very large, and deal with it is a challenge.

In [79] the authors present a new machine learning algorithm that combines Random Forest algorithm with a modified AdaBoost algorithm to classify tumors from the MRI Brain tissues. The medical images are the input for Random Forest and AdaBoost to improve the accuracy of the classification. Besides, ANBC algorithms have been used in [80] to train offline foot contact detection and perform analysis in real-time. The authors obtain an accuracy around 95%, which means that this algorithm works well in real-time environments.

Other studies work with Softmax algorithm and others, as [81] conclude that this algorithm works well with datasets that are linearly separable. In [70], [82]–[84] different studies based on machine learning algorithms like *BAG and Random Forest*, *Decision Tree*, *HMM* and *DTW* are shown. This work includes a description of the different algorithms, the results, and comparisons between other solutions using different datasets. Note that all of these related works train the algorithms offline and use meta-algorithms to create better datasets. Using this information, it is possible to adapt our algorithms to the datasets depending on the strengths of each one. [85] describes how to use the KNN classifier to detect potential COVID-19 infected patients. This algorithm is interesting because of its simplicity and the accuracy reached when working with datasets without noise.

For multi-dimensional datasets, [86] describes how to adapt GMM classifier to identify different features in the speech in different languages. This solution is an alternative of



NN used in this work. Finally, in [87] the authors describe an approach that integrates genetics algorithm (GA) with support vector machine (SVM) classifier and particle swarm algorithm for software fault prediction. The work's strengths are its accuracy and the possibility of working in real-time and large datasets.

Reaching good results in predictions depends on the used algorithms and the data to train them. In this field, a study and evaluation of different machine-learning algorithms have been done. Table 2.2 shows a summary of these features, including a brief description for each algorithm. However, not all have been included in Chapter 6 due to poor performance or incompatibility with the nature of data used in this paper.

Algorithm	Description
AdaBoost	<i>Adaptive Boosting</i> . It is a classification algorithm that combines multiple weak classifiers into a single “strong classifier”.
ANBC	<i>Adaptive Naive Bayes Classifier</i> . It is a naive but powerful classifier that works very well on both basic and more complex recognition problems
BAG	<i>Bootstrap Aggregating</i> . It is a meta-algorithm that, combined with a decision tree method, reduces the variance and tries to avoid overfitting during the training process.
DT	<i>Decision Tree</i> . It corresponds to simple classifiers that work well on even complex classification tasks. It cuts the space into rectangular regions, classifying a new value by finding which region it belongs to.
DTW	<i>Dynamic Time Warping</i> . It is a powerful classifier that is designed to work well for recognizing temporal gestures.
GMM	<i>Gaussian Mixture Model Classifier</i> . It is basic classification algorithm that can be used to classify an N-dimensional signal.
HMM	<i>Hidden Markov Models</i> . There are two versions: continuous and discrete. Both powerful classifiers that work well on temporal classification problems when the training dataset is large.
KNN	<i>K-Nearest Neighbour</i> . It is a simple classifier that works well on basic recognition problems; however, it can be slow for real-time prediction and is not robust to noisy data.
Random Forest	It is an ensemble learning method that operate by building decision trees during training and returning the class with the majority vote over all the trees in the ensemble.
Softmax	It is a simple but effective classifier based on logistic regression. It works well on problems that are linearly separable
SVM	<i>Support Vector Machine</i> . It works well on many classification problems, even problems in high dimensions and that are not linearly separable.

Table 2.2: Machine Learning algorithms studied and tested.



## 2.4. Summary

This chapter describes different approaches related to this PhD. thesis, providing a complete study of the state of the art, and focusing on node-level and application-level monitoring, scheduling techniques and application malleability to migrate processes and deal with the performance interferences between applications. Also, it includes advanced techniques (such as Neural Networks and Machine Learning algorithms) to make predictions about the future state of the cluster and improve, as well, the scheduling based on those predictions. Hence, with these proposals, the goal of this PhD. thesis is to provide a framework with better performance than many previous approaches, without the need for virtualization, collecting data from the system and the applications, scheduling the tasks based on new multi-criteria policies and the exploitation of the monitoring data.

The next chapter describes a global view of the framework architecture, and shows a description and of each component and the interactions between them.

### 3. SYSTEM ARCHITECTURE AND DESIGN

This chapter presents an overview of the developed monitoring framework and shows a description of each component.

LIMITLESS is a light-weight and scalable framework that is designed to run on distributed systems. However, and taking into account that its purpose is to monitor LSDS, it can be executed to monitor a single node with a little overhead. The framework provides information about the platform hardware and resources, the current state of each compute node and the applications that are in execution. It is also dynamically reconfigurable, and are in communication with the scheduler to improve its decisions. Figure 3.1 shows the architectural representation of how LIMITLESS integrates the whole components. As it can be seen, the monitoring tool in LIMITLESS includes four main components: (1) the *System monitor*, (2) the database, (3) a visualizer, (4) and an *Analytic component* to exploit the data. The first component is in charge of collecting the performance metrics from each machine. The second one corresponds to *ElasticSearch* [88] database, which provides persistent storage. It is a NoSQL database that is fast storing and processing big amounts of information. Then, the selected application to display the information is *Kibana*, which can be connected to *ElasticSearch* easily. Finally, the analytic component processes the whole collected data, extracting the most important information.

It is important to note that the information flow follows two global directions: from the system monitor to the data-analytic component and the visualizer (*Kibana* in this case) and from the Analytic component to the system monitor. The first direction corresponds to the collected monitoring data, whilst the second one corresponds to the application models, created by the analytic component, and sent to the monitor for performing in-transit processing of the monitoring data. In the following we provide a more detailed description of each component.

#### 3.1. LIMITLESS System monitor

The system monitor is designed to collect performance information from the compute nodes in large-scale distributed systems. This information is used in two ways: (1) is provided for general purposes to the users, and (2) is used to improve other monitoring and scheduling tasks. In comparison with other monitoring tools, one interesting feature is the possibility of changing the configuration parameters dynamically. One of them, for example, is the monitoring period, which can be modified whenever the user wants,

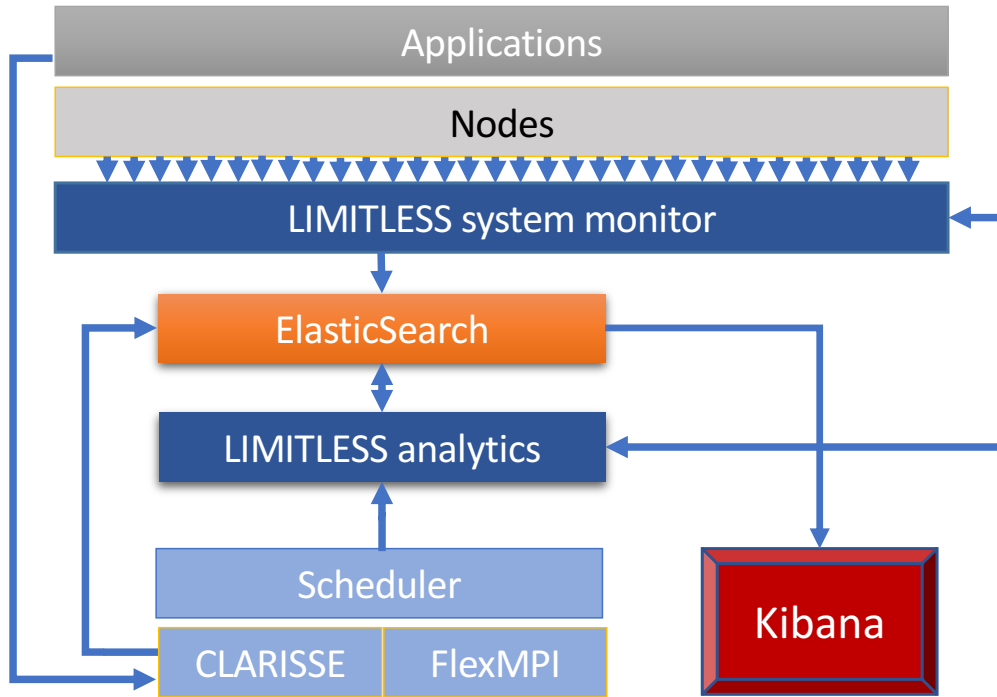


Figure 3.1: General overview of the LIMITLESS architecture and interrelation with other components (Scheduler, CLARISSE and FlexMPI).

including having different periods in different nodes. The range of this monitoring period starts in milliseconds and can be augmented to seconds, minutes, and hours. The most important thing about this feature is that there is no necessity to stop and start anything. Each component can detect and upgrade its state on the fly. Besides, LIMITLESS benefits from the interoperability of other components to improve the detail of the monitoring process and to reduce the communication overhead.

One of the main restrictions in this kind of framework is scalability. For that reason, the system monitor provides scalability in two different ways, trying to reduce the global overhead of its tasks. The first one corresponds to the distribution of the monitor logic in a topological hierarchy. The objective is to deploy and connect the components through a graph-based scheme, which should be the same as the real network topology. The second one consists of implementing optimizations to reduce the network monitoring overhead, which is the main limiting factor for scalability in large-scale platforms.

A general example of deployment consists of executing the three monitoring processes that collect the information: *LIMITLESS Daemon Monitor* (LDM), *LIMITLESS DaeMon Aggregator* (LDA) and *LIMITLESS DaeMon Server* (LDS). One instance of LDM is executed in all nodes in the cluster. LDM is in charge of collecting the performance

counters related to the system and the applications periodically. Then, a set of LDA instances should be executed in determining nodes (depending on the user) because these processes are in charge of retransmitting the information from the LDMs to the LDSs. LDA is also responsible for executing analysis and an important optimization, which will be described in Chapter 4. Finally, one instance of LDS should be executed in a node (can be more than one; it depends on if the user wants replication) because (1) it receives and stores all the information collected in the Elasticsearch database, and (2) sends notifications to the administrator if something wrong is detected.

Figure 3.2 shows a generic deployment of LIMITLESS. The red lines indicate the fault tolerance mechanisms, being the solid lines the watchdog processes (WD), and the dotted lines Triple Modular Redundancy (TMR). It means that this example includes replication at aggregation and server layers. The objective of the redundancy is to increase the monitor scalability and resilience. As it will be explained later, the component replication provides fault tolerance by creating multiple connections between the different levels of the hierarchy. In addition, each LDA includes the feature of performing in-transit data-processing with the collected metrics in order to reduce the traffic with the LDSs.

Each LDMs is configured to collect periodically different performance-related information from the compute nodes. LIMITLESS is designed for both homogeneous and heterogeneous clusters (for example, of a system where only certain compute-nodes have GPUs).

Table 3.1 shows the different performance metrics that can be collected from compute-nodes, depending on if it has GPU or not. These parameters provide useful information to understand which is the current real state of a machine, and it includes CPU and memory consumption, global cache hits and misses, I/O and network bandwidth utilization, energy that has been consumed by the CPU during the sampling interval, the temperature reached by each socket during the sampling interval, and detailed information from the GPU. Finally, the monitor provides a list of running processes every sampling interval.

Table 3.2 describes the main performance events related to InfiniBand networks. There are more counters, but the indicated counters are the most representative. In a summary, the counters indicate the octets that have been transmitted, the number of packets transmitted, and the octets and packets received. Finally, a counter that indicates the delay between transmissions, which can give an idea of how saturated the connection channel is.

Table 3.3 shows the information obtained due to *IPMITOOL* [89], and it is information that cannot be collected directly from the SO partition. This data summarizes the power consumption, the temperature, and the speed of the different fans. The main important counter is power consumption, which can help to design a scheduling policy related to

energy and cooling.

Table 3.1: Parameters that can be collected in a monitored node.

Parameter	Description
<b>CPU</b>	Number of CPUs and cores, percentage of CPU in use.
<b>RAM Memory</b>	Total available RAM memory (in GB), percentage of RAM used.
<b>Cache memory</b>	System monitor can obtain different cache-related events, including the percentage cache hits and misses for each level, and the CPU stalled cycles.
<b>FLOPS</b>	Floating Point Operations Per Second.
<b>I/O</b>	Percentage of I/O traffic during the sampling interval, percentage of writes in I/O traffic.
<b>Network</b>	Host IP address, available network bandwidth, percentage of network bandwidth in use.
<b>Energy</b>	Joules consumed during the sampling interval.
<b>Temperature</b>	Temperature in °C reached during the sampling interval of each CPU.
<b>GPU(*)</b>	Percentage of memory in use, percentage of GPU in use, temperature, energy consumed during the sampling.
<b>Processes</b>	A list of running processes (with %CPU > 0.0).

Table 3.2: Parameters collected in InfiniBand networks.

Parameter	Description
<b>PortXmitData</b>	This counter indicates the total number of fabric packet flits transmitted.
<b>PortRcvData</b>	The total number of data octets, divided by 4, (counting in double words, 32 bits), received
<b>PortXmitPkts</b>	Total number of packets transmitted on all VLs from this port.
<b>PortRcvPkts</b>	Total number of packets received.
<b>PortXmitWait</b>	The number of ticks during which the port had data to transmit but no data was sent during the entire tick

Table 3.3: Parameters collected with IPMITOOL.

Parameter	Description
<b>Power consumption</b>	Machine power consumption during sampling interval.
<b>Temperature</b>	CPU temperature (for each socket), and more if other sensors are enabled.
<b>Fan speed</b>	For each fan connected to the motherboard.

The lower-level component of this system monitor is LDM, which is a process executed on each machine and is in charge of collecting the performance metrics, as it has been

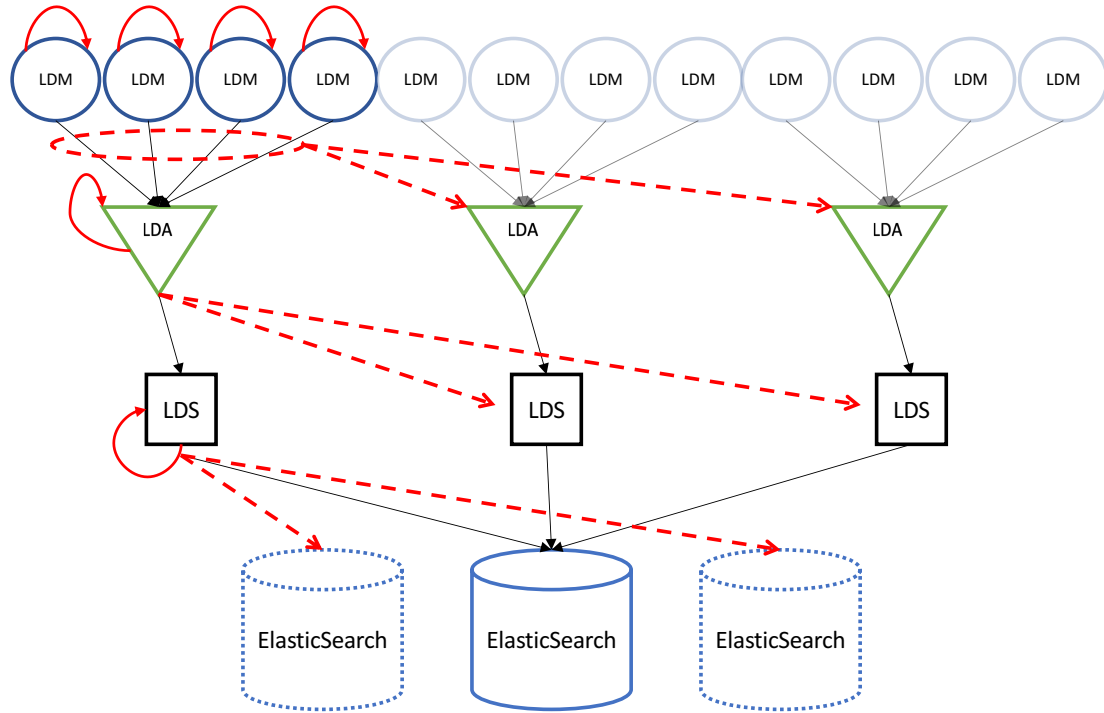


Figure 3.2: LIMITLESS - Deployment with fault tolerance mechanisms enabled in the first branch of the topology.

explained before. Each *time interval* (that is a configurable parameter) it gets and sends the collected metrics from the node to the next component, which can be a LDA or an LDS, depending on the deployment.

The next component in the communication process is LDA. These processes can be executed depending on the topology and the number of nodes, and their objectives are to provide scalability retransmitting the packets from the nodes below it. It is important to know how many LDMs can manage each LDA to configure a good deployment (this is analyzed in Chapter 6) avoiding running more processes than necessary due to bad topology designs. Instead of retransmitting one package per LDM, LDA is able to group packages to send the information in one. Besides, its functionality can be extended because LDA can execute partial analysis of pre-processing after the information reaches the next component (for example, for pre-process the received data instead of delegate this complex task to the LA).

Finally, the last component is LDS, which is the receiver of the whole metrics collected. Its main function is to receive, process and store the information. Once the information has been received and uncompressed, two actions are taken: (1) the LDS checks that there are not hotspots (if there is a hotspot, a notification will be sent), and (2) the information is



Figure 3.3: Relation between ES, LA and the application scheduler.

stored in ES for later analysis and visualization.

A detailed description of the functionality of each component can be found in Chapter 4, and it includes examples, algorithms with pseudo-code, and explanations of their designs.

### 3.2. Smart Analytic Component

The *LIMITLESS Analytic (LA)* component is a set of functionalities that gives the framework the necessary tools to analyze and extract interesting information from the LIMITLESS log, which stores all the packets received. These tools include different algorithms to provide predictions with the historical application profile data. Algorithms that are based on modeling, regression, machine learning, and neural networks.

Figure 3.6 shows how the analytic module is integrated into the framework. There are two alternatives, that are illustrated in the same figure: the first one (on the left side) is when the same node that executes the LDS, also executes the LA and the ES components; the second one (on the right) is when one node executes both LDS and LA, but the storage in ES is delegated to a dedicated node.

The reason is that LA is included in the same box as LDS because, typically, the LDS runs exclusively in a compute-node, and that node is underused because the main work of LDS is to receive, process and send the information collected, and It may not imply big computation and memory loads (it depends on the sampling interval and the number of incoming packets). So that, LA leverage that fact to take advantage of the unused resources.

In Figure 3.6, the left example is commonly used in small and medium clusters because it requires only one node to execute three components. However, in medium and large clusters, the nodes are usually separated depending on if they are for computation or storage. So that, the example on the right should be used to run ES in a storage node. The unique difference between both deployments is one parameter in the configuration file. This parameter specifies the IP of the ES node. In the first example, this value will be *localhost*, however in the second one will be the IP of the ES node.

### 3.3. Application scheduler

The *LIMITLES Analytic (LA)* component is in coordination with the application scheduler to provide resource allocation and dynamic scheduling policies. This relation is described in Figure 3.3.

The cooperation between these components has two main objectives: resource allocation and improving the application scheduling process. The first use happens when a user wants to execute an application and send the *application context* to the scheduler. After that, the scheduler sends the same information to LA, which will allocate the computational resources needed based on the current state of the cluster. Then, LA returns to the scheduler a list of compute nodes where to run the application, and the scheduler executes the application on those nodes.

The second use case is always running, but its effect is detected when the scheduler has a queue of applications ready to run (and other applications already running in the nodes). In this case, the scheduler and LA try to improve the process of executing the applications taking into account the available resources, the queue of applications, and the historical information about those applications (which is exploited using machine learning and neural network algorithms). This improvement is done in two ways: *coarse-grain* and *fine-grained* scheduling. Both alternatives are based on scheduling applications in shared nodes. The first one uses the information collected by the monitor to detect interferences between applications that are allocated in the same nodes. The term *interference* means that the applications which are sharing a node have a performance degradation, typically because both applications are using the same resources. So that, coarse-grain monitoring tries to share nodes between applications to use efficiently the computational resources, also detecting interference between applications to migrate one of them to another exclusive node (avoiding the interference and the performance degradation). On the other hand, *fine-grained* scheduling performs the same idea but using machine learning and neural network algorithms to detect applications phases, predict the future performance of the applications, and schedule taking into account the compatibility of those phases between  $n$  applications. It means that two applications with the same profile could share a node if their main phases (those which produce interference) are executed distanced in time (if phases with the same profile are not executed in the same moment, the interference will not occur).

All of these scheduling policies are described in Chapter 5, and evaluated with different use cases in Chapter 6.



### 3.4. InfiniBand support

One of the fundamental subsystems in HPC clusters and datacenters is the network, because the applications and services perform millions of communications between computing and storage nodes. Hence, high-speed and low-latency networks are required to manage those communications, otherwise the network could become a bottleneck.

There are a lot of network technologies, and HPC systems require the transmission of the highest possible amount of information in the shortest time interval. One of the most successful technology is the InfiniBand Architecture (IBA), which is commonly deployed in datacenters and supercomputers (for example, many of those included in Top500 list [90]).

Support for this technology has been included in the LIMITLESS monitor to collect performance metrics of these networks. The main objective of this integration is to collect the network performance metrics and study the congestion effect on the applications and take advantage of the communication with the scheduler to design new policies to mitigate the impact on the network executing applications in nodes without congestion. The collected performance counters are described in Table 3.4 and are based on the information obtained by the IBA subnet agents from their *Host Channel Adapters* (HCAs). The procedure for collecting all the performance counters consists of request all the information to the IBA Subnet Manager (SM), which is the process in charge of configuring the local subnet and ensuring its continued operation. It works in cooperation with the subnet agents, which are processes that handle the communications between HCAs and the SM. Due to this cooperation, LIMITLESS can ask for the performance counters to the SM to obtain all the information.

LIMITLESS obtains all the available IBA performance counters. However, the most important to have a general knowledge about the network state are the last five in Table 3.4: *PortXmitWait*, *PortRcvPkts*, *PortXmitPkts*, *PortRcvData* and *PortXmitData*. These five performance counters give the user information about the bandwidth usage and if there is any kind of congestion in the selected ports. The other performance counters give details about the behaviour of the network, links, errors, etc.

### 3.5. Cooperation with third-party components

The following components are integrated with LIMITLESS, but there are third-party components. For each one, the objective of each integration and how this framework uses it are described.

Table 3.4: Performance counters collected from IBA subnet manager.

<b>PortSelect</b>	<b>PortXmitConstraintErrors</b>
<b>CounterSelect</b>	<b>PortRcvConstraintErrors</b>
<b>SymbolErrorCounter</b>	<b>CounterSelect2</b>
<b>LinkErrorRecoveryCounter</b>	<b>Total number of packets received</b>
<b>LinkDownedCounter</b>	<b>LocalLinkIntegrityErrors</b>
<b>PortRcvErrors</b>	<b>ExcessiveBufferOverrunErrors</b>
<b>PortRcvRemotePhysicalErrors</b>	<b>QP1Dropped</b>
<b>PortRcvSwitchRelayErrors</b>	<b>VL15Dropped</b>
<b>PortXmitDiscards</b>	<b>PortXmitWait</b>
<b>PortXmitData</b>	<b>PortRcvData</b>
<b>PortXmitPkts</b>	<b>PortRcvPkts</b>

### 3.5.1. FlexMPI

FlexMPI is a runtime that is implemented to provide dynamic *malleability* for iterative MPI applications based on performance information. FlexMPI also includes a set of interfaces (API) to allow programmers to use it in other programs or applications. The source code of FlexMPI as well as many examples can be found in [91]. Moreover, this runtime was previously developed in [92] and is implemented as a library on top of the MPICH.

LIMITLESS uses FlexMPI in two ways: the first use case is for application migrations from nodes with poor performance to nodes where they can run better (thanks to the monitoring information), and using its performance counters to provide more data about the system and the applications that are running on it. That means the system monitor has two levels of data: information about the current performance of the real nodes, and information about the performance of the applications that are running in the cluster.

Figure 3.4 shows an example of an MPI application executor with FlexMPI runtime. The explanation of how it runs is based on intercepting selected MPI calls and inserting check points that performs the logic of the algorithms. When the application calls an MPI routine, it is wrapped by FlexMPI (Figure 3.4 arrow 1), the library executes some operations and then, the PMPI interface calls the original MPI routine (with a context that provides malleability). The code is executed in a transparent way: the operations that wrap the original routine create a context that includes the data structures and parameters to allow malleability instructions with the MPI original function. It is important to say that not all MPI calls are intercepted, for example the synchronization and datatype-management routines, that are directly executed by MPI.

The control point logic is in charge of the coordination of all the application processes

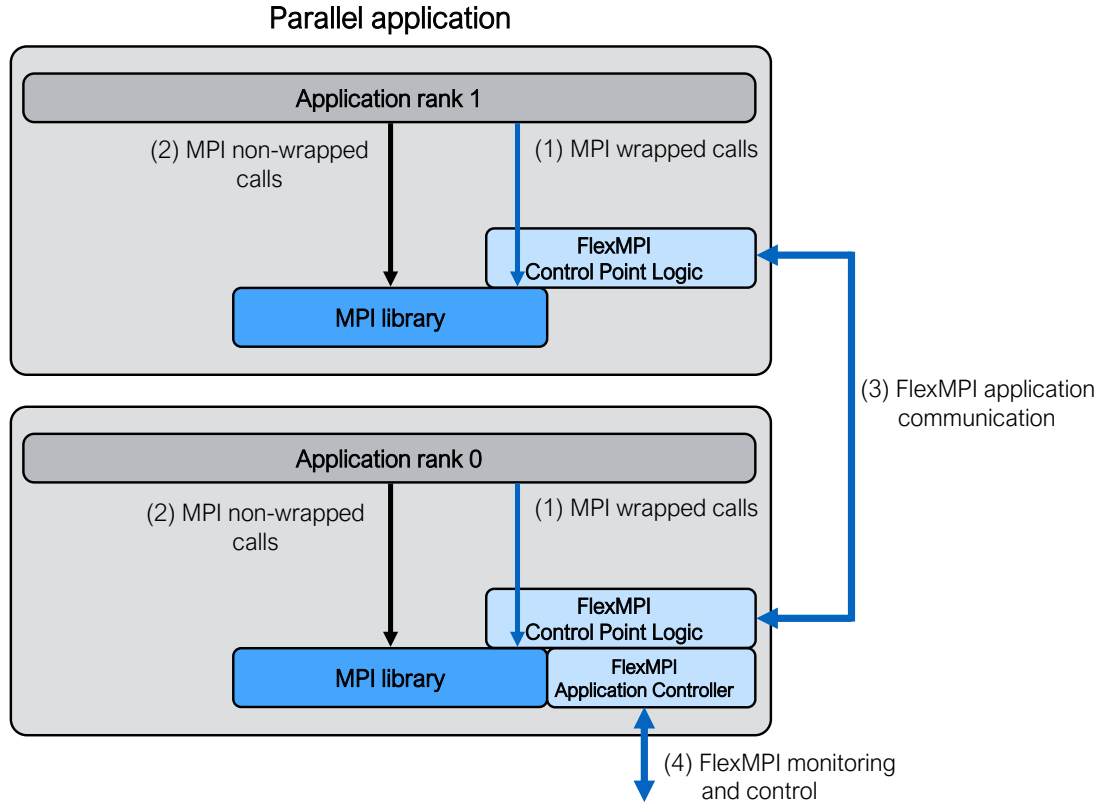


Figure 3.4: Explanation of how and MPI application is executed in FlexMPI environment.

(Figure 3.4 arrow 3). This logic performs two different actions: (1) performance data collection from all the applications and their processes and (2) sharing the control information received by the application controller. All this information is sent to the analytic component where it will be processed, in combination with system data, to improve the scheduling of the current and future running applications.

Table 3.5: Information provided by FlexMPI to the framework for each iteration and for each process.

Parameter	Description
<b>FLOPs</b>	Floating-point operations per second for each process
<b>MFLOPS</b>	Mega Floating point Operation per Second for each process.
<b>RTIME</b>	Time that each process spends in CPU.
<b>IOTIME</b>	Time that each process spends performing I/O operations.
<b>CTIME</b>	Time that each process spends executing communication operations.
<b>PTIME</b>	Is the sum of RTIME and CTIME at any given time.

Table 3.5 shows the parameters that LIMITLESS obtains from FlexMPI to improve the scheduling policies. This information is useful because it is directly related to the

performance of the applications, and knowing, for each application, their phases, profile and duration of each one, allows the creation of dynamic scheduling policies that combine phases instead of complete executions of the applications.

### 3.5.2. CLARISSE

CLARISSE is a runtime that provides interference-aware I/O scheduling [93]. The objective of this integration is to detect interference between the applications that are running in the same node in order to move one of them if I/O interference is detected. If one application (or more if there are more than 2 applications running) reduces its performance, the system monitor confirms that there is interference. At the beginning of the execution of a certain application, the runtime collects the performance metrics. So that, if during the execution the system detects lower values in the last samples collected, it means that the interference exists.

The first step consists of detecting the interference and taking a decision about how to avoid them is the second. There are different options depending on the type of the applications and the scheduler policy. For example, one solution consists of migrating one of the applications to another node. Another example consists of reducing the number of processes of one of them too. Both solutions are available if the applications use MPI. However, if there is interference between two applications and it is not possible dealing with their processes, one of them should be stopped and re-launched in another node.

The interference detection algorithm needs the related performance metrics for each running application ( $S_i^{init}$ ). If one application is new (there are no application metrics), it will be executed in an exclusive node to generate them without interference. The collected information contains data about the user, the system, how many communications the application performs, and other performance counters. To obtain that information, there are two ways: (1) getting the performance counters from previous executions, and (2) executing the application in an exclusive node, where there is no interference with other applications.

The general overview of how CLARISSE runs is explained in Chapter 4, however, how it is integrated with the other components can be seen in Figure 3.1.

### 3.5.3. ElasticSearch and Kibana

LIMITLESS uses ElasticSearch (ES) as a database, knowing that ES is powerful than this. It is an analytic engine that provides near real-time search for all types of data. It is able to

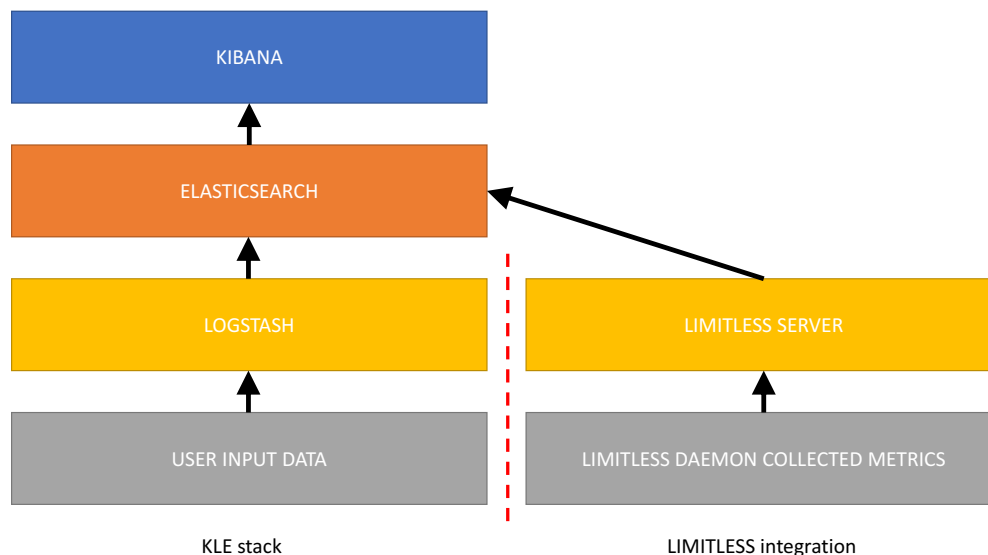


Figure 3.5: ELK stack vs *ES + K + LIMITLESS* architecture.

store and index efficiently structured or unstructured information, numerical or geospatial data, etc. By default, ES indexes the information in every field, and each indexed field has its own optimized data structure. Two examples of this are: text fields use inverted indexes. However, numeric data uses BKD trees (there are a type of trees that are commonly used for searching multidimensional data). That ability to use the specialized data structures to return search results is what makes ES so fast and efficient providing data access [88].

Besides, ES can be schema-less, which means that the information can be stored without indicating how to deal with each field of each document. That is why it is so easy to use, because the user sends the data without parse or organize them. LIMITLESS organizes the metrics that are going to be sent in a bulk operation as a time-series. It is not important for ES and its performance, but due to that, the information can be returned to the user through the connection API in JSON format.

In terms of scalability and resilience, ES has demonstrated that has great availability and it can scale well. It has good scalability because it was designed to be distributed, and the incoming packets are divided into small chunks that are inserted in different queues, each one managed by a different pool of threads. Besides, the users can include as many servers as they want (ES nodes), that implies that the whole data and queries will be distributed automatically, improving the response due to availability and replication. Also, it means that the resilience increases. However, instead of replicating all data, ES also uses the Cross-cluster replication (CCR) algorithm. CCR provides a way to synchronize the indices from the primary cluster to another remote cluster that can be used as a backup. It is important because high-availability architectures demand avoiding the idea of having

one single point of failure, and CCR achieves this goal [94].

Moreover, ES is the main engine for the ELK stack, which also includes Kibana. It is a visualization tool that is built on top of ES and takes advantage of the functionalities of ES [95]. Figure 3.5 shows how the ELK components interact to send the input data from the receivers to ES, and how Kibana is the last component in the stack, which is in charge of displaying the information in charts and tables.

In conclusion, ES has been selected because of its performance, scalability and integration with Kibana to display the data. However, as it will be seen, it is possible to apply other solutions, for example integrating a SQL or NoSQL databases with proprietary or non-proprietary visualizers. At the beginning of this work, the first solution applied involves InfluxDB [96] in coordination with fFLASHING. The objective was to show the global state of a cluster every second. The problem of this solution was the performance of the communication between InfluxDB and fFLASHING: this communication has more delay than *ES + Kibana*. In addition to that, to provide graphical representations of the collected metrics, a combination *InfluxDB + Grafana* has been used, as other related works like [97]. Grafana [98] is another visualizer, similar to Kibana but with fewer external plugins and the support of companies. In this case the performance is not a problem, including when the number of nodes to manage increases. However, over time, the performance is reduced due to the amount of data provided due to the processes that manage and display the historical data. But there are solutions to deal with this problem (for example auto-deletion of the  $n$ -oldest samples).

All of this means that, due to the provided API, there is not only one solution to store and plot the data. However, the experience shows us that ELK is a good framework to deal with time-series representations and to perform fast queries and analytics, and that is the main reason why it has been selected to be integrated with LIMITLESS.

### 3.6. Summary

This chapter describes the different components included in LIMITLESS framework. It includes a system monitor to collect the machine performance metrics, two runtimes that are coordinated with the system monitor to provide performance information at application-level, a persistent storage module based on Elasticsearch, an analytic module to provide features like modeling, prediction or knowledge data extraction, and a visualization dashboard based on Kibana to display, in a user-friendly format, the current state of the system and the historical performance data. It also includes an integration with a high-speed network technology (IBA) to collect performance information from these networks,

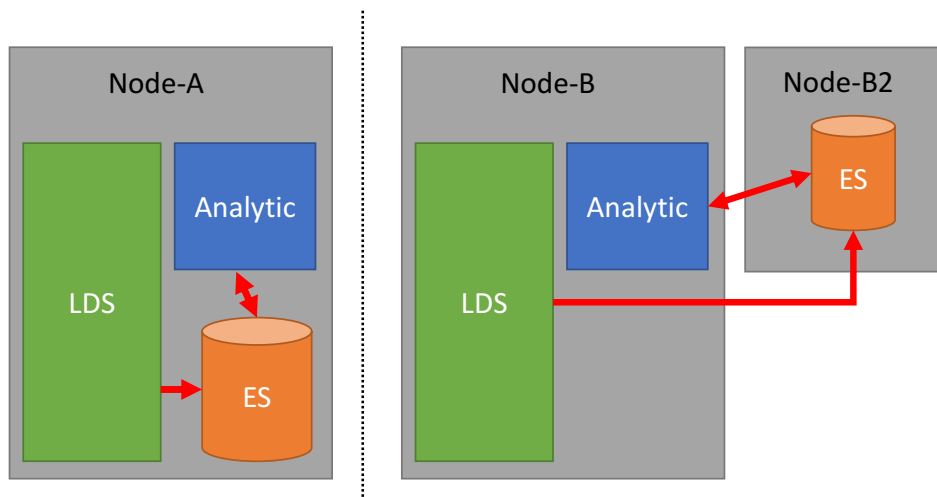


Figure 3.6: Analytic component - Integration examples.

which is widely used in datacenters and supercomputers.

The following chapter (Chapter 4) describes in deeper how each component has been implemented, including pseudo-code and algorithms to keep everything clear and understandable.

## 4. FRAMEWORK LOGIC AND ALGORITHMS

Chapter 3 shows the different components of LIMITLESS and what is the main function of each one. In this chapter the logic and the algorithms that are designed for the proposed framework are explained. To do so, the descriptions are going to be presented in combination with graphical examples and *pseudo-code* in order to improve the understandability.

This chapter is organized into six sections: the first one explains the monitoring function and how performance counters are collected, the second one describes different monitoring features, then, the smart analytic support for modeling and predicting are described, followed by the multi-criteria scheduling explanation. Then, a section shows the alternatives that allow the visualization and communication with the framework. Finally, the last section explains how this framework is used to improve the control congestion on InfiniBand networks.

### 4.1. Performance counter collection

One of the main components of LIMITLESS is the monitoring system, because it provides the information that is used to improve scheduling, detect hotspots, make models from the applications, etc. In order to obtain the state of the machines, the monitor processes the information that the operating system stores, and obtains the performance related to a certain application using PAPI [99][100], which is an API for accessing hardware performance counters available on most modern microprocessors [101]. Using this API facilitates the task of collecting different performance metrics for each application, and LIMITLESS leverages the FlexMPI component to have access to the application performance metrics using PAPI.

The Linux kernel has two main functionalities: manage the access to the physical devices, and establish how the processes interact with that devices. The filesystem called */proc* stores different information with details about the hardware and every process that are currently running in the system [102]. This special filesystem has the structure of a directory; thus, the files are *virtual files* with zero size and without type (binary, text, etc.). However, the files can be read and provide the related system information.

One common example of command that access to virtual files in *proc* are: *top* [103]. This command shows in the standard output (screen) real-time information of a running system, providing interesting data such as CPU usage, available memory, current running



```

Processes: 427 total, 2 running, 425 sleeping, 1450 threads
Load Avg: 1.21, 1.35, 1.34 CPU usage: 0.96% user, 2.25% sys, 96.78% idle
SharedLibs: 314M resident, 49M data, 25M linkedit.
MemRegions: 74360 total, 1872M resident, 67M private, 650M shared.
PhysMem: 7672M used (2055M wired), 518M unused.
VM: 3452G vsize, 2305M framework vsize, 71718850(0) swapins, 73962832(0) swapouts.
Networks: packets: 51172126/47G in, 97336327/101G out. Disks: 12440804/599G read, 4750830/580G written.
20:54:43

PID    COMMAND      %CPU    TIME      #TH      #WQ      #PORT    MEM      PURG      CMPRS      PGRP      PPID      STATE      BOOSTS
13108  top           13.3    00:02.56  1/1      0        28      8960K    0B        0B        13108    13102    running    *0[1]
134    WindowServer  3.7     02:08:07  15       6        2532+    813M-    9120K+    96M      134      1        sleeping    *0[1]
0      kernel_task   2.6     01:57:04  240/8    0        0        224M    0B        0B        0        0        running     0[0]
12901  com.apple.Ap  1.4     00:13.33  3        2        78       908K     0B        464K     12901    1        sleeping    0[1]
13100  Terminal      1.2     00:04.81  7        2        263-     82M      28M-     0B       13100    1        sleeping    *0[18]
13090  com.apple.We  0.4     00:06.23  5        1        109      103M-    6400K    16M      13090    1        sleeping    0[3686]
11803  gamecontroll  0.3     01:39.52  3        2        61       1572K    0B        340K     11803    1        sleeping    *0[193116+]
11747  com.apple.We  0.1     10:27.50  4        1        120      1101M    0B        577M-    11747    1        sleeping    0[180217]
13110  screencaptur 0.1     00:00.38  3        1        315+     3804K+   40K      0B       416      416      sleeping    *0[632+]
209    airportd      0.1     03:54.50  8        6        253      12M+     0B        7500K    209      1        sleeping    *501[68]

```

Figure 4.1: Top command - Example output.

processes, etc., as can be seen in Figure 4.1. All of them, given by the Linux kernel in */proc*, processed and displayed in a user-friendly format.

There are other examples of functions and system calls that can provide some performance data efficiently, for example *rstatd()* that obtains performance information from the kernel. However, the information that it provides is not enough to have a complete view of the machine, but it is interesting for some management tasks (for example, disable compute-nodes because of their high loads, power off machines which are up many days for their maintenance, etc.) [104]. The parameters that *rstatd()* return are described in Table 4.1. It shows general information about the CPU load, transfers per second for I/O and the number of network packets sent and received during the sampling interval. However, these values are related to computation, communication and storage resources, and only CPU load is in percentage (which is easy to understand).

In comparison with LIMITLESS monitor parameters, there are some of them that this process does not obtain, for example, temperatures, energy consumption, or the hardware configuration. On the contrary, *rstatd* include some other interesting features like *system load average*, which shows the number of active tasks. LIMITLESS has been designed for scalable and distributed architectures, and it includes the possibility of obtaining a list of running processes instead. however, its objective is not to provide information related to the operating system on the running processes.

The reason of using */proc* is that the monitor can obtain all available metrics that the operating system generates, and it can do it efficiently accessing only to the desired virtual files and getting the desired performance counters with low overheads (avoiding intermediate processes or system calls).

Finally, the framework includes another important feature: a heartbeat for every node.

Table 4.1: Information provided by *rstatd* process.

Parameter	Description
<b>CPU user time</b>	The percentage of the CPU time taken up executing user commands.
<b>CPU nice time</b>	The percentage of the CPU time for all processors globally taken up by low-priority processes.
<b>CPU system time</b>	The percentage of the CPU time for all processors globally taken up executing system tasks, including system calls to kernel routines.
<b>CPU idle time</b>	The percentage of the CPU time the processor was idle.
<b>Rate of IO transfers</b>	The number of disk transfers per second.
<b>Memory pages data</b>	The number of pages moved from the disk/cache/swap/storage to other destination.
<b>Interrupt counter</b>	The number of interruptions per second.
<b>Context switch count</b>	The number of processor context changes per second.
<b>System load avg</b>	The average number of active system tasks.
<b>Network packets</b>	Total number of packets sent/received and with errors.

If the server stops receiving data from a node for more than ten times the sampling interval, it assumes that this node is down, displaying this situation in the charts.

## 4.2. Monitoring system

This section provides a detailed description of the different components of the monitoring system. The main components have been already described in Chapters 3 and 4 (LDM, LDA, LDS and LA), but this chapter will give more information, including algorithms and implementation details about their integration.

Figure 4.2 shows the monitoring components, their connections and the information flow directions. Each node will run an instance of LDM (represented as blue circles), is configured to collect periodically different performance-related information from the compute nodes, and it is designed for both homogeneous and heterogeneous clusters (for example, of a system where only certain compute-nodes have GPUs). Each set of LDMs (depending on the topology designed by the administrator) sends the collected metrics to a LDA (represented as green triangles), which are in charge of retransmitting the information from the LDMs to LDSs. In this example, there is only one LDS, but it also depends on the topology design. The LDS will receive each monitoring packet from the nodes and will process and store them in ES. Hence, the information flow always follows the same directions, from LDMs to LDAs, from LDAs to LDAs or LDSs (depending on the topology), and from LDSs to ES database(s).

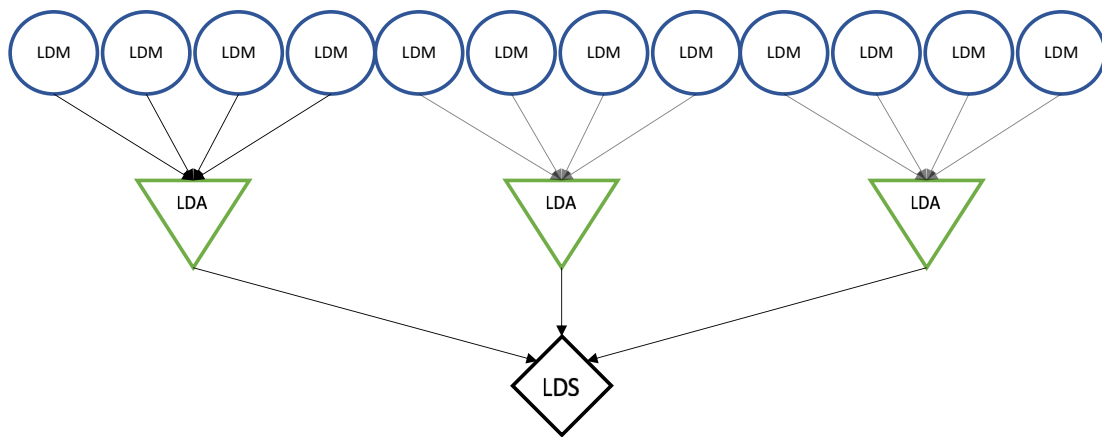


Figure 4.2: Systems monitor architecture.

The monitoring system can be configured to perform deployments based on different topologies. This allows designing different deployments, and the best option is to perform the same topology as the cluster's communication network. As an example, a deployment with one LDS, three LDAs, and four LDMs per LDA (12 monitored nodes) can be seen in Figure 4.2. This deployment can be designed manually, creating an *XML* file with the correct labels and defining the monitoring hierarchy branches (a branch is each set of nodes that send the collected metrics to an LDA (which is directly connected to an LDS)).

An example of how to design a topology can be seen in Listing 4.1, which shows a representation of the Figure 4.2. Taking into account the following labels, whatever topology can be built.

- *node-master*: Represents the LDS information (for instance, the ip and listening port).
- *node-group*: Indicates that the following labels represent a branch, with nodes (LDMs) and retransmitters (LDAs). If no retransmitters are indicated, the information will be sent to node-master.
- *node*: Represents each LDM.
- *retransmitter*: Represents an LDA that waits information from LDMs in the port defined, and will retransmit it to the node-master (LDS).

```

1  <topology>
    <node-master ip="compute-9-1" port="5000">
        <node-group>
            <retransmitter ip="compute-9-1" port="5004">
                <node ip="compute-9-1">
2          <node ip="compute-9-2">
            <node ip="compute-9-3">
            <node ip="compute-9-4">
        </node-group>
        <node-group>
11         <retransmitter ip="compute-11-1" port="5004">
            <node ip="compute-11-1">
            <node ip="compute-11-2">
            <node ip="compute-11-3">
            <node ip="compute-11-4">
16        </node-group>
        <node-group>
            <retransmitter ip="compute-1-1" port="5004">
                <node ip="compute-1-1">
                <node ip="compute-1-2">
21         <node ip="compute-1-3">
            <node ip="compute-1-4">
        </node-group>
    </topology>

```

Listing 4.1: XML file that defines the topology of Figure 4.2

In addition to the topological deployment, each component can be independently configured. As a practical example, let's assume a system with two different applications executed in two different sets of compute nodes. The first one is CPU-bounded and has a steady use of the resources. The second one alternates multiple phases of CPU, communication and occasionally I/O. In this case, the compute node that executes first application could be monitored with a coarse-grain sampling interval of several seconds, reducing the monitoring traffic without losing details about the application behaviour. In contrast, the sampling interval of the compute node related to the second application should be much smaller in order to capture the details of the different application phases. Note that these configuration parameters can also be adjusted by the monitor or provided as hints by other cluster components like FlexMPI (which performs application monitoring)

and the application scheduler.

Besides, the system monitor configuration can be dynamically configured without the need of interrupting the remaining monitor components. This is achieved due to each monitor component periodically checks whether the related configuration has been modified. If so, the component is reconfigured according to these changes that include the following features:

- Change the sample interval that the LDM collects the compute-node performance metrics.
- Change the referenced network address of another component that the current one is connected with.
- Change the connection ports of other applications or services.
- Change the fault-tolerance policy of a certain component.

LIMITLESS monitor allows to configure 21 parameters for the different modules on startup, which can be a tedious and confusing process for non-skilled users. In order to make the deployment task easier, the framework provides a script-file that performs the deployment of the platform, by reading the configuration data for deployment from a *XML* file that defines the topology deployment, sampling intervals, etc. Each component will have its own configuration file because the launching parameters differ depending on the component. If any files do not exist, default parameters are used.

The system monitor provides scalability in two ways to reduce the overhead and minimize the communication. The first one corresponds to the distribution of the monitor logic in a topological hierarchy to deploy and connect the components through a graph-based scheme. The second one consists of applying filtering techniques in the LDMs (in-node analysis) to reduce the network traffic if the metrics are within a range with the previous one. The objective is to reduce the communications if the performance metrics are within a range (defined by the user).

The distributed algorithm behind the *LIMITLESS Daemon Monitor* can be seen in Algorithm 1. This algorithm collects the performance counters in the nodes and sends them to the next component.  $sample_i$  represents the vector that stores the collected information (performance counters) at the  $i$ -th sampling interval. These performance counters include information like CPU usage, memory used, number of CPUs, cores, and energy consumption, among others (they are enumerated in Chapter 3. After each LDM collects the new metrics, an in-node analysis is performed to evaluate if the information

should be sent (if the metrics are enough different, given a tolerance  $tol_j$ , from the previous sample, denoted as  $ref\_metric$ ). If the algorithm determines that any metric is out of the given tolerance, the whole vector will be sent to an LDA. In another case, the system assumes that the current state is the same and the data won't be sent. Besides, once the information is sent, the LDM process sleeps until the next sampling interval. For every new event, that may represent a change in the system, *obtain\_status()* returns the current status of node  $i$ . The possible status values are “NONE”, for no interference, and the aforementioned “RAM”, “NET”, and “CACHE” hot spots. The hot spots are quantified as the percentage of use of RAM and network bandwidth and the last-level cache miss ratio. Note that when one of these values reaches a predefined threshold, the related interference status value is automatically generated and the aggregator notifies the scheduler about the interference type by means of *notify\_scheduler()* (arrow 3).

---

**Algorithm 1** LDM distributed algorithm.

---

```

1: while running do
2:   while  $dosample_i = collect\_node\_metrics()$ 
3:      $i++$ 
4:     for each  $metric_j \in sample_i$  do
5:       if  $metric_j \notin [ref\_metric_j - tol_j, ref\_metric_j + tol_j]$  then. // In-node analysis
6:          $send(sample_i)$ 
7:         break
8:       end if
9:     end for
10:     $update(ref\_metric)$ 
11:     $waitTillNewCollectionTime(frequency)$ 
12:  end while
13: end while

```

---

#### 4.2.1. Communication between components

The messages exchanged between the different components are made through *User Datagram Protocol (UDP)* sockets. Other alternatives have been considered, like *Transmission Control Protocol (TCP)* and *Message Queued Data Transfer (MQDT)*. However, UDP has been used due to different criteria: its packets are around 60% smaller than TCP packets, there is no connection to create and maintain, the user has more control of when a data is being sent out and, in the experiments, UDP is faster. This option also includes some disadvantages, like the possibility of increasing the packet loss rate, they can arrive out of

order and there is no congestion control. Taking this information into account, UDP sockets have been used but we have also included some features to mitigate these disadvantages.

The standard UDP sockets have been optimized, like MQDT does, with a message queue to store the incoming metrics. Then, concurrently, the processing threads get them and process the information stored. Besides, each packet includes the timestamp of when its metrics have been collected in order to allow the identification of the correct sequence of the metrics. This means that there are two features that minimize the disadvantages of UDP at the same time that keep its advantages.

#### **4.2.2. Monitoring policies**

After analyzing the state of the art, there are two kinds of monitoring tools, depending on the results offered. There are users who want to receive the information continuously (as soon as the metrics are collected). However, other users prefer to receive only certain information. That is why the monitoring tool presented in this PhD. thesis includes both alternatives.

In the continuous policy, each LDM collects and sends, every sampling interval, the complete set of metrics collected. This policy provides a clear and updated view of the entire system.

Otherwise, the event-based monitoring policy does not provide the collected metrics for every sampling interval. Instead of that, this policy sends those vectors of metrics that overcome the predefined conditions. This feature is designed to receive only relevant information or metrics that indicate a hot spot (which can be a potential failure in the system). Before the use of this policy, we introduce the concept of “event”. In this case, an event arises when one metric overcomes a predefined threshold (for example, when CPU values are greater than 80%). With this implementation, each LDM will send only those vectors of metrics that contain values greater than the threshold. This policy has two main advantages: (1) the communication overhead (network traffic) is reduced, which increases the scalability, and (2) the framework components have less load due to the lack of messages to process (the size of the database is also reduced). All of this allows the user to receive only notifications when the events occur. In addition to this, when an event is detected and notified, the visualizer shows that situation, and the LDS sends another notification to the scheduler to enhance the scheduling task [61].

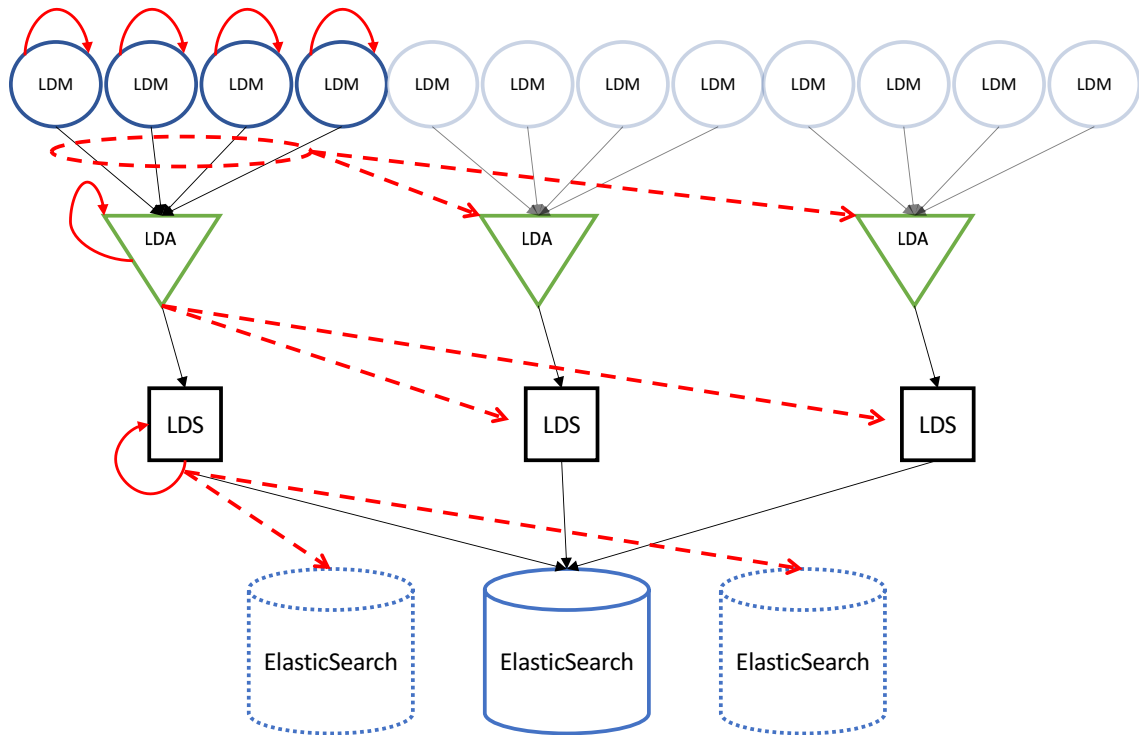


Figure 4.3: LIMITLESS - Deployment with fault tolerance mechanisms enabled in the first branch of the topology.

#### 4.2.3. Fault tolerance

In addition to the basic deployment method, LIMITLESS provides fault tolerance policies. These policies are focused in providing resilience with or without redundancy, and both policies can be enabled at the same time (they are complementary). The first policy is *Triple Modular Redundancy* (TMR) and the second is *WatchDog processes* (WD).

The first alternative, TMR, provides fault tolerance adding multiple communication channels between the components. It means that each component will be connected to another three components in the next level (LDMS to LDAs, LDAs to LDSs, and LDSs to ES).

This policy includes two different working modes:

- **Collective communication:** the performance metrics are sent to the connected components, providing data replication at expenses of a higher network traffic.
- **Peer-to-peer communication:** in this mode the metrics are only sent to one of the next-level connected component. The idea behind this, consist of dealing with failures without creating extra network traffic and keeping only one communication



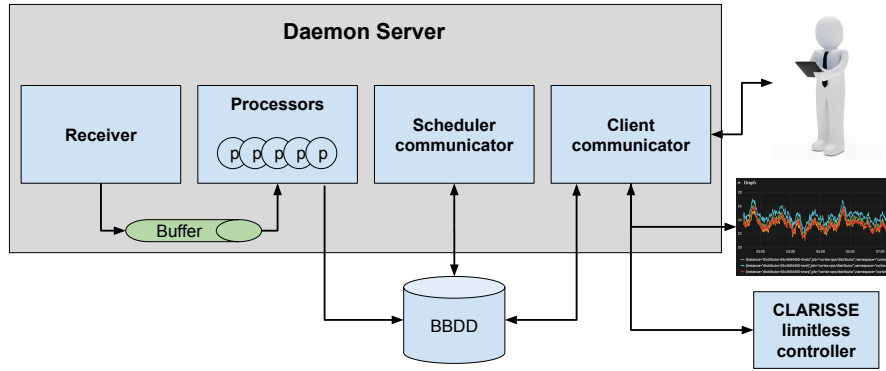


Figure 4.4: LDS architecture.

channel. In this mode, since each component only sends the information to one next-level component, in the case of failure, the data is then sent to the following component chosen according to a round-robin list.

The second alternative, WD, executes the different components as a service that is always running and checking if the component processes are also running. So that, it includes a control process that checks if the main process (LDM, LDA, LDS, etc.) is running properly or not. If there is a failure in any component, the WD process will attempt to execute again the component maintaining the configuration. If the process cannot be re-launched, the node will be flagged as non-operational.

Figure 4.3 shows the fault tolerance mechanisms in red colour over a generic deployment. The red lines indicate the fault tolerance mechanisms, being the solid lines the watchdog processes (WD), and the dotted lines Triple Modular Redundancy (TMR).

#### 4.2.4. Limitless Daemon Server

As it has been explained before, the LDS is the component in charge of receiving, processing and saving the monitoring packets from the LDMs (through the LDAs). These functions require computational resources, and that is why this component executes each function in different threads. As Figure 4.4 shows, a thread is in charge of receiving incoming packets (*receiver*). To extract the information collected, there is a pool of threads in charge of getting, disassembling, and processing it concurrently (module labeled *processors*). Another thread is responsible for communicating with the scheduler, using the monitoring information to improve its task. These threads are also in charge of evaluating the existence of hotspots and send the information to the ElasticSearch database. Finally, another thread is in charge of performing three different actions because there can be done using the same communication API: it listens to the client petitions, which can be requested by the user, the visualizer, or the CLARISSE and FlexMPI components.

**Algorithm 2** LDS - Communication algorithm between the scheduler and the LDS.

---

```

1: // Thread 1: Receiver
2: while running do
3:   rcvd_packet = socket_listener()
4:   store_packet_inBuffer(rcvd_packet)
5: end while
6: // Thread 2: Client communicator
7: while running do
8:   rcvd_query = socket_client_listener()
9:   query = create_multicriteria_query(rcvd_query)
10:  result = DB_exec_query(query)
11:  send_to_client(result)
12: end while
13: //Scheduler communicator
14: while running do
15:   rcvd_query = socket_scheduler_listener()
16:   [appi,  $\Delta p$ , excl] = exec_multicriteria_query(rcvd_query)
17:   if allocate(appi,  $\Delta p$ , excl) then
18:     if is_new_application(appi) then
19:       nodes = allocate_new( $\Delta p$ )
20:     else
21:       nodes = reallocate(appi,  $\Delta p$ , excl)
22:     end if
23:     return_scheduler(nodes)
24:   end if
25: end while
26: // Pool of threads: Processors
27: while running do
28:   lock_buffer_mutex() //Concurrent queue
29:   Frame = get_packet_fromBuffer()
30:   unlock_buffer_mutex()
31:   db_store(Frame)
32:   if evaluate_hotspots(Frame) then
33:     send_notification() //to the scheduler
34:   end if
35: end while

```

---

It is important to know that the *scheduler communicator* also acts as a resource manager. When the users want to execute their applications, this process allocates the required resources. To describe better this functionality, Algorithm 2 shows the pseudo-code, which performs the required operations to allocate the resources and execute the applications. The function named *allocate()* is in charge of allocating the resources requested by the scheduler. The resource allocation process needs three input parameters: (1) the application id, which typically corresponds to the name ( $app_i$ ), (2) the number of processes that the application will execute ( $\Delta p$ ), and (3) the flag to request exclusive or non-exclusive nodes ( $excl = 1$  or  $excl = 0$  respectively). By means of the id  $app_i$ , the component can distinguish between applications that should be executed, or re-executed, and applications that are currently running and should be migrated to another node using malleability. Recognizing the application is important the allocation policy depends on it. If the applications are new, the scheduler executes the function *allocate\_new()*, which returns the compute nodes allocated to run them. These allocated nodes can be exclusive or shared. In the case of applications that are already running and should be migrated, the scheduler executes the function *reallocate()*. It returns the new allocated nodes, taking into account the value of the third input parameter *excl*. The next step is to send the list of the selected nodes to the scheduler by means of executing the function *return\_scheduler()*. Note that  $\Delta p$  can be positive or negative, depending on if the application needs to increase or decrease its number of processes. If  $\Delta p$  is positive, new processes of the same application are going to be executed, and the allocation takes into account the current layout to use the same compute nodes as the original processes. If the allocation of the same nodes is not possible, the scheduler will use the topological information to allocate the nearby nodes.

### 4.3. Communication API and Visualization tools

This section describes the communication API included in LIMITLESS, which allows other processes to get information about the system performance and hardware, and three visualizers tested in real environments: one of them has been developed during this PhD. thesis and the others are third-party software.

LA includes an *Application Programming Interface* (API) with different functions that allow other components or processes to obtain the collected data, and it also allows LDSs to store the information in ES. The idea of this API is to close the framework communications to the outer components, maintaining only a trusted communication channel to put and get the data from ES.

The communication with this API is done using sockets, and the communication IP

and port are parameters set in the configuration files just before the monitor deployment. Using sockets, any process can request information to the framework, and the information that can be obtained includes: hardware information, historical data collected and last collected metrics. All of that for all or one machine in the cluster. Table 4.2 summarizes the functions related to the system management and Table 4.3 the functions related to the acquisition of the information.

Table 4.2: Management functions available in LIMITLESS API .

Name Function	Description
db_initialization()	Creates all the tables needed to manage all the information.
db_insert_conf()	Inserts the hardware information for a certain machine.
db_insert_cr()	Inserts performance data collected for a certain machine (general percentage of usage).
db_insert_coreload()	Stores the load for each core from each machine.
db_insert_IODev()	Inserts information about the IO devices (number and % of writes and usage) for all nodes.
db_insert_net()	Inserts information about the Network interfaces (number, % of usage and speed) for all machines.
db_insert_cores()	Inserts information about the number, energy consumed and temperature for each core for all the machines.
db_insert_gpu()	Inserts information about the GPUs (id, temperature, % memory usage, energy and % of computation used) for all the machines.
db_insert_summary()	Inserts a summary of general-purpose data for a global overview.

Once the information has been collected and stored, the framework displays it for the users. LIMITLESS includes a web application based on NodeJS that shows the state of the whole system with a colour scheme. The system is represented as a matrix, where each cell represents a node, and each one has a background colour depending on the load of the represented node (heatmap). This tool is called **FLASHING** (LArge Scale Heatmap vIsualizer whit Nodejs). The application includes three sheets to visualize the IP list, which show the IPs registered, a summary of the main performance metrics for all nodes, and the heatmaps. This last view includes another four sheets to display the heatmap depending on a certain performance metric: CPU, memory, GPU and IO (other metrics can be added, but this tool is a proof of concept). The colour scheme indicates a load of each node depending on the selected metric: green indicates values lower than 30%, yellow values between 36% and 75%, and higher values are painted in red.

Figures 4.5 and 4.6 shows the execution of **FLASHING**. The first one shows every performance counter for all compute nodes, and the second one shows the heatmap filtered by CPU load. Note that the cell colours between the first and the second image may

Table 4.3: Query functions available in LIMITLESS API .

Name Function	Description
db_query_table()	Returns the information stored in a certain table.
db_query_spec_cr()	Returns the information for all nodes filtering the columns indicated by parameters
db_query_spec_conf()	Returns the hardware data for all nodes filtering the columns indicated by parameters.
db_query_all_cr()	Returns the general information for all nodes.
db_query_all_conf()	Returns the hardware information for all nodes.
db_query_all_io()	Returns the information about the IO devices for all nodes.
db_query_all_cores()	Returns the information about the cores for all nodes.
db_query_all_gpu()	Returns the information about the GPUs for all nodes.
db_query_all_net()	Returns the information about the network interfaces for all nodes.
db_query_all()	Returns the whole information stored.
db_query_all_now()	Returns the last hardware data and general performance data received for all nodes.
db_query_all_summary()	Returns the average of the performance counters stored for all nodes.

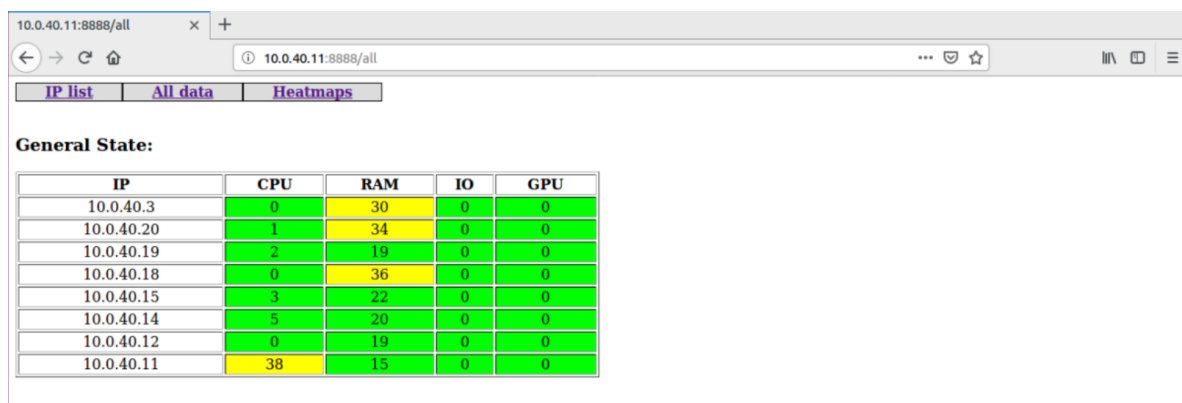


Figure 4.5: fLASHING - Visualizing general performance metrics for all nodes registered and monitored by LIMITLESS.

be different because of the performance fluctuation and because the images have been captured in different times. In this case, between both images, the main differences appear in the last two nodes, which shows CPU loads of 0% and 38% in Figure 4.5, but in Figure 4.6 are painted yellow and red. It means that the CPU loads are increased to more than 30% and 75%, respectively.

The following two methods are similar: both visualizers are based on web applications, include different dashboards to display the information, charts, gauges, tables, etc. They also include many plugins to display the metrics in a friendly way, to perform analysis

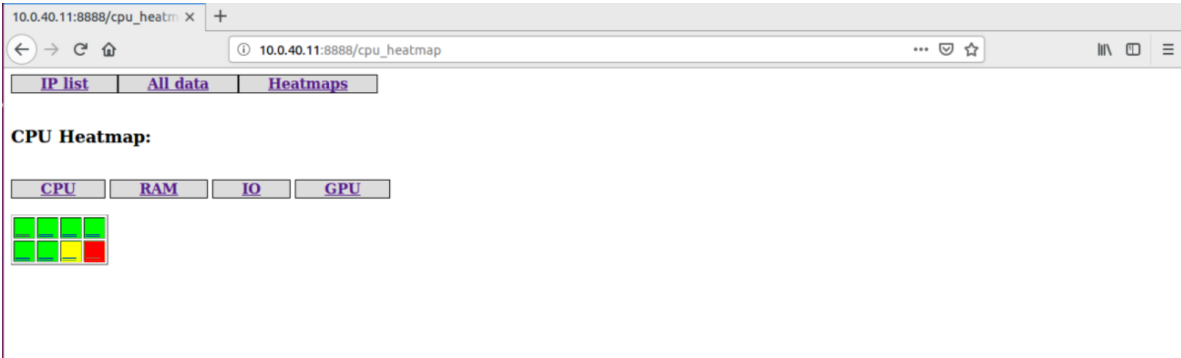


Figure 4.6: fLASHINg - Visualizing the CPU heatmap from the nodes registered and monitored by LIMITLESS.

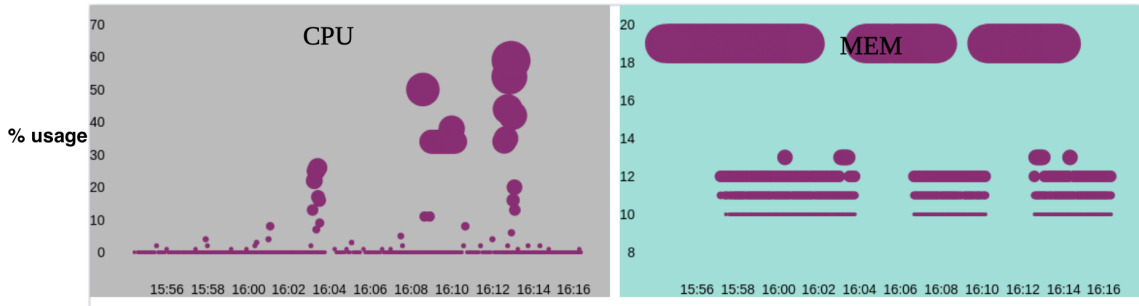


Figure 4.7: Kibana - Visualizing general state of the cluster. Each point represents an aggregation of nodes (bigger sizes represent more nodes, and smaller, less nodes).

operations to the data. Their use with large volumes of data has been proven. Besides, the visualizer utilization is basically the same: both visualizers can establish a connection with Elasticsearch as a data source, defining the tables to import to the web tool. Due to this, once the information has been imported, the user has to create a new dashboard (visualization page) and then he/she can insert new charts for the performance metrics.

Figures 4.7, 4.8 and 4.9 shows different views of Kibana dashboards. The first image represents one view to have a global vision taking into account CPU and main memory. The Y-axis represents the usage percentage for all charts, and the size of the points indicates the accumulation of nodes that belong to that percentage. The second image shows the general information for one certain node. In this case, there is a drop-down list to select the IP that corresponds to the node that the user wants to check (for instance, 10.0.40.19). Finally, the last figure shows a dashboard that provides some analysis of the data, indicating average, most repeated values, a counter of metrics, etc.



Figure 4.8: Kibana - Visualizing the general performance metrics for a certain node with IP 10.0.40.19 in a certain moment.

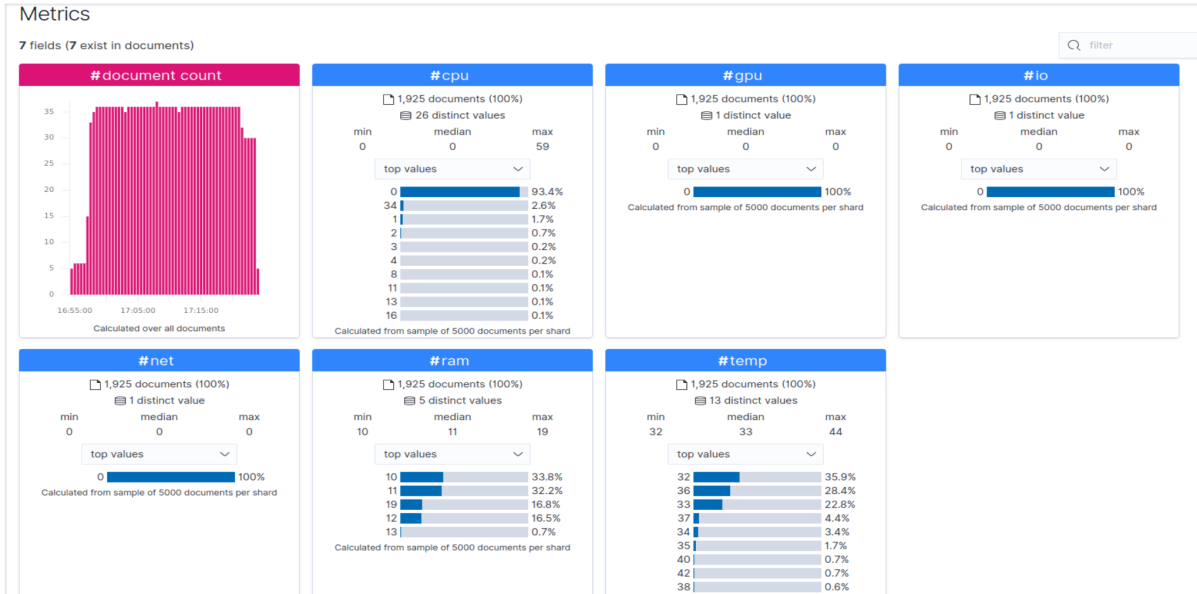


Figure 4.9: Kibana - Visualizing the Kibana dashboard which analyzes the data and provides functions to exploit the information (due to new plugins).

#### 4.4. Improving control congestion for InfiniBand networks

One of the major challenges in HPC and data-center infrastructures is how to design the interconnection network and how to deal with the network congestion. As it is an important topic, the different alternatives to create that network, for example, InfiniBand (IBA), include algorithms to mitigate its effects. However, detecting network congestion is a difficult task because of its variety and dynamism. If the detection is not fast enough, the system will react too late.

When there are some applications executing in a cluster, and they generate a lot of communications, network congestion may appear. In this case, this optimization focuses on the congestion over IBA networks. When an IBA-based network starts congesting, the origin of the problem is in the network paths, which are clogged channels where the applications are contending for them. These saturated paths (also called congested flows) can delay the communications at switches that there are not in the conflicting zone (these paths are *victim flows*). This problem is known as HoL blocking (Head-of-Line), and the percentage of performance degradation can reach important values. For this reason, one of the major challenges is the designing of new policies or algorithms that can detect and reduce network congestion.

In the specific case of IBA, the developers have spent a lot of resources to design their own congestion control (CC), which automatically minimizes the negative effect of the congestion over the performance. This CC is based on a strategy named *closed-loop*, which delegates the detection of the congestion to the switches in the network. The IBA switches can detect congestion in the output channels (ports), and the process of congestion control starts when it overcomes a predefined threshold. The process consists of communicating two switches through special flags to determine the congestion degree. Once the congestion is detected, the switches reduce the traffic through the congested channels, retransmitting them through other paths. Finally, when the congestion is reduced, the switches start to dispatch the packets normally.

The objective of LIMITLESS, in this case, is limited to collect, as fast as possible, the performance counters of the IBA network and send them to an optimized CC, using OpenSM and the transmission via *MANagement Datagrams* (MADs), in order to update the IBA CC parameters dynamically. OpenSM allows LIMITLESS to communicate with the IBA Subnet Manager, and with the correct messages, thresholds, flags, and parameters of the IBA devices can be modified. It means that we could manage the behavior of the CC to design new strategies to reduce the negative effects of the congestion in the performance. Hence, with the OpenSM and the monitoring information about the IBA network, this



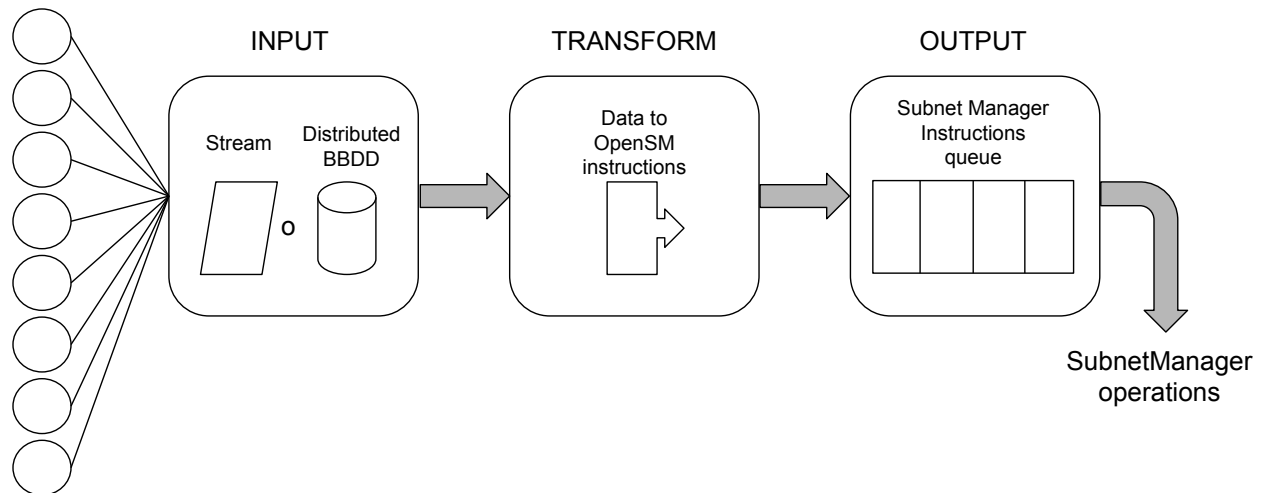


Figure 4.10: Representation of the pipeline from when the data is collected until instructions are sent to the congestion control.

feature tries to dynamically modify the configuration IBA CC parameters at the *Host Channel Adapters* (HCAs) generating traffic flows contributing to congestion.

Figure 4.10 shows the processes involved in this proposal to manage the congestion control of the network based on the monitoring data. Three main processes can be distinguished: the first one corresponds to the data collection from the compute-nodes (*INPUT*), which is served as a stream, or through a distributed database that communicates through the IB network. The second process (*TRANSFORM*) is in charge of receiving the input metrics and transform them to instructions in OpenSM, so that the Subnet Manager understands them. Finally, the last process in the pipeline executes the actions in OpenSM language to manage the congestion control based on them (*OUTPUT*). Note that this is a proof of concept, and the information collected is centralized before generating the Subnet Manager instructions. However, as future work, this process should be distributed in order to communicate, directly, the performance counter collection in the compute-nodes to the Subnet Agents (instead of sending the instructions to the Subnet Manager).

The following description explains how LIMITLESS contributes to increasing the accuracy and the reactivity of the CC. At first, LIMITLESS gets the *PortXmitWait* parameter, which provides a measure about how much time a certain HCA delays the traffic injection. If the delay value is greater than a threshold, it confirms that the communication channel is contributing to congestion. This situation is the cause that the HCAs do not receive the monitored packets in a short period of time, which implies an increment of the victim flows. Combining the monitoring information provided by LIMITLESS with the CC strategies, the system tries to update the OpenSM parameters to activate the injection throttling.

When an HCA truly contributes to congestion, a high value for *PortXmitWait* indicates that the congesting channels are waiting to be injected because the congestion tree has been propagated throughout the network. In this situation, it is needed to activate the injection throttling manual and dynamically, and one option to do it consists of updating the *CCTI\_increase* parameter in the HCA configuration. Based on the value of *PortXmitWait*, if it is over an *upper\_limit*, then the *CCTI\_increase* is enabled (in IBA-CC this parameter is disabled by default). On the contrary, reduced values of *PortXmitWait* confirm that the congestion is vanishing. If the congestion is reduced and assumable, the injection throttling is disabled and the *CCTI\_increase* value is initialized to zero.

The main results of applying this optimization (LIMITLESS + CC) are shown and described in Section 7.

#### 4.5. Summary

This chapter describes how the implementation of each component has been done. The first section shows how the performance metrics are collected, and why the */proc* processing is the best option. The second section describes the implementation of the monitoring system, explaining the communications, how to use it, the different optimizations developed, and all of that with pseudo-code and algorithms. The third section is dedicated to one of the main components in the framework: the analytic component. It is in charge of analyzing the whole data, modeling applications, predicting future states of the cluster and performing other operations to improve the framework performance (for example, reduce the monitoring network traffic, or improve the scheduling policies). The fourth section describes the concept of *multi-criteria scheduling* and the two different policies designed to schedule the applications based on shared nodes: coarse and fine-grained monitoring. Finally, the last section describes the communication API to allow other processes get information collected, and two alternatives to display the collected information, which have been tested in combination with LIMITLESS.

The following chapter (Chapter 5) shows a deep description about the Analytic component, which is in charge of generating models, identifying applications in execution, predicting the performance of the applications in the nodes, and performing two new scheduling policies based on monitoring.

## 5. ANALYTIC COMPONENT

Chapter 4 shows the implementation of the different components of LIMITLESS and what is the main function of each. This chapter focuses on the implementation details about the Analytic Component described in Chapter 3. To do so, the descriptions are going to be presented in combination with graphical examples and *pseudo-code* in order to improve the understandability.

This chapter is organized into two sections: the first one describes the different algorithms used to provide smart functions, and the second one presents two novel scheduling policies based on the algorithms presented in the first section.

### 5.1. Smart analytic support

In this subsection, there is a deep description of the different algorithms developed to provide smart functions to LIMITLESS. The main function of this component can be separated into three fields: application modeling, predicting algorithms, and classification/identification algorithms. The objective of using these algorithms is to exploit the data captured by the monitoring tool to (1) obtain profiles and predictors from the applications, and (2) to improve the scheduling process.

From the scheduling point of view, this component is used to leverage the performance information collected from the applications and the platform to predict future behaviours of the applications and, then, reduce the communication between the LDMs and the LDAs. A second goal consists of enhancing the application scheduling by means of identifying and predicting applications that be executed in the same compute node without degrading their performance. Once the applications have been identified, the LA performs predictions of the future performance metrics based on two kinds of algorithms: single-variable and multivariable analysis. The single-variable techniques are application pattern matching, prediction based on a historical window, and prediction based on neural networks. The multivariable analysis consists of using different machine learning algorithms: classification and searching.

Starting with single-variable prediction algorithms, the first method is based on pattern or model generation. When an application starts its execution, the framework stores its performance metrics as a pattern. In the following executions, the LA will try to predict the performance metrics using those patterns. 3 describes the logic of this solution, and there

---

**Algorithm 3** Application performance model logic based on pattern matching. Variables  $n_i$  and  $m_i$  corresponds, respectively, to the  $i^{th}$  measured and recorded performance metrics.

---

```

1: // Limitless Analytic
2: INPUT (from Elastic search):  $m_i$ 
3:  $\{n_i\} = ES\_read\_metrics(i)$ 
4: if  $\|n_i - m_i\| < threshold$  then
5:    $n_{i+1} = ES\_read\_metrics(i + 1)$ 
6:   return( $n_{i+1}$ )
7: else
8:   return("Prediction failed")
9: end if

```

---

can be seen that each sampling interval, the LA evaluates if the real metrics correspond to those stored in the pattern. If the metrics are similar (their difference is below a determined threshold), the following metrics will be generated from the pattern (assuming the same behaviour). If the predictions fail, the pattern is discarded. This alternative is lightweight, but it only works with applications that do not change their performance between different executions.

The second technique consists of providing predictions based on a regression of the last  $n$  samples received. This technique performs the interpolation of the  $n$  previous samples. The number of the sampling intervals to consider computing the interpolation is five (the sample window). The result of this operation is similar to the process of find tendencies, and its result is the predicted value. As the first technique, this alternative predicts well for periodic applications, which perform different phases (IO, communication, CPU, etc.). In this case, the duration of each phase should be similar during different executions to increase the predictor accuracy. The objective of using interpolation is to refine the predictions instead of using the stored value in the pattern. It allows certain tolerance to performance variability between executions.

The last single variable technique is based on the using of neural networks to learn, automatically, the performance patterns from the data collected from the previous executions, and then, to predict the following states when a known application starts. Each predictor includes a set of neural networks, one per variable that the user wants to predict. Each one is based on the Multi-Layer Perceptron (MLP) algorithm, and it is built with three layers fully connected with 120, 60, and one neuron respectively. Once one application has already been executed (an application ensemble is created), the training starts. After the training, the networks are capable of set their weights to recognize the patterns. However, the second execution of the application verifies the accuracy of the predictions (its results are used as a test). If the predictor is not accurate enough, more executions are needed to provide more training data. Note that one of the main advantages of this alternative is that there is no necessity to use historical data.

The last prediction technique is based on a multi-variable correlation using the machine learning algorithms. The first studied algorithm is *Nearest Neighbour* (NN) [105], [106], then the solution is extended with *AdaBoost* and *Support Vector Machine* algorithms due to their accuracy in the analysis, which will be described in Chapter 6, although other algorithms have been also tested.

In this approach, given a set of  $k$  performance metrics collected by a LDM in a current sample, our solution predicts the application performance and uses this information for two different actions: to reduce the network communication between the monitor components and to improve the application scheduling. The main idea behind the second goal is to obtain the application profile using the monitor and then, use this information for predicting the application performance when it is executed in the same node with other applications. Executing applications concurrently in the same nodes aims to optimize the resource utilization by leveraging the computational resources (cores) not used by one of the applications. It can result in less energy utilization (given that less compute nodes may be used). In case of large workloads, it contributes to reduce the overall makespan reduction and increase the platform throughput because the system has more compute nodes available to run extra applications. However, running multiple applications in the same compute node may produce performance degradation due to contention in the sharing of the compute node's resources between applications with different characteristics. We call this effect *application interference*. For this reason, this decision (sharing or not the compute node) has to be taken based on the application characteristics. In this work, we will leverage monitoring information to determine that application interference does not occur.

For the first goal, an algorithm capable of predicting the following performance metrics is enough, and we use NN. Having the set of  $k$  metrics, this algorithm finds the most similar  $k$ -metrics to this set in the complete historical data. It means that the algorithm has found the  $i$ -th vector of metrics in the set, being  $i$  the execution time, and then the predictor returns the vector of metrics that corresponds to the instant  $i + 1$ . One advantage of using this approach is that it can process really fast and efficiently a large set of data (like the historical log that stores the different application ensembles). The second advantage is that this method is multivariable, which means that there is only one ML algorithm per application (instead of one per metric, and per application).

For the second goal, it is not enough to predict the state of the cluster, but we also need information about the application profile that distinguishes its execution phases. That is, we need fine-grained monitoring. For this reason, we use support for machine learning classification algorithms. When the scheduler wants to execute applications in shared

nodes, sends the appropriate information to LA, which analyzes the scenario, predicts the future state of the applications, and identifies their phases to run both of them in the best moment (which is when the phases that cause interference do not overlap).

## 5.2. Multi-criteria scheduling

This section describes two novel techniques for application scheduling based on monitoring information. This process is done taking into account scheduling policies for shared nodes, where two, or more, applications can be executed in the same node at the same time (depending on the available resources). The first scheduling process executes applications in the same node when there is no interference between them (it means that there is no performance degradation when both applications are running concurrently). The second process executes applications but taking into account their internal phases. This method is based on decomposing applications in phases, get the profile of each one, and execute applications when their phases will not produce interference.

The policies of these processes are multi-criteria because, although they are based on detecting interference between the applications, the administrator can establish higher criteria that will always be taken into account. For example, the scheduler can perform the allocations taking into account the energy consumed, I/O or network usage, makespan, etc. It means that the scheduler will dispatch the applications based on those policies (a set of metrics to improve can be indicated), and then it will also try to share nodes to reduce the makespan, using the available resources efficiently. This can be done due to new scheduling policies that try to maximize all metrics, but prioritizing one to avoid conflicting criteria (for example, executing applications using 100% of CPU, and saving energy. Both metrics are in conflict, so that, one should be marked as principal).

Following, both methodologies to schedule the applications are described.

### 5.2.1. Scheduling based on monitoring: *Coarse-grain scheduling*

Figure 5.1 shows a general overview of the monitoring framework (without including the analytic, storage and visualization modules). The center of the figure shows the cluster's compute nodes, which are organized in two racks of three nodes each. The top of the figure shows the LIMITLESS monitoring tool, which is in charge of monitoring the applications and platform resources. The monitoring tool is deployed using one LDM per compute node to collect the performance metrics, one LDA per rack to gather the information from the LDMs (arrow 1) and then to send it to LDS (arrow 2). The LDS is the process in

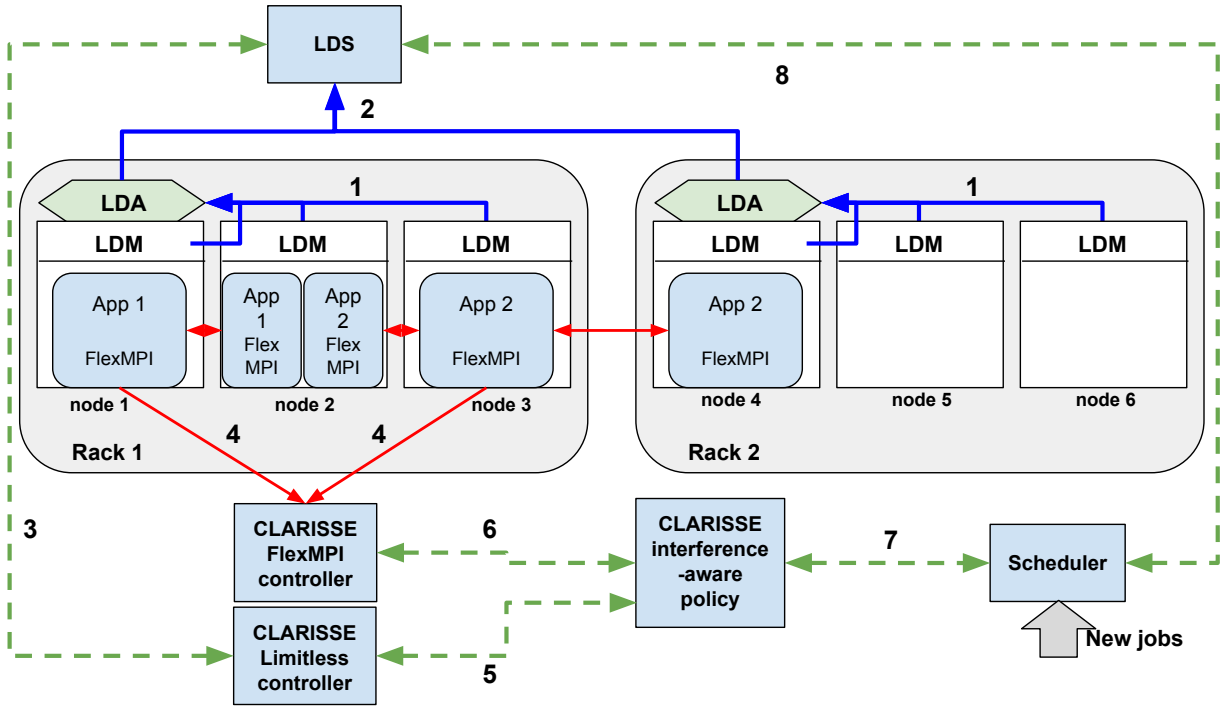


Figure 5.1: Systems monitor and scheduling architecture for coarse-grain scheduling.

charge of storing the whole received information in an Elasticsearch database. Besides, it processes the information received, analyzes the performance, and, in case of detecting a hot spot, it sends this information to the *CLARISSE* component (arrow 3).

Finally, the bottom of the figure represents the scheduling features, including FlexMPI and CLARISSE that are integrated components. FlexMPI is an implementation of OpenMPI that provides application-level monitoring and malleability of MPI applications, and it is connected with the CLARISSE's controller (arrow 4) to analyze the performance metrics. CLARISSE's interference-aware component leverages the monitoring information gathered from the monitoring system and FlexMPI (arrows 5 and 6) to check if there is a hot spot in any compute node that produces a performance degradation in the applications running. If degradation is detected, the policy manager is in charge of sending the instructions to the scheduler, with the objective of migrating the applications (arrow 7) and FlexMPI (arrow 6). Note that, in this scheduling policy, LIMITLESS performs two roles: (1) it monitors the system and (2) acts as a resource allocator providing the list of nodes where the applications have to be executed or migrated.

In every cluster, there are a certain number of queues where the applications wait until the scheduler dispatches them to hosts. These queues can have different priorities, different target hosts, different policies, or are designed for different users. Once a user wants to run an application, this is put into a ready-to-run queue, and the scheduler, when there are enough resources, executes it into a compute-node. This process is called *scheduling*.

There are different policies to provide this scheduling, but the most common consists of running applications in nodes when these are free without applications running on them (*exclusive-policy*). However, sometimes the scheduler can be configured to share nodes between applications, but not computational resources (for example CPUs or cores). It means that the throughput could be increased (depending on the applications) and the resources can be used in a more efficient way (*shared-policy*).

To explain better the meaning of the Figure 5.1, following there is a practical example description that uses the coarse-grain scheduling in shared nodes. Before starting, assume that the cluster nodes are homogeneous with 12 cores each, and there is an execution queue of two applications with 18 and 30 processes. The first application *App1* is executed in the first place, in nodes 1 and 2. Note that node 1 executes 12 processes (one per core) and node 2 the rest 6 processes. During a short time, FlexMPI monitors the performance of *App1* while it is executing without node sharing. Then, the second application *App2* is executed in the free nodes 3, 4, and 5, executing 12, 12, and 6 processes respectively. FlexMPI now monitors the performance of *App2*. Note that this monitoring process is done when the applications are running without sharing resources. For this reason, there is no risk of interference, and the performance metrics correspond to the real performance of each application. Once CLARISSE has received the performance of each application, the system tries to share nodes with unused resources. At this moment, the 6 processes allocated in node 5 from the *App2* are dynamically moved using malleability to compute 2, where there are already 6 processes executing from *App1*. The point of this policy consists of using the resources efficiently, trying to reduce the utilization of nodes: currently, four nodes are in use instead of five. It means energy-saving. Note that the stage of collecting the original performance metrics (without interference) can be avoided if the framework has already them (for instance, due to previous executions of the same applications). In this case, the framework could apply the sharing-node policy directly.

Let's now assume that *App1* and *App2* suffer performance degradation in node 2 due to cache-related interference. The monitoring tool detects this interference when the system performance metrics arrive at the LDS. Then, the LDS sends a notification with the risk to the CLARISSE controller. In order to evaluate the interference, CLARISSE activates the application-level monitoring in both applications, collecting the current performance metrics. The last step is to compare the current metrics with those without interference (initial performance in exclusive nodes). Once performance degradation is confirmed, the six processes of *App2* in node 2 are migrated to another free node (node 5 or 6, depending on the allocator process), where the application will recover its expected performance due to continuing its execution in an exclusive node. At this moment, node 2, as well as the node where *App2* has been moved, can be shared.



As it can be seen, the system monitoring tool is scalable because the LDAs manage many LDM instances, and is connected to LDSs in a hierarchical way. At the application level, there is one connection between FlexMPI and each application, and the rest of the connections are peer-to-peer basis between the components. And the application monitoring provided by FlexMPI is executed on demand when there is a possibility of interference between the applications that share a node.

### 5.2.2. Scheduling based on monitoring: *Fine-grained scheduling*

This scheduling process is similar to the one described above, but, instead of detecting interference between applications, this process tries to find which phases of the applications cause interferences. The main challenge of this process is to know which execution, from the application ensemble stored in ES (and managed by LA) corresponds to the current execution, with the objective of predicting and detecting its phases. For this reason LA uses machine learning algorithms to identify which execution from the application ensemble take as application model (based on ML Classification algorithms). Once the application and its phases have been identified, LA performs a series of analysis to determine if another application (or more if they fit in the non-interference phase) can be run together without interference, which would mean that LIMITLESS would be using the resources efficiently, reducing the makespan, and allowing new executions in the nodes that have released the applications in shared nodes.

The idea includes three steps: running concurrently two applications to know if there is application interference. Then, if it happens, dividing the application models (of both applications) into different phases that will be recognizable by the classification algorithms. Once these phases are identified, and the profile of each one (CPU-intensive, communications-intensive, etc.) is known, the phases of different applications that do not generate interference can be combined, allowing to run in the same compute node to maximize the machine resource utilization. Algorithm 4 shows the general algorithm pseudocode for this *fine-grained scheduling algorithm*, which consists of the study and schedule the phases of applications, instead of their complete execution. This technique allows LIMITLESS to combine long-time with short-time execution applications depending on the profile of their phases. The scheduler, for each application ready to be executed, sends a notification to the LAN component (Line 1) and obtains the related application model (Line 2). Then, the LAN component obtains the list of the running applications from the scheduler (Line 4). For each one, it identifies its model, profile and phases (Line 5). The next step consists of evaluating if there is interference between the models of the application ready to run and the currently running. If there is no interference between

---

**Algorithm 4** Fine-grained scheduling algorithm. Variable  $app_k$  represents a given application  $k$  that may be in execution or ready for being executed.  $\mathbb{M}app_k$  represents the application model used by the Analytic component, and  $node_k$  is the compute node where the application is being executed.

---

```

1: for each  $app_k$  in Ready_Queue do
2:    $\mathbb{M}app_k = model(app_k, AdaBoost, SVM)$ 
3:    $node_k =$ 
4:   for each  $app_j$  in Execution do
5:      $\mathbb{M}app_j = model(app_j, AdaBoost, SVM)$ 
6:     if  $interference(\mathbb{M}app_k, \mathbb{M}app_j) == FALSE$  then
7:        $node_k = request\_node(app_j)$ 
8:     end if
9:   end for
10:  if  $node_k ==$  then
11:     $node_k = request\_new\_node()$ 
12:  end if
13:   $execute(app_k, node_k)$ 
14: end for

```

---

two applications (Line 6) and there are enough available resources in the same compute node, then the LAN component uses the same node for running the new application (Line 7). However, if interference is detected or there are not enough resources, then the LAN component requests to allocate  $app_k$  in a new node. Finally, in Line 13 the LAN notifies the scheduler which is the compute node where application  $app_k$  should be executed.

This scheduling process has been done taking into account multiple factors. At first, the scheduler notifies LA that the application  $App_1$  will be executed in  $node_i$  (for this use case). Once  $App_1$  has started, the monitor starts providing the collected metrics and the application model of  $App_1$  is obtained from ES. This process of identifying the current application from the application ensemble is done by executing two machine learning algorithms (AdaBoost and SVM) and making a consensus between their results. Then, the scheduler gets a new application  $App_2$  from the application queue and notifies its execution to LA. With this information, LA obtains the application models from  $App_2$  and evaluates if there could be interference between both applications taking into account their models and the current state of the  $App_1$ , because it is running for a while. If no interference is detected (theoretically), LA allows the execution of  $App_2$  in  $node_i$ , where  $App_1$  is running. Otherwise, a new node  $node_j$  is returned by LA, and the scheduler executes  $App_2$  in  $node_j$ . Note that this scheduling process is complementary to *coarse-grain scheduling*. If real interference is detected during the execution of  $App_1$  and  $App_2$  in  $node_i$ ,  $App_2$  would be moved to another free node.

The objectives of this strategy are to detect when an application  $App_i$  is running in a certain node  $node_i$  and detect the current execution phase of  $App_1$  when the scheduler wants to execute another application  $App_2$  in the same node.

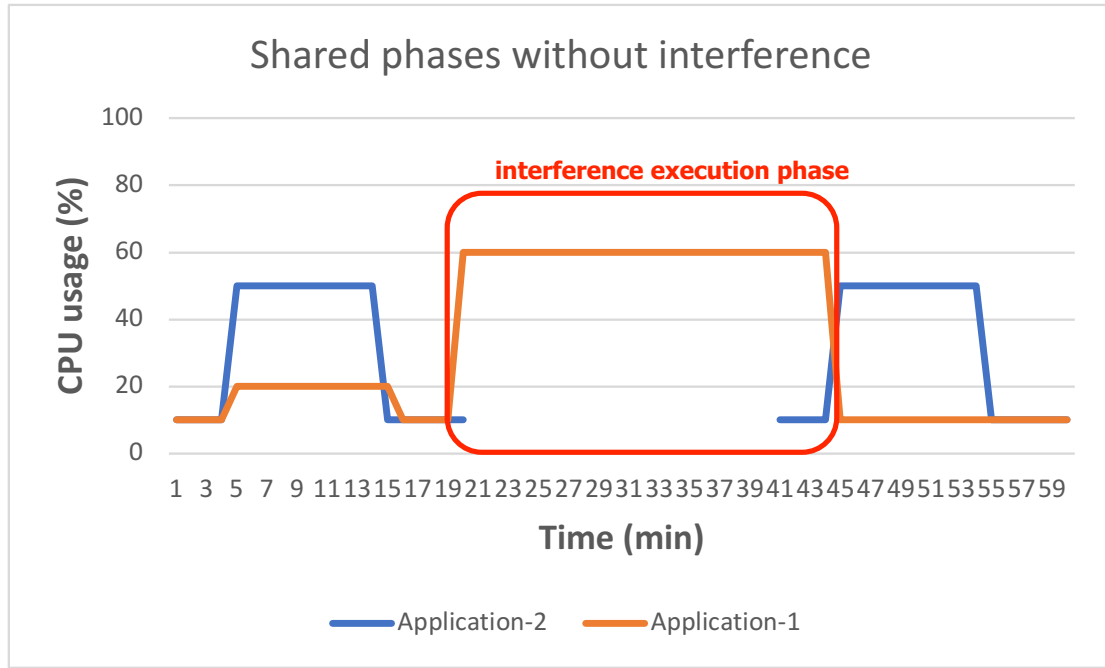
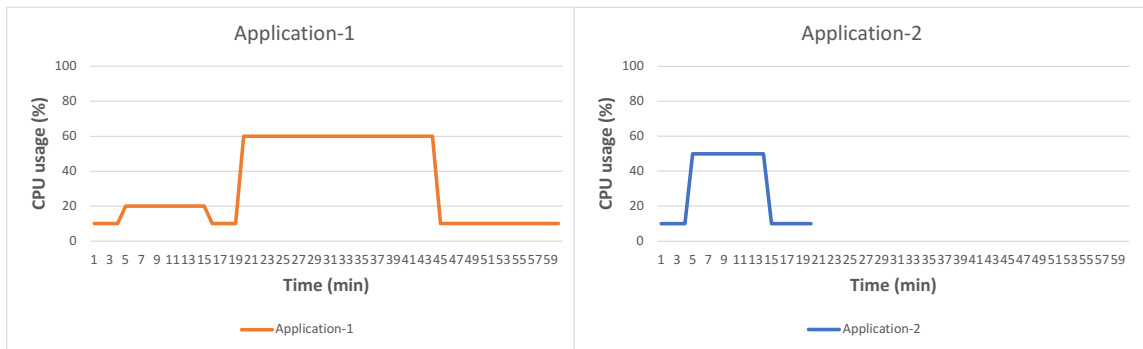


Figure 5.3: Fine-grain scheduling result - Execution phases where both applications can share a node.



(i) Application 1 - CPU pattern.

(ii) Application 2 - CPU pattern.

Figure 5.2: CPU execution patterns of two random applications that will share a node.

An example of how this scheduling policy tries to find interference between applications can be seen in Figures 5.2 and 5.3. Both charts in Figure 5.2 represent the CPU pattern of two random applications that are in the execution queue. Once the *Application-1* is executed in a certain node, the analytic component tries to find out if the next application in the queue, *Application-2*, can be executed in the same *node-x*. This process is done by predicting the performance of both applications and finding out if there are enough resources at different points in time. The result of this process can be seen in Figure 5.3, which shows how the analytic component is able to detect that there would be a lack of computational resources (required CPU is greater than 100%) if *Application-1* and

*Application-2* share a node when *Application-1* is executing the phase between the 20 and 45 minutes. It means that both applications can share the node in two phases: since the beginning of the *Application-1* to minute 20, and since minute 45 to the end.

The combination of applications in shared nodes (with available computational resources) should reduce the makespan of the executing queue because of the leveraging of those free resources, also increasing the throughput. However, as this scheduling process is based on ML algorithms, and they are not perfect in terms of accuracy, there are possibilities of executing applications with real interference in the same node (due to prediction and identification failures), producing a bit of performance degradation. But in these cases, CLARISSE would detect those conflicts and they would be avoided migrating applications to other free nodes. So that, the worst scenario would be one with a set of applications ready to be run, where all of them causes interference between them. In this case, the combination *LA + Scheduler* would detect the profile of the applications (and their phases) and would execute all of them as an exclusive policy, avoiding interference from the beginning.

### 5.3. Summary

This chapter is dedicated to one of the main points of the LIMITLESS framework, the analytic component, which is in charge of analyzing the whole data, modeling applications, predicting future states of the cluster and performing other operations to improve the framework performance (for example, reduce the monitoring network traffic, or improve the scheduling policies). The second section describes the concept of *multi-criteria scheduling* and the two different policies designed to schedule the applications based on (1) monitoring data and (2) the use of shared nodes: coarse and fine-grained monitoring.

The following chapter (Chapter 6) shows a formal model, simulations, and another developed tool to help users with the deployment and the topology design related to the monitoring tool included in LIMITLESS.

## 6. THEORETICAL MODELING

This chapter presents the model behind the system monitor of LIMITLESS framework, with the objective of define a mathematical description of it. The goal is to present some results in this way to make clear and understandable the capabilities and limitations of its implementation. It includes one formal model for the system monitor (structure, communications and node loads), it also analyzes the impact of different general optimizations, and finally, a study that considers the fault tolerance scheme. Also, this chapter includes a simulation example using OMNET++ [107] to evaluate the scalability of the model, and the description of a new developed tool to estimate the communication cost of the input topological designs.

### 6.1. System monitor model

Following, the main structures of this system monitor are presented. At this point, the defined objects are identified as a node, instead of processes, because we are assuming that each node executes only one function. It means that each node object is directly related to its main process: LDM, LDA or LDS, and the organization of these nodes can be seen in Figures 6.1 and 6.2.

***Definition 1:***

$D \rightarrow \text{DaemonMonitorNode}$

$A \rightarrow \text{AggregatorNode}$

$S \rightarrow \text{Servernode}$

$C \rightarrow \text{Communication}$

$ES \rightarrow \text{ElasticSearch}$

The system is organized based on sets, so that the set of a component ( $S_x$ ) include all nodes connected to that element  $x$ . Now, knowing the different structures, Figure 6.1 shows the relation between the framework components and the sets defined in the architectural model, which is described as:

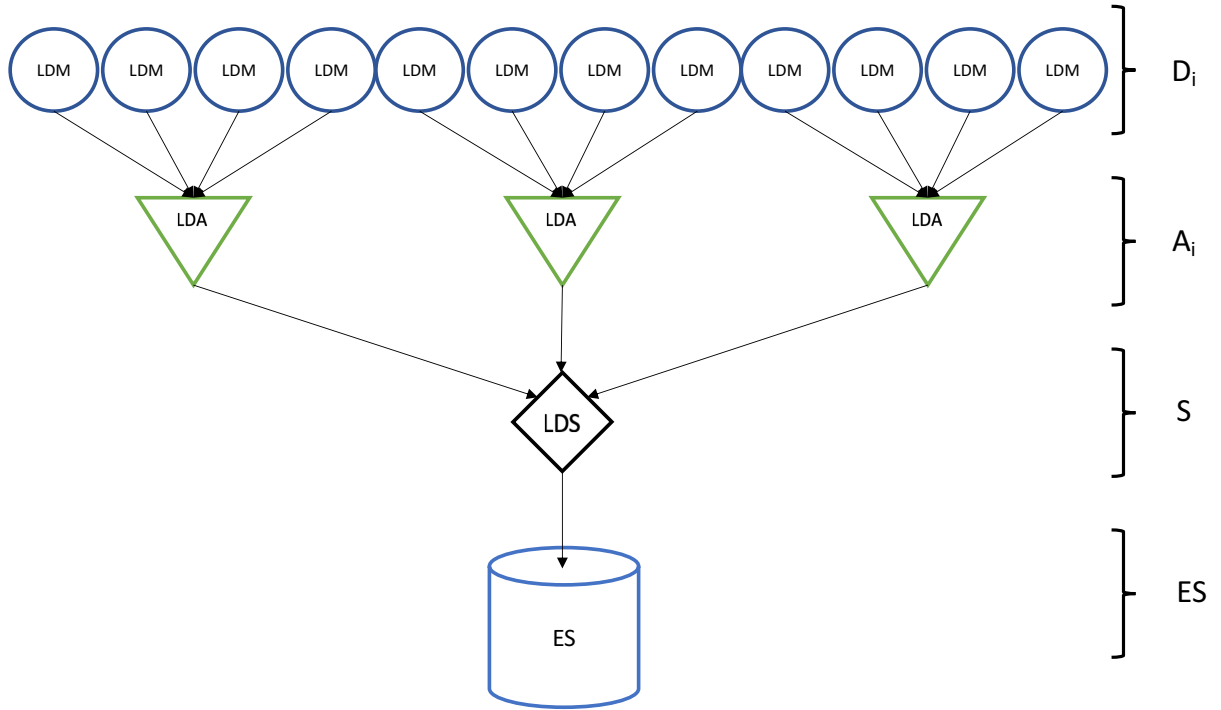


Figure 6.1: Relation between framework components and the sets defined in the theoretical model in Definitions 1 and 2.

**Definition 2:**

$$D_i = \{\{D_0, \dots, D_n\} \rightarrow n \in S_D\}$$

$$A_i = \{\{D_0, \dots, D_n\} \cup \{A_0, \dots, A_m\} \rightarrow n, m \in S_A\}$$

$$S = \{\{A_0, \dots, A_p\} \cup \{D_0, \dots, D_q\} \rightarrow p, q \in S_S\}$$

$S_A$  represents a set of daemon monitor nodes and a set of aggregator nodes that are connected to  $A_i$ .  $S_S$  represents a set of aggregator nodes and daemon nodes that are connected to  $S$ .

Let's assume that we are working with a set of tree-based topologies that send the information to an ElasticSearch  $ES$  database, and for each branch of the tree  $S_n$ , indexes  $i$  and  $j$  define the horizontal position of an element, and  $q$  defines the aggregation level for each subtree, assuming that the maximum number of levels is 2. The graphical representation of this model can be seen in Figure 6.2. In this topology, the previous subsets can be particularized as follows:

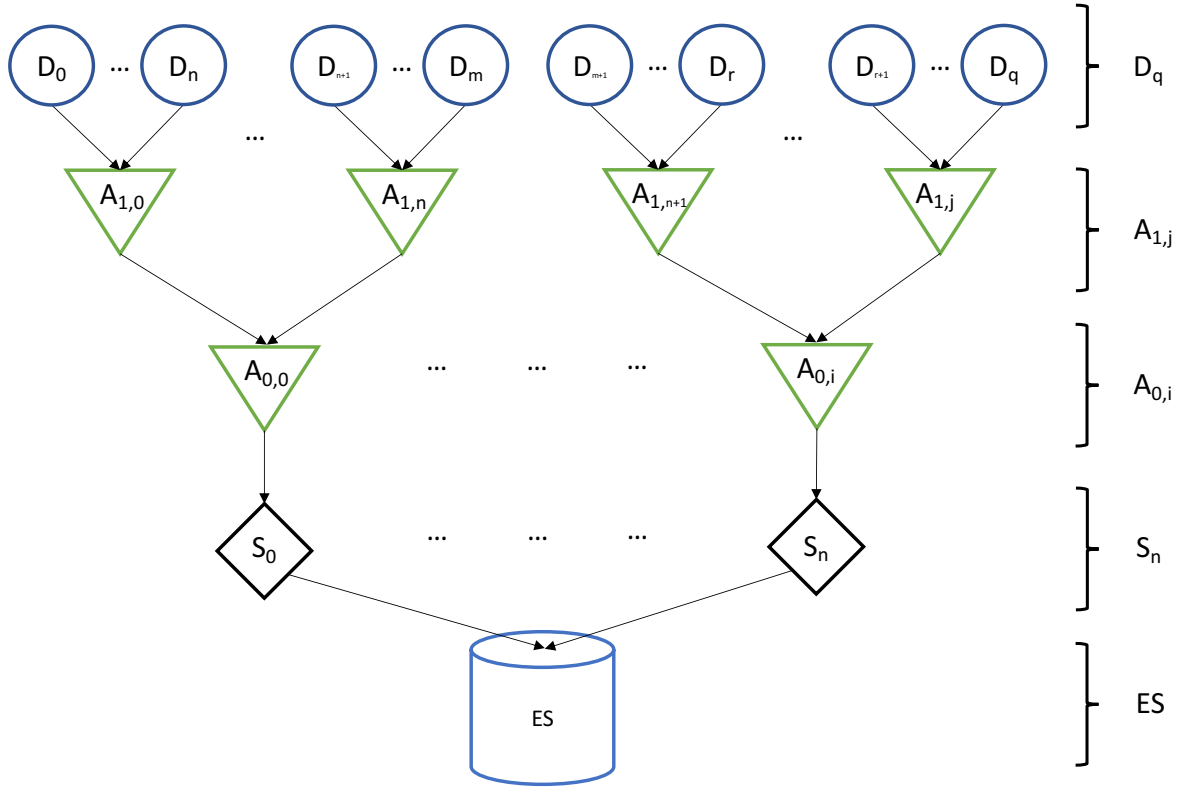


Figure 6.2: Relation between framework components and the sets defined in Definitions 3.

**Definition 3:**

$$ES = \{S_0, \dots, S_n\}$$

$$S_n = \{D_{0,0}, \dots, D_{0,q}\} \cup \{A_{0,0}, \dots, A_{0,q}\}$$

$$A_{0,i} = \{D_{1,0}, \dots, D_{1,i}\} \cup \{A_{1,0}, \dots, A_{1,i}\}$$

$$A_{1,j} = \{D_{2,0}, \dots, D_{2,q}\}$$

Next formulas represent the connections (network communication channel) between components in order to proceed to explain the results mathematically. These connections are represented in Figures 6.1 and 6.2 by arrows, and always follows the same directions: from  $D$  to  $A$ , from  $A$  to  $A$  or to  $S$ , and from  $S$  to  $ES$ .

**Definition 4:**

$$\begin{aligned}
(\forall D_i \in S_A) \exists C_{D_i,A} \\
(\forall D_k \in S_S) \exists C_{D_k,S} \\
(\forall A_j \in S_S) \exists C_{A_j,S} \\
(\forall S_l \in ES) \exists C_{S_l,ES}
\end{aligned}$$

As you can see, each aggregator node ( $A_i$ ) will receive one packet per daemon (D) and aggregator (A) in its set. In the same way, the server (S) will receive one packet per daemon (D) in its set and as many packets as each aggregator (A) in its set has received. Thus, we can define the number of connections to an Aggregator (A) as:

**Definition 5:**

$$\begin{aligned}
C_{A_{0,i}} &= (\sum C_{A_{1,i},A_{0,i}} + \sum C_{D_{1,i},A_{0,i}}) \rightarrow i \in S_A \\
C_{A_{1,j}} &= \sum C_{D_{2,j},A} \rightarrow j \in S_{A_{1,j}}
\end{aligned}$$

And the number of connections to a server ( $C_{S_n}$ ) with the following equation:

**Definition 6:**

$$C_{S_n} = (\sum C_{A_{0,i},S} + \sum C_{D_{0,j},S}) \rightarrow i, j \in S_{S_n}$$

Extending the equation to reflect the total number of connections in LIMITLESS monitoring tool to the  $ES$  database  $C_{ES}$  (we are assuming one database instance):

**Definition 7:**

$$C_{ES} = \sum_{i=0}^q C_{S_i} \rightarrow q \in S_{ES}$$

**6.1.1. Communication model**

Each connection between D-A, D-S, A-S or S-ES will have a certain network overhead during the monitor running. This overhead will depend on the number of messages sent



during a certain time, which frequency is the reverse of the *sample\_period* (time between measures, defined by the user in the deployment).

Let  $C_{D_i, A_j}$  represent the connection between the component  $D_i$  and  $A_j$  in the system. Thus, the number of messages between two elements  $D_i$  and  $A_j$  of the monitor can be calculated as:

**Definition 8:**

$$T_s = \frac{time}{sample\_period}$$

Assuming that  $P_D$  bytes is the size of each packet with monitoring data sent by  $D_i$ , the number of bytes sent along *time* through the connection link will be:

**Definition 9:**

$$O_{D_i, A_j} = P_D * T_s$$

Assuming that  $P_A$  bytes is the size of each packet with monitoring data sent by  $A_j$  to another  $A_k$  or to  $S$ , the number of bytes sent along *time* through the connection link will be:

**Definition 10:**

$$O_{A_i, A_j | S} = P_A * T_s$$

Assuming that  $P_S$  bytes is the size of each packet with monitoring data sent by  $S_q$  to  $ES$ , the number of bytes sent along *time* through the connection link will be:

**Definition 11:**

$$O_{S, ES} = P_S * T_s$$

The communication overhead  $O_i$  can be modeled generally as follows:

**Definition 12:**

$$\begin{aligned}
O_A &= (\sum O_{D_i,A} + \sum O_{A_j,A}) \rightarrow i, j \in S_A \\
O_S &= (\sum O_{A_k} + \sum O_{D_p,S}) \rightarrow k, p \in S_S \\
O_{ES} &= \sum O_{S_p} \rightarrow p \in ES
\end{aligned}$$

Taking into account all the partial equations, the global overheads can be summarized as follows:

**Definition 13:**

$$\begin{aligned}
O_{A_j} &= (\sum (P_D * T_s) + \sum (P_A * T_s)) \rightarrow j \in S_A \\
O_{A_j} &= |C_{D_i,A}| * (P_D * T_s) + |C_{A_i,A_j}| * (P_A * T_s) \rightarrow i, j \in S_A \\
O_{A_j} &= (|C_{D_i,A}| * P_D + |C_{A_i,A_j}| * P_A) * T_s \rightarrow i, j \in S_A
\end{aligned}$$

**Definition 14:**

$$\begin{aligned}
O_{S_j} &= (\sum (P_D * T_s) + \sum (P_A * T_s)) \rightarrow j \in S_S \\
O_{S_j} &= |C_{D_i,S}| * (P_D * T_s) + |C_{A_i,S}| * (P_A * T_s) \rightarrow i \in S_S \\
O_{S_j} &= (|C_{D_i,S}| * P_D + |C_{A_i,S}| * P_A) * T_s \rightarrow i \in S_S
\end{aligned}$$

**Definition 15:**

$$\begin{aligned}
O_{ES} &= \sum (P_S * T_s) \\
O_{ES} &= |C_{S_i,ES}| * (P_S * T_s) \rightarrow i \in S_{ES}
\end{aligned}$$

The formulas described before represent the communication overhead between compute nodes and aggregator nodes ( $O_A$ ), between compute nodes and/or aggregator nodes, and server ( $O_S$ ), and between servers and ElasticSearch database ( $O_{ES}$ ). The relation of this overheads and the communications between the framework components can be seen in Figure 6.3.

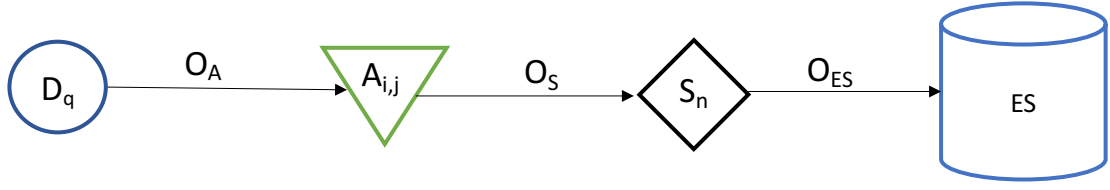


Figure 6.3: Description of the communication overheads between the framework components. It corresponds to the overheads defined in the set of Definitions 13, 14 and 15.

### Communication Limits

Assuming a bandwidth ( $B$ ) of the network link, reducing the size of packets sent by sample ( $B$ ) is critical as the maximum number of messages that could be sent through the link would be:

**Definition 16:**

$$CL_{Ei,Ej} = \frac{B}{P_D * 8}$$

$CL$  limits, not only the maximum number of packets per second, but also the aggregation capacity in the *Aggregator* nodes and the *Server*, giving a critical parameter to build the graph of the monitoring system as the monitored system scales. In *LIMITLESS*, the packet size in a common compute-node use to be small than 70B. Then, we theoretically could send:

**Definition 17:**

$$CL_{Ei,Ej} = \frac{B}{70 * 8}$$

Bandwidth	Samples per Second
1 Gbps	1.7e6
10 Gbps	1.7e7

Table 6.1: Maximum number of packages allowed by CL

Table 6.1 shown the maximum number of messages that could be sent through a network connection. As may be seen, network capacity is not a strong limitation to have a second, and even subsecond, *sample period*. The amount of resources needed to provide that number of packets is really high.

### 6.1.2. Server workload

This section explains how the time affects the server in terms of workload, which is a product of the computation done per packet received. So that, each connection  $C_{iA|C}$  implies that a  $Node_{A|S}$  receives  $T_s$  packets per input connection that has the  $Node_i$  along an execution *time*.

In this way, the total number of packets received ( $Ld_{component}$ ) can be obtained. Each aggregator node in the first level ( $A_{0,i}$ ) will receive one packet per daemon ( $D_{1,j}$ ) in its set, and  $n$  packets per aggregator ( $A_{1,k}$ ) its set, each *sample\_period*; In the same way, each aggregator node in the second level ( $A_{1,k}$ ) will receive one packet per daemon ( $D_{2,r}$ ) in its set each *sample\_period*:

**Definition 18:**

$$Ld_{A_{0,i}} = \left( \sum_{j=0}^q C_{D_{1,j}} \right) * T_s + Ld_{A_{1,k}}$$

$$Ld_{A_{1,k}} = \left( \sum_{r=0}^q C_{D_{2,r}} \right) * T_s$$

Following this model, each server ( $S_n$ ) will receive one packet per daemon (D), and as many packets as aggregators in its set had received during that *sample\_period*.

**Definition 19:**

$$Ld_{S_n} = \sum_{i=0}^q Ld_{A_{0,i}} + \left( \sum_{j=0}^q C_{D_{0,j}} \right) * T_s$$

It means that the maximum number of packets that ES server will receive in a certain amount of time *time*, is defined by the following equation, and it represents the main

bottleneck of the system  $Ld_S$ .

**Definition 20:**

$$Ld_{ES} = \sum_{i=0}^q Ld_{S_i} \rightarrow i \in S_{ES}$$

## 6.2. In-node threshold filter

In order to try to reduce the number of packets sent by the daemon nodes (D), we have introduced a function in each daemon with the objective of analyze the data collected and determine if the information is relevant to be sent. The method to do this consists of add a window range  $+/- \Delta_m$  (defined by the user), and if measurements are within this range, the information will be sent. Otherwise, no.

Mathematically, this algorithm can be modeled as follows: knowing that  $M_i$  represents a raw of metrics collected in a compute-node,  $M_k$  is the last sample sent, and  $F(M_i)$  is the function which defines the threshold filter:

**Definition 21:**

$$\begin{aligned} M_i &= \{Cpu, Mem, Net, Io\} \\ M_k &\rightarrow k \in [0, (i-1)] \\ F(M_i) &= \\ &\begin{cases} 0 & M_i \in (M_k - \Delta_m, M_k + \Delta_m) \\ 1 & M_i \notin [M_k - \Delta_m, M_k + \Delta_m] \end{cases} \end{aligned}$$

With this optimization the communication model change reducing the amount of traffic through the network thanks to the threshold filter over each packet sent between two elements  $O_{f-E1,E2}$ :

**Definition 22:**

$$O_{f-E1,E2} = \sum_{i=1}^{T_s} P * F(M_i)$$

The equation that defines the server with this optimization is:

**Definition 23:**

$$O_S = (\sum O_{f-A_jS} + \sum O_{f-D_iS}) \rightarrow i, j \in S_S$$

### 6.3. Fault tolerance

To deal with failures in the different levels of this framework it has been implemented two mechanisms:

- Watchdog processes: The three different modules (daemon, aggregator and server) have control about their state (running or not). Each 30 seconds, each process calls a function to check if it is running in the system. If the result is that it is not running, it re-launches itself with the same configuration to continue its job.
- TMR: To maintain the communications between the modules in case of network failure or superior-nodes failures it has been implemented a triple redundancy to send the data. Each compute-node and aggregator node have three different directions to send the information: the main path and two backups. In case of failure sending the data, the module will try to send the information to the second path, and in case of failure too, it will try with the third option. This method implies that there are, at least, three daemon aggregators (for TMR in daemon layer), and three servers (for TMR in last aggregator layer).

The first method has not cost (is lower than 0.1% in CPU usage) because it consists of a simple check of a process state each 30 seconds and, only if the process is not running, a command is executed.

The second method has two different configurations:

1. Always send the information to the same node unless there is a failure, in this case try to send to the first backup node and then, to the second backup node.
2. Always send the information to three referenced nodes (redundancy).

Option 1. has no cost because there aren't new computations and the information sent is the same. However, Option 2. increases the amount of information sent through the network and increases the number of packets that the referenced nodes must process. So

that, with this second option the formulas must be adapted including the set of  $A S_A$  to  $S_{A^*}$ , that now grows up to include the set of other two aggregator nodes in its same depth level:

**Definition 24:**

$$S_{A^*i,j} = \{S_{Ai,j} \cup S_{Ai,k} \cup S_{Ai,p}\} \rightarrow i, j, k, p \in S_A \rightarrow k, p$$

$$O\_TMR_{Ai,j} = (\sum O_{DiAi,j} + \sum O_{AjAi,j}) \rightarrow i, j \in S_{A^*i,j}$$

$$O\_TMR_{S_z} = (\sum O_{Ai} + \sum O_{D_jS}) \rightarrow i, j \in \{S_{S1} \cup S_{S2} \cup S_{S3}\}, z \in \{1, 2, 3\}$$

$O\_TMR_{Ai,j}$  represents the overhead of an aggregator node with TMR and redundancy, and includes the overhead of two other aggregator nodes in its same parent set ( $S_{A^*i,j}$ ). It includes the amount of data from its compute-nodes, its aggregator-nodes and the overhead from other two aggregators in its same level of scalability.  $O\_TMR_{S_z}$  represents the overhead of the three servers, which is the same than without TMR and redundancy but, in this case, instead of having one main server, the information is replicated three times. Then, when the servers try to consolidate the information in the shared database the redundancy is managed.

#### 6.4. Simulation model

The idea of using simulations to tests some features had been taken from [108], where authors describe a platform to model and simulate cloud computing systems. As the monitor is distributed and can be executed in both environments, real and virtual platforms, and due to the difficulty of having access to a large cluster (more than eighty two nodes), the scalability limit of the monitoring tool included in LIMITLESS has been evaluated based on simulation techniques, through the using of SIMCAN [109] and OMNET++ [107].

In this case, OMNET++ and SIMCAN are configured to run under the same conditions as the real cluster that the tests have been performed, which consists of a set of nodes in the same rack (everyone executes LDM functions), and a node separated by two switches (which executes LDM functions) because it is in another network segment. To do this, the simulator includes INET Framework [110] [111], an open-source model library, that provides protocols, agents, and other features for designing and validating new scenarios

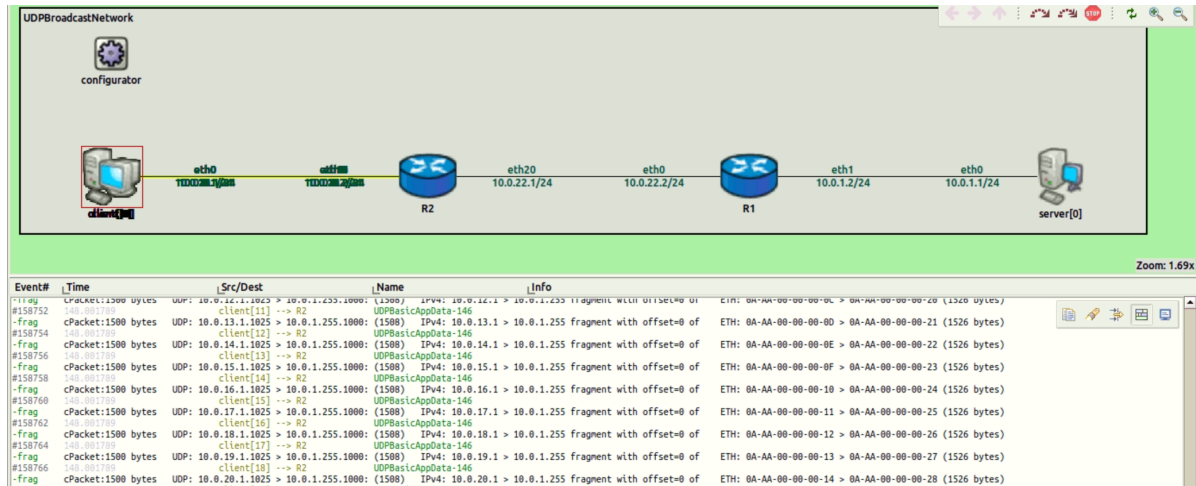


Figure 6.4: OMNET++. Example with a rack of 20 LDM nodes and one LDS. The devices in the middle are switches that allow the topology representation.

based, among other things, on:

- Models for the Internet stack (TCP, UDP, IPv4-6, etc.).
- Wired and Wireless protocols (Eth, PPP, IEEE 802-11)
- MANET protocols.

The simulation model corresponds to a general hierarchical deployment, which contains  $n$  homogeneous nodes with the following configuration: each node executes the LDM process, and the hardware includes one network interface and I/O device. All nodes that execute LDM functions are connected with another node that executes the LDS function, and the connection is done between an intermediate switch. The communication links are 1Gbps Ethernet, and the simulated use case corresponds to intensive monitoring, which consists of setting the sampling interval in one second, and only one thread to operate in LDAs and LDS. Note that LDS is, actually, a multithread process, which means that the performance in real scenarios is higher. Figure 6.4 shows an example of the described configuration in the simulator. The blue figures represent the intermediate switches, the machine on the right represents LDS, and the set of machines on the left are the nodes that execute LDM. Figure 6.5 shows how SIMCAM simplifies the process of configuring nodes in a network, only including the hardware features and the number of devices that are needed. These images show an example. However, complex topologies can be designed and simulated to test different configurations.

In addition to the basic configuration, the original models from INET have been modified to fit functionality of the real functions. This means that each compute-node includes



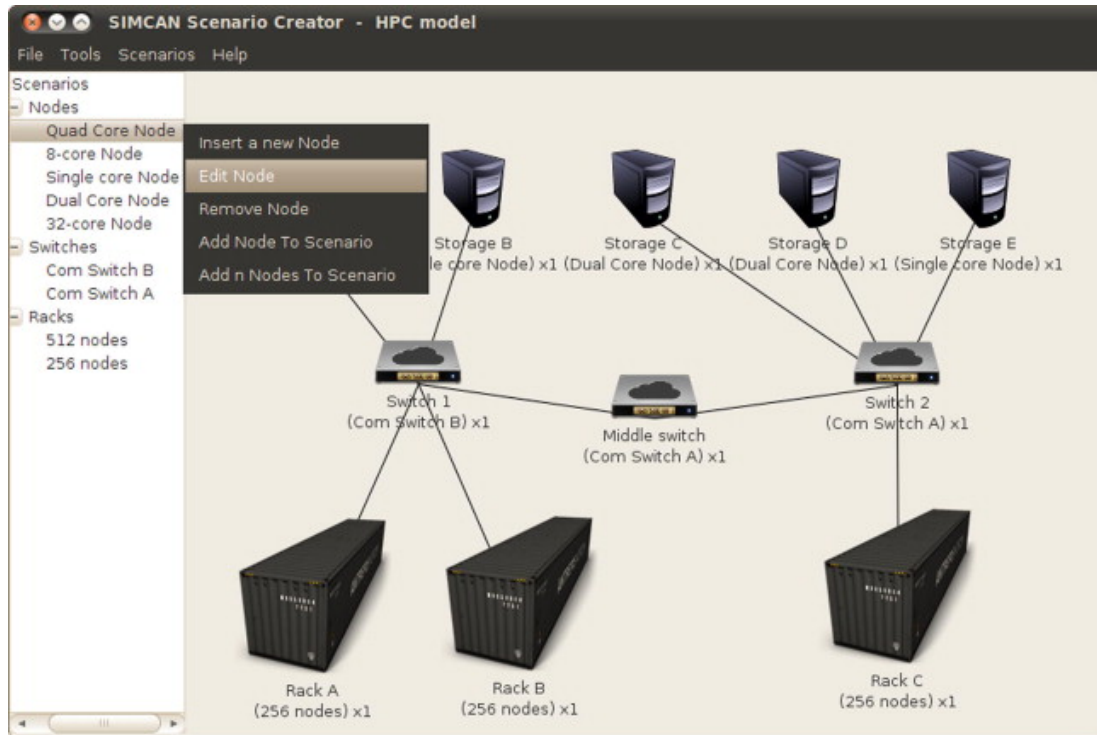


Figure 6.5: Screenshot of the SIMCAN Scenario Creator tool.

a processing overhead, which simulates the LDS functions of receiving, processing, and storing the incoming metrics. These overheads have been carefully calculated:

1. Calculating how long it takes for a package to be received, processed, and stored, along the time in a real use case with an LDS with one thread.
2. Obtaining the average time and the standard deviation.
3. Creating a model for LIMITLESS in OMNET++ that includes this processing overhead in the server. To perform the worst case, the final overhead for processing is  $avg\_time + std\_dev$  seconds, where  $avg\_time$  is the average time that a thread spends in receiving, processing, and storing a packet, and  $std\_dev$  is the standard deviation of the same variable. The objective of considering both is to create the worst-case simulation.

Finally, in order to simulate in a more realistic way, the model includes a feature to run the simulations with a log created with real data (monitoring information directly collected from a real platform). The objective of this feature is to perform simulations based on the real platform configuration, design and information collected. So that, the packets that each monitor sends, are obtained from a log which stores the metrics collected in the real cluster, resulting in sending the same information through the network as a real cluster and producing more accurate simulations.

After running different simulations, the results can be observed in a log file, which includes the whole packets created and transferred, and the timing of each event. The most important values in the log are the number of packets created for each monitor, the number of packets received by the server, and the queue of waiting packets. If the last value is greater than zero, it means that the server can't manage the volume of incoming metrics. However, if that value is zero and the packets created are the same as processed, it means that the model designed is valid (from a simulating point of view).

Under these circumstances, the results show that each LDA and LDS can manage up to 200 connections to them (LDMs or LDAs). Note that this result corresponds to an unreal case (taking into account the current technology), but it represents the worst-case scenario.

#### 6.4.1. Simulation model validation

In order to validate the simulation model described before, the same configuration and experiments have been performed on a real platform. This configuration includes different topologies to test different conditions, variations in the number of nodes that are monitored, same speed interconnection and latencies, and LDSs in one-thread processor mode. Besides, the monitor has been updated to write a log that shows the size of the incoming packets queue, and the time spent to process and send it.

Table 6.2: Simulation vs Real - Experimentation under different conditions in both simulated and real environments. Every use case simulates one hour.

Nodes	Interval (s)	Sim. sent/recv.	Real avg. sent/recv.	Unreceived msg.
1	1	3598 / 3598	3599 / 3599	0 %
	5	721 / 720	720 / 720	0 %
	10	361 / 360	360 / 360	0 %
5	1	17991 / 17900	17995 / 17995	%
	5	3601 / 3600	3595 / 3590	0.83%
	10	1801 / 1800	1795 / 1795	0%
10	1	35981 / 35980	35987 / 35978	0.63%
	5	7201 / 7200	7200 / 7200	0.34%
	10	3601 / 3600	3600 / 3600	0%
20	1	71961 / 71960	71960 / 71936	1.16%
	5	14401 / 14400	14347 / 14036	2.16%
	10	7201 / 7200	7150 / 7040	1.53%

Table 6.2 shows the number of packets sent and received both in real and simulated experiments. The results show that the real environment is well represented by the model in OMNET++, that can be verified seeing the correlation between the packets sent and received in the experimentation. Note that the simulated sends and receives differs in one packet. This behaviour is identified as a simulation issue because when the simulation

timer reaches the maximum time, the whole simulation processes are stopped, including the server process, which can't receive the last message sent. If the finalization process is delayed, the whole packets are well received. On the other hand, real experiments have also been repeated ten times and the table shows the average results, as well as the percentage of packets that the server has not received on each one. Note that in the worst case, the unreceived packets represent the 2.16% of the total sent packets, and these results are obtained with the server configured with one processor thread (in the same way as the model).

### 6.5. Communication cost calculator

The monitoring tool has two limiting factors: the network bandwidth and the computation in some components, which are servers and aggregators. The previous model explains, theoretically, in a formal way, how to deal with the communications between components. In the same way, the computation in aggregators and servers has been simulated thanks to OMNET++. In order to give feedback to the user when he/she wants to design a topology, a tool called *LIMITLESS Deployment Checker* has been implemented in Python.

This tool is able to detect if the configuration selected for a platform is correct or if there is possibility of future performance issues (overloaded aggregators or servers due to the number of communications from other components). To check the configuration, the program accepts the parameters described in Table 6.3.

Table 6.3: LIMITLESS - Deployment Checker parameters.

Parameter	Description
<b>Num. nodes</b>	The number of nodes where the monitor will be deployed.
<b>Aggs per server</b>	The number of aggregators that will be connected to each server.
<b>Number of servers</b>	The number of servers to receive the information.
<b>TMR mode</b>	<i>Triple modular redundancy</i> mode: 0 enabled, 1 enabled in round-robin mode, and 2 to disable it.
<b>Threshold filter</b>	Value for the in-node threshold filter to estimate the amount of packets not sent.
<b>Num. Nets</b>	The number of network interfaces in each node.
<b>Num. I/Os</b>	The number of I/O devices in each node.
<b>Num. GPUs</b>	The number of GPUs in each node.
<b>Sampling interval</b>	Every <i>sampling interval</i> the LDM collects and sends the metrics.
<b>Network speed</b>	The speed of the network in Gbps. By default 1Gbps is configured.

The checker evaluates the viability of the desired deployment in two ways: calculating

the percentage of the network bandwidth that will be used by the monitoring information, and the limit provided by OMNET++ by means of simulation, to advice the user about the possibility of performance issues due to servers or aggregators saturation.

To run the deployment checker, the user has to execute a command line like the following:

```
$ python depChecker.py 400 2 2 2 10 1 1 1 1 1 (cmd line 1)
```

```
# 400 nodes.  
# 2   LDA per LDS.  
# 2   LDS nodes.  
# 2   TMR mode disabled.  
# 10  Threshold filter value.  
# 1   Number of network interfaces per node.  
# 1   Number of I/O devices per node.  
# 1   Number of GPUs per node.  
# 1   Sampling interval.  
# 1   Gbps network channel speed.
```

The parameters in the command line are in the same order as the Table 6.3, indicating that the example is going to check if the configuration for 400 nodes, two aggregators per server, two servers, TMR in mode two (disabled), threshold set to ten, and one for the rest of the parameters: number of network interfaces, IO devices and GPUs, sampling interval and network speed.

When an administrator, or user, wants to deploy the monitor, there are two ways: designing a model and simulate it in OMNET++ or executing this *Deployment Checker* with the desired parameters. Using this tool is fast and easier, but it could be less accurate. The execution example displayed in Figure 6.6 corresponds to the execution of command-line 1, and its results show that the configuration is correct. This tool is complementary to OMNET++ simulations because it reduces the time spend to decide which configuration the user wants to deploy.

```

#####
#####
##### LIMITLESS DEPLOYMENT CHECKER #####
#####
#####
Nodes to monitor: 400
TMR mode selected: 2
Network bandwidth: 1 Gbps --> 125000 B/s

Nodes per aggregator (bigger rack): 100
Node transmission: 24 bytes
Aggregator reception: 2400 bytes
Server reception: 9600 bytes
Design accepted.
Bandwidth between nodes and aggregators: 1.920000 %
Bandwidth between aggs/nodes and servers: 7.680000 %

Process finished with exit code 0

```

Figure 6.6: LIMITLESS Deployment Checker - Example output of the execution of *cmd line 1* without TMR.

Another interesting example can be found in the next command-line, where the point is the TMR mode set in 1. In modes zero and one, each component is linked with other three in the next layer (this feature is described in Chapter 5) to send the information to all of them. However, the difference between them is the redundancy: the first method sends all to the three components, but the second uses the channels only to send the information until it verifies that the information has been received by one of the next components. That avoids redundancy and reduces the amount of data transmitted. Figure 6.7 shows the result for this example.

```
$ python depChecker.py 800 2 3 1 10 1 1 1 1 1 (cmd line 2)
```

```

# 800 nodes.
# 2  LDA per LDS.
# 3  LDS nodes.
# 1  TMR mode enabled without redundancy.
# 10 Threshold filter value.
# 1  Number of network interfaces per node.
# 1  Number of I/O devices per node.
# 1  Number of GPUs per node.
# 1  Sampling interval.
# 1  Gbps network channel speed.

```

```

#####
#####
##### LIMITLESS DEPLOYMENT CHECKER #####
#####
#####

Nodes to monitor: 800
TMR mode selected: 1
Network bandwidth: 1 Gbps --> 125000 B/s

Nodes per aggregator (bigger rack): 135
Node transmission: 24 bytes
Aggregator expected reception: 3240 bytes
Server expected reception: 19440 bytes
Aggregator possible reception (failure): 9720 bytes
Server possible reception (failure): 58320 bytes
Design accepted.
Bandwidth between nodes and aggregators: 2.592000 %
Bandwidth between aggs/nodes and servers: 15.552000 %
Bandwidth between nodes and aggregators possible (failure): 7.776000 %
Bandwidth between aggs/nodes and servers possible (failure): 46.656000 %

Process finished with exit code 0

```

Figure 6.7: LIMITLESS Deployment Checker - Example output of the execution of *cmd line 2* with TMR in mode 1 (no redundancy).

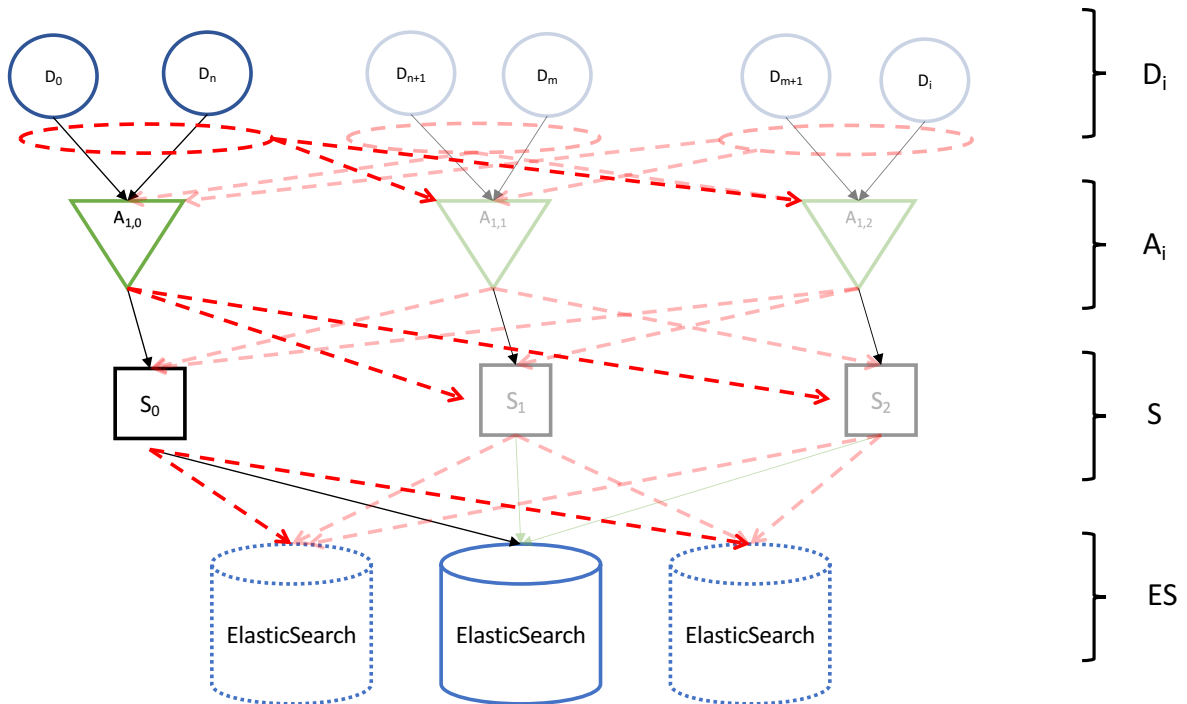


Figure 6.8: LIMITLESS - TMR connection in the worst case: each component will send the same information to three next components in the framework.

In the output observed in Figure 6.7, the lines that contains the string *failure* stand out. These lines indicate the amount of information received in case of a failure in the transmission, when the TMR is enabled (Figure 6.8 shows an example of how the monitoring packets are multiplied by three). The results correspond to the worst case, where all the connections established by the TMR fail, but only one is active and points to one server or aggregator. It means that the component will receive 3x packets in a row. The script checks if the transmission is possible and prints a message about that. In this example, the message is *Design accepted.*, and in the following case, an error message is displayed because something wrong is detected in the configuration: the number of packets per second that the LDS receives would overload the network. This use case can be seen in Figure 6.9), which corresponds to the execution of the command-line 3.

```
$ python depChecker.py 2000 2 3 1 10 1 1 1 1 1 (cmd line 3)
```

```
# 2000 nodes.  
# 2   LDA per LDS.  
# 3   LDS nodes.  
# 1   TMR mode enabled without redundancy.  
# 10  Threshold filter value.  
# 1   Number of network interfaces per node.  
# 1   Number of I/O devices per node.  
# 1   Number of GPUs per node.  
# 1   Sampling interval.  
# 1   Gbps network channel speed.
```

```

#####
#####
##### LIMITLESS DEPLOYMENT CHECKER #####
#####
#####

Nodes to monitor: 2000
TMR mode selected: 1
Network bandwidth: 1 Gbps --> 125000 B/s

Nodes per aggregator (bigger rack): 335
Node transmission: 24 bytes
Aggregator expected reception: 8040 bytes
Server expected reception: 48240 bytes
Aggregator possible reception (failure): 24120 bytes
Server possible reception (failure): 144720 bytes
There are too many nodes for each aggregator. Lost packets expected
Bandwidth between nodes and aggregators: 6.432000 %
Bandwidth between aggs/nodes and servers: 38.592000 %
Bandwidth between nodes and aggregators possible (failure): 19.296000 %
Bandwidth between aggs/nodes and servers possible (failure): 115.776000 %

Process finished with exit code 0

```

Figure 6.9: LIMITLESS Deployment Checker - Example output of the execution of *cmd line 3* the with TMR enabled and errors due to consuming the network bandwidth and the maximum nodes that an aggregator or a server can manage.

The advantage of this script is that there is no necessity of designing the model in OMNET, which requires spend time in a graphical tool to build the architecture, and time with files configuring the communications, the relations, and the functionalities of each component. The result provided by the *development checker* is less precise than the simulation because it is based on the mathematical limits calculated in this Chapter (instead of simulation), but is easy to use, faster because it is not necessary to build a complete design and wait for the simulation result (which can last several minutes, including hours), and does not require a specific installation (only Python).

## 6.6. In-node threshold calculation

As it has been commented before, the network communication is crucial for the correct performance of the framework. For this reason, trying to reduce the network usage between the compute-nodes, where the monitors share resources with the applications, and the aggregators and servers, the optimization called *In-node analysis* has been developed. However, to facilitate the task of studying which is the best value for the threshold, another script in Python is provided to perform tests in the real environment.

If one user wants to find the best threshold value, two issues arise: it depends on the granularity of the monitoring results expected, and the time that there is needed to execute



the tests. It depends on the granularity because lower values in the threshold implies more metrics received and more precise monitoring. However, greater values reduce the precision but also reduce communication because of the absence of information and the non-sending packets. It is a trade-off that the user has to evaluate (an evaluation of this algorithm, including results about precision and packets sent, is detailed in Chapter 6). On the other hand, the time to evaluate different threshold values can be unaffordable if the time of common applications in the system is too large.

For those reasons, the threshold calculator has been developed. Once the framework has collected information for a suitable time (the monitoring has information of typical use of the cluster), the generated log can be used to evaluate this optimization with the script. In a short time, the program gives to the user the original number of messages sent and received, and the number of messages that would be saved with the determined value for the threshold.

To use the script, the user has to indicate three parameters: the log with the monitored data, the value for the threshold, and the name for the output log. When the script finishes, the output log contains the performance metrics that would be received if the threshold was the defined value, and on the screen, the number of saved communications is indicated.

The next box shows an example, followed by the figures that shows the results. Figure 6.10 represents the terminal where the script has been executed. The use case designed for this feature corresponds to the execution of Jacobi algorithm, with original memory and CPU metrics plotted in Figure 7.25.

```
$ python testThreshold.py log_node12 1 output_log12 (cmd line 4)

# log_node12      Input file: monitoring data from Jacobi execution.
# 1               Threshold value.
# output_log12    Output file.
```

The example uses the log collected during the execution of Jacobi algorithm, which takes around eighteen minutes, with sampling interval set in one second. Figure 6.10 prints on screen the result of the threshold simulation: from a log that contains 995 messages with monitoring data, assuming a tolerance of 1%, the number of messages that would not have been sent is 502 (49% saved).

Taking into account this potential traffic reduction, to evaluate the trade-off between the granularity and the accuracy, Figure 6.12 shows the results of the simulation, that can be compared with the original in Figure 6.11. As this is an example, a value of

```
#####  
#####  
##### THRESHOLD EVALUATOR #####  
#####  
#####  
  
Number of messages: 995  
Number of messages not sent: 502  
  
Process finished with exit code 0
```

Figure 6.10: Example - Execution of the command-line *cmd line 4* based on a monitoring log collected in a real cluster.

1% is acceptable taking into account the accuracy and the network traffic reduction. An evaluation of this feature, including an example with different threshold values, can be seen in Chapter 6.

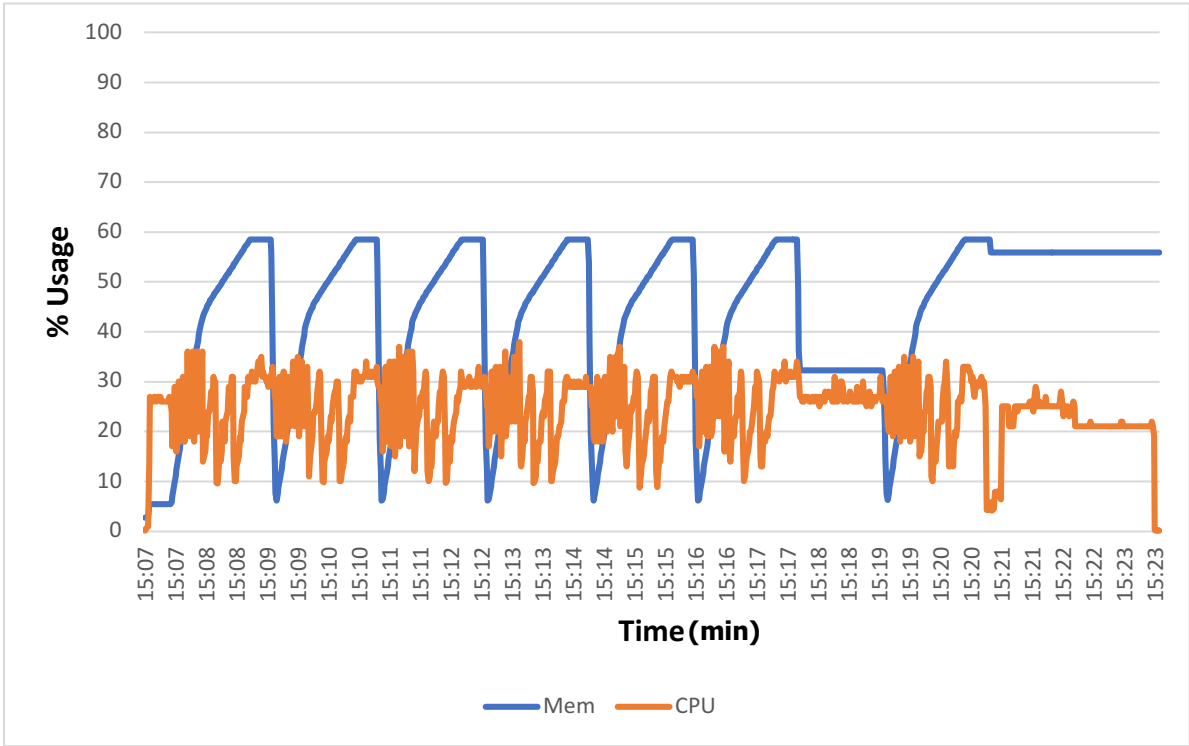


Figure 6.11: Application model - Original Memory and CPU performance of Jacobi algorithm execution.

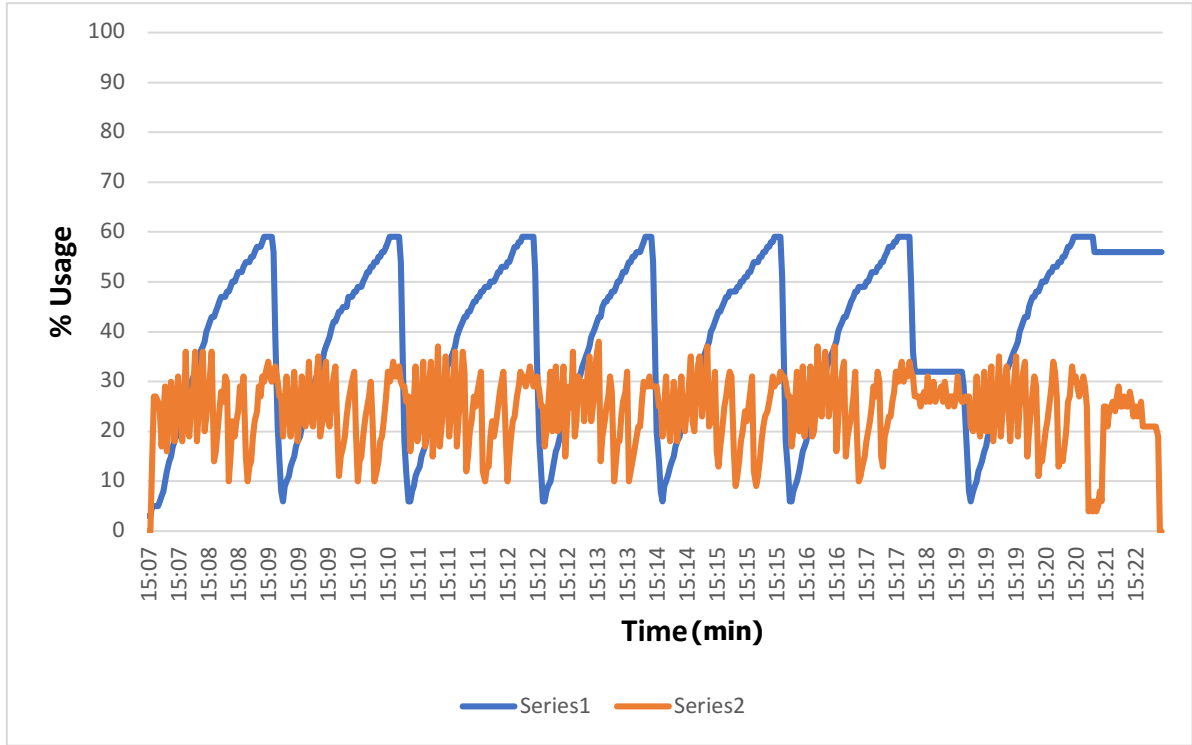


Figure 6.12: Application model - Memory and CPU performance simulated with value 1% in in-node threshold.

## 6.7. Summary

This chapter describes, in a formal way, the different components of the monitor, the communications between them, and how the limits are estimated. Also, the formal model of the different optimizations (the threshold filter and the fault tolerance strategies), and their impact on the communication performance have been included.

On the other hand, two different jobs have been done to calculate the scalability of the monitor thanks to simulation, and a program, developed in Python, to calculate which value can be a good option to set the threshold value.

All of these sections indicate different ways to obtain information to design, or deploy, in a better way the monitor. That information helps to understand the viability of the designed topology, from a communication point of view, the correctness of the design, thanks to simulation, and how to set an assumable threshold filter value thanks to the Python cost calculator developed.

In the following chapter, the results and de evaluation of all of this work are presented, including different experiments and comparisons with other similar tools.

## 7. EVALUATION AND RESULTS

In this chapter, the results of using LIMITLESS framework over different clusters (both production and development clusters) are presented. It includes a comparison between other common system monitors, from the performance and overhead point of view, results obtained thanks to using monitoring for schedule applications, and results about the different optimizations, including the models and predictions in the *smart analytic* component. The objective is to show, in different environments, the potential of the framework. Also, examples of the data collected will be displayed, and examples of the visualization offered to the user.

### 7.1. Monitoring overhead

The monitoring overhead is directly related to the sampling interval and the platform. Each LDM uses one core during a fraction of a second to collect and send (or not if the optimizations are enabled) the metrics. Taking it into account, the overhead will not be the same in a quad-core machine as in an octa-core, because the overhead is calculated as the percentage of CPU time consumed in a period of time. It means that the total overhead will depend on the number of cores used during the monitoring, and the time spent to perform the actions, which is directly related to the frequency and the FLOPS of the machine.

To provide an indicative value for the overhead, the calculation has been done in a common compute node: Intel(R) Xeon(R) Silver 4214 CPU @ 2.20GHz with 12 real cores, and 24 virtual cores (*Hyperthreading*).

The methodology to obtain the overhead of each LDM consists of running the monitor during a certain period of time, and then, ask to the system how much CPU time has been required. With this data, the calculation is defined as:

$$Overhead = \frac{T_{cpu}}{T_{total}} * 100$$

Where  $T_{cpu}$  is the real cpu time consumed,  $T_{total}$  is the total time that the monitor has been running, and the product is to obtain the percentage value.

The experimentation has been performed using four different sampling intervals: a short interval time that provides fine-grained monitoring every second, a medium interval time for every five seconds, and an acceptable interval time for a general-purpose, which is ten seconds. Originally, a long-term monitoring for every 60 seconds was planned, but the overhead obtained is negligible. The results can be seen in Table 7.1, reflecting the total times, the CPU consumed time and the global overhead obtained.

Table 7.1: Summary - LIMITLESS monitor overhead under different sampling intervals.

Case	Interval (s)	Total time (H)	CPU time (s)	Overhead (%)
Fine-grained mon.	1	24	144	0.160
medium-grained mon.	5	24	24	0.027
General monitoring	10	24	11	0.012

Note that the results displayed in Table 7.1 are lower than 1%. This value is important because running in production clusters implies that the interference with other applications must be minimum. The best solution is to collect the information consuming fewer resources, and these results support us.

For some researching applications, the sampling interval of one second could be a long or a short time interval. It depends on how much time do the applications need to finish their executions (for example, applications that perform little tasks will need a shorter sampling interval, but applications that are executed during hours or days could be monitored by seconds, minutes or hours, depending on the user objective). For the applications in the first group, LIMITLESS is able to collect and send information in sub-second intervals. However, the overhead increases rapidly and sometimes monitoring data packet is lost. The minimum interval time for a correct execution of the LDM code is 25 milliseconds. With this value the monitor does not produce any error, but some packets could be lost due to the network and the server synchronization (they do not arrive to the server due to network traffic, or when the server is restarting the connection<sup>2</sup>). So that, lower sampling intervals could produce losing monitoring packets, but on the contrary, the LDMs send more of them per second.

Although the monitor is designed for use in distributed systems, a LDM instance can be executed locally in a PC or a compute-node without sending the monitoring data, but storing it into a log-file (*local running*). If the monitor is running in this mode, sampling intervals of 25ms are accepted because there is no packet lost (every generated packet is written into a file, instead of sending it). For distributed systems, after performing some tests collecting the basic data (without cache, GPU, InfiniBand, Power consumption, fan

<sup>2</sup>To avoid an infinite wait in the server due to *recv()*, *recvfrom()* or *recvmsg()* syscalls, each determined time the connection is restarted. If a packet arrives during this process, that takes a portion of a second to finish, it is discarded

speed, etc., because they require additional time), an acceptable sampling interval is 250 milliseconds because it keeps the overhead under the 1% giving four collections of metrics per second. The overheads of sub-second monitoring are available in Table 7.2.

Table 7.2: Monitoring overhead with the minimum sampling interval in a compute-node with Intel(R) Xeon(R) Silver 4214 CPU with 12 real cores, and 24 virtual cores (Hyperthreading).

Case	Interval (s)	Total time (H)	CPU time (s)	Overhead (%)
Min interval local	0.025	24	7968	9.20
Min interval distrib.	0.250	24	648	0.75

Finally, in order to show how the monitored machine produces variations in the total overhead, other tests have been performed in one compute node Intel(R) Xeon(R) Gold 6138 CPU @ 2.00GHz with 2 CPUs and 20 cores per CPU. In this case, the overhead can be seen in Table 7.3. As it can be seen, this compute node has more cores than the last. However, maybe due to the low CPUa frequency the overhead is greater.

Table 7.3: Monitoring overhead with the minimum sampling interval in a compute-node with Intel(R) Xeon(R) Gold 6138 CPU with 20 real cores, and 40 virtual cores (Hyperthreading).

Case	Interval (s)	Total time (H)	CPU time (s)	Overhead (%)
Fine-grained mon.	1	24	504	0.583
medium-grained mon.	5	24	122	0.141
General monitoring	10	24	61	0.071

As a summary, this section shows the overheads of different configurations to understand the impact of different sampling intervals on the machines, including a sub-second sampling interval analysis.

## 7.2. LIMITLESS monitoring tool vs Collectd

This section shows a comparison between the LDM component of the LIMITLESS monitoring tool, and Collectd, a well-known monitor that has been described in the state of the art (Chapter 2). Both “*applications*” execute the same tasks: data collection and store/-transmission, and the configurations have been changed to perform the same tests. The objective is to compare the overhead of both tools, as well as their outputs, to determine the quality of the developed work.

This section includes three use cases, which are the following: a short interval time that monitors every second, a medium interval time for every five seconds, and a longer

interval time of ten seconds. These tests are design to see the trend of the overhead at low and medium sampling intervals.

The testing machine is a desktop computer with an Intel(R) Core(TM) i5 CPU 760 at 2.80GHz and with two real cores with *Hyperthreading*, 18GB of RAM memory, and 1TB of HD storage. For this new testing machine, the monitoring overhead is shown in Table 7.4, obtaining values for one, five and ten seconds intervals (the defined use cases). As the machine is a personal computer and its processor is a dual-core, the overhead is greater than in a compute-node because it has less computational resources (this is the reason why the overhead depends on the machine).

Table 7.4: LIMITLESS monitor overhead in a local PC.

Interval (s)	Total time (H)	CPU time (s)	Overhead (%)
1	24	259	0.30
5	24	62.4	0.07
10	24	29.7	0.03

Each test has been running 24 hours with the predefined sampling interval. The real CPU consumption has been collected from the system log, and the overhead has been calculated as in the last section. Table 7.5 shows the comparison results between Collectd and LIMITLESS.

Under the same load conditions, measurement interval, and collected metrics, LIMITLESS has better performance as Collectd, obtaining lower overhead in all the use cases. Moreover, the Collectd documentation recommends using an interval of 10 seconds, and seeing the overhead for that case, the recommendation is based on the designer's evaluation of the trade-off between accuracy and interference. Note that LIMITLESS with 1 second interval consumes less resources than 5 seconds of Collectd providing the same information.

Table 7.5: Summary - Collectd overhead under different sampling intervals in a local PC.

Interval (s)	Total time (H)	CPU time (s)	Overhead (%)	O. LDM (%)
1	24	2062	2.3	<b>0.3</b>
5	24	410	0.5	<b>0.07</b>
10	24	165	0.19	<b>0.03</b>

Once the overhead has been compared, in the same way and under the same circumstances, the objective is to compare the outputs of both applications. For that reason, they have monitored the same machine for the same interval (at the same time), when the Jacobi application were running. The objective is to obtain the performance metrics for both monitors whilst the machine is working (not in idle state).

The results for this test can be observed in Figure 7.1, 7.2, 7.3 and 7.4. The first

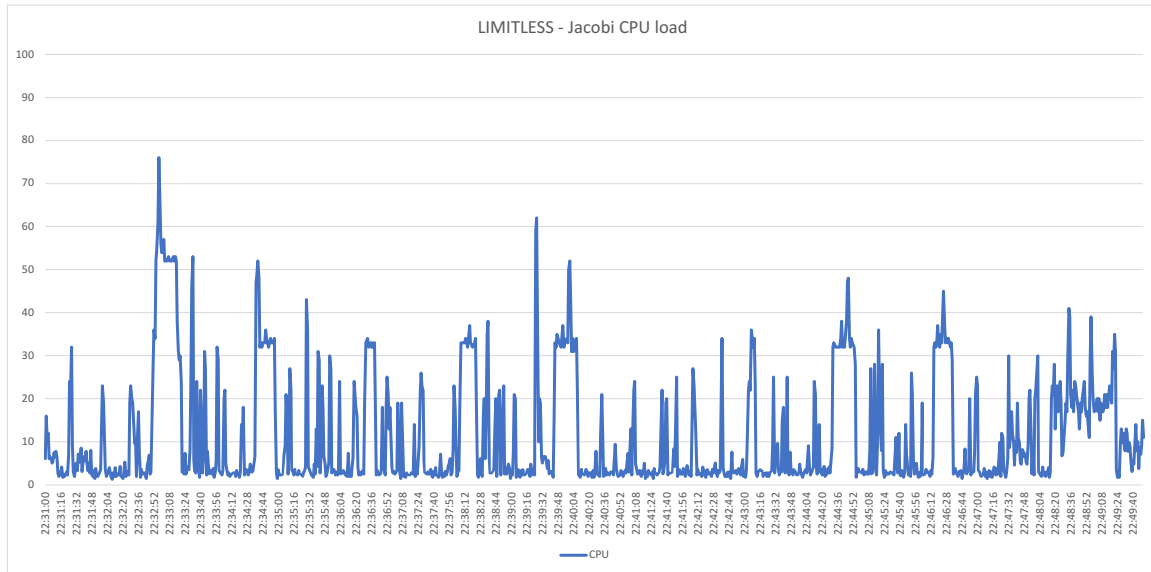


Figure 7.1: LIMITLESS - CPU usage collected during the execution of Jacobi.

two figures represent the CPU and Memory consumption captured by LIMITLESS during the execution of Jacobi algorithm. In them, you can see that the application has CPU phases, combined with a long memory-intensive phase. The next two figures represent the CPU (Figure 7.3) and the memory (Figure 7.4) captured by Collectd. In those images, you can see that both results are very similar to those collected by LIMITLESS. If the quantitative analysis is performed, Collectd shows higher peaks than LIMITLESS in CPU measurements with a difference of +5%, and similar values for low workloads (note that the representation changes in the Y-axis, starting in zero with the first increment of forty, but then the increment is set to twenty until one hundred is reached). In terms of memory, both monitors show the same behavior: LIMITLESS shows values between 28% and 39%, which translated into GB of memory means 4.7GB and 6.6GB, which is approximately what Collectd shows.

In a summary, two monitors have been compared, both obtain the same output metrics, but LIMITLESS is less intrusive than Collectd due to its reduced overhead.

### 7.3. Scheduling based on monitoring

This section describes two different scheduling techniques: *coarse-grain* and *fine-grained scheduling*. Both alternatives use monitoring information (current and historical) to improve the scheduling process. The first one provides scheduling at the application or task level, while the second provides it at the application-phases level. It means that the *coarse-grain scheduling* deals with applications and tries to share nodes between complete application executions, and the *fine-grained scheduling* decomposes the application in



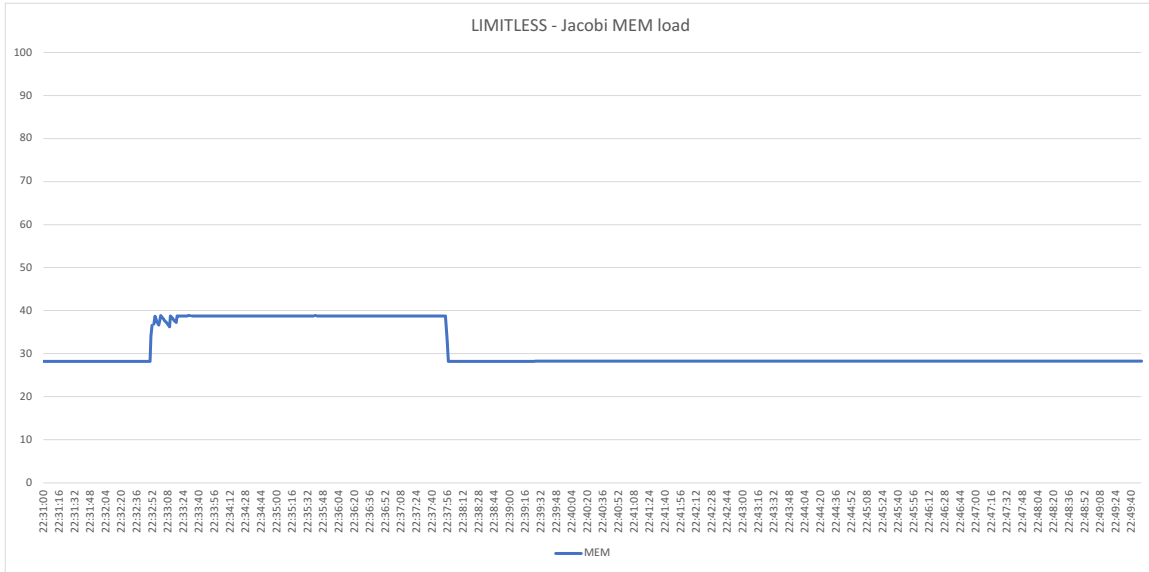


Figure 7.2: LIMITLESS - MEM usage collected during the execution of Jacobi.

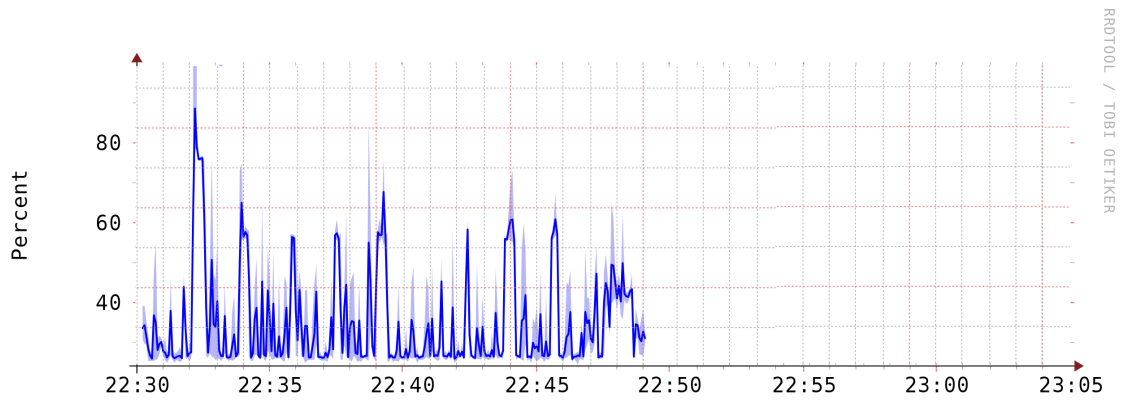


Figure 7.3: Collectd - CPU usage obtained during the execution of Jacobi.

phases using the generated models, and tries to share nodes between phases of applications if they are compatible.

### 7.3.1. Coarse-grain scheduling based on monitoring information

To evaluate the behavior of LIMITLESS when it uses the monitoring information to improve scheduling, a scenario with different combinations of applications has been designed. The idea here is to reduce the make-span of the complete execution of the application queues.

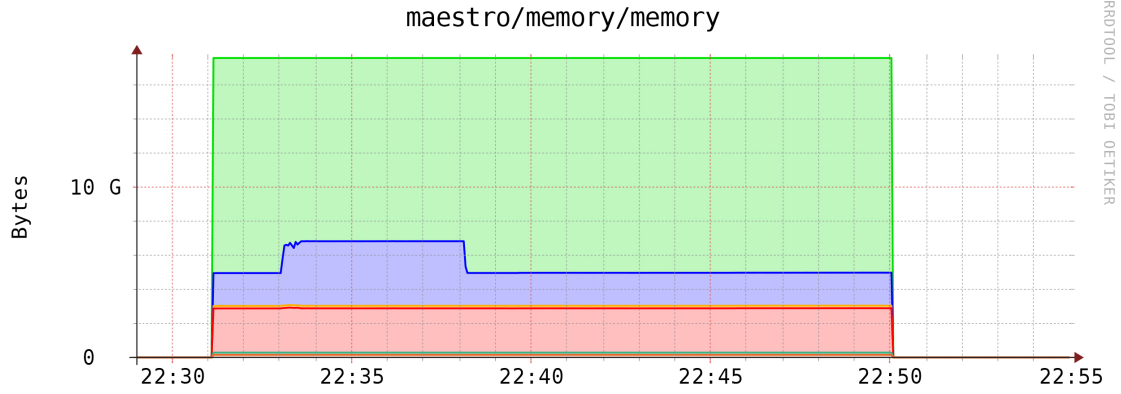


Figure 7.4: Collectd - Memory usage obtained during the execution of Jacobi.

### Platform and experiments description

For these experiments, the platform that has been used is a heterogeneous cluster with two racks. The first rack contains two nodes with Intel(R) Xeon(R) E5 with eight cores each and 256GB of RAM memory. On the other hand, the second rack contains six nodes with Intel(R) Xeon(R) E7, 128GB of RAM memory, and twelve cores per node.

Table 7.6: Use cases characteristics for the evaluation.

Code	Name	Class	Access pattern	Size	Intensive on
A	EpiGraph	Application	irregular	small	CPU & network
$B_{ll}, B_{hl}$	CG	Kernel	irregular	small	CPU
C	Jacobi	Kernel	regular	medium	CPU
$D_m$	CPU	Synthetic	regular	medium	CPU
$D_{xl}$	CPU	Synthetic	regular	large	CPU & memory
E	CPUNET	Synthetic	regular	medium	CPU & network
F	IMEM	Synthetic	irregular	medium	memory

As use cases, a collection of applications and real and synthetic kernels have been chosen. All those *benchmarks* have been integrated with FlexMPI to apply malleability techniques (which consist of increase or decrease the number of processes of MPI applications, allowing the creation of new processes in new nodes, and destroy other processes in used nodes). Table 7.6 shows each use cases characteristics.

The application *EpiGraph* [112] [113] is a stochastic and parallel simulator of the propagation of the influenza and COVID-19 viruses. It uses an un-directed weighted graph of 703,258 nodes and 8,806,520 edges, that corresponds to the individual-connections in the simulation.

*CG* is an application that performs the Conjugate Gradient iterative method, that operates with sparse matrices and executes sparse-matrix-vector multiplications (SpMV).

This algorithm is applicable to those cases that are too large to be solved directly by other methods (for example, the Cholesky decomposition). This kernel also includes two different input matrices that analyze the impact of the data locality in the algorithm. The first matrix, which is identified by  $B_{ll}$  (CG kernel with **low locality**), corresponds to a square sparse matrix with 500,000 rows and 40 million of non-zero values. Those non-zero entries are randomly distributed to generate low data locality on the vector accesses during the SpMV. The second matrix is identified by the code  $B_{hl}$  (CG kernel with **high locality**). This case is a random sparse matrix of the same size and number of non-zero values as the last, but, in order to provide better data locality on the vector accesses, those non-zero values are randomly distributed creating a block diagonal matrix of 20,000 entries.

*Jacobi* is an application that executes the kernel of the Jacobi iterative method operating with dense matrices. The basis of the method consists of creating an iterative convergent and defined sequence. The limit of this sequence is the solution of the system. For that reason, if the algorithm stops after a finite number of steps, an approximation to the value of  $x$  of the solution of the system is reached.

A set of synthetic kernels are also added to provide complete scenarios with different features. The kernel with code *CPU* is similar to the application *Jacobi*, with the difference that this process does not perform communications (it is a pure CPU application). Besides, it executes two processes: the first one is  $D_m$  which has a memory footprint for six dense matrices of 20,000 entries, and the second is  $D_{xl}$  (**extra-large**), which uses the same matrices but with 50,000 entries (120GB). *IMEM* is a memory-intensive application that operates with several matrices using indirections, and its main function in these use cases consist of generating interference in the cache memory due to the processing of six dense matrices with 20,000 entries (19.2GB). Finally, *CPUNET* is similar to  $D_m$  but alternating CPU with communication-intensive phases. This application generates network interference.

## Results

This part of the evaluation corresponds to the use case that checks the performance of coarse-grain scheduling with different classes of interference. Moreover, this section illustrates the overhead and the impact related to the framework. This use case represents a workflow (a sequence of applications) that is executed in three shared compute nodes. The results of the execution of the proposed workflow can be seen in Table 7.7. It shows the application execution order (in the column *id*) and the results of the scheduling process. The table includes, for each code, the name, number of processors used, and the memory footprint. The column *Shared* shows the id of the applications that have shared a compute node with the current application.

To understand better the results shown in Table 7.7, the following paragraphs will describe the different situations separately. The first use case corresponds to applications 1 and 2, which have been executed in the same shared node, where they increase the miss ratio in the last-level cache. At first, this event is detected by the LIMITLESS system monitor. Then, it sends notifications to the scheduler and FlexMPI, which starts the application-level monitoring for both applications. The columns T1, T2, and T3 represent the execution time when the applications run without sharing a node, when there are running in the same node with interference, and after the conflict has been solved respectively. The table also shows the difference in the execution time between *App 1* and *App 2*, where the first one doubles it while the second is unaffected. The reason is that it does not have a temporal data locality. In order to avoid this interference, *App 2* is migrated to another free node. After that, both applications run exclusively without interferences between them.

The column *Overhead* shows the reconfiguration overhead, which is the process in charge of migrating an application to another node (because of the interference). The migration task requires four operations, which are included in the *Overhead*. The first step consists of creating 8 processes in the node that are going to be shared. The second step is the destruction of the 8 processes allocated in the initial exclusive node. Once interference is detected, the third step is the creation of 8 processes in a new free node. Finally, the framework has to destroy the 8 processes allocated in the shared node. Note that the first two reconfigurations can be avoided if the framework has already stored the performance metrics of the applications (for instance, due to previous executions).

Table 7.7: Example of a workflow that generates interference. T1, T2, T3 are the execution time per 10 iterations. Overhead represents the overhead of the migration process measured in seconds.

<b>Id.</b>	<b>Code</b>	<b>Procs.</b>	<b>Size(Gb)</b>	<b>Shared</b>	<b>T1(s)</b>	<b>T2(s)</b>	<b>T3(s)</b>	<b>Overhead</b>	<b>Iterations</b>
1	$B_{ll}$	4	0.30	2	3.20	6.40	3.20	-	$1.5 * 10^4$
2	$F$	20	17.9	1	29.6	29.2	29.1	32.2	$6 * 10^2$
3	$B_{hl}$	4	0.30	4	2.80	2.80	2.80	-	$1.2 * 10^4$
4	$F$	20	17.9	3	29.5	29.6	29.5	-	$5 * 10^2$
5	$E$	20	1.20	6-7	6.90	9.00	7.00	-	$4 * 10^2$
6	$E$	20	1.20	5	7.00	8.78	7.60	2.80	$4 * 10^3$
7	$D_m$	20	1.20	5	0.03	0.03	0.03	-	$2 * 10^6$
8	$D_{xl}$	6	94.6	9	21.0	27.8	21.6	-	$5 * 10^2$
9	$D_{xl}$	16	111.8	8	9.40	9.60	9.40	101.1	$1.5 * 10^3$

The second use case involves applications 3 and 4, which are executed in a shared node without producing performance degradation (there is no interference between these applications). The cause is that there have a good data locality, and there is no necessity for reconfiguring.

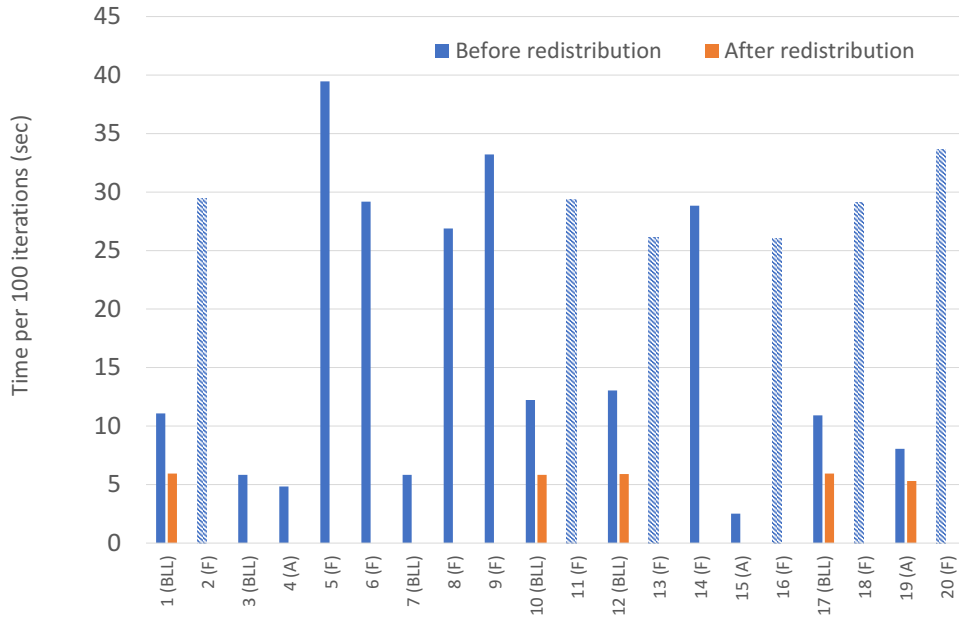


Figure 7.5: Scenario A - Coarse-grain scheduling evaluation. Each bar of each application shows the execution time per 100 iterations. Applications with striped bars are applications that create interference. Applications with two bars exhibit change in the execution time due to interferences.

In the third use case, there is, initially, communication interference between applications 5 and 6, and it affects both applications negatively. Due to this interference, application 6 is migrated to a free node allocated in a different rack, which increases the execution time  $T_3$  due to slower network bandwidth. If a process allocates a small amount of data the overhead is related to the processes of creation and destruction. Once application 6 has been moved, application 7 is executed in the same node as application 5. However, there is no interference between them because their profiles are different.

Finally, in the last use case, applications 8 and 9 generates interference between them because they consume more memory than the node has. When the interference is detected, application 9 is migrated to an exclusive node. Note that, in this case, the overhead is greater than the other use cases due to the superior amount of data used.

Figure 7.5 shows the performance evaluation for Scenario A using the coarse-grain scheduling policy. This scenario consists of 20 jobs, being each job an independent application, and they are executed as a workflow. The x-axis shows the name of the application and the y-axis represents the execution time per 100 iterations. Note that some applications, like 4(A) in Scenario A, may have a much smaller execution time per iteration than the others, but their impact on the overall time might be important due to executing a larger number of iterations. Detecting a hot spot during the execution is based

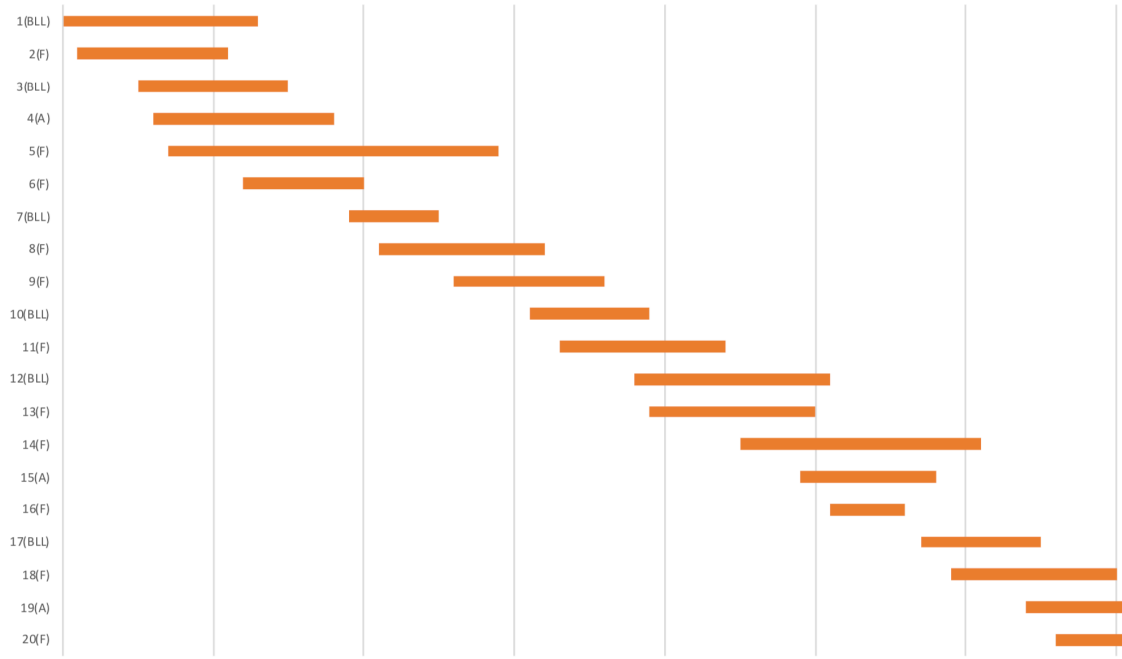


Figure 7.6: Scenario A - Gantt diagram of the execution when the nodes can be shared, but the interference detection is not active. The length of diagram corresponds to the makespan, with a value of 14.889 seconds.

on predefined thresholds. Those threshold values are: more than 90% of memory usage, a miss rate greater than 40% of global last-level cache, and a network bandwidth usage greater than 40%.

Scenario A is a medium-conflict workflow with jobs of classes  $A$ ,  $B_{LL}$  and  $F$ . Note that the first two applications will have a performance degradation due to  $F$ , and the performance of  $F$  is unaffected by interference. Figure 7.6 shows the Gantt diagram associated to this scenario. The point is that 6 conflicting applications produced hot-spots, that resulted in six cases of performance degradation. However, another interference case existed ( $7(B_{LL})$ ). Six performance-degraded cases were detected and avoided, except the  $7(B_{LL})$  case in which the degradation did not exceed the predefined threshold and no action was taken.

Figure 7.5 shows these use cases with two bars per application. The left bar represents the execution time when there is interference with other applications, taking into account that this time is to before redistribution. The left bar represents the execution time when there is interference with other applications, taking into account that this time is to before redistribution. The right bar represents the final execution time when the interference has been avoided. In the case of  $7(b_{LL})$ , this application does not improve the execution time. So that, the graphical result is the opposite: the initial execution time does not include the interference, and the final time is longer because it includes that interference. In the cases that the second bar is not shown (after redistributing the processes), it means that they do

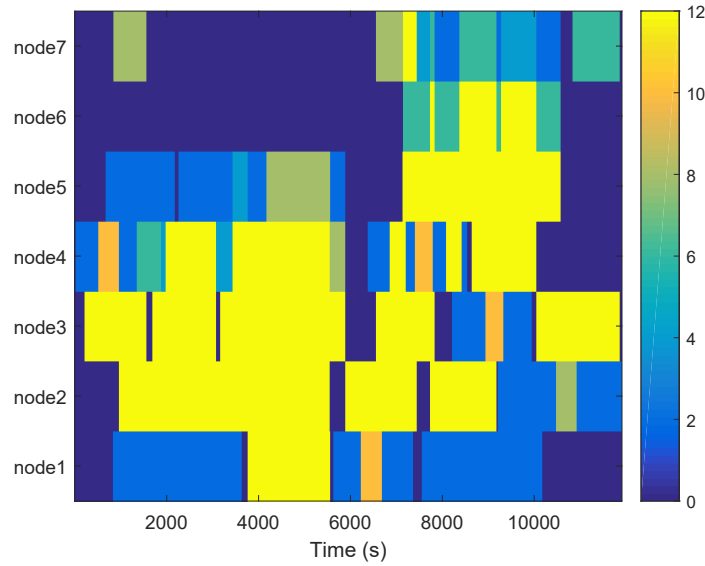


Figure 7.7: Scenario A - System load using the coarse-grain scheduling policy, including shared nodes and interference detection.

not exhibit performance degradation, and these bars are omitted for clarity.

The system identifies an interfering application like the one that, after starting its execution, reduces the performance of another application that is already running on the same shared node. These applications are represented by striped bars in Figure 7.5. During the execution, the framework (the cooperation between the system monitor, CLARISSE, and FlexMPI) detects and moves the interfering applications to other compute nodes. Note that, in the case of interfering applications, the execution time is the same after and before the migration process. That is why only one striped bar is shown. As it can be deduced, the migration process has a certain overhead related to the processes creation/destruction and data redistribution. Overall, the total overhead is of 17.8 seconds (taking into account the complete workflow execution) per process creation/destruction and 183.6 for data redistribution.

Before explaining the certain results of this scenario, it is important to know one assumption: actually, there are five compute nodes allocated to execute the workflow (nodes 1 to 5), but there are another two nodes (nodes 6 and 7) that are reserved to be used by CLARISSE. In there, only the applications that generate interference will be executed. Besides, node 8, which is not displayed, is used to execute processes of new applications. The objective is to obtain the initial performance metrics (which should not be affected by other applications). After collecting them, the application can be scheduled with a shared

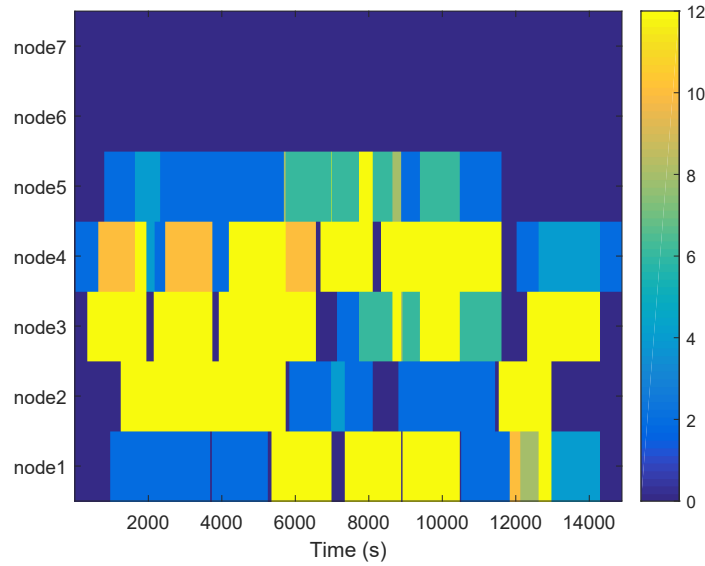


Figure 7.8: Scenario A - System load using shared nodes, but disabling CLARISSE interference detection.

policy.

Figure 7.7 shows a diagram with the CPU use of each compute node during the workflow execution, including the interference detection. This evaluation takes a short time and following this, these processes are migrated to a shared node. In total, the workflow makespan is 11,909 s (including overheads) and the total energy consumption (taking into account the eight compute nodes) is 6.9 MJ.

Figure 7.8 shows CPU use when CLARISSE is not used, and the conflicts are not avoided. In this case, the makespan is 14,889 s. Two main reasons explain why the makespan has been incremented in comparison with the previous strategy. At first, given that the conflicts are not avoided, the conflicting applications take longer to complete their executions, and then, nodes 6 and 7 are not used because they are reserved for CLARISSE. Note that there is a trade-off between the number of computational resources involved in the execution and the application execution time. For this scenario, the total energy consumption is 7.7 MJ. Despite using less computational resource, the increase in the conflicting application execution time produces a larger amount of energy consumption.

Finally, Figure 7.9 shows system use when each application is executed in exclusive nodes (without CLARISSE and without share nodes with other applications). Here the applications do not experience performance degradation due to the lack of conflicts. However, some computational resources may be underused. For example, nodes 1 and



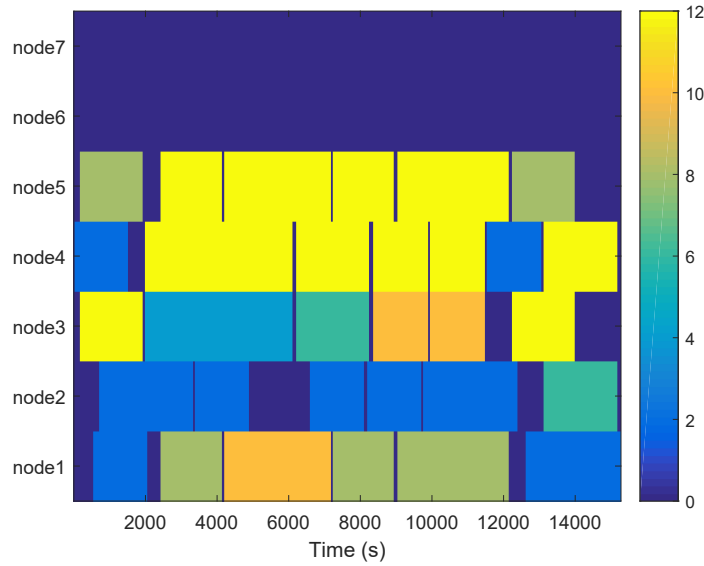


Figure 7.9: Scenario A - System load using a typical exclusive policy without shared nodes and interference detection.

2 have a reduce workload as some jobs only use two processes. Now, the makespan is 15,277 s and the energy consumption is 7.9 MJ. In this case, both measures are larger than the previous policies, which are based on sharing the compute-nodes.

The second scenario (*scenario B*) consists of another workflow of the previously described applications but using different input data. This configuration produces a scenario with more conflicts. The objective of this scenario is to test the *interference detection* and migration with a more intensive workflow. In this case, 9 applications produce interference (generating performance degradation) over the other 6 applications. Figure 7.11 shows the Gantt diagram associated to this scenario.

The combination of the monitor and the scheduler allows to detect the interferences between applications, mitigating their effects thanks to the migration of the conflicting applications. The migration overhead in this case is 31.3 s for process creation and 501.9 for data redistribution. For applications  $13(B_{LL})$  and  $16(B_{LL})$  there are several jobs that produce conflict with them: jobs  $14(DM)$ ,  $15(DM)$ ,  $17(DM)$  and  $20(F)$ . Due to the number of conflicts, the scheduler tries to migrate all of them but, because of their large number, the original applications ( $13(B_{LL})$  and  $16(B_{LL})$ ) cannot be executed without conflicts for a significant amount of time. It means that the applications will suffer a performance degradation because there are not enough *exclusive* resources to manage all conflicts (HPC clusters cannot be upgraded in execution time, however more resources could be requested

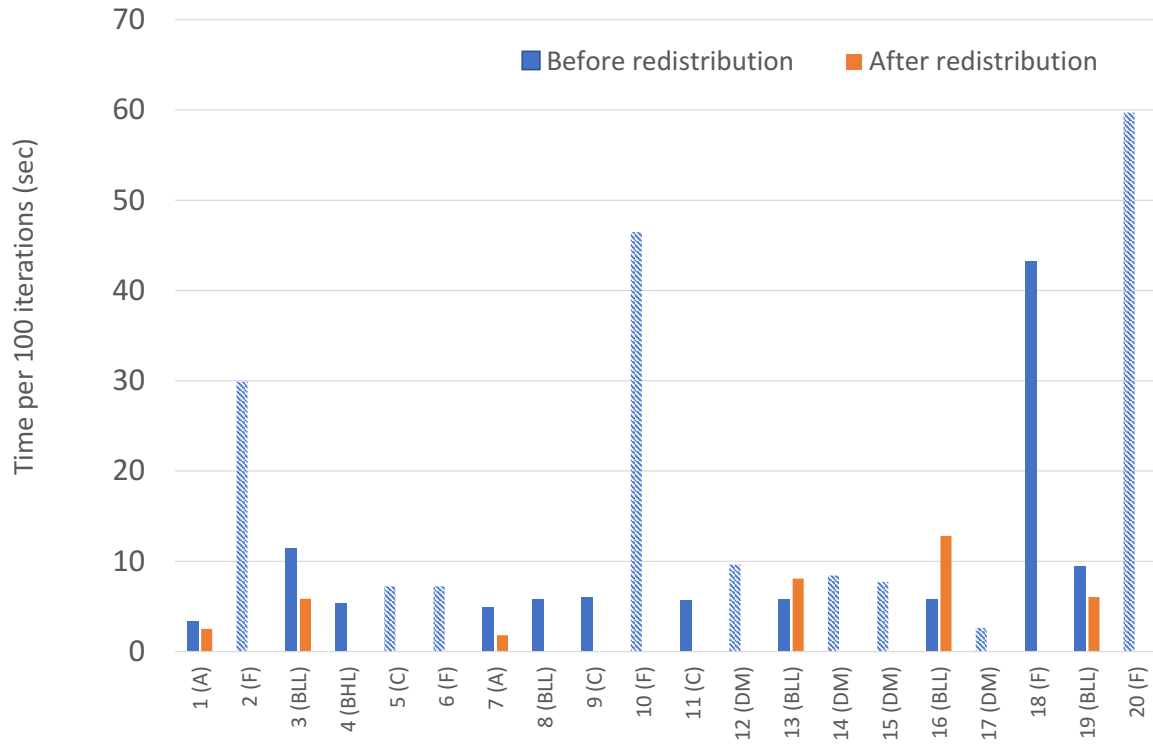


Figure 7.10: Scenario B - Coarse-grain scheduling evaluation. Each bar of each application shows the execution time per 100 iterations. Applications with striped bars with are applications that create interference. Applications with two bars exhibit change in the execution time due to interferences.

in cloud environments).

Despite that, the makespan and energy consumption of this scenario with the interference detection enabled is 10,759 s and 6.3 MJ. If the workflow is executed without the interference detection, the values are larger than before, producing a makespan of 13,256 s and an energy consumption of 6.9 MJ. Finally, executing the workflow exclusively, the makespan and the energy consumption grow up to 14,929 s and 7.4 MJ.

Table 7.8 shows a summary of the experiments in this section. For both scenarios, the global results about the makespan and the energy consumption are displayed. Also, the values are indicated depending on the scheduling policy, where *Policy 1* is based on the interference detection in shared nodes, *Policy 2* on using shared nodes without managing the interferences, and *Policy 3* on running all applications exclusively.

Taking these scenarios into account, the framework provides a more efficient execution in both cases, reducing the makespan and the energy consumption.

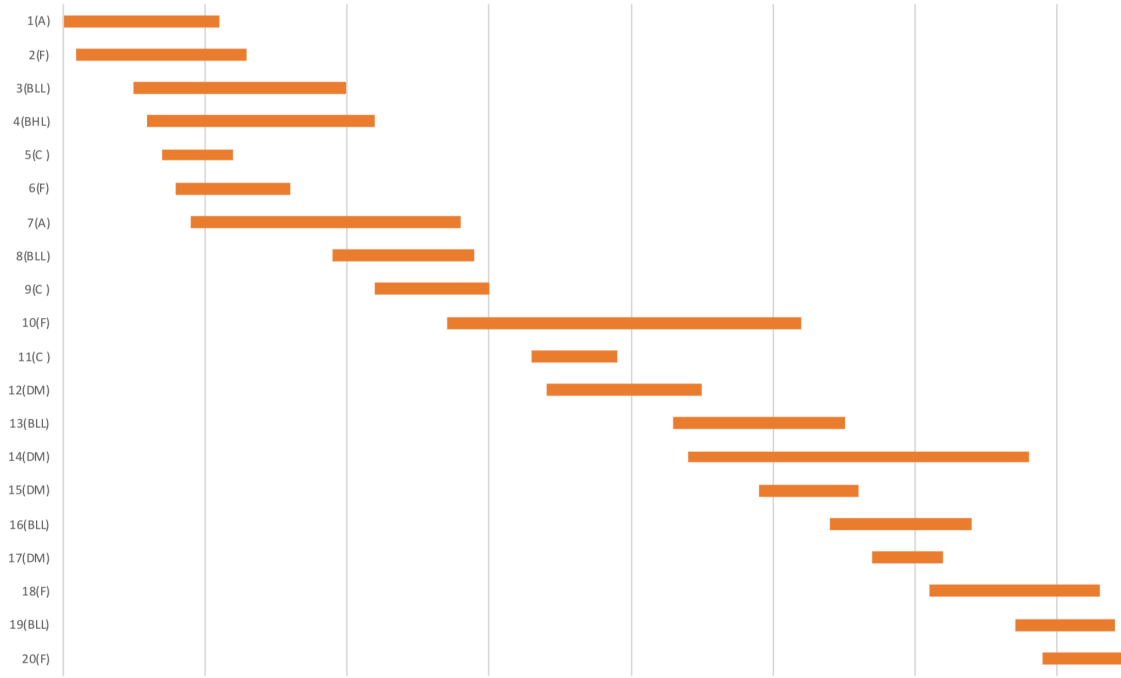


Figure 7.11: Scenario B - Gantt diagram of the execution when the nodes can be shared, but the interference detection is not active. The length of diagram corresponds to the makespan, with a value of 13.256 seconds.

Table 7.8: Summary - Comparison between results obtained in Scenario A and Scenario B with all of the policies.

Scenario	Policy 1		Policy 2		Policy 3	
	Makespan	Energy	Makespan	Energy	Makespan	Energy
Scenario A	11,909 s	6.9 MJ	14,889 s	7.7 MJ	15,277 s	7.9 MJ
Scenario B	10,759 s	6.3 MJ	13,256 s	6.9 MJ	14,929 s	7.4 MJ

### 7.3.2. Fine-grained scheduling based on monitoring

This subsection describes how LIMITLESS uses the data collected and the machine-learning classification features to support the feature *fine-grained scheduling*.

#### Platform and experiments description

For these experiments, the platform that has been used is a heterogeneous cluster with two racks. The first rack contains two nodes with Intel(R) Xeon(R) E5 with eight cores each and 256GB of RAM memory. On the other hand, the second rack contains six nodes with Intel(R) Xeon(R) E7, 128GB of RAM memory, and twelve cores per node.

As in the previous scheduling mode, the platform that has been used is a heterogeneous cluster with two racks. The first rack contains two nodes with Intel(R) Xeon(R) E5 with

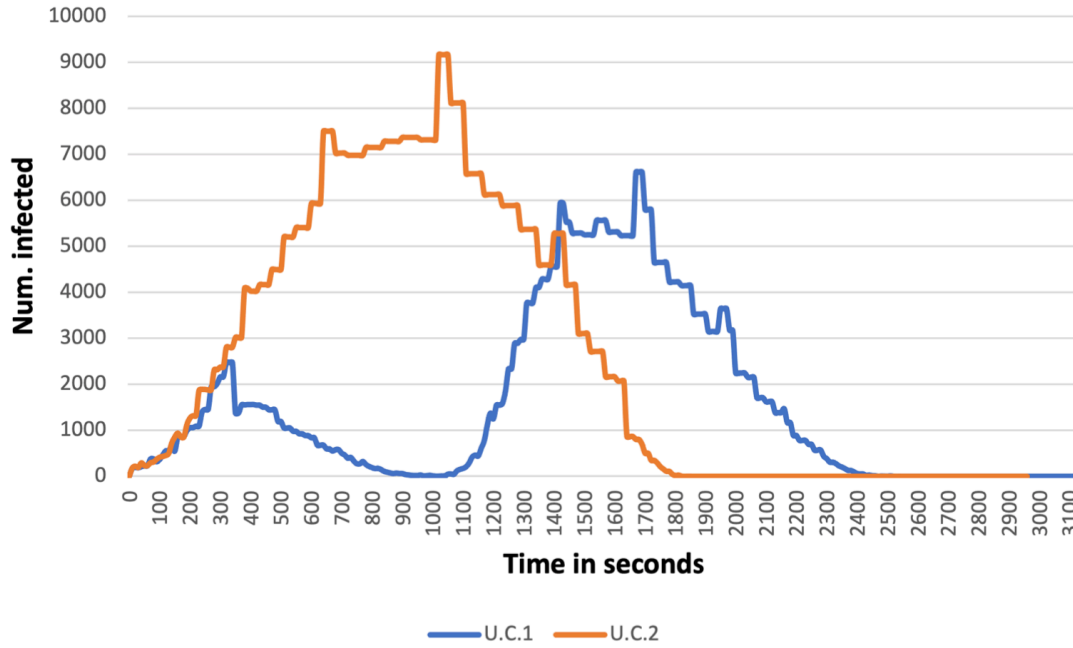


Figure 7.12: Distribution of the number of infections with COVID-19 during EpiGraph simulation for use cases 1 and 2.

eight cores each and 256GB of RAM memory. On the other hand, the second rack contains six nodes with Intel(R) Xeon(R) E7, 128GB of RAM memory, and twelve cores per node.

In this case, four use cases, which consist of combining the execution of EpiGraph, which executes 6 processes, with *N*-MG benchmarks that execute six processes, which are memory-intensive and performs long and short-distance communications, for 100 seconds. The difference between these use cases is the infection propagation: the use cases one and three simulates one COVID-19 infection wave for Spain in 2020, and the use cases two and four two COVID-19 infection waves (see Figure 7.12) for Spain in 2021. The figure represents the number of infections for each one wave using a small scenario of a city with 500,000 inhabitants. These represent the execution takes around 2950 seconds, and Figures 7.13 and 7.14 show the communication usage during each execution. This communication is expressed as KB per second (input and output), and it shows the communication pattern of each use case. Note that the communication pattern is strongly correlated with the infection curves: when the number of infections increases, EpiGraph becomes more cpu-intensive, and the communication intensity decreases. On the other hand, when there are less infections, the simulator performs less computations, and the communication phases are executed more frequently, increasing the communication intensity.

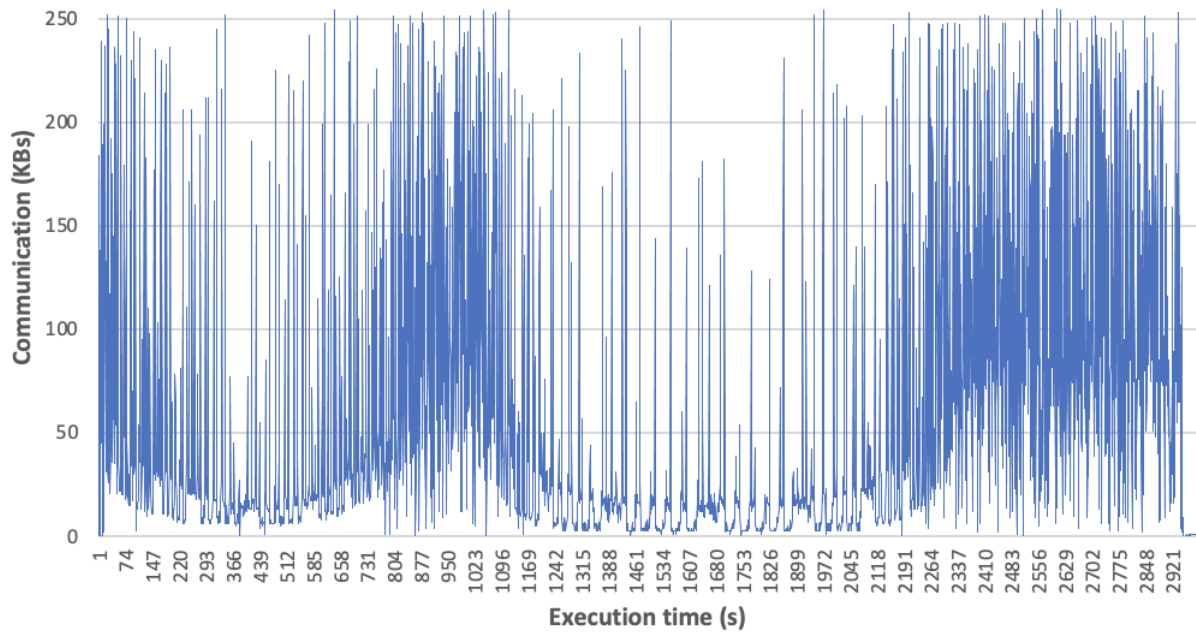


Figure 7.13: EpiGraph use case 1 - Communication during the execution which shows two waves.

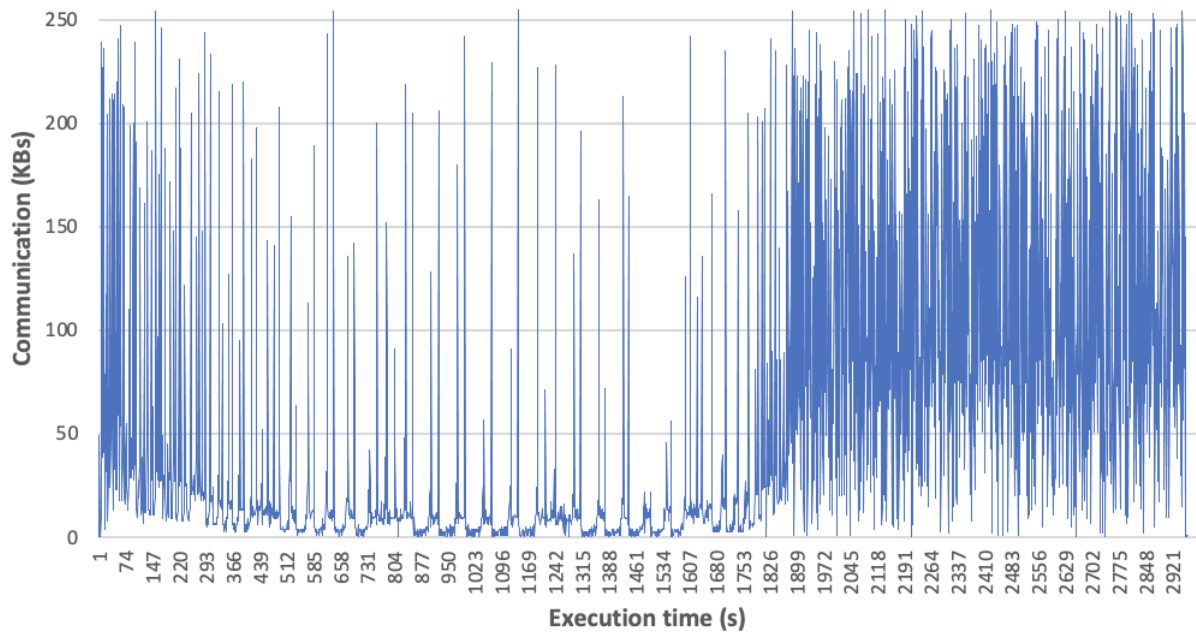


Figure 7.14: EpiGraph use case 2 - Communication during the execution which shows one wave.

The aims of this strategy are (1) to detect when EpiGraph is running in a certain node, and (2) detect the current execution phase. With this information, that can be acquired from the monitoring data and machine learning algorithms, the scheduler can run instances of MG when there is no interference with EpiGraph. Note that this profiling process is done offline when there is enough data about applications, and some classification algorithms used to identify phases and applications can be seen in 7.9, including the accuracy in our experiments of identifying applications. Besides, these tests have been performed in two nodes with twelve cores with *Hyperthreading*. We have decided to do not use this feature and limit the maximum number of processes (of EpiGraph and MGs) to 12 per node. This is why the combined executions CPU use is 50% instead of 100%.

## Results

Based on the profile of each application, EpiGraph and MG have interference when the first one increases its communications. The cause is that the first one makes more communications when the number of infected people decreases. However, MG is constant in CPU and communication intensities, and the interference becomes important when EpiGraph generates those peaks of network usage. This behaviour can be seen in Figure 7.15 shows how EpiGraph increases MG's iteration times when EpiGraph's iteration times are shorter (note that smaller iteration times for EpiGraph means larger communication intensities). Figure 7.16 shows the same but in a use case with one bigger wave. As it can be seen in these cases, both applications run concurrently generating an important degradation between them: the degradation is 32.5% for the first case, and 32.2% for the second. Detecting when EpiGraph performs low-intensity communication phases allows LIMITLESS to run MG tasks during those phases, using the information of the predictors. The scheduler will execute MG during the peaks of the infection propagation and EpiGraph will be exclusively executed during the periods with low COVID-19 incidence, that is when the communication frequency reaches the maximum values.

Executing these use cases in a two compute-nodes means that the applications will

Algorithm	60 s.	100 s.
AdaBoost	45%	75.6%
ANBC	31.1%	45.8%
BAG	34.5%	56.9%
MinDistance.	45.8%	80.7%
SVM	86.9%	100%

Table 7.9: Accuracy of the different classification algorithms using patterns of 60 and 100 seconds of EpiGraph.

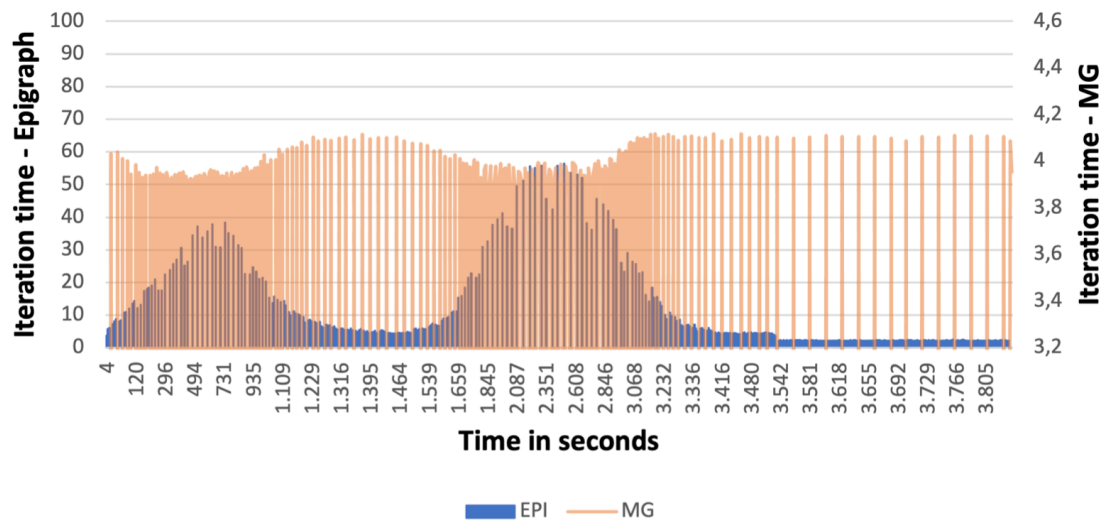


Figure 7.15: Use case 1 - Relation between EpiGraph and MG when they are executed in the same node at the same time (two waves).

run at the same time. However, the second compute node will run exclusively instances of MG (the number depends on the scheduling and the compatible phases). The first use case is executed with 14 MG instances as a workflow. The second one is executed with 12 MG instances as a workflow. Finally, the last two use cases are executed with 25 MG instances. As it can be seen in Figure 7.17 for use case 1, Figure 7.18 for use case 2, Figure 7.19 for use case 3 and Figure 7.20 for use case 4. Combining different executions of different applications, when there is no interference between them, decreases the use of the computational resources. In all the figures, the execution of MG instances can be noticed when the CPU of the machine increases from 25% to 50%. Note that figures 7.18 and 7.20 are similar because the node 1 does not change respect the second use case. The second node (not shown in the figure) runs the instances of MG exclusively when it is not possible to run them in the same node as EpiGraph. Note that the number of the instances executed in the second node depends on the current execution stage of EpiGraph (whether it creates interference or not) and the remaining MG instances that have to be executed. In the considered scenario these numbers (instances of MG executed in exclusively in the second compute node) are 9 for the first use case, zero for the second, 15 for the third, and 12 for the last one.

The combination of both applications produces between 2.1% and 2.4% of performance

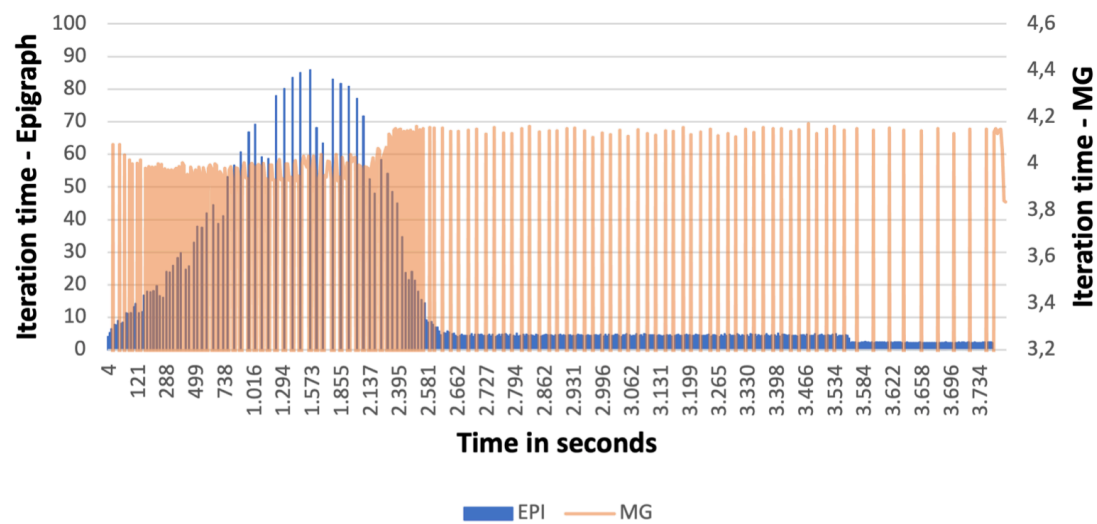


Figure 7.16: Use case 2 - Relation between EpiGraph and MG when they are executed in the same node at the same time (one wave).

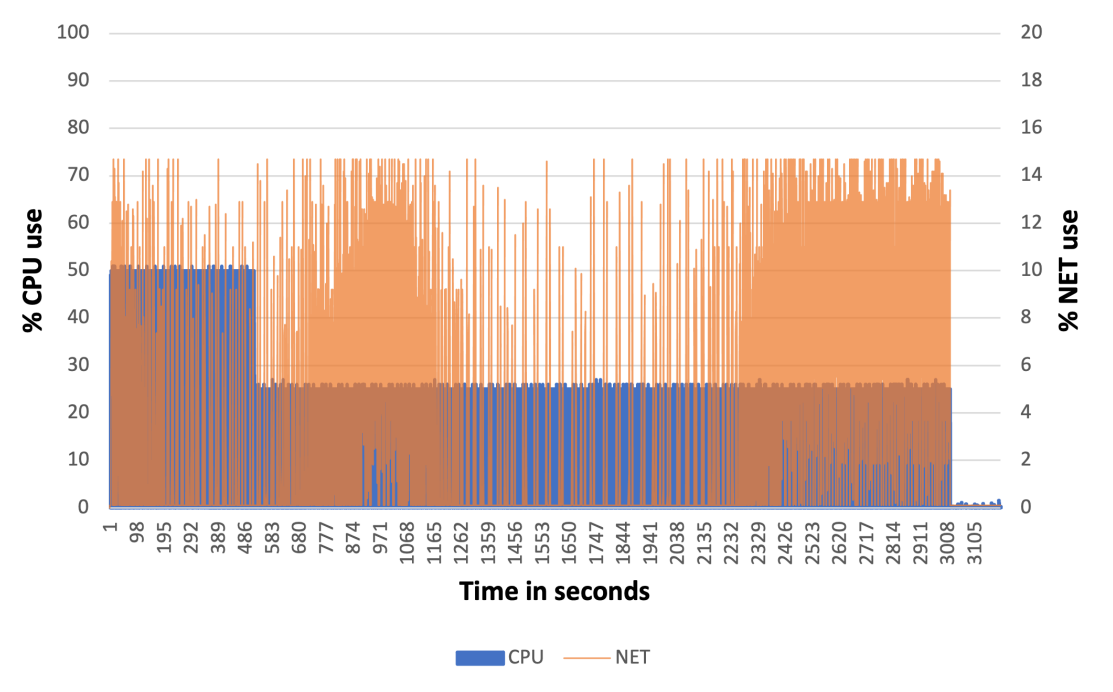


Figure 7.17: Use case 1 (Node 1) result combining EpiGraph and MG when LIMITLESS detects compatible phases between both applications.



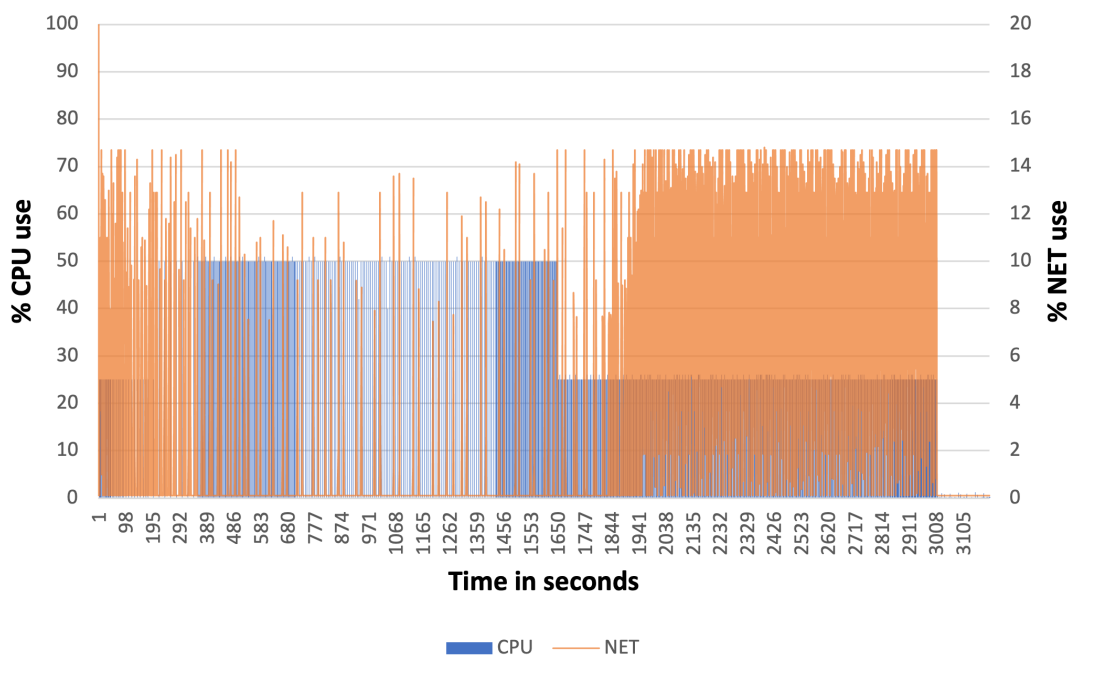


Figure 7.18: Use case 2 (Node 1) result combining EpiGraph and MG when LIMITLESS detects compatible phases between both applications.

degradation in the four use cases. It is caused by the overlap of MG with some CPU phases of EpiGraph due to the precision in identifying the execution phase. LIMITLESS uses a voting process between the machine learning classifiers to validate each phase identification. As this process is not 100% accurate, it is possible to obtain certain interference degree.

The results are summarized in Table 7.10 that shows the computational resources used on each use case, expressed as total CPU time used. The first column of the table shows the results when different exclusive nodes are used for EpiGraph and MG, i.e., each application are executed separately. Note that in this case not all the cores of each compute node are used. For instance, in the first use case, the accumulated CPU time (52,200 seconds) will be equal to 12 times the use time of the first compute node (used by EpiGraph) plus 12 times the use time of the second compute node (used by MG instances), where 12 is the number of cores in each node. When fine-grained scheduling is enabled, some MG instances are executed in the same node as EpiGraph (compute node 1) decreasing the utilization of the second node. This leads to a reduction in the accumulated CPU time to 46,200 seconds. As it can be seen, the *fine-grained* scheduling allows the system to take advantage of the unused resources existing in a node improving the system utilization.

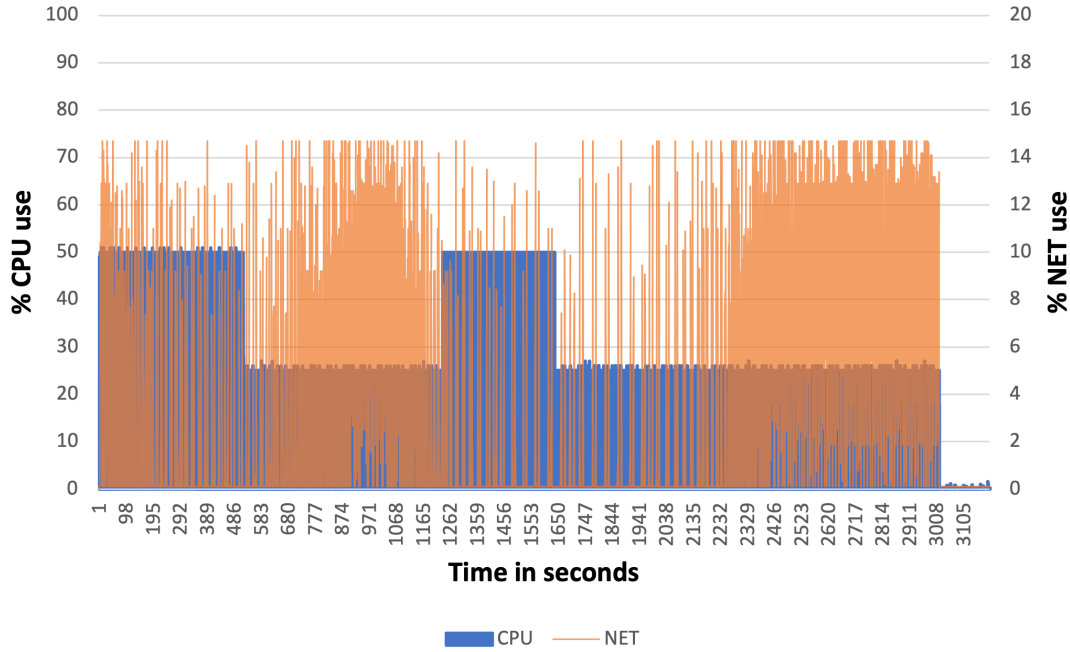


Figure 7.19: Use case 3 (Node 1) result combining EpiGraph and MG when LIMITLESS detects compatible phases between both applications.

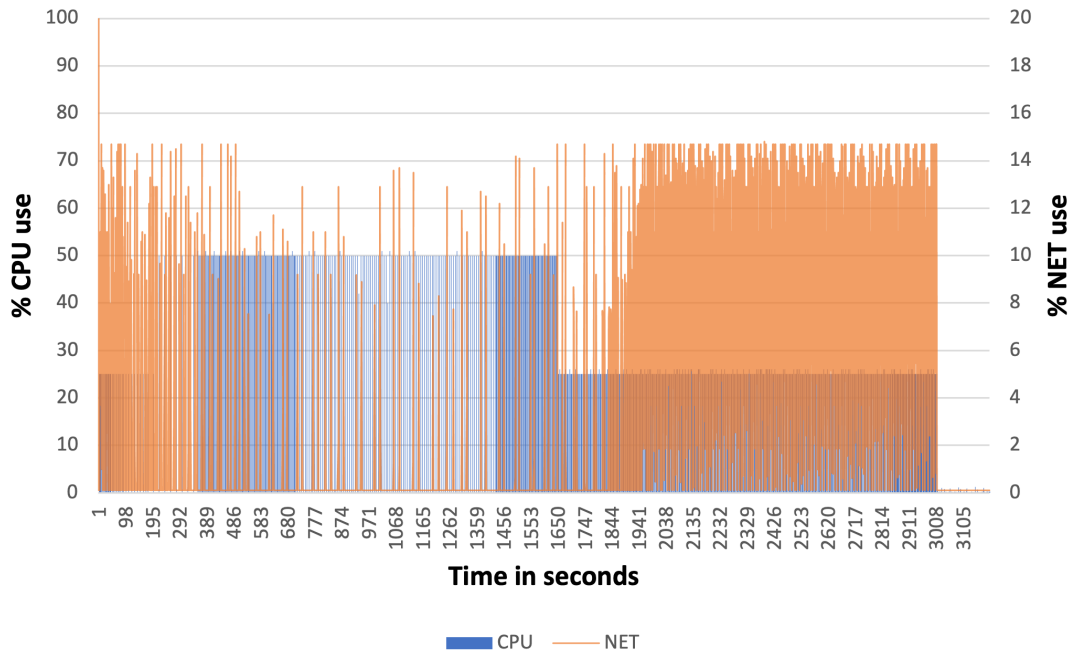


Figure 7.20: Use case 4 (Node 1) result combining EpiGraph and MG when LIMITLESS detects compatible phases between both applications.

#### 7.4. Optimizations

The main goal of these optimizations consist of reducing the network traffic due to monitoring communications.

### 7.4.1. In-node analysis optimization

This section shows a quantitative analysis of the optimization *in-node analysis* to try to reduce the communication between the LDMs and the LDAs. Following, an evaluation of the effect of this optimization, focused on reducing network usage, is shown. In order to test the in-node analysis under different conditions, a synthetic benchmark has been designed and deployed in two nodes. One node is in charge of executing the LDS process, while the second is in charge of executing the LDM. The benchmark consists of a workflow that executes three computation phases: the first ten minutes the node is in idle state, the second ten minutes the node executes a computation algorithm that produces a constant CPU load around 75%, and the last ten minutes, another computation algorithm is performed, but including *sleep* instructions periodically. The CPU load is displayed in Figure 7.21.

Figure 7.22 shows the network traffic related to the monitor when in-node analysis is enabled as well as when in-node analysis is disabled. The results of the experiment show that the in-node analysis reduces the network traffic dramatically (up to 90% in phases 1 and 2). Even with variable load (phase 3) almost 50% of the monitor traffic can be reduced. Note that there is a trade-off between the tolerance value and the amount of information that the server receives. In our experiments we found that tolerance values between 5% and 10% provides accurate measurements with important reductions in network use.

For a deeper analysis, in Figure 7.23 the CPU load of a single compute node for a 10% tolerance is represented for a period of 24 hours. In the same way, Figure 7.25 shows the same information but for main memory usage. During this period, two workflows were running in the system.

Figure 7.27 represents the difference between the metrics obtained with 10% tolerance and without tolerance. Note that these values are the error produced by using the tolerance threshold. As it can be seen, the biggest error obtained in some samples is 10% but on average is 0.27% - and the total percentage of metrics with an error is 5.6%. In terms of network traffic, the use of tolerance drastically decreases the number of packets sent from

U.C.	CPU time (s) Exclusive nodes	CPU time (s) Fine-grained scheduling	Saved resources %
1	52200	46200	11.5
2	49800	35400	28.9
3	65400	53400	19.4
4	65400	49800	23.9

Table 7.10: Execution summary of the use cases. Accumulated CPU time for each use case and scheduling.

the monitors to the aggregator. In this example, this number decreases by more than 87% (from 5314 packets to 680).

Performing different tests under different conditions (both real and simulated) shows that a tolerance value between 5 and 10 % provides a good relationship in the accuracy-traffic reduction trade-off.

Figures 7.24 and 7.26 show the results of applying this optimization graphically, and they can be directly compared with the original performance graphs in Figures 7.23 and 7.25 respectively.

#### 7.4.2. In-transit analysis optimization

As it has been said before, one of the main limitations of scalability is communication. When systems try to gather and centralize the information, the network becomes a bottleneck due to the amount of data. This section describes the optimization that reduces de communication between the LDAs and the LDSs, based on performance prediction. It also includes a practical evaluation that shows the overall accuracy of the different prediction alternatives, as well as the reduction in the network usage by the monitor.

The evaluation of this optimization focuses on the LIMITLESS Analytic (LAN), which can be seen in Figure 3.1. It is in charge of performing the smart functions to predict the performance of the executing applications.

The evaluation has been done using simulation and a real platform. This real cluster is a heterogeneous cluster with two racks. The first rack contains two nodes with Intel(R) Xeon(R) E5 with eight cores each and 256GB of RAM memory. On the other hand, the

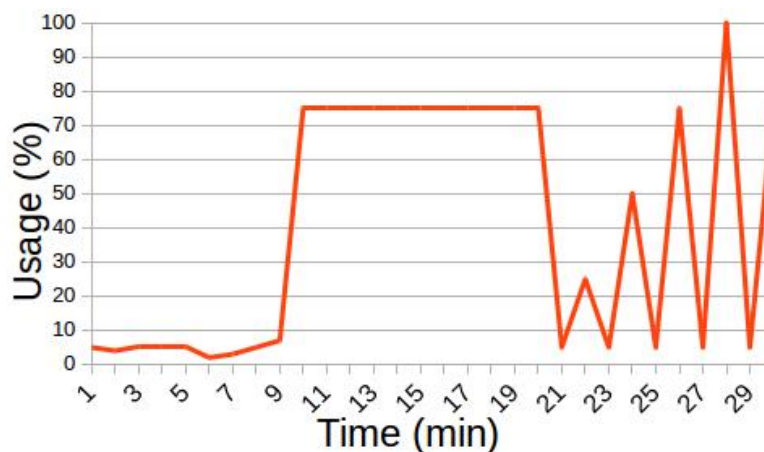


Figure 7.21: Synthetic benchmark with three different CPU phases to evaluate the in-node analysis impact.

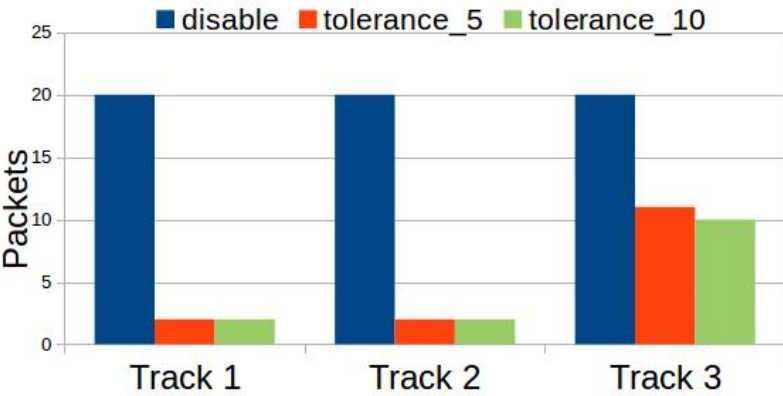


Figure 7.22: A comparison between the number of monitoring packets sent with and without in-node analysis optimization, including two tolerance values.

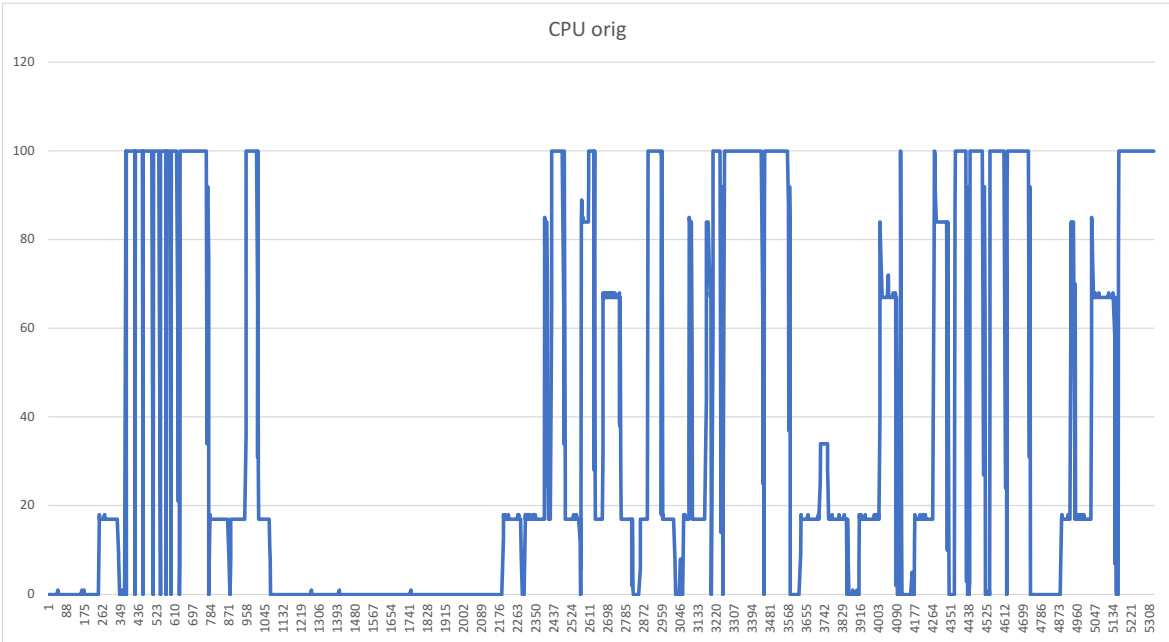


Figure 7.23: In-node analysis A - CPU monitored during 24h

second rack contains six nodes with Intel(R) Xeon(R) E7, 128GB of RAM memory, and twelve cores per node. The connection between nodes is made through a 10 Gbps Ethernet. The I/O is based on Gluster parallel file system.

Application Modeling and comparison TOP command vs. LIMITLESS

One interesting feature of a monitor is the ability to detect applications. Thanks to the communication with the scheduler, LIMITLESS is able to detect applications that are running exclusively in a node, which means that the performance metrics collected for that node will be produced by that application. As it has been demonstrated before, the

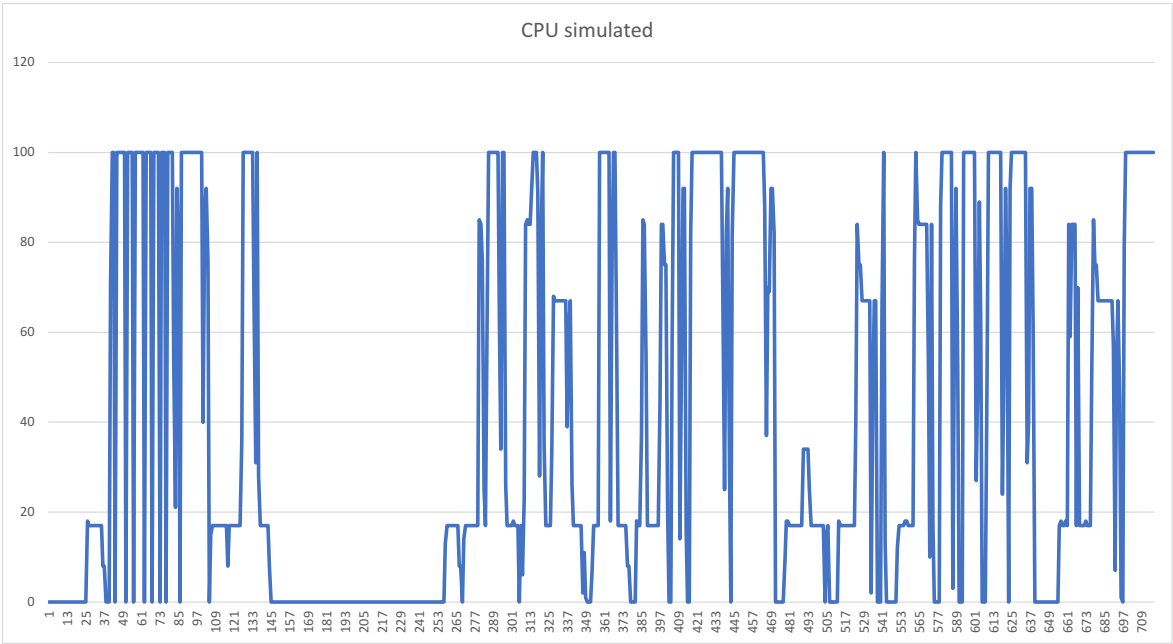


Figure 7.24: In-node analysis A - CPU monitored using 10% tolerance.

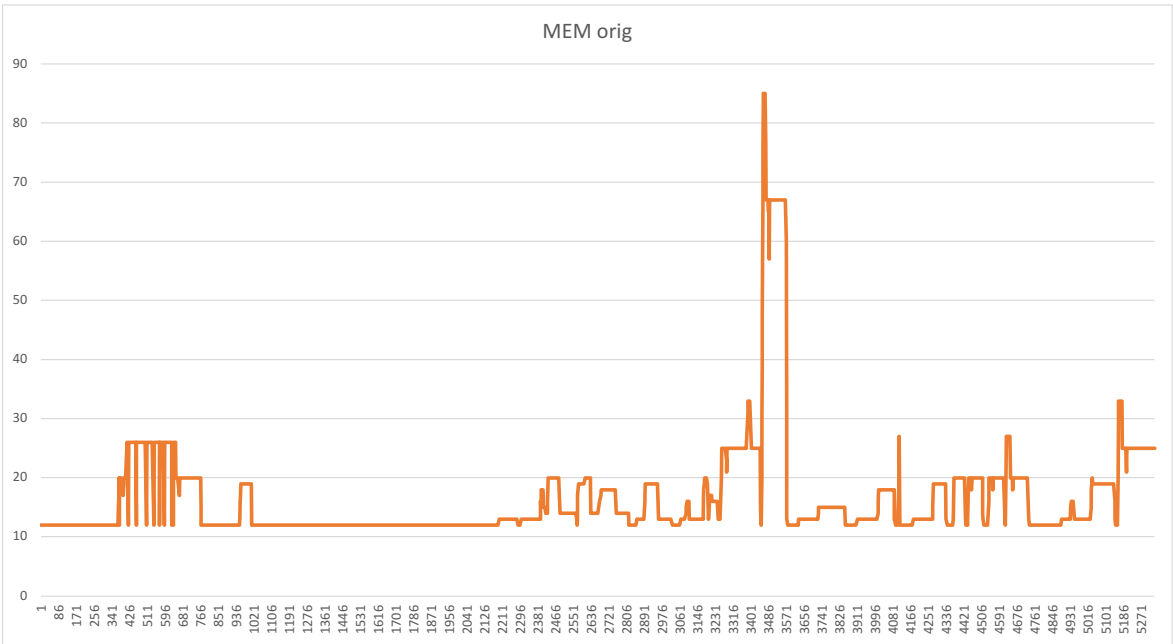


Figure 7.25: In-node analysis B - Memory monitored during 24h

performance counters collected by LIMITLESS are so similar to the metrics offered by Collectd, but in this case, a comparison has been done with the top command. The objective is to show the results, including another comparison between two methods of obtaining performance data. So that, this subsection will focus on the application modeling and the results obtained combining their executions instead of explaining differences between collection methods.

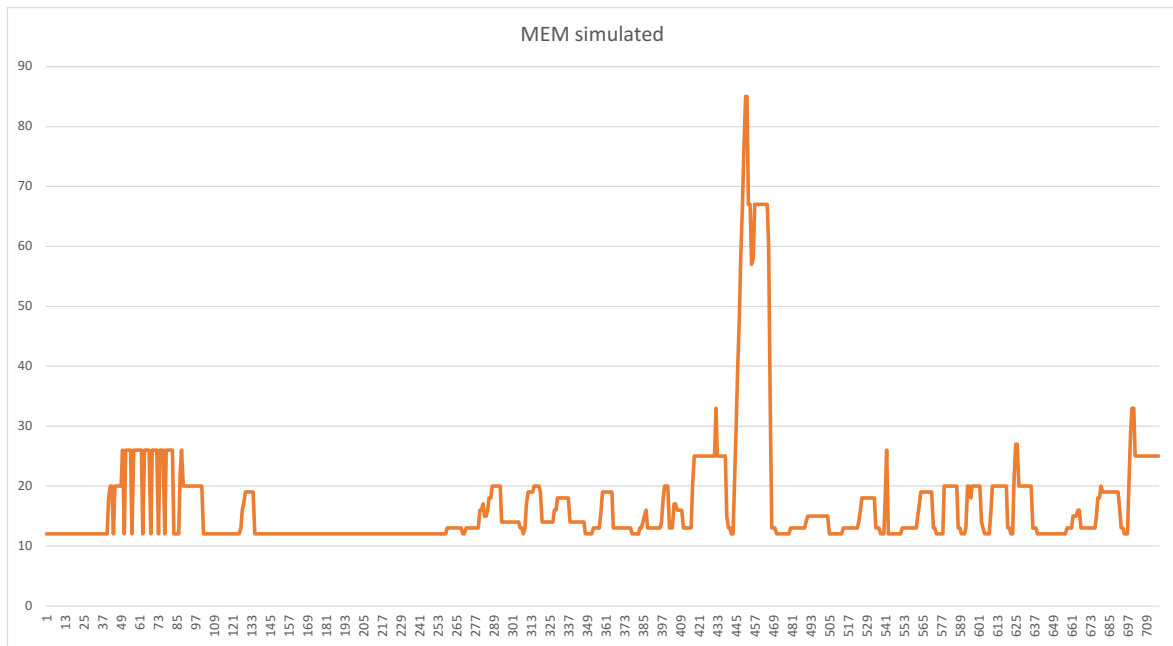


Figure 7.26: In-node analysis B - Memory monitored using 10% tolerance.

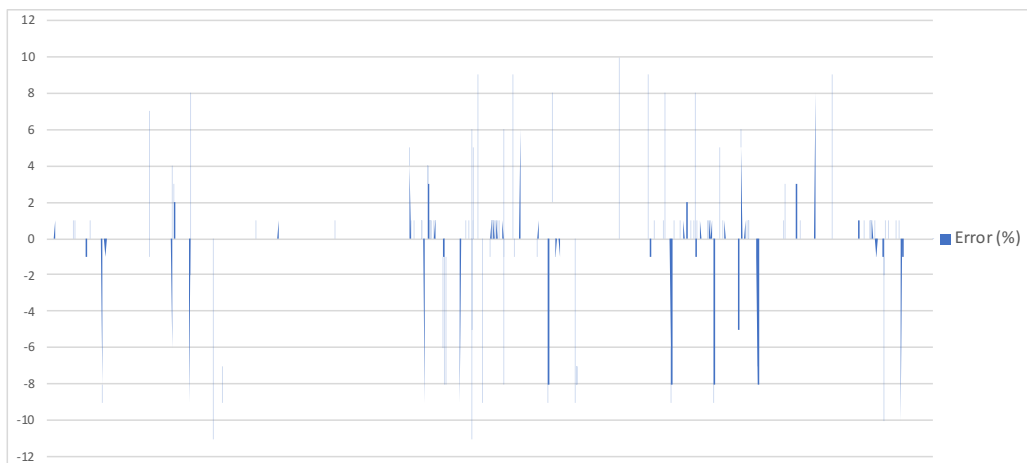


Figure 7.27: In-node analysis A - Error as difference of CPU performance with/without tolerance, setting this value un 10% (original figure in Figure 7.24).

The following results show how LIMITLESS creates profiles from the applications when they have been run in exclusive nodes. The applications modeled are described in Table 7.11. Note that all applications have been executed running 6 processes. The objective of this configuration is to show the individual performance of each application, and then, different combinations between them to measure the consumed resources (the total resources used should be the combination of both individual models) in compute-nodes with 12 cores.

The Jacobi method is, in numerical analysis, an iterative method used to solve systems of linear equations of the type  $Ax = b$ . The algorithm is based on finding approximate

solutions from the diagonal matrix and iterating until it converges.

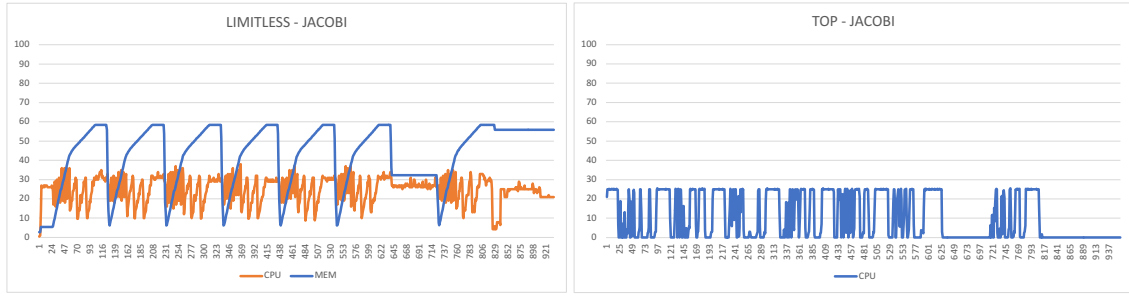
Application	Description
<b>Jacobi</b>	Iterative algorithm for determining the solutions of linear equations. It combines CPU and I/O phases during its execution. Developed using MPI.
<b>SP</b>	Scalar Penta-diagonal solver. CPU intensive. Developed using OpenMP.
<b>IS</b>	Integer Sort. It performs random memory access with constant use of CPU and memory. Developed using MPI.
<b>BT-IO</b>	It is a test of different parallel I/O techniques.
<b>Epigraph</b>	It is an efficient graph-based algorithm for designing vaccine antigens. It performs CPU and communication phases, depending on the number of the infected every iteration. Developed using MPI.

Table 7.11: Applications modeled by LIMITLESS.

The following figures show the different models for each application. Note that these models correspond to one execution and certain parameters (for example the class of the problem, which sets the size of the matrices or the number of iterations of each application).

The first use case corresponds to the execution of the Jacobi algorithm. It is a CPU-intensive application with a combination of memory and IO operations in MPI. The model generated thanks to the performance counters collected can be seen in Figure 7.28, which includes two figures inside it. At first, Figure 7.28i is the graphical result obtained by LIMITLESS. On the other side, Figure 7.28ii represents the CPU performance based on the information retrieved by the `top` command. Measuring CPU with high variations `top` is less precise than LIMITLESS because it computes the performance counters less time (note that `top` displays information about many processes instead one, and in this experiments the interval time is set to one second). So that, in many cases the displayed information corresponds to max and min values. As `top` does not obtain buffered and shared memory information, Figure 7.29 has been obtained using the tool *Free*: it provides information about the total amount of the physical and swap memory, as well as the free and used memory. With this tool, Figure 7.28i can be validated in combination with the CPU returned by `top`.





(i) Model generated by LIMITLESS from the original application Jacobi. Each series represents the percentage of use.

(ii) Model generated by top counters from the original application Jacobi. The series represents the CPU usage as a percentage.

Figure 7.28: LIMITLESS model generation versus top counters. Jacobi algorithm use case.

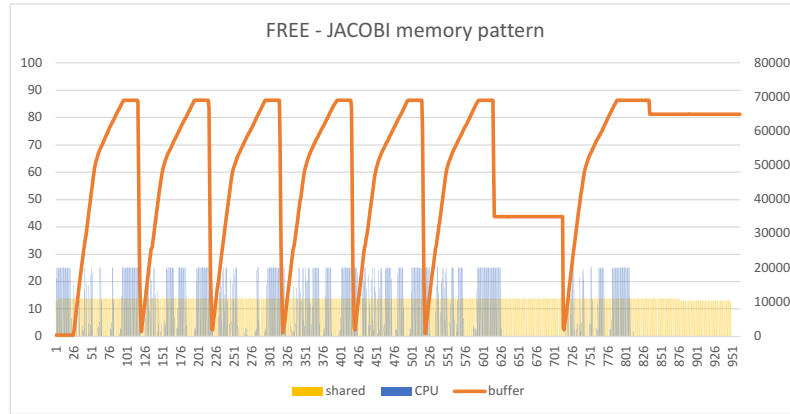
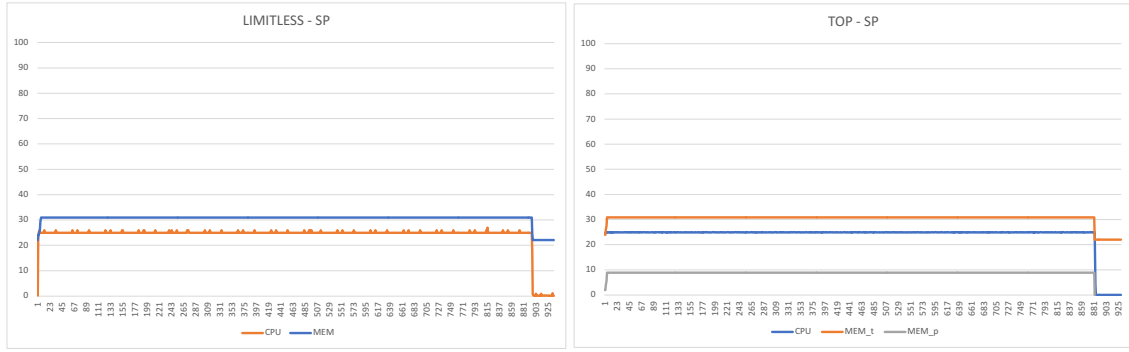


Figure 7.29: Free - Jacobi memory pattern. The total memory consumption is represented as a percentage of use.

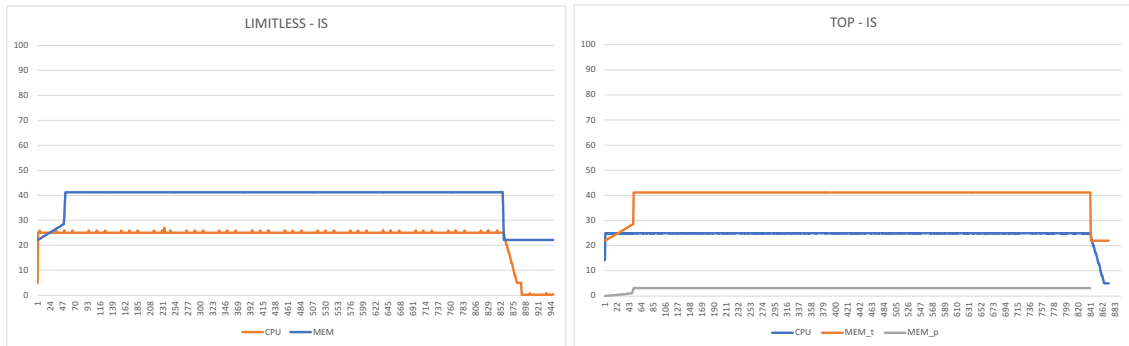
The second use case corresponds to the execution of the Scalar Penta-diagonal solver. It is a CPU-intensive application with a high usage of main memory due to big matrixes storage. The model generated thanks to the performance counter collected can be seen in Figure 7.30, which includes two figures inside it. At first, Figure 7.30i is the graphical result obtained by LIMITLESS. On the other side, Figure 7.30ii represents the same information based on the information retrieved by the top command, but another series called  $mem_p$  appears.  $mem_t$  is equivalent to  $mem$  in LIMITLESS and represents the total memory consumed: real and virtual. However,  $mem_p$  is the value returned by top, which is the resident memory used. Adding manually the virtual memory, the result is  $mem_t$ .



(i) Model generated by LIMITLESS from the original application SP. Each series represents the percentage of use. (ii) Model generated by top counters from the original application SP. Each series represents the percentage of use.

Figure 7.30: LIMITLESS model generation versus top counters. Scalar Penta-diagonal solver use case.

The third use case corresponds to the execution of the MPI distributed Integer-Sort algorithm. It is a CPU-intensive application with a high usage of main memory and random memory accesses. The model generated thanks to the performance counter collected can be seen in Figure 7.31, which includes two figures inside it. At first, Figure 7.31i is the graphical result obtained by LIMITLESS. On the other side, Figure 7.31ii represents the same information based on the information retrieved by the top command. As in the last use case, another series called  $mem_p$  appears, but taking into account the virtual memory consumed and the dynamic buffers, the total memory used can be seen in  $mem_t$  series.



(i) Model generated by LIMITLESS from the original application IS. Each series represents the percentage of use. (ii) Model generated by top counters from the original application IS. Each series represents the percentage of use.

Figure 7.31: LIMITLESS model generation versus top counters. Integer Sort use case.

The next use case corresponds to the execution of the Block Tri-diagonal solver algorithm, performing different parallel I/O operations. It is a CPU-intensive application due to matrix operations, but then it performs a lot of I/O operations. The model generated

thanks to the performance counter collected can be seen in Figure 7.32, which includes two subfigures inside it. At first, Figure 7.32i is the graphical result obtained by LIMITLESS. On the other side, Figure 7.32ii represents the same information based on the information retrieved by the top command. Besides, top does not provide I/O information, but LIMITLESS does. The result can be seen in Figure 7.33. This figure shows how BTIO performs different I/O operations and following different access patterns. The result is the percentage of writes during the sampling interval.

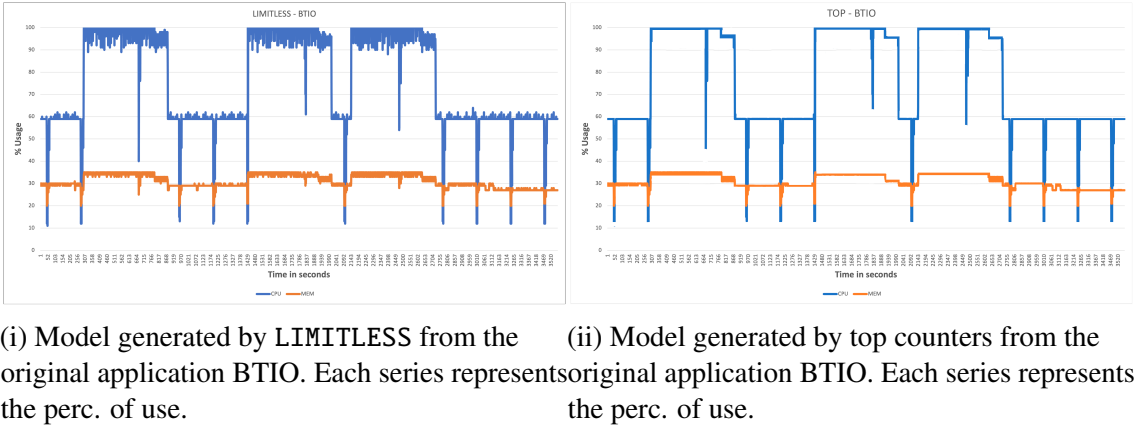


Figure 7.32: LIMITLESS model generation versus top counters. BT-IO use case.

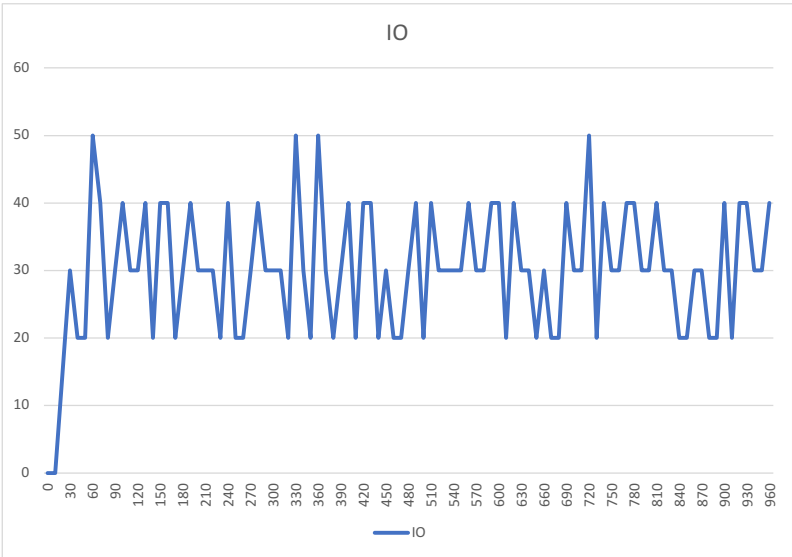


Figure 7.33: LIMITLESS - BTIO read/write operations over time. The Y-axis represents the percentage of read/writes with respect to the total.

The last use case corresponds to the execution of the Epigraph, a scalable, fully

distributed simulator that performs large scale and realistic stochastic simulations of the propagation of some viruses (for example, COVID-19 [114]). The model generated thanks to the performance counter collected can be seen in Figure 7.34, which includes two subfigures. At first, Figure 7.34i is the graphical result obtained by LIMITLESS. On the other side, Figure 7.34ii represents the same information based on the information retrieved by the top command. In this case, top does not provide communication information, but LIMITLESS does. The result can be seen in Figure 7.35. This figure shows the communication pattern during the execution of the application. It is directly related to the number of people infected.

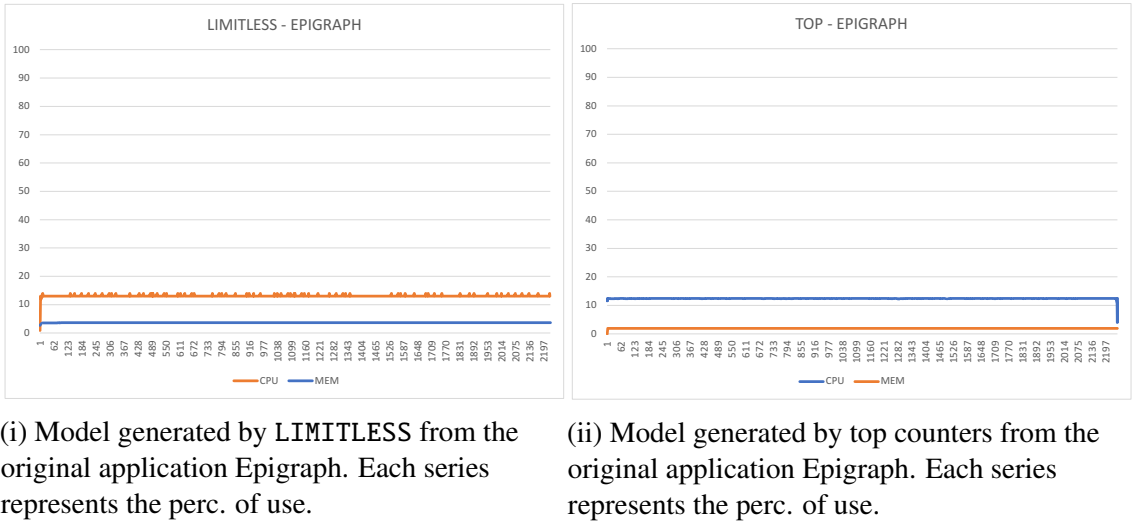


Figure 7.34: LIMITLESS model generation versus top counters. Epigraph use case.

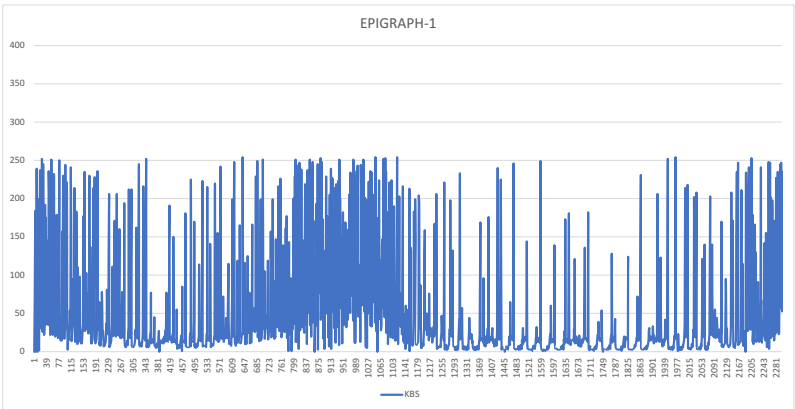


Figure 7.35: LIMITLESS - Epigraph communication usage. The Y-axis represents the communication usage in Kbps.

## Performance prediction

This subsection focuses on describing the algorithms included in LIMITLESS to make predictions about the performance of the applications. It also provides a deep evaluation that shows the results of the different algorithms and confirms the benefit of using them, not only for implementing optimizations but for designing new scheduling policies based on predictions. The LA is the component in charge of performing these predictions, leveraging the large amount of data collected. There are two objectives for developing these algorithms: the first one is to optimize the communications between the LDAs and the LDSs (*in-transit analysis*), and the second is to improve the scheduling process by designing a new scheduling policy (*fine-grain scheduling policy*).

To reach those goals, this section evaluates four different algorithms, three of them based on uni-variable analysis, and one based on multivariable analysis. The proposed algorithms based on analyzing each variable separately are *application pattern matching*, *prediction based on a historical window*, and *neural networks*. Finally, the algorithm proposed to make multivariable predictions is based on machine learning. All of them have been described in Chapter 5.

Both neural networks and machine learning techniques need a big amount of information to train correctly, and usually, this process may require a considerable amount of time. Because of that, in the case of the neural network, there is a lag between the application executions and the possibility to use the predictions generated. However, in the case of the machine learning algorithm the delay is much smaller and assumable, and it can be used since the beginning (but the accuracy increases with more execution examples). The training process in machine learning needs to spend few seconds, even with an important number of input monitoring data. For instance, we have performed a test with an application ensemble of four days, collecting metrics each second, and generating a 28MB log, and the process was done in less than 5 seconds. The same process with machine learning takes more than 50 seconds.

The next two figures show the performance pattern for the Jacobi and BT-IO applications. The objective is to show the different performance metrics when the applications are executed in exclusive nodes. Figure 7.36 shows the main performance metrics of the Jacobi application. In this case, the application has been executed using four processes. Figure 7.37 shows the same performance metrics of BT-IO, but using eight processes. Note that the objective of these figures is to show the profile of each application; they are not the use cases.

For the experimentation, three different use cases have been considered. The first

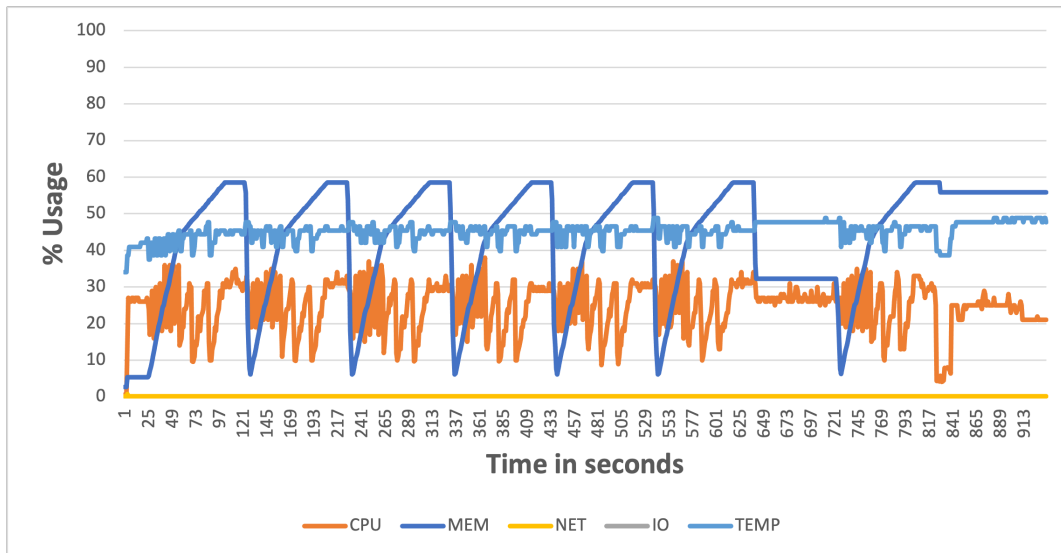


Figure 7.36: Performance metrics for Jacobi method executed in an exclusive node.

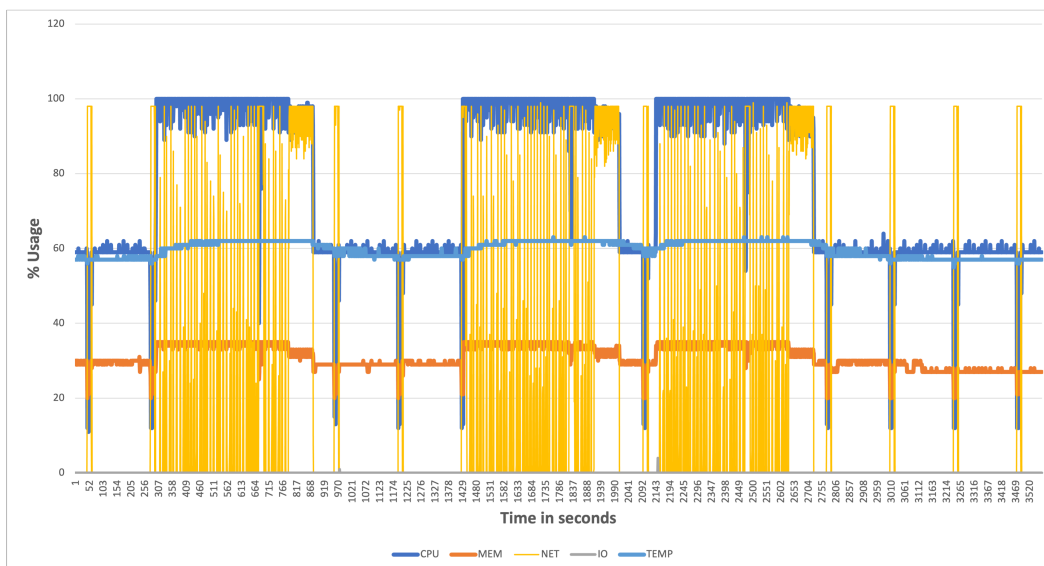


Figure 7.37: Performance metrics for BT-IO benchmark from NAS Parallel Benchmarks.

one corresponds to the parallel implementation of a Jacobi algorithm, which has been already described before. This use case executes eight processes using an input matrix with 15,000 entries. As the application executes its processes exclusively in one compute node (most of the time), each phase of its execution is approximately constant during all the execution and between different executions. Figure 7.38 shows the percentage of CPU, which is consumed during the execution of this use case. The instants where there is a CPU reduction corresponds to the I/O phases, which are executed periodically. As it can be seen, there is an important increase in CPU around the second 25,000, which is related to a synchronization process.

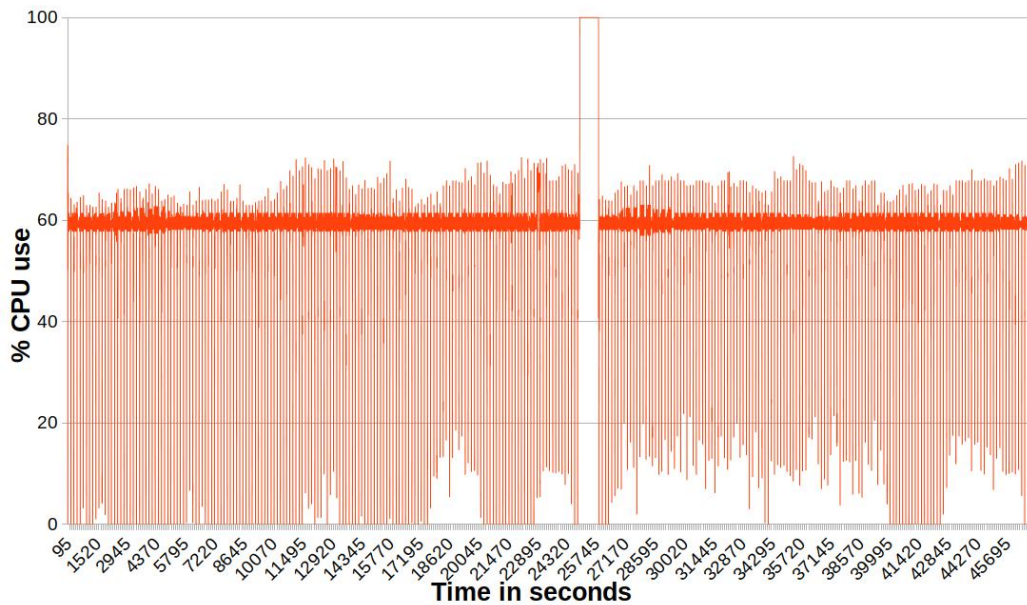


Figure 7.38: Use case 1 - CPU use of Jacobi method executed in an exclusive node.

The second use case consists of executing simultaneously two instances of the Jacobi method in the same nodes. Due to this, the CPU and communication phases keep barely constant. However, the I/O phases change their duration because both applications are executing I/O operations at the same time. That is the reason why there is interference between both applications: they share the I/O bandwidth, reducing their access (read and write) speed, and producing an increased phase duration. Figure 7.39 shows the CPU use when two Jacobi instances are running in the same compute node. Each application executes 6 processes, and the CPU consumption is nearly 100%. As it can be seen, there is not a constant distance between the CPU reductions (I/O phases) due to the interference.

The third use case corresponds to BT-IO from NAS Parallel Benchmarks configured as class C, which means that is 12X complex than the standard BT-IO case. This benchmark alternates CPU and I/O phases. Figure 7.40 shows the CPU usage for this benchmark.

The following results show only CPU information, but the framework processes the complete vector of metrics. Note that there are two kinds of metrics depending on the variability of their values: high variability and low variability values. For instance, CPU belongs to the first group due to its volatility. However, temperature belongs to the second group because it is more constant and typically has limits. The reason why the evaluation shows only CPU results is that it belongs to the group that is most difficult to predict. It shows more clearly the proposal benefits.

On the other hand, as each execution generates different values, including executions with same parameters, produce exact predictions is really hard. For this reason, LIMITLESS

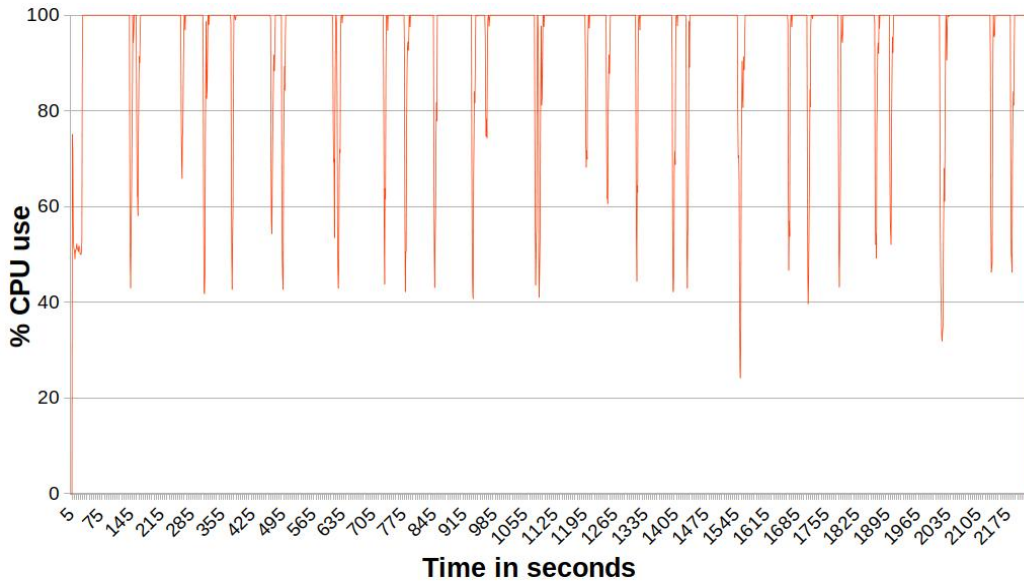


Figure 7.39: Use case 2 - CPU use of two Jacobi instances executing in the same node. There is I/O interference between them.

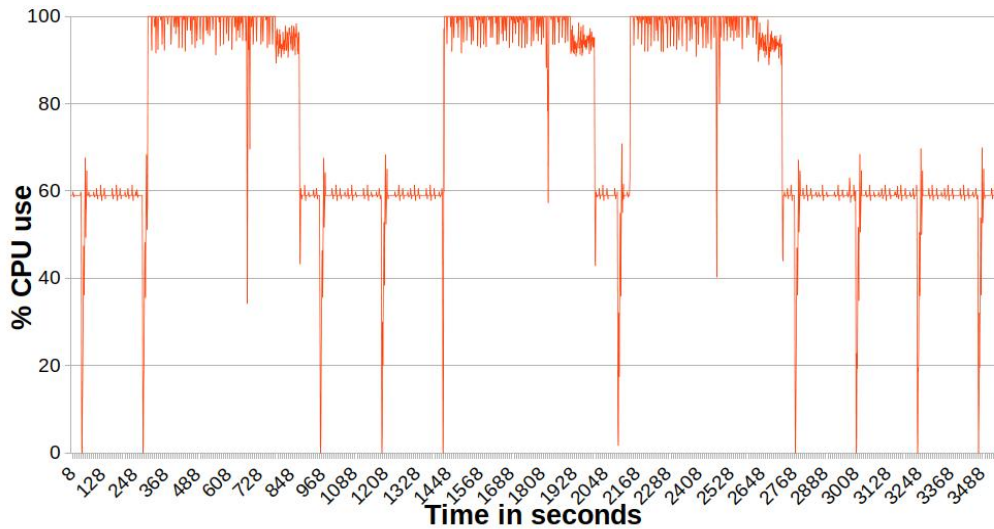


Figure 7.40: Use case 3 - CPU use of BTIO benchmark from NAS Parallel Benchmarks.

considers good values those that are within a predefined range.

Table 7.12 shows the accuracy of the predictions based on the performance metrics that LIMITLESS has collected from the execution of the applications. The accuracy corresponds to the percentage of predictions that have the same value as expected, including an acceptance margin of 3% in each algorithm. Comparing the use cases one and two (one instance of Jacobi executed in an exclusive node and two instances of Jacobi executing in the same node), the first one has a steadier behavior, producing more accurate predictions for this case. In the optimization *in-transit analysis*, the traffic reduction between LDAs



and LDSs is directly related to the precision of these predictors. Each correct prediction allows the LDA to avoid sending one monitoring packet to the LDS.

Note that the key of *in-transit analysis* consists of sharing the same predictor (created and trained by LA) between LDAs and LDSs. In terms of transmission overhead, due to the predictors sharing between different components, the first two algorithms (pattern matching and historical window) have no input parameters, thus, there is no transmission of data. However, neural networks and machine learning algorithms need the data related to their configuration (for instance, the weight of each neuron in the neural network). In these cases, there is an overhead of 28KB for sharing the neural network-related configuration, and 23KB for the machine learning algorithm.

Table 7.13 shows the accuracy of the predictions based on the performance metrics that CLARISSE and FlexMPI have collected from the applications (which are directly stored in ElasticSearch). In this case, there was not possible to collect performance information from the BTIO application because it is written in Fortran. Note that using these runtimes brings a more accurate representation of the application that enhances the modeling. They use timestamps for collecting the duration of the different phases (CPU and I/O). This monitorization improves the accuracy of the predictions, as can be seen in Table 7.13.

	<b>UC 1: Jacobi excl.</b>	<b>UC 2: Jacobi interf.</b>	<b>UC 3: BTIO</b>
<b>Pattern matching</b>	25.2%	25.2%	24.7%
<b>Historical window</b>	61.5%	82.7%	50.0%
<b>Neural networks</b>	99.5%	93.3%	98.0%
<b>Machine Learning</b>	90.8%	90.5%	88.5%

Table 7.12: Predictors accuracy at node level monitoring - Accuracy expressed as a percentage of correct predictions.

	<b>UC 1: Jacobi excl.</b>	<b>UC 2: Jacobi interf.</b>
<b>Pattern matching</b>	48.5%	53.2%
<b>Historical window</b>	98.6%	98.7%
<b>Neural networks</b>	99.3%	99.5%
<b>Machine Learning</b>	98.7%	98.8%

Table 7.13: Predictors accuracy at application-level monitoring - Accuracy expressed as a percentage of correct predictions.

All these algorithms have been included in the optimization process called *in-transit analysis*, having an impact on the number of monitoring packets sent from LDAs to LDSs.

In table 7.14 can be seen the global traffic reduction when these prediction algorithms are in use. Note that those values include the prediction of four performance metrics: CPU, memory, I/O, and communication. As it can be seen, the final results exhibit almost the same results as the experiments. However, the addition of more metrics to predict reduces the accuracy due to the accumulated variability.

	<b>Use case 1: Jacobi excl.</b>	<b>Use case 2: Jacobi interf.</b>	<b>Use case 3: btio</b>
<b>Pattern matching</b>	24%	23%	24%
<b>Historical window</b>	60%	80%	50%
<b>Neuronal networks</b>	96%	92%	96%
<b>Machine Learning</b>	90%	90%	88%

Table 7.14: Percentage of network traffic saved of all the prediction algorithms, including CPU, IO, Memory and Network collected metrics.

## 7.5. Improving control congestion for InfiniBand networks

This section shows that LIMITLESS in combination with congestion control mechanisms is able to dynamically configure the InfiniBand congestion control (CC) efficiently. Following, the platform and the experiments performed using the GPCNeT benchmark are described. Note that this work is the result of a cooperation between the author of this PhD. thesis and his supervisors, and a group of researchers at Universidad de Castilla-La Mancha (UCLM).

### 7.5.1. Platform and experiments description

The cluster is composed of 50 server nodes HP Proliant DL120 Gen9, with processors Intel Xeon E5-2630v3 with 8 cores at 1.80 GHz and 16 GB of RAM each. The network connection is built with IBA-based hardware: 50 Mellanox IS5022 switches with 8 ports with QDR technology. Each link offers 32 Gbps (instead of 40Gbps) due to the encoding protocol. Finally, the connection between nodes corresponds to a KNS topology using the IBA-based hardware, which can be seen in Figure 7.41.

This topology exhibits 12 switches (in blue) and 36 switches that operate as routers (in orange). This has been done to perform the re-routing functionality required by KNS. Routers are labeled with “R” with the  $X$ -coordinate and the  $Y$ -coordinate. End-nodes are labeled as “H” followed by the coordinates of each one. Finally, switches are labeled as “Sw” followed by the number of the dimension  $D$  where that switch is connected and the position of that switch  $N$ .

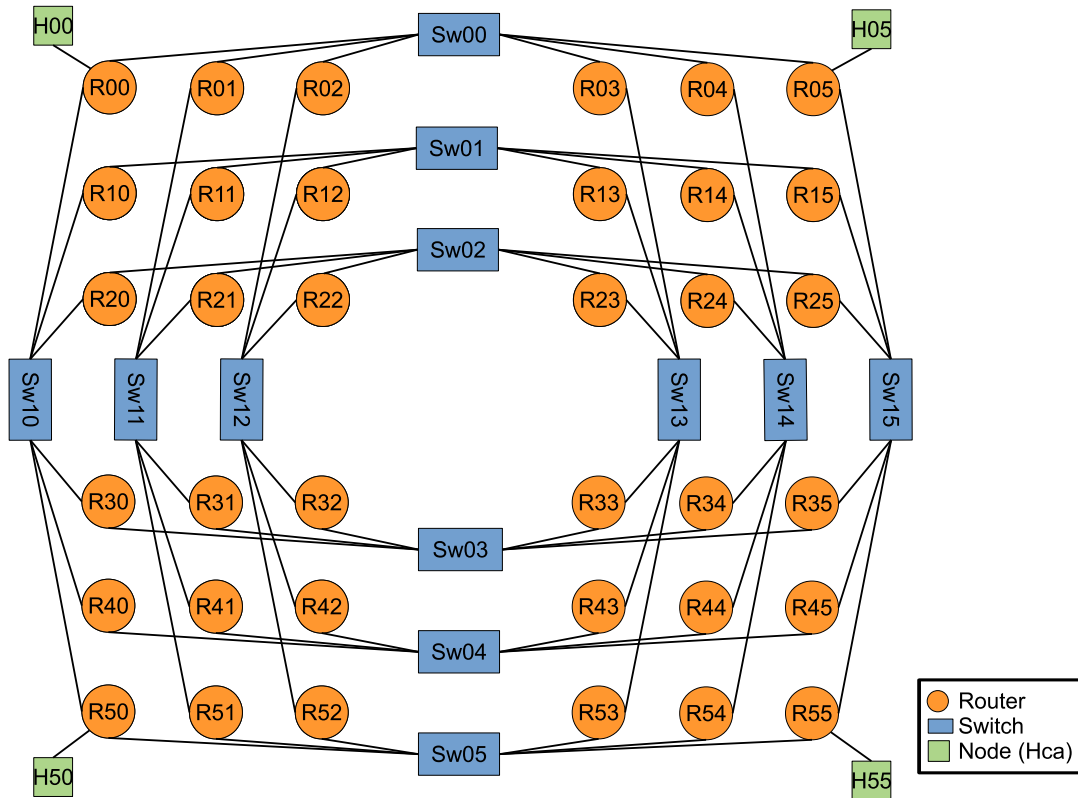


Figure 7.41: 36-node KNS topology built on the cluster.

The experimentation has been done using the GPCNeT benchmark [115], which is used to measure the impact of congestion in HPC systems. GPCNeT is a benchmark based on MPI that runs among all the nodes, but, in this case, it has been modified to set manually which nodes will provide results that do not depend on the random distribution between the different tests.

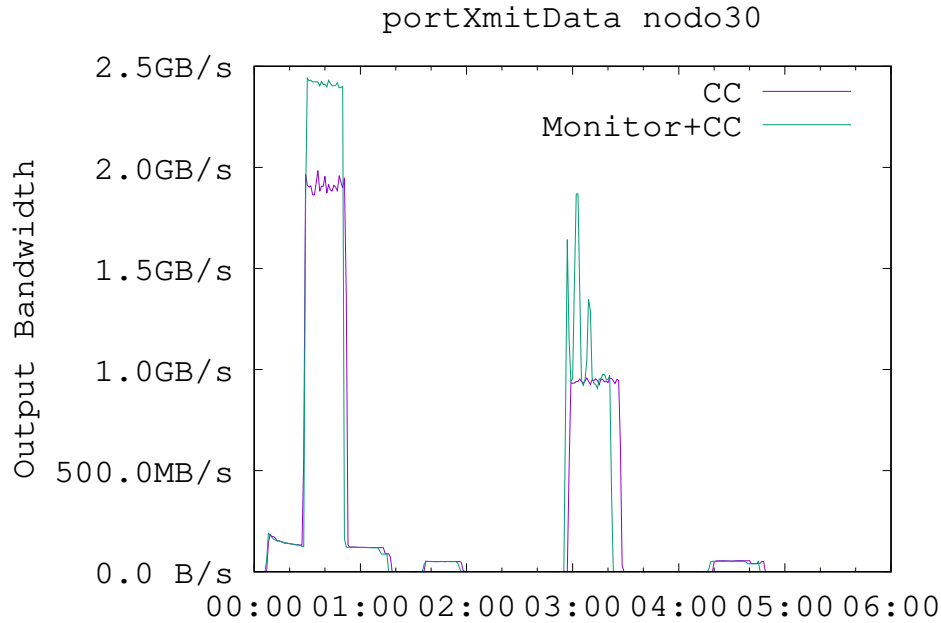
For each test in the evaluation, GPCNeT processes the mean and 99th percentile in MiB/s per MPI rank. Initially, the results are collected without congestion, and then, the congestor nodes inject traffic. GPCNeT will calculate the congestion, comparing the results obtained with the results obtained without congestion. In this case, the collected metrics provided by LIMITLESS have been studied to understand the congestion using metrics such as *portXmitData* and *portXmitWait*.

### 7.5.2. Results

The experiment execution consists of 8 MPI ranks per node in the 36 nodes in the KNS topology, and the tests have been executed using different values for *CCTI\_increase* because the objective is to observe the impact of this factor on the CC performance and efficiency. The most relevant parameters of the IBA-CC mechanism can be seen in Table 7.15.

Table 7.15: Congestion control configuration.

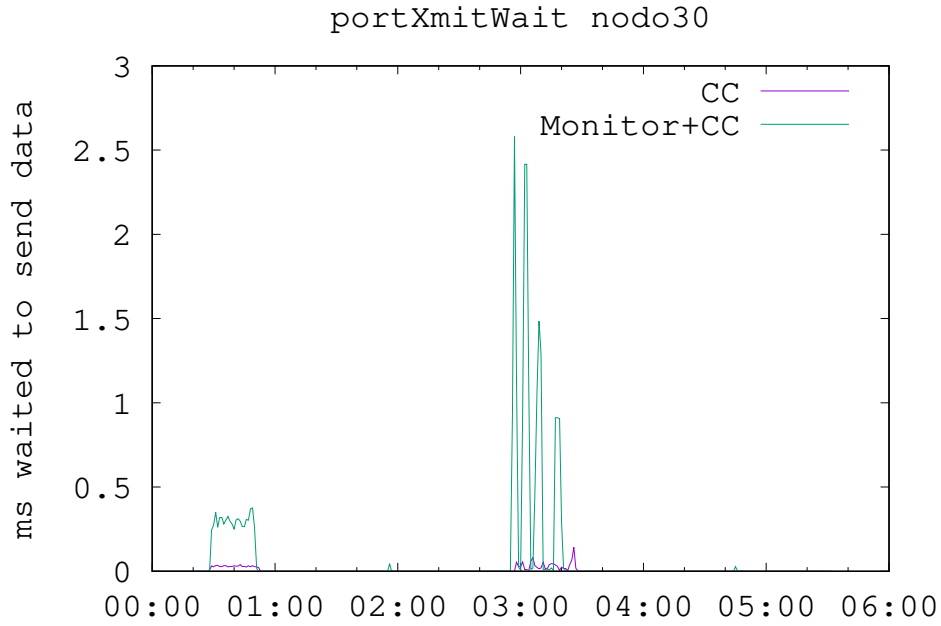
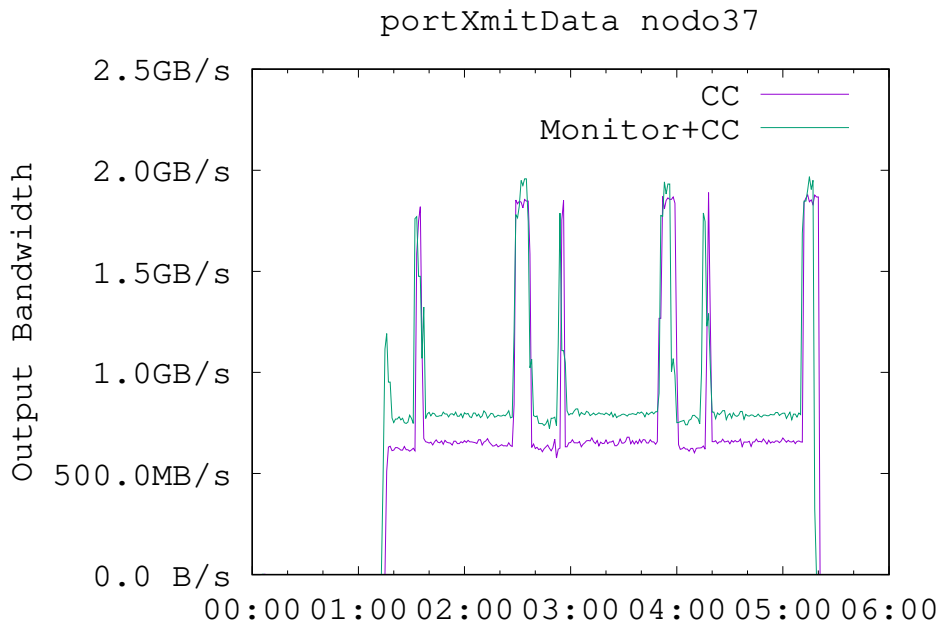
Parameter	Value
<i>Threshold</i>	0x7
<i>Packet_Size</i>	1
<i>Marking_Rate</i>	0x000f
<i>CCTI_Timer</i>	150
<i>CCTI_Increase</i>	{1,2,5,10,20}

Figure 7.42: *portXmitData* in node 30.

Following Figures 7.42, 7.43, 7.44 and 7.45 show the metrics collected by LIMITLESS system monitor in comparison with the time for the *PortXmitData* and *PortXmitData* performance counters during GPCNeT benchmark execution. As it can be seen in the figures, congestion starts around 1:20 from the beginning. The policies used in the experiments are: IBA-CC exclusively (CC) and LIMITLESS (LIMITLESS + CC). The IBA-CC *CCTI\_Increase* parameter is set to 5, which is an intermediate value in the category passive-aggressive of throttle degree at end-nodes.

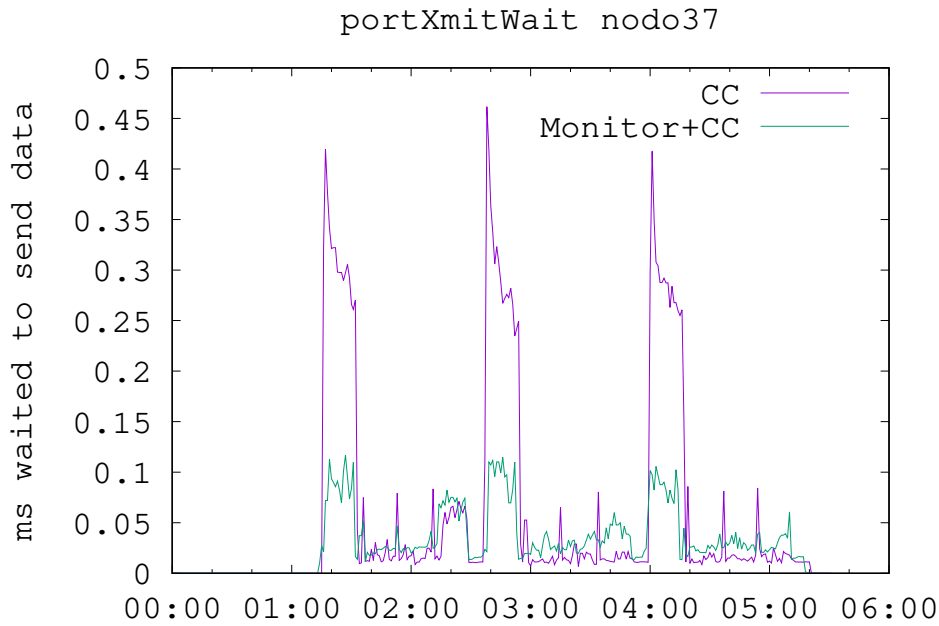
As it has been said, the study focuses on the *PortXmitData* and *PortXmitWait* performance counters. The first one shows an approximation of the bandwidth reached by each end-node, while the second measures the number of cycles that a node cannot send data because of the congestion. These values are used to estimate the congestion impact at this end-node.

Figure 7.42 shows the bandwidth results for the *PortXmitData* performance counter in node 30, when the IBA-CC and LIMITLESS+CC configurations are used. Since approximately 1:20, as it can be seen, when we use the LIMITLESS configuration the bandwidth

Figure 7.43: *portXmitWait* in node 30.Figure 7.44: *portXmitData* in node 37.

reaches a maximum of 2.5 GByte/s, while when the default IBA CC is used, it remains at 2 Gbyte/s. That increase of the bandwidth corresponds to the *P2PBW+Sync* test, as this is where the most data is sent.

On the other hand, Figure 7.43 shows the time that the same node 30 has to wait to transmit packets. The time needed to send data using the *LIMITLESS* is longer than that required when using the IBA CC alone in no-congestion conditions. However, this

Figure 7.45: *portXmitWait* in node 37.

difference is marginal and due to anodyne congestion. When the scenario becomes more intensive, the time to send data exceeds the *upper\_limit* set for detecting congestion, and the IBA CC starts working after the time needed for the CC to react. Note that, when the congestion is high, *PortXmitWait* is updated around the 3:00 minutes of execution time.

Figures 7.44 and 7.45 show the same information in node 37. As the figures show, the utilization of LIMITLESS+CC obtains higher values of bandwidth than the standard IBA CC.

Following figures show the results for the P2PBW+Sync (Random Ring Point-to-point Bandwidth with Synchronization) experiment. They also show the comparison between the standard CC and the LIMITLESS+CC alternative. Different values of *CCTI\_increase* have been used to check its impact on the network bandwidth.

Figures 7.46 and 7.47 show the results of the P2PBW+Sync scenario when there is no congestion, measuring them in average and focusing on the 99 percentile. Note that higher values of *CCTI\_Increase* parameter produces higher levels of intrusion of the IBA CC, reaching lower levels of bandwidth. However, using the implementation with LIMITLESS, regardless of the *CCTI\_Increase* value, the bandwidth keeps at maximum values.

Finally, figures 7.48 and 7.49 show the results when GPCNeT generates congestion in the network. In this case, the traffic in the network is intensive, and the reached bandwidth is lower. It can be seen smaller values on *portXmitWait* when LIMITLESS+CC is active because this alternative seems to react faster adjusting the injection rate. It can also be seen that, in average, higher values for *CCTI\_increase* do not produce significant variations in

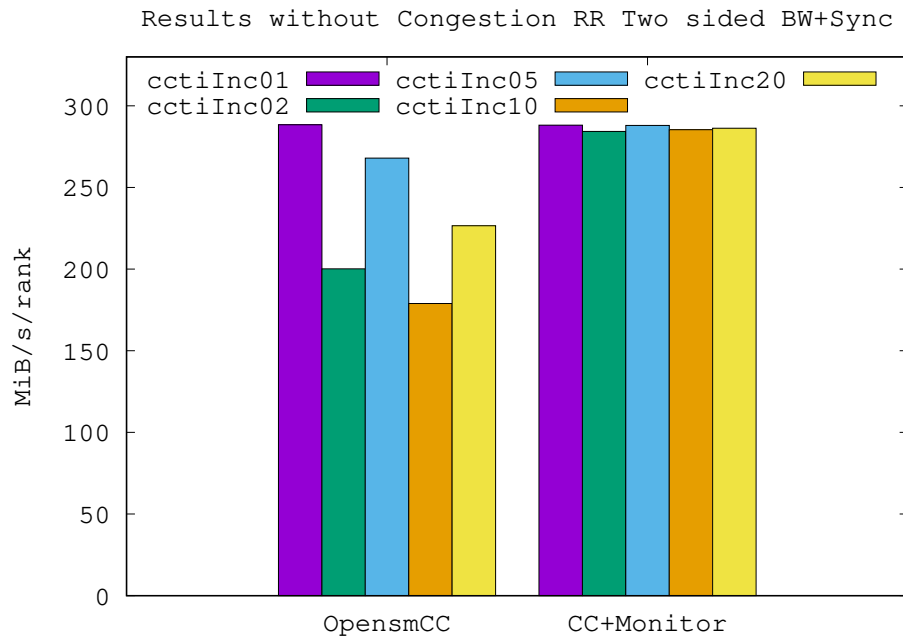


Figure 7.46: P2PBW+Sync. No congestion (Average).

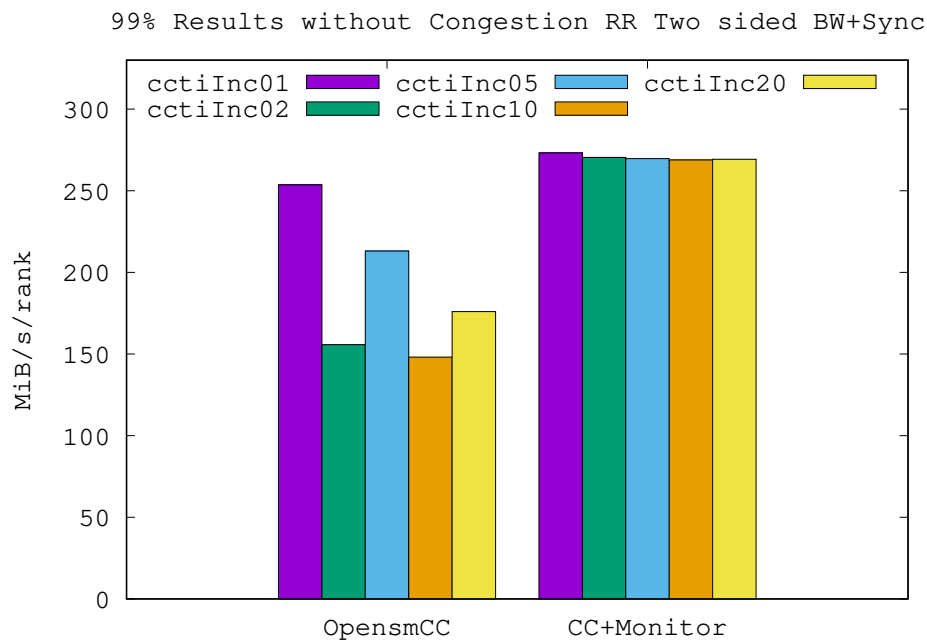


Figure 7.47: P2PBW+Sync. No congestion (99%-tile).

the bandwidth, and the standard CC detects congestion when the network is not actually congested. On the other hand, LIMITLESS+CC can estimate better the congestion, based on these results, because it reacts faster and refine the congestion control.

In a summary, LIMITLESS+CC dynamically sets the *CCTI\_Increase* value at given HCA based on the collected *portXmitWait* and *portXmitData* performance counters (which gives an idea of the congestion). This solution allows the adjustment of the injection rate

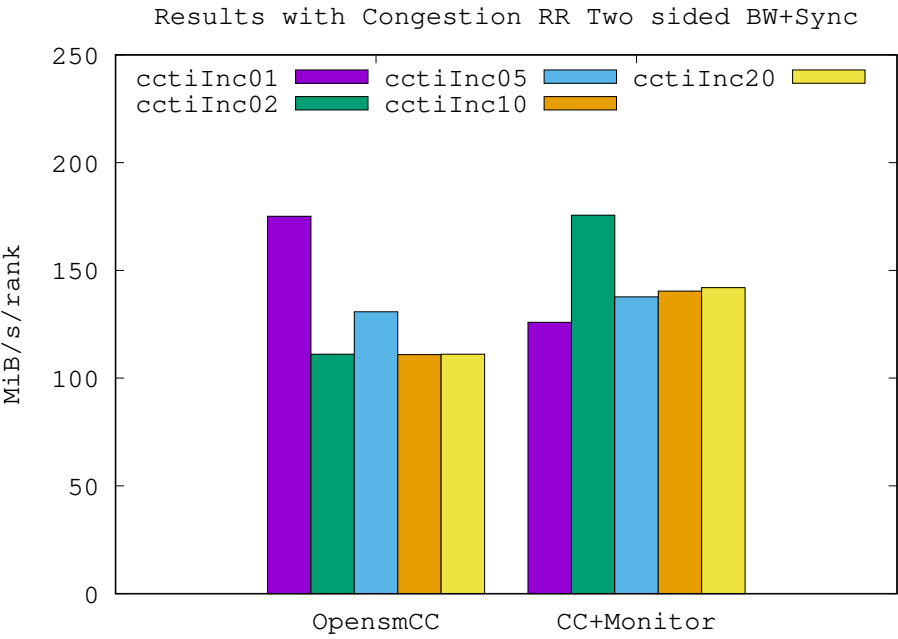


Figure 7.48: P2PBW+Sync with congestion (Average).

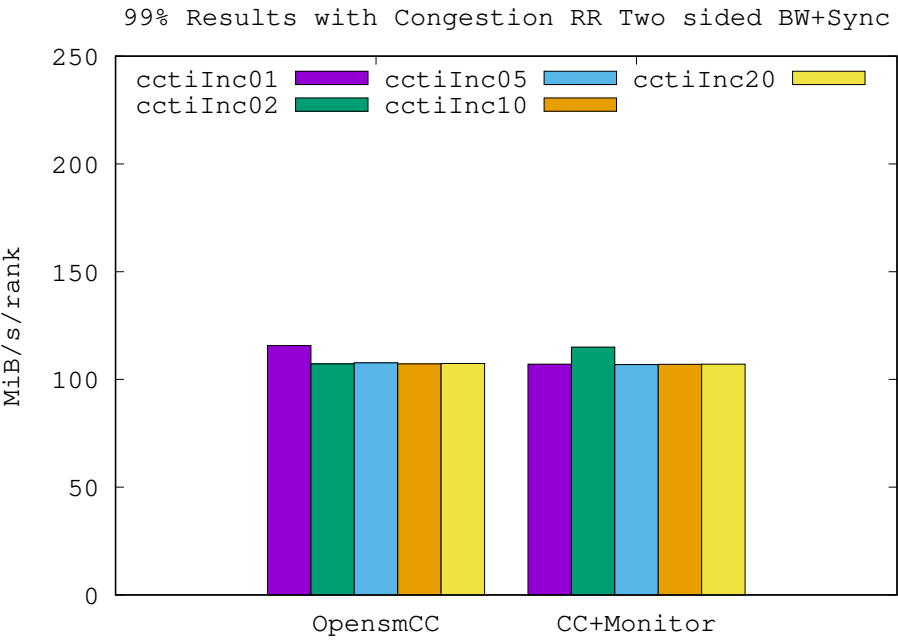


Figure 7.49: P2PBW+Sync with congestion (99%-tile).

of the traffic flows, regardless of whether the congestion is higher or lower.

7.6. Summary

This chapter shows the different results obtained for each feature that is a candidate for an evaluation: monitoring overhead, a comparison with other monitor, the benefits of the



new scheduling policies designed, the impact of the optimizations to reduce the monitor communication overhead, etc., and includes results for each section described in Chapter 3 (Architecture) and Chapter 5 (Implementation).

The next chapter shows the conclusions of this PhD. thesis and the planned future works, as well as the contributions and the objectives completed.

## 8. CONCLUSION AND FUTURE WORK

This PhD. thesis describes the design and development of a monitoring and scheduling framework that has been designed for large-scale computing infrastructures. It schedules applications with different multi-criteria policies, and includes various optimizations to reduce the impact of the monitoring to the machines, and brings smart support thanks to the use of ML and NN.

The features that it provides include topological-aware deployment, dynamic reconfiguration, in-situ and in-transit processing for reducing the monitor traffic, performance modeling, event detection and notification, and two new and advanced scheduling policies based on monitoring information combined with Machine Learning algorithms. This work is aimed to address some of the European research priorities in the area of HPC technology. One of the main characteristics of LIMITLESS is its integration with other platform software components like the application scheduler, CLARISSE and FlexMPI runtimes.

The monitoring tool has a really low overhead, as it has been demonstrated, in comparison with other similar tools studied in a deep researching of the state of the art. Besides, it is easy to deploy, does not need special installation packages or instructions, and it is fully and dynamically configured without the necessity of restart the system to apply the changes. It includes elements to facilitate the visualization and exploitation of the data, can be connected to other software through the developed API, and has some desirable software characteristics: it is scalable, extensible and robust.

This work includes some optimizations to reduce the overhead generated by the communications between the components, including optimizations inside the monitoring nodes and the other elements of the hierarchy, and out of the monitoring nodes applying Machine Learning techniques for identifying and predicting applications to design new scheduling policies, that can reduce (as has been seen in Chapter 6) the monitoring traffic up to 85%. Also, new scheduling techniques have been described: *fine-grain scheduling*, which is based on running applications at the same time in the same node, leveraging the phases which do not produce performance degradation in any of them. With this solution LIMITLESS can reduce the makespan on the described use cases in more than 25%, and *coarse-grain scheduling*, which is based on share applications in a node if the complete concurrent execution of both applications does not produce performance degradation in any of them. With this scheduling mode, the makespan on the described use cases can be

reduced between 20% and 30%.

At the end of the PhD. thesis, the status of the objectives listed in Section 1.3 is the following:

- **O1. Explore new techniques and abstractions to allow HPC applications the exploiting of the parallelism, locality, elasticity and adaptability of LSDS.** A monitoring framework has been designed to be able to work in LSDS, providing useful information about the platform and the status of each machine with the objective of having a global view of the infrastructure, and using that information to assign applications to the better nodes depending on their performance, and to design new scheduling policies.
- **O2. Design the monitoring tool to be able to run in heterogeneous and homogeneous machines.** The designed framework is capable of operating in heterogeneous and homogeneous systems because the methodology for collecting the data is independent of the devices in the machines. It has been tested in four different clusters, homogeneous and heterogeneous, with good results.
- **O3. Explore different techniques to collect information in a machine in two levels: node and application.** As it has been described, the monitoring information provided includes information about the hardware performance of the machines, and the applications that are running in them. All of them available to be displayed in visualizers or processed by user applications.
- **O4. Design new scheduling policies based on multiple criteria.** LIMITLESS includes two different scheduling policies for shared nodes: *coarse-grain* and *fine-grain* scheduling. Both of them try to execute different applications in the same node, reducing the makespan, but this combination can be done based on multiple criteria. For example, scheduling taking into account the available resources and the energy consumption, or I/O and network usage.
- **O5. Explore how to bring machine learning and neural networks support to the proposed framework.** LIMITLESS includes a module to perform machine learning operations over the collected data. Thanks to this feature, the framework can exploit the data and can make predictions and application identifications efficiently and with high accuracy. All of those results are used to improve the scheduling process, and to optimize the communications between the framework components (if the predictions are correct, there is no necessity of send future monitoring packets).

It is considered all of them achieved.

### 8.1. Contribution

Altogether, this PhD. thesis provides the following contributions: LIMITLESS - Light-weight MonItoring Tool for Large Scale Systems

- **A light-weight monitoring tool designed to be used in Large-Scale Distributed Systems.** The monitoring tool included in LIMITLESS has a very low overhead in comparison with other monitors. Its architecture and components provide scalability levels to make possible the execution in very large-scale system (taking into account the trend towards exascale).
- **Mechanisms to coordinate the monitoring tool with other system components to increase the global results.** The monitoring tool can be integrated with other components and system runtimes to take advantage of their functionalities and collect more information to feed into the scheduler and analytic component.
- **An API that allows the users to get every information collected since the beginning of the execution of the framework.** This API includes functions to communicate C/C++ applications directly to the LDS component, which is in charge of store and load the monitoring information received. There are two groups of functions, depending on the data that the user wants: data about the current state of the system (current data) or previous data collected (historical data).
- **Optimization techniques to reduce the communication overhead of the framework,** which is one of the most important factors in monitoring, because the whole information should finish in a central or distributed storage, which implies moves of large amount of data through the network. Application of filtering techniques and the methodology called *in-transit optimization* manage to reduce the number of communications by a large percentage.
- **A component that integrates some Machine Learning algorithms to allow predictions, classifications and profiling.** The LA component includes smart analytic functionalities that provide functions to predict future states of the system (for scheduling and for reducing the communications in *in-transit* processing), classification to identify which applications are currently running in the compute-nodes (using the application models, LA can train ML algorithms to recognize applications), and profiling to decompose the applications, find out their profiles, how many phases do they perform (also their performance), and all of them to improve the scheduling with new and smart scheduling policies that can be updated dynamically.

- **A deep study and evaluation of the framework that shows its overhead, its performance, and the results of the analytic component.** At first, a deep study of the state of the art has been done in three different areas: monitoring, scheduling and Machine Learning techniques, describing solutions and the importance of the works. On the other hand, many tests of each framework component have been done and organized in Chapter 6, where the use cases, considerations and results can be seen with complete descriptions and explanations about the process.

## 8.2. Future work

There are several future researching lines resulting from this PhD. thesis:

- Include a new feature in the framework to automatically detect the cluster topology with the objective of auto-deploy the monitor with an efficient layout. Currently the system has to be deployed designing manually a topology (flat or hierarchical), but now we are researching the way to facilitate this task optimizing the process of discovering the resources in a network.
- Include more refined prediction algorithms for improving the in-transit processing algorithm (optimization to reduce the network usage overhead), and to improve the predictions to perform scheduling based on future states of the system.
- Integrate this smart framework into a worldwide used workload manager: SLURM. The objective is to design a plugin for that software to take advantage of its popularity, to give other researchers the possibility of use our framework to reduce the makespan of their workloads, to have a clear view of the performance of their systems, and to predict future events based on the monitoring historical data.

## BIBLIOGRAPHY

- [1] A. Cascajo, D. E. Singh, and J. Carretero, “Performance-Aware Scheduling of Parallel Applications on Non-Dedicated Clusters,” *Electronics*, vol. 8, no. 9, p. 982, 2019. doi: 10.3390/electronics8090982.
- [2] Cascajo, Alberto and Singh, David E. and Carretero, Jesus, *Avances en Arquitectura y Tecnología de Computadores. Actas de las Jornadas SARTECO 2019*, U. de Extremadura, Ed. Universidad de Extremadura, Servicio de Publicaciones, 2019, pp. 530–537. [Online]. Available: <https://dehesa.unex.es/handle/10662/9626>.
- [3] A. Cascajo, D. E. Singh, and J. Carretero, “LIMITLESS - Light-weight Monitoring Tool for Large Scale Systems,” in *Proceedings - 29th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, PDP 2021*, Institute of Electrical and Electronics Engineers Inc., Mar. 2021, pp. 220–227. doi: 10.1109/PDP52278.2021.00042.
- [4] A. Cascajo, D. E. Singh, and J. Carretero, *Adaptive scheduling of HPC applications using malleability and dynamic migration*, 2019. [Online]. Available: <https://scheduling2019.sciencesconf.org/271456> (visited on 06/18/2021).
- [5] S. Jha, J. Qiu, A. Luckow, P. Mantha, and G. C. Fox, “A Tale of Two Data-Intensive Paradigms: Applications, Abstractions, and Architectures,” in *2014 IEEE International Congress on Big Data*, IEEE, Jun. 2014, pp. 645–652. doi: 10.1109/BigData.Congress.2014.137.
- [6] S. Lang, P. Carns, R. Latham, R. Ross, K. Harms, and W. Allcock, “I/O performance challenges at leadership scale,” in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis - SC '09*, New York, New York, USA: ACM Press, 2009, p. 1. doi: 10.1145/1654059.1654100. [Online]. Available: <http://dl.acm.org/citation.cfm?doid=1654059.1654100>.
- [7] *Global trends in internet traffic, data centre workloads and data centre energy use, 2010-2010*, 2020. [Online]. Available: <https://www.iea.org/data-and-statistics/charts/global-trends-in-internet-traffic-data-centre-%5C%20workloads-and-data-centre-energy-use-2010-2019>.

- [8] *Is Current Progress in Artificial Intelligence Exponential?* 2020. [Online]. Available: <https://medium.com/@reevesastronomy/is-current-progress-in-artificial-intelligence-exponential-8e18f126d2cb>.
- [9] W. M. Jones, J. T. Daly, and N. DeBardeleben, "Application monitoring and checkpointing in HPC: Looking towards exascale systems," in *Proceedings of the 50th Annual Southeast Regional Conference*, ser. ACM-SE '12, Tuscaloosa, Alabama: ACM, 2012, pp. 262–267.
- [10] T. Evans *et al.*, "Comprehensive resource use monitoring for hpc systems with TACC Stats," in *Proceedings of the First International Workshop on HPC User Support Tools*, ser. HUST '14, New Orleans, Louisiana, 2014, pp. 13–21.
- [11] M. L. Massie, B. N. Chun, and D. E. Culler, "The ganglia distributed monitoring system: design, implementation, and experience," *Parallel Computing*, vol. 30, no. 7, pp. 817–840, Jul. 2004. doi: 10.1016/J.PARCO.2004.04.001. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0167819104000535>.
- [12] *sFlow.org - Making the Network Visible*, 2018. [Online]. Available: <https://sflow.org/> (visited on 10/25/2018).
- [13] *Ganglia Monitoring System*, 2020. [Online]. Available: <http://ganglia.sourceforge.net/> (visited on 10/29/2018).
- [14] *Nagios - The Industry Standard In IT Infrastructure Monitoring*, 2018. [Online]. Available: <https://www.nagios.org/> (visited on 10/25/2018).
- [15] E. Imamagic and D. Dobrenic, "Grid infrastructure monitoring system based on Nagios," in *Proceedings of the 2007 workshop on Grid monitoring - GMW '07*, New York, New York, USA: ACM Press, 2007, p. 23. doi: 10.1145/1272680.1272685. [Online]. Available: <http://portal.acm.org/citation.cfm?doid=1272680.1272685>.
- [16] S. Andreozzi *et al.*, "GridICE: a monitoring service for Grid systems," *Future Generation Computer Systems*, vol. 21, pp. 559–571, 2005. doi: 10.1016/j.future.2004.10.005. [Online]. Available: [https://ac.els-cdn.com/S0167739X04001669/1-s2.0-S0167739X04001669-main.pdf?%7B%5C\\_%7Dtid=60adef6a-82f1-49b9-a5e8-fa51786a915b%7B%5C%7Dacdnat=1540810667%7B%5C\\_%7D5a5f142ae9b011db2d8926503006ee04](https://ac.els-cdn.com/S0167739X04001669/1-s2.0-S0167739X04001669-main.pdf?%7B%5C_%7Dtid=60adef6a-82f1-49b9-a5e8-fa51786a915b%7B%5C%7Dacdnat=1540810667%7B%5C_%7D5a5f142ae9b011db2d8926503006ee04).
- [17] H. Newman, P. Galvez, R. Voicu, and C. Cirstoiu, "MonALISA : A Distributed Monitoring Service Architecture," Tech. Rep. [Online]. Available: <http://monalisa.caltech.edu/documentation/MOET001.pdf>.

- [18] J. Montes, A. Sánchez, B. Memishi, M. S. Pérez, and G. Antoniu, “GMonE: A complete approach to cloud monitoring,” *Future Generation Computer Systems*, vol. 29, no. 8, pp. 2026–2040, Oct. 2013. doi: 10.1016/J.FUTURE.2013.02.011. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0167739X13000496>.
- [19] E. Volk *et al.*, “Towards Intelligent Management of Very Large Computing Systems,” in *Competence in High Performance Computing 2010*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 191–204. doi: 10.1007/978-3-642-24025-6\_16. [Online]. Available: [http://link.springer.com/10.1007/978-3-642-24025-6%7B%5C\\_%7D16](http://link.springer.com/10.1007/978-3-642-24025-6%7B%5C_%7D16).
- [20] M. Badger, *Zenoss core network and system monitoring*. Packt Pub, 2008, p. 280. [Online]. Available: [https://books.google.es/books?hl=es%7B%5C\\_%7Dlr=%7B%5C\\_%7Ddid=B3YBMfU%7B%5C\\_%7Du8sC%7B%5C\\_%7Doi=fnd%7B%5C\\_%7Dpg=PT1%7B%5C\\_%7Ddq=zenoss%7B%5C\\_%7Dots=WdvjzcAkfn%7B%5C\\_%7Dsigt=t2q2qRP48piXuCaug9ggSxbYZ-8%7B%5C\\_%7Dv=onepage%7B%5C\\_%7Dq=zenoss%7B%5C\\_%7Df=false](https://books.google.es/books?hl=es%7B%5C_%7Dlr=%7B%5C_%7Ddid=B3YBMfU%7B%5C_%7Du8sC%7B%5C_%7Doi=fnd%7B%5C_%7Dpg=PT1%7B%5C_%7Ddq=zenoss%7B%5C_%7Dots=WdvjzcAkfn%7B%5C_%7Dsigt=t2q2qRP48piXuCaug9ggSxbYZ-8%7B%5C_%7Dv=onepage%7B%5C_%7Dq=zenoss%7B%5C_%7Df=false).
- [21] S. Sanchez *et al.*, “Design and Implementation of a Scalable HPC Monitoring System,” in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, IEEE, May 2016, pp. 1721–1725. doi: 10.1109/IPDPSW.2016.167. [Online]. Available: <http://ieeexplore.ieee.org/document/7530073/>.
- [22] Trinity, 2018. [Online]. Available: <https://www.lanl.gov/projects/trinity/> (visited on 10/30/2018).
- [23] Collectd - The system statistics collection daemon, 2020. [Online]. Available: <https://collectd.org/> (visited on 10/30/2018).
- [24] R. P. Centelles, M. Selimi, F. Freitag, and L. Navarro, “DIMON: Distributed monitoring system for decentralized edge clouds in guifi.net,” in *Proceedings - 2019 IEEE 12th Conference on Service-Oriented Computing and Applications, SOCA 2019*, Institute of Electrical and Electronics Engineers Inc., 2019. doi: 10.1109/SOCA.2019.00009.
- [25] M. J. Sottile and R. G. Minnich, in *Proceedings. IEEE International Conference on Cluster Computing*, 2002. doi: 10.1109/CLUSTER.2002.1137727.
- [26] K. Stefanov, V. Voevodin, S. Zhumatiy, and V. Voevodin, “Dynamically Reconfigurable Distributed Modular Monitoring System for Supercomputers (DiMMon),”



- Procedia Computer Science*, vol. 66, 2015. doi: <https://doi.org/10.1016/j.procs.2015.11.071>.
- [27] J. Sperhac *et al.*, “Federating XDMoD to Monitor Affiliated Computing Resources,” in *Proceedings - IEEE International Conference on Cluster Computing, ICC*, vol. 2018-September, Institute of Electrical and Electronics Engineers Inc., 2018, pp. 580–589.
- [28] R. Chakode, *Monitoring with Graphite: Architecture and Concepts | MetricFire Blog*, 2019. [Online]. Available: <https://www.metricfire.com/blog/monitoring-with-graphite-architecture-and-concepts/>.
- [29] *Amazon CloudWatch: Monitoreo de infraestructuras y aplicaciones*, 2018. [Online]. Available: <https://aws.amazon.com/es/cloudwatch/>.
- [30] *IBM Tivoli Monitoring - Overview*, 2018. [Online]. Available: [https://www.ibm.com/support/knowledgecenter/en/SSTFXA%7B%5C\\_%7D6.3.0/com.ibm.itm.doc%7B%5C\\_%7D6.3/install/itm%7B%5C\\_%7Dover.htm](https://www.ibm.com/support/knowledgecenter/en/SSTFXA%7B%5C_%7D6.3.0/com.ibm.itm.doc%7B%5C_%7D6.3/install/itm%7B%5C_%7Dover.htm) (visited on 11/05/2018).
- [31] *IBM Tivoli Monitoring for Virtual Environments*. [Online]. Available: [https://www.ibm.com/support/knowledgecenter/en/SS9U76%7B%5C\\_%7D7.2.0.2/com.ibm.tivoli.itmvs.doc%7B%5C\\_%7D7.2.0.2/VE72fp2%7B%5C\\_%7Dqsg%7B%5C\\_%7Den.html](https://www.ibm.com/support/knowledgecenter/en/SS9U76%7B%5C_%7D7.2.0.2/com.ibm.tivoli.itmvs.doc%7B%5C_%7D7.2.0.2/VE72fp2%7B%5C_%7Dqsg%7B%5C_%7Den.html) (visited on 11/05/2018).
- [32] *Nagios Core Documentation*, 2018. [Online]. Available: [https://assets.nagios.com/downloads/nagioscore/docs/nagioscore/4/en/index.html%7B%5C#%7D%7B%5C\\_%7Dga=2.149213937.691373301.1541418878-1705882606.1540813970](https://assets.nagios.com/downloads/nagioscore/docs/nagioscore/4/en/index.html%7B%5C#%7D%7B%5C_%7Dga=2.149213937.691373301.1541418878-1705882606.1540813970) (visited on 11/05/2018).
- [33] *High Availability Solution for Nagios XI*, 2018. [Online]. Available: <https://www.nagios.com/news/2015/10/press-release-nagios-enterprise-and-linbit-release-high-availability-solution-for-nagios-xi/> (visited on 11/05/2018).
- [34] M. Li, M. Baker, and John Wiley & Sons., *The grid : core technologies*. Wiley, 2005, p. 423.
- [35] *HLRS High Performance Computing Center Stuttgart - TIMaCS*. [Online]. Available: <https://www.hlrs.de/about-us/research/past-projects/timacs/> (visited on 11/05/2018).
- [36] *Collectl*, 2021. [Online]. Available: <http://collectl.sourceforge.net/> (visited on 06/02/2021).

- [37] T. Dwyer, A. Fedorova, S. Blagodurov, M. Roth, F. Gaud, and J. Pei, “A practical method for estimating performance degradation on multicore processors, and its application to hpc workloads,” in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, IEEE Computer Society Press, 2012, p. 83.
- [38] M. Bhadauria and S. A. McKee, “An approach to resource-aware co-scheduling for cmps,” in *Proceedings of the 24th ACM International Conference on Supercomputing*, ACM, 2010, pp. 189–199.
- [39] A. B. Yoo, M. A. Jette, and M. Grondona, “SLURM: Simple Linux Utility for Resource Management,” *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 2862, pp. 44–60, 2003. doi: 10.1007/10968987\_3.
- [40] X. Jiang *et al.*, “Energy-Efficient Scheduling of Periodic Applications on Safety-Critical Time-Triggered Multiprocessor Systems,” *Electronics*, vol. 7, no. 6, p. 98, Jun. 2018. doi: 10.3390/electronics7060098. [Online]. Available: <http://www.mdpi.com/2079-9292/7/6/98>.
- [41] A. Mahmood *et al.*, “Energy-Aware Real-Time Task Scheduling in Multiprocessor Systems Using a Hybrid Genetic Algorithm,” *Electronics*, vol. 6, no. 2, p. 40, May 2017. doi: 10.3390/electronics6020040. [Online]. Available: <http://www.mdpi.com/2079-9292/6/2/40>.
- [42] X. Su, F. Lei, X. Su, and F. Lei, “Hybrid-Grained Dynamic Load Balanced GEMM on NUMA Architectures,” *Electronics*, vol. 7, no. 12, p. 359, Nov. 2018. doi: 10.3390/electronics7120359. [Online]. Available: <http://www.mdpi.com/2079-9292/7/12/359>.
- [43] S. Rajkumar, N. Rajkumar, and V. G. Suresh, “Automated object counting for visual inspection applications,” in *International Conference on Information Communication and Embedded Systems (ICICES2014)*, IEEE, 2014.
- [44] T. Jones, “Linux kernel co-scheduling for bulk synchronous parallel applications,” in *Proceedings of the 1st international workshop on runtime and operating systems for supercomputers*, ACM, 2011, pp. 57–64.
- [45] J. Breitbart, J. Weidendorfer, and C. Trinitis, “Case study on co-scheduling for hpc applications,” in *2015 44th ICPP Conference Workshops*, Sep. 2015, pp. 277–285.
- [46] J. Weidendorfer and J. Breitbart, “Detailed characterization of hpc applications for co-scheduling,” in *Proceedings of the 1st COSH Workshop on Co-Scheduling of HPC Applications*, 2016, p. 19.

- [47] A. Sedighi, M. Smith, and Y. Deng, “Fud—balancing scheduling parameters in shared computing environments,” in *2017 IEEE 4th International Conference on Cyber Security and Cloud Computing (CSCloud)*, IEEE, 2017, pp. 363–368.
- [48] D. Klusáček, H. Rudová, R. Baraglia, M. Pasquali, and G. Capannini, “Comparison of multi-criteria scheduling techniques,” in *Grid Computing*, Springer, 2008, pp. 173–184.
- [49] M.-A. Vasile, F. Pop, R.-I. Tutueanu, V. Cristea, and J. Kołodziej, “Resource-aware hybrid scheduling algorithm in heterogeneous distributed computing,” *Future Generation Computer Systems*, vol. 51, pp. 61–71, 2015.
- [50] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella, “Multi-resource packing for cluster schedulers,” *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, pp. 455–466, 2014.
- [51] T. T. Tran, M. Padmanabhan, P. Y. Zhang, H. Li, D. G. Down, and J. C. Beck, “Multi-stage resource-aware scheduling for data centers with heterogeneous servers,” *Journal of Scheduling*, vol. 21, no. 2, pp. 251–267, Apr. 2018.
- [52] A. Raveendran, T. Bicer, and G. Agrawal, “A framework for elastic execution of existing mpi programs,” in *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, May 2011, pp. 940–947. doi: 10.1109/IPDPS.2011.240.
- [53] R. d. R. Righi, V. F. Rodrigues, C. A. da Costa, G. Galante, L. C. E. de Bona, and T. Ferreto, “Autoelastic: Automatic resource elasticity for high performance applications in the cloud,” *IEEE Transactions on Cloud Computing*, vol. 4, no. 1, pp. 6–19, Jan. 2016. doi: 10.1109/TCC.2015.2424876.
- [54] D. E. Singh and J. Carretero, “Combining malleability and I/O control mechanisms to enhance the execution of multiple applications,” *Journal of Systems and Software*, vol. 148, pp. 21–36, Feb. 2019. doi: 10.1016/j.jss.2018.11.006.
- [55] M. Rodriguez-Gonzalo, D. E. Singh, J. G. Blas, and J. Carretero, “Improving the Energy Efficiency of MPI Applications by Means of Malleability,” in *Proceedings - 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2016*, Institute of Electrical and Electronics Engineers Inc., Mar. 2016, pp. 627–634. doi: 10.1109/PDP.2016.98.
- [56] S. Blagodurov and A. Fedorova, “Towards the contention aware scheduling in HPC cluster environment,” *Journal of Physics: Conference Series*, vol. 385, p. 012010, Oct. 2012. doi: 10.1088/1742-6596/385/1/012010.

- [57] A. Gupta, B. Acun, O. Sarood, and L. V. Kalé, “Towards realizing the potential of malleable jobs,” in *2014 21st International Conference on High Performance Computing (HiPC)*, Dec. 2014, pp. 1–10. doi: 10.1109/HiPC.2014.7116905.
- [58] K. Ekanadham *et al.*, “Application Oriented Resource Management on Large Scale Parallel Systems,” *IBM RESEARCH, YORKTOWN HEIGHTS*, pp. 56–63, 1995. [Online]. Available: <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.56.1178>.
- [59] A. Agelastos *et al.*, “Continuous whole-system monitoring toward rapid understanding of production HPC applications and systems,” *Parallel Computing*, vol. 58, pp. 90–106, 2016.
- [60] T. Rohl, J. Eitzinger, G. Hager, and G. Wellein, “LIKWID monitoring stack: A flexible framework enabling job specific performance monitoring for the masses,” in *Proceedings - IEEE International Conference on Cluster Computing, ICC*, vol. 2017-September, Institute of Electrical and Electronics Engineers Inc., 2017, pp. 781–784.
- [61] A. Cascajo, D. E. Singh, and J. Carretero, “Performance-aware scheduling of parallel applications on non-dedicated clusters,” *Electronics*, vol. 8, no. 9, p. 982, 2019.
- [62] Y. Yu, L. Guo, J. Huang, F. Zhang, and Y. Zong, “A cross-layer security monitoring selection algorithm based on traffic prediction,” *IEEE Access*, vol. 6, pp. 35 382–35 391, 2018.
- [63] S. M. Rashti, M. Mollanoori, M. S. Nia, and N. M. Charkari, “A prediction-based algorithm for target tracking in wireless sensor networks,” in *2009 International Conference on Ultra Modern Telecommunications and Workshops*, 2009.
- [64] H. Tang, G. Tang, and L. Meng, “Prediction of the bridge monitoring data based on support vector machine,” in *Proceedings - International Conference on Natural Computation*, vol. 2016-January, IEEE Computer Society, 2016, pp. 781–785.
- [65] X. Kang and M. Xu, “Explore of monitoring data pattern prediction of gas tunnel,” in *2011 International Conference on Remote Sensing, Environment and Transportation Engineering, RSETE 2011 - Proceedings*, 2011, pp. 4046–4049.
- [66] R. Lijia, L. Hong, and L. Yan, “On-line monitoring and prediction for transmission line sag,” in *Proceedings of 2012 IEEE International Conference on Condition Monitoring and Diagnosis, CMD 2012*, 2012, pp. 813–817.

- [67] S. Bhulai, “Nearest neighbour algorithms for forecasting call arrivals in call centers,” in *Smart Innovation, Systems and Technologies*, vol. 39, Springer Science and Business Media Deutschland GmbH, 2015, pp. 77–87.
- [68] N. Wiebe, A. Kapoor, and K. M. Svore, Tech. Rep. eprint: 1401.2142v1.
- [69] D. M. Nelson, A. C. Pereira, and R. A. De Oliveira, “Stock market’s price movement prediction with LSTM neural networks,” in *Proceedings of the International Joint Conference on Neural Networks*, vol. 2017-May, Institute of Electrical and Electronics Engineers Inc., 2017, pp. 1419–1426. doi: 10.1109/IJCNN.2017.7966019.
- [70] K. Li, J. Daniels, C. Liu, P. Herrero, and P. Georgiou, “Convolutional Recurrent Neural Networks for Glucose Prediction,” *IEEE Journal of Biomedical and Health Informatics*, vol. 24, pp. 603–613, 2020. doi: 10.1109/JBHI.2019.2908488.
- [71] I. M. Nasser and S. S. Abu-Naser, “Predicting Tumor Category Using Artificial Neural Networks,” 2019. [Online]. Available: <http://dspace.alazhar.edu.ps/xmlui/handle/123456789/303>.
- [72] T. Y. Yang, C. G. Brinton, C. Joe-Wong, and M. Chiang, “Behavior-Based Grade Prediction for MOOCs Via Time Series Neural Networks,” *IEEE Journal on Selected Topics in Signal Processing*, vol. 11, no. 5, pp. 716–728, Aug. 2017. doi: 10.1109/JSTSP.2017.2700227.
- [73] L. Zhang, G. Yu, D. Xia, and J. Wang, “Protein–protein interactions prediction based on ensemble deep neural networks,” *Neurocomputing*, vol. 324, pp. 10–19, Jan. 2019. doi: 10.1016/j.neucom.2018.02.097.
- [74] M. Chen, Y. Hao, K. Hwang, L. Wang, and L. Wang, “Disease Prediction by Machine Learning over Big Data from Healthcare Communities,” *IEEE Access*, vol. 5, pp. 8869–8879, 2017. doi: 10.1109/ACCESS.2017.2694446.
- [75] M. Sabah, M. Talebkeikhah, D. A. Wood, R. Khosravanian, M. Anemangely, and A. Younesi, “A machine learning approach to predict drilling rate using petrophysical and mud logging data,” *Earth Science Informatics*, vol. 12, no. 3, pp. 319–339, Sep. 2019. doi: 10.1007/s12145-019-00381-4. [Online]. Available: <https://link.springer.com/article/10.1007/s12145-019-00381-4>.
- [76] X. Zeng, S. Yan, F. He, and Y. Shi, “Multi-variable grey model based on dynamic background algorithm for forecasting the interval sequence,” *Applied Mathematical Modelling*, vol. 80, pp. 99–114, Apr. 2020. doi: 10.1016/j.apm.2019.11.032.

- [77] P. Mercati *et al.*, “Multi-variable Dynamic Power Management for the GPU Subsystem,” in *Proceedings - Design Automation Conference*, vol. Part 128280, Institute of Electrical and Electronics Engineers Inc., Jun. 2017. doi: 10.1145/3061639.3062288.
- [78] K. Nagarajan, “A Predictive hill climbing algorithm for real valued multi-variable optimization problem like PID tuning,” *International Journal of Machine Learning and Computing*, vol. 8, no. 1, pp. 14–19, Feb. 2018. doi: 10.18178/ijmlc.2018.8.1.656.
- [79] D. Selvathi and H. Selvaraj, “Segmentation of brain tumor tissues in MR images using multiresolution transforms and random forest classifier with adaboost technique,” in *26th International Conference on Systems Engineering, ICSEng 2018 - Proceedings*, Institute of Electrical and Electronics Engineers Inc., Feb. 2019. doi: 10.1109/ICSENG.2018.8638244.
- [80] H. Ma, W. Yan, Z. Yang, and H. Liu, “Real-Time Foot-Ground Contact Detection for Inertial Motion Capture Based on an Adaptive Weighted Naive Bayes Model,” *IEEE Access*, vol. 7, pp. 130 312–130 326, 2019. doi: 10.1109/ACCESS.2019.2939839.
- [81] R. Bamler and S. Mandt, “Extreme Classification via Adversarial Softmax Approximation,” *arXiv*, Feb. 2020. arXiv: 2002.06298.
- [82] T. H. Lee, A. Ullah, and R. Wang, “Bootstrap Aggregating and Random Forest,” in *Advanced Studies in Theoretical and Applied Econometrics*, Springer, 2020, pp. 389–429. doi: 10.1007/978-3-030-31150-6\_13.
- [83] H. Lu and X. Ma, “Hybrid decision tree-based machine learning models for short-term water quality prediction,” *Chemosphere*, vol. 249, p. 126 169, Jun. 2020. doi: 10.1016/j.chemosphere.2020.126169.
- [84] B. Mor, S. Garhwal, and A. Kumar, “A Systematic Review of Hidden Markov Models and Their Applications,” *Archives of Computational Methods in Engineering volume*, vol. 28, pp. 1429–1448, 2021. doi: 10.1007/s11831-020-09422-4.
- [85] W. M. Shaban, A. H. Rabie, A. I. Saleh, and M. A. Abo-Elvoud, “A new COVID-19 Patients Detection Strategy (CPDS) based on hybrid feature selection and enhanced KNN classifier,” *Knowledge-Based Systems*, vol. 205, p. 106 270, Oct. 2020. doi: 10.1016/j.knosys.2020.106270.

- [86] I. Shahin, A. B. Nassif, and S. Hamsa, “Novel cascaded Gaussian mixture model-deep neural network classifier for speaker identification in emotional talking environments,” *Neural Computing and Applications*, vol. 32, no. 7, pp. 2575–2587, Apr. 2020. doi: 10.1007/s00521-018-3760-2. [Online]. Available: <https://doi.org/10.1007/s00521-018-3760-2>.
- [87] H. Alsghaier and M. Akour, “Software fault prediction using particle swarm algorithm with genetic algorithm and support vector machine classifier,” *Software: Practice and Experience*, vol. 50, no. 4, pp. 407–427, Apr. 2020. doi: 10.1002/spe.2784.
- [88] C. Gormley and Z. Tong, *Elasticsearch: The Definitive Guide: A Distributed Real-Time Search and ...* - Clinton Gormley, Zachary Tong - Google Libros, First Edition. O’Reilly Media, 2015. [Online]. Available: <https://books.google.es/books?id=U19aBgAAQBAJ%7B%5C%7Dprintsec=frontcover%7B%5C%7Dhl=es%7B%5C%7Dsource=gbs%7B%5C%7Dge%7B%5C%7Dsummary%7B%5C%7Dr%7B%5C%7Dcad=0%7B%5C%7Dv=onepage%7B%5C%7Dq%7B%5C%7Df=false>.
- [89] *IPMITOOL - Linux man page*. [Online]. Available: <https://linux.die.net/man/1/ipmitool> (visited on 03/11/2021).
- [90] Top500.org. (2020). “Top 500 List,” [Online]. Available: [www.top500.org](http://www.top500.org) (visited on 12/19/2020).
- [91] D. E. Singh, G. M. Martín, M. C. Marinescu, and J. Carretero, *FlexMPI source code software*, <http://www.arcos.inf.uc3m.es/flexmpi/>, 2019.
- [92] G. Martín, D. E. Singh, M. C. Marinescu, and J. Carretero, “Enhancing the performance of malleable MPI applications by using performance-aware dynamic reconfiguration,” *Parallel Computing*, vol. 46, pp. 60–77, Jun. 2015. doi: 10.1016/j.parco.2015.04.003.
- [93] F. Isaila, J. Carretero, and R. Ross, “CLARISSE: A Middleware for Data-Staging Coordination and Control on Large-Scale HPC Platforms,” in *Proceedings - 2016 16th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, CCGrid 2016*, Institute of Electrical and Electronics Engineers Inc., Jul. 2016, pp. 346–355. doi: 10.1109/CCGrid.2016.24.
- [94] *What is Elasticsearch? | Elastic*. [Online]. Available: <https://www.elastic.co/es/what-is/elasticsearch> (visited on 03/11/2021).
- [95] Y. Gupta, *Kibana Essentials*. 2015, p. 303.

- [96] *InfluxDB: Purpose-Built Open Source Time Series Database* | InfluxData. [Online]. Available: <https://www.influxdata.com/> (visited on 03/11/2021).
- [97] E. Betke and J. Kunkel, “Real-time I/O-monitoring of hpc applications with SIOX, elasticsearch, grafana and FUSE,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 10524 LNCS, Springer Verlag, 2017, pp. 174–186. doi: 10.1007/978-3-319-67630-2\_15. [Online]. Available: [https://link.springer.com/chapter/10.1007/978-3-319-67630-2%7B%5C\\_%7D15](https://link.springer.com/chapter/10.1007/978-3-319-67630-2%7B%5C_%7D15).
- [98] *Grafana: The open observability platform* | Grafana Labs. [Online]. Available: <https://grafana.com/> (visited on 03/11/2021).
- [99] D. Terpstra, H. Jagode, H. You, and J. Dongarra, “Collecting performance data with PAPI-C,” in *Proceedings of the 3rd International Workshop on Parallel Tools for High Performance Computing 2009*, Springer Verlag, 2010, pp. 157–173. doi: 10.1007/978-3-642-11261-4\_11. [Online]. Available: [https://link.springer.com/chapter/10.1007/978-3-642-11261-4%7B%5C\\_%7D11](https://link.springer.com/chapter/10.1007/978-3-642-11261-4%7B%5C_%7D11).
- [100] H. Jagode, A. Danalis, H. Anzt, and J. Dongarra, “PAPI software-defined events for in-depth performance analysis,” *The International Journal of High Performance Computing Applications*, vol. 33, no. 6, pp. 1113–1127, Nov. 2019. doi: 10.1177/1094342019846287. [Online]. Available: <http://journals.sagepub.com/doi/10.1177/1094342019846287>.
- [101] P. J. Mucci, S. Browne, C. Deane, and G. Ho, “PAPI: A Portable Interface to Hardware Performance Counters,” Tech. Rep.
- [102] *The proc File System*. [Online]. Available: <https://web.mit.edu/rhel-doc/4/RH-DOCS/rhel-rg-en-4/ch-proc.html> (visited on 03/11/2021).
- [103] *Top - Linux man page*. [Online]. Available: <https://man7.org/linux/man-pages/man1/top.1.html> (visited on 03/11/2021).
- [104] “Linux Administration,” in *Pro Oracle Database 10g RAC on Linux*, Apress, Jan. 2008, pp. 385–400. doi: 10.1007/978-1-4302-0214-1\_15. [Online]. Available: [https://link.springer.com/chapter/10.1007/978-1-4302-0214-1%7B%5C\\_%7D15](https://link.springer.com/chapter/10.1007/978-1-4302-0214-1%7B%5C_%7D15).
- [105] P. Cunningham and S. J. Delany, “K-nearest neighbour classifiers–,” *arXiv preprint arXiv:2004.04523*, 2020.
- [106] N. Wiebe, A. Kapoor, and K. M. Svore, “Quantum algorithms for nearest-neighbor methods for supervised and unsupervised learning,” *Quantum Inf. Comput.*, vol. 15, pp. 316–356, 2015.



- [107] OMNET++ Discrete event simulator. [Online]. Available: <https://omnetpp.org>.
- [108] A. Núñez, J. L. Vázquez-Poletti, A. C. Caminero, G. G. Castañé, J. Carretero, and I. M. Llorente, “ICanCloud: A Flexible and Scalable Cloud Infrastructure Simulator,” *Journal of Grid Computing*, vol. 10, no. 1, pp. 185–209, Mar. 2012. doi: 10.1007/s10723-012-9208-5.
- [109] A. Núñez, J. Fernández, R. Filgueira, F. García, and J. Carretero, “SIMCAN: A flexible, scalable and expandable simulation platform for modelling and simulating distributed architectures and applications,” *Simulation Modelling Practice and Theory*, vol. 20, no. 1, pp. 12–32, 2012.
- [110] INET Framework. [Online]. Available: <https://inet.omnetpp.org/Introduction.html>.
- [111] L. Mészáros, A. Varga, and M. Kirsche, “Inet framework,” in *EAI/Springer Innovations in Communication and Computing*, Springer Science and Business Media Deutschland GmbH, 2019, pp. 55–106. doi: 10.1007/978-3-030-12842-5\_2. [Online]. Available: [https://link.springer.com/chapter/10.1007/978-3-030-12842-5%7B%5C\\_%7D2](https://link.springer.com/chapter/10.1007/978-3-030-12842-5%7B%5C_%7D2).
- [112] G. Martín, M. C. Marinescu, D. E. Singh, and J. Carretero, “Parallel algorithm for simulating the spatial transmission of Influenza in EpiGraph,” in *ACM International Conference Proceeding Series*, Association for Computing Machinery, 2013, pp. 205–210. doi: 10.1145/2488551.2488585. [Online]. Available: <http://dl.acm.org/citation.cfm?doid=2488551.2488585>.
- [113] G. Martín, D. E. Singh, M. C. Marinescu, and J. Carretero, “Towards efficient large scale epidemiological simulations in EpiGraph,” *Parallel Computing*, vol. 42, pp. 88–102, Feb. 2015. doi: 10.1016/j.parco.2014.09.004.
- [114] EpiGraph - Simulating COVID-19 at large scale. [Online]. Available: <https://www.arcos.inf.uc3m.es/epigraph/> (visited on 03/11/2021).
- [115] Sudheer Chunduri et al, “Gpcnet: Designing a benchmark suite for inducing and measuring contention in hpc networks,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19, Denver, Colorado: Association for Computing Machinery, 2019. doi: 10.1145/3295500.3356215. [Online]. Available: <https://doi.org/10.1145/3295500.3356215>.