



# Exploration of parallel graph-processing algorithms on distributed architectures

Julien Collet

## ► To cite this version:

Julien Collet. Exploration of parallel graph-processing algorithms on distributed architectures. Other [cs.OH]. Université de Technologie de Compiègne, 2017. English. NNT : 2017COMP2391 . tel-01800156

**HAL Id: tel-01800156**

**<https://theses.hal.science/tel-01800156>**

Submitted on 25 May 2018

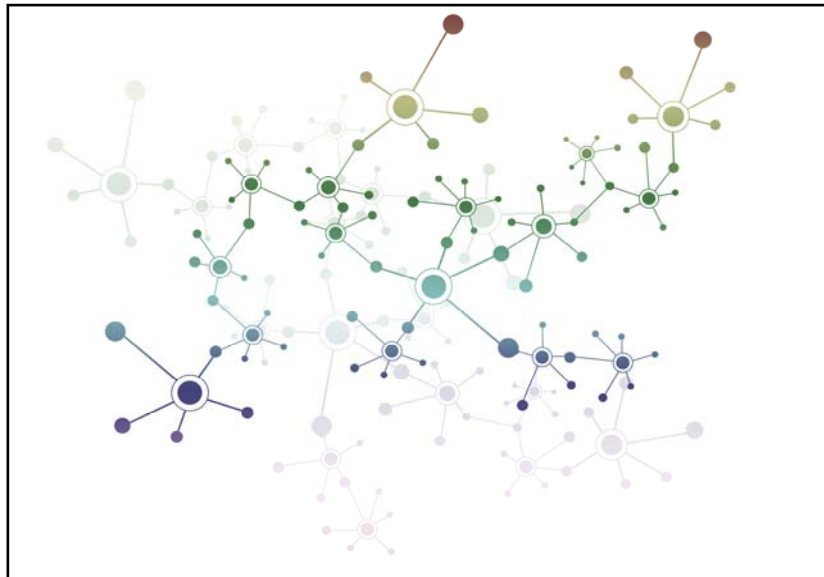
**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Par **Julien COLLET**

*Exploration of parallel graph-processing algorithms  
on distributed architectures*

Thèse présentée  
pour l'obtention du grade  
de Docteur de l'UTC



Soutenue le 6 décembre 2017

**Spécialité** : Technologies de l'Information et des Systèmes :  
Unité de recherche Heudyasic (UMR-7253)

D2391

Thèse de doctorat de  
L'Université de Technologie de Compiègne

---

Département de Génie Informatique  
École doctorale 71, Sciences pour l'Ingénieur

Technologies de l'Information et des Systèmes

---

# Exploration of parallel graph-processing algorithms on distributed architectures

---

PAR Julien Collet

POUR OBTENIR LE GRADE DE  
DOCTEUR de L'Université de Technologie de Compiègne

MEMBRES DU JURY:

**Président du Jury :** Dritan NACE, Professeur à l'Université de Technologie de Compiègne

**Examineur :** Lorène ALLANO, Ingénieur-Chercheur, CEA-LIST

**Rapporteur :** Corinne ANCOURT, Professeur à l'Ecole des Mines de Paris

**Rapporteur :** Didier EL BAZ, Directeur de recherche LAAS-CNRS

**Directeur de thèse :** Jacques CARLIER, Professeur émérite, Université de Technologie de Compiègne

**Co-Directeur de thèse :** Renaud SIRDEY, Directeur de recherche, CEA-LIST

**Encadrant :** Tanguy SASSOLAS, Ingénieur-Chercheur, CEA-LIST

**Co-Encadrant :** Yves LHUILLIER, Ingénieur-Chercheur, CEA-LIST

**Date de soutenance :** 6 décembre 2017



*Je dédie cette thèse à mon  
grand-père, Michel Cottonnet*



# Remerciements

Ce manuscrit de thèse vient clore un chapitre couvrant trois années de travail — et quelque part, bien plus. Je voudrais ici prendre un peu d’espace pour remercier les personnes qui ont contribué à l’écrire. De par leur nombre, il m’est pratiquement impossible d’être absolument exhaustif. Ainsi, je voudrais commencer par remercier et présenter mes excuses à ceux qui ne se retrouveraient pas dans les quelques mots qui suivent.

Tout d’abord, j’aimerais exprimer ma gratitude envers le laboratoire LCE de l’institut LIST du CEA pour m’avoir accueilli en son sein et, en particulier, Raphaël DAVID et Nicolas VENTROUX, chefs successifs du LCE, pour leur bienveillance, leurs conseils avisés et leur sympathie.

Je voudrais ensuite remercier Corinne ANCOURT et Didier EL BAZ pour avoir accepté de rapporter ma thèse, ainsi que Dritan NACE pour avoir accepté de présider le jury de thèse. J’aimerais également adresser de chaleureux remerciements à Lorène ALLANO pour m’avoir permis d’accéder aux plateformes de calcul du LADIS et de découvrir le fascinant domaine de la génomique, ainsi que pour avoir accepté de prendre part au jury en qualité d’examineur.

Aussi, j’aimerais dire ici toute ma reconnaissance à Jacques CARLIER, directeur de thèse, pour son aide continue et ses nombreux conseils avisés tout au long de ces trois années de dur labeur. J’adresse également mes plus francs remerciements à Renaud SIRDEY, directeur de thèse, pour sa disponibilité et sa bienveillance pendant les moments de doute, pour avoir apporté une hauteur de vue nécessaire et avoir été un précieux soutien tout au long de cette thèse.

Il m’est impossible de ne pas remercier ici Tanguy SASSOLAS et Yves LHUILLIER, encadrants de thèse, pour leur disponibilité, leurs conseils et leur implication, mais également leur sympathie, leur tolérance et leur sens de l’humour qui font qu’aujourd’hui la thèse se termine relativement paisiblement. Tanguy, si j’ai souvent pesté — à tort ! — contre ton souci permanent du détail, force est de reconnaître que cette rigueur (dont je manquais probablement un peu en arrivant) m’aura été très précieuse, à chaque instant. Yves, merci de n’avoir jamais perdu patience pendant ces trois ans malgré les sollicitations incessantes — et pas toujours pertinentes, hélas ! —, ton expertise m’aura beaucoup fait progresser durant cette thèse. Enfin, travailler avec vous a été un vrai plaisir, que j’espère partagé. Pour ceci et pour beaucoup d’autres choses encore, je veux vous remercier à nouveau.

Au cours de ces trois ans, j’ai eu l’occasion de rejoindre l’ENSTA ParisTech en tant qu’enseignant-vacataire. À ce titre, je voudrais donc remercier sincèrement Omar HAMMAMI pour sa confiance et pour cette belle opportunité, mais également pour m’avoir donné l’occasion d’exploiter des plateformes de calcul de l’ENSTA dans le cadre de cette thèse.

---

J'ai une pensée toute particulière pour mes collègues futurs docteurs — voire jeunes diplômés pour certains ! — avec qui j'ai l'impression d'avoir bien plus que simplement partagé un lieu de travail pendant plusieurs semaines, mois ou années. Pour commencer, je voudrais remercier Vincent pour les débriefings de réunions de thèses (souvent modérément animés et, parfois, un peu plus), pour l'aide apportée tout au long de ces quelques années et pour tous les bons souvenirs que je garde de ce temps-là — on démarrerait nos thèses ensemble il y a trois ans et, malgré tout, on touche enfin au but : *not bad*. Je tenais également à remercier Joël, nouvellement promu à la meilleure place de l'open-space, celle du *chef* — nos réunions du lundi ainsi que les intermèdes philosophiques (et moins philosophiques aussi) me manquent déjà. De même, j'aimerais remercier Emna, qui est capable de rendre presque supportable une formation doctorale — je n'oublie pas non plus les pépites musicales que tu m'as fait découvrir et qui me hantent encore. J'aimerais également adresser mes remerciements à Johannes — je ne désespère pas d'un jour comprendre, grâce à toi, quelque chose aux réseaux de neurones — et à Jason qui, je l'espère, continuera à promouvoir l'usage unilatéral de *beamer*. Enfin, merci aussi à Mariam, pour notre correspondance très constructive pendant la quasi-totalité de la thèse — et toutes mes félicitations ! Je n'oublie pas non plus les *anciens*, Aziz et Alexandre, qui ont guidé mes premiers pas de thésard au laboratoire — merci pour les franches rigolades et pour avoir fait le déplacement le grand jour.

Je voudrais également adresser mes remerciements à David, pour ses analyses très objectives sur de multiples sujets (sport, cuisine et tourisme en particulier), à Benoit, en souvenir de cours d'anglais mémorables, à Mickaël, pour tous les soucis que j'ai pu te causer en salle serveur, à Thomas, pour les relectures toujours très pointues des chapitres qui composent le présent manuscrit, à Alexandre, pour ses conseils tout au long de la thèse, à Emmanuel, pour avoir toléré mon accent espagnol déplorable, à Mohammed et ses fameux *loupins*, et, enfin, à Gabriel et Enrique, à qui je souhaite également bonne route dans mon désormais ex-bureau. J'aimerais finalement remercier l'ensemble des membres des laboratoires LCE, L3S et L3A que j'ai pu côtoyer, pour l'atmosphère générale qui m'a permis de ne pas (trop) sombrer dans la folie pendant ces trois ans, mais également pour avoir supporté toutes mes obsessions, mes petites manies et mes prises de position — toujours très modérées — avec une grande patience et une certaine bienveillance. J'espère que vous garderez un aussi bon souvenir que moi de ces années.

Je voulais également remercier, sans ordre particulier et pour des raisons différentes, Florian, Faustine, Bastien, Camille, Etienne (si, si), Lucia, Mickael, Weronika, Alice, Norbert, Eleonore, Sophie, Lukáš *and all of you guys at the University of Ostrava*, les *embiddet people*, les amis lyonnais de CPE — quel que soit le côté de l'amphi qu'ils aient occupé d'ailleurs —, sans oublier toutes les personnes qui, par discrétion ou par inadvertance, ne sont pas citées ici. Chacun à votre manière, vous avez donné une belle couleur à ces trois années et je vous en suis reconnaissant. Bien sûr, je n'oublie pas Robin, complice infailible depuis bien trop longtemps déjà — et pour combien de temps encore ?



---

*Last, but not least*, je voudrais remercier ceux qui sont à mes côtés depuis le début, ma famille et, en particulier mes parents, Isabelle et Alain, et mon frère, Thomas. Je sais n'avoir pas toujours été *facile* pendant ces trois années et je veux vous dire encore merci de m'avoir accompagné de la meilleure des façons.

Je termine — enfin ! — sur ces quelques mots, qui prennent (en ce qui me concerne) un sens assez particulier après ces trois années.

*Rien n'est plus difficile pour chacun d'entre nous que de situer ce  
qu'il a fait et de se situer soi-même à sa juste mesure*

— Jean d'Ormesson

---

# Contents

<b>Abstract</b>	<b>1</b>
Abstract . . . . .	1
Résumé . . . . .	2
<b>Introduction</b>	<b>3</b>
Context of the thesis . . . . .	3
Research focus and problematic . . . . .	5
Outline of this manuscript . . . . .	7
<b>1 High performance computers and parallel programming paradigms</b>	<b>9</b>
1.1 Landscape of high performance parallel machines . . . . .	11
1.1.1 Supercomputers and high-end architectures . . . . .	11
1.1.2 Data-crunching many-cores . . . . .	13
1.1.3 Commodity clusters . . . . .	14
1.1.4 Summary . . . . .	14
1.2 Abstractions and design models for parallel programming . . . . .	15
1.2.1 Execution models . . . . .	16
1.2.2 Memory models . . . . .	18
1.3 Towards parallel computing for data-mining . . . . .	20
1.3.1 Integrated Development Environment for data-mining . . . . .	20
1.3.2 Language extension approaches . . . . .	21
1.3.3 Library-based approaches . . . . .	23
1.4 Concluding remarks . . . . .	24
<b>2 Graph-processing at scale on distributed architectures</b>	<b>27</b>
2.1 Graph-processing: Programming models and state of the art . . . . .	29
2.1.1 Programming models: from BSP to vertex-centric programming . . . . .	29
2.1.2 Landscape of vertex-centric frameworks . . . . .	30
2.2 Anatomy of a GraphLab program . . . . .	31
2.2.1 Initialization and input file parsing . . . . .	32
2.2.2 Partitioning of the graph structure . . . . .	32
2.2.3 Graph-processing using the GAS execution engine . . . . .	33
2.2.4 Result output and execution termination . . . . .	36
2.2.5 Synthesis . . . . .	37
2.3 Performance evaluation of GraphLab programs . . . . .	37
2.3.1 Benchmarking off-the-shelf application against real-life performance tuning . . . . .	38
2.3.2 Throughput metrics for operating performance benchmarking . . . . .	39

2.4	Available distributed memory architectures . . . . .	40
2.4.1	Presentation of the compute clusters . . . . .	40
2.4.2	Comparing hardware approaches . . . . .	42
2.5	Synthesis . . . . .	43
<b>3</b>	<b>Practical deployment of parallel graph applications</b>	<b>45</b>
3.1	Program trace analysis . . . . .	46
3.1.1	Algorithms and graph models of computation . . . . .	47
3.1.2	Vertex-centric implementation . . . . .	49
3.1.3	Experimental protocol, materials and methods . . . . .	52
3.1.4	Experimental results . . . . .	52
3.1.5	Synthesis . . . . .	58
3.2	De Bruijn graph filtering for <i>de novo</i> assembly . . . . .	59
3.2.1	De novo assembly of short reads using de Bruijn graphs . . . . .	60
3.2.2	Problem modelization . . . . .	64
3.2.3	DBG preprocessing algorithm . . . . .	67
3.2.4	Vertex-centric implementation of the algorithm . . . . .	70
3.2.5	Materials and methods . . . . .	71
3.2.6	Experimental results . . . . .	72
3.2.7	Synthesis . . . . .	76
3.3	Concluding remarks . . . . .	78
<b>4</b>	<b>Distributed architecture exploration for efficient graph-processing</b>	<b>81</b>
4.1	Assessing performance and operating ranges . . . . .	82
4.1.1	Operating ranges . . . . .	82
4.1.2	Targeting an efficient operating point . . . . .	83
4.1.3	Experimental parameters . . . . .	84
4.2	Comparison of distributed architectures for graph-processing . . . . .	84
4.2.1	Analysis of GraphLab program profiles . . . . .	84
4.2.2	Throughput-based analysis . . . . .	87
4.2.3	Whole-process update rates . . . . .	90
4.2.4	Asynchronous execution performances . . . . .	92
4.2.5	Performance scalability . . . . .	93
4.2.6	Synthesis . . . . .	95
4.3	Throughput-oriented dimensioning of a cluster . . . . .	96
4.3.1	From operating ranges to machine capacity . . . . .	96
4.3.2	Replication factor . . . . .	99
4.3.3	Throughput-based methods for cluster dimensioning . . . . .	99
4.4	Conclusion and perspectives . . . . .	101

<b>5</b>	<b>More efficient graph-oriented cluster design</b>	<b>105</b>
5.1	Flash-based victim swap towards graceful performance degradation . . . .	106
5.1.1	Motivations . . . . .	106
5.1.2	Evaluation of flash-based swap memory . . . . .	107
5.1.3	Perspectives in graph-processing cluster design . . . . .	109
5.2	Microserver-based architectures for efficient graph-processing . . . . .	109
5.2.1	Motivations . . . . .	110
5.2.2	Hardware architecture . . . . .	110
5.2.3	Using GraphLab on ARMv8 architectures . . . . .	110
5.2.4	Single-node operating performances . . . . .	112
5.2.5	Distributed operating performances . . . . .	113
5.2.6	Relevance of ARM-based platform for graph-processing . . . . .	115
5.3	Conclusion and perspectives . . . . .	116
	<b>Conclusions and perspectives</b>	<b>119</b>
	Synthesis . . . . .	119
	Perspectives . . . . .	122
	Short-term . . . . .	122
	Longer-term . . . . .	123
	<b>Personal publications</b>	<b>125</b>
	International conference papers . . . . .	125
	Poster presentations . . . . .	125
	Talks . . . . .	125
	Submitted publications . . . . .	126
	<b>Bibliography</b>	<b>127</b>



# List of Figures

1	TOP500 performances . . . . .	4
1.1	Fork-Join execution model . . . . .	16
1.2	Message-passing execution model . . . . .	17
1.3	MapReduce and BSP execution models . . . . .	17
1.4	Parallel memory models . . . . .	19
2.1	GraphLab Gather-Apply-Scatter model . . . . .	34
2.2	Distributed GAS model . . . . .	35
2.3	GraphLab vertex consistency policies . . . . .	36
3.1	Program trace analysis toy example . . . . .	48
3.2	Input graph datasets characteristics . . . . .	51
3.3	Update rate figures for the three input kernels . . . . .	53
3.4	Whole-process update rate figures for the three input kernels . . . . .	53
3.5	Performances and pagefaults figures for the three input kernels . . . . .	54
3.6	Timing analysis for matmult kernel . . . . .	55
3.7	Operating performances scalability . . . . .	56
3.8	Asynchronous engine execution performances . . . . .	57
3.9	Ingress method performance comparison . . . . .	58
3.10	Read errors in sequencer data . . . . .	61
3.11	$k$ -mer decomposition and associated de Bruijn graph of a sequence . . . . .	61
3.12	Typical sequencing experiment set-up . . . . .	62
3.13	De Bruijn graph edge weight distribution . . . . .	65
3.14	Impact of the sequencing depth on the $k$ -mer spectrum . . . . .	66
3.15	$k$ -mer spectrum of the ID0_40X dataset . . . . .	66
3.16	Edge determination using capacity propagation . . . . .	68
3.17	Error-induced bubble in a 3-mer de Bruijn graph . . . . .	68
3.18	Edge capacity determination and dead-end branch removal . . . . .	69
3.19	Impact of the algorithm parameters . . . . .	70
3.20	Algorithm accuracy with varying error rates . . . . .	73
3.21	Impact of the tolerance interval and the sequencing depth on the accuracy of the algorithm . . . . .	74
3.22	Algorithm convergence for varying error rates . . . . .	75
3.23	Wall execution time for increasing input graphs . . . . .	75
3.24	Speedup observed for three parts of the program . . . . .	76
3.25	Execution time with varying cluster configurations . . . . .	77
3.26	Performances of various distributed cluster configurations . . . . .	77

## LIST OF FIGURES

---

4.1	Profiling figures for the program trace application . . . . .	85
4.2	Profiling figures for the DBG filtering algorithm . . . . .	86
4.3	Throughput performance comparison of the distributed clusters . . . . .	88
4.4	Update rate figures for the DBG filtering algorithm . . . . .	89
4.5	Global performances comparison of three distributed clusters . . . . .	91
4.6	Whole-process update rate figures for the DBG filtering algorithm . . . . .	92
4.7	Asynchronous throughput performances of the HC and LSCC platforms . .	93
4.8	Performance scalability of the LSCC and HC platforms for the trace anal- ysis use-case . . . . .	94
4.9	Per machine update rate . . . . .	97
4.10	Evolution of the replication factor . . . . .	100
4.11	System comparison (program trace analysis) . . . . .	102
5.1	Performance comparison of the victim-swap approach with the baseline . .	107
5.2	Comparison of wall-clock execution time of the victim-swap approach with the baseline . . . . .	108
5.3	Operating performance of the TX2 board on the trace analysis use-case . .	113
5.4	Distributed performances of the TX2-based platform (trace analysis use-case)	114
5.5	Distributed performances of the TX2-based platform (genomic use-case) . .	115



# List of Tables

1.1	Evolution of supercomputer architectures . . . . .	12
1.2	Architectural trends in embedded high-performance manycores . . . . .	13
2.1	Comparison of vertex-centric frameworks . . . . .	30
2.2	Description of the hardware architectures used . . . . .	41
4.1	Unit capacities of the three platforms . . . . .	99
5.1	Description of the Nvidia Jetson TX2 hardware architecture . . . . .	111



# List of Algorithms

1	Algorithm of the parallel input file parser . . . . .	49
2	Algorithm of the program trace analysis vertex-function . . . . .	50



# Abstract

## Abstract

With the advent of ever-increasing graph datasets in a large number of domains, parallel graph-processing applications deployed on distributed architectures are more and more needed to cope with the growing demand for memory and compute resources. Though large-scale distributed architectures are available, notably in the High-Performance Computing (HPC) domain, the programming and deployment complexity of such graph-processing algorithms, whose parallelization and complexity are highly data-dependent, hamper usability. Moreover, the difficult evaluation of performance behaviors of these applications complexifies the assessment of the relevance of the used architecture.

With this in mind, this thesis work deals with the exploration of graph-processing algorithms on distributed architectures, notably using GraphLab, a state of the art graph-processing framework. Two use-cases are considered. For each, a parallel implementation is proposed and deployed on several distributed architectures of varying scales. This study highlights operating ranges, which can eventually be leveraged to appropriately select a relevant operating point with respect to the datasets processed and used cluster nodes.

Further study enables a performance comparison of commodity cluster architectures and higher-end compute servers using the two use-cases previously developed. This study highlights the particular relevance of using clustered commodity workstations, which are considerably cheaper and simpler with respect to node architecture, over higher-end systems in this applicative context.

Then, this thesis work explores how performance studies are helpful in cluster design for graph-processing. In particular, studying throughput performances of a graph-processing system gives fruitful insights for further node architecture improvements. Moreover, this work shows that a more in-depth performance analysis can lead to guidelines for the appropriate sizing of a cluster for a given workload, paving the way toward resource allocation for graph-processing.

Finally, hardware improvements for next generations of graph-processing servers are proposed and evaluated. A flash-based victim-swap mechanism is proposed for the mitigation of unwanted overloaded operations. Then, the relevance of ARM-based microservers for graph-processing is investigated with a port of GraphLab on a NVIDIA TX2-based architecture.

## Résumé

Avec l’explosion du volume de données produites chaque année, les applications du domaine du traitement de graphes ont de plus en plus besoin d’être parallélisées et déployées sur des architectures distribuées afin d’adresser le besoin en mémoire et en ressource de calcul. Si de telles architectures larges échelles existent, issue notamment du domaine du calcul haute performance (HPC), la complexité de programmation et de déploiement d’algorithmes de traitement de graphes sur de telles cibles est souvent un frein à leur utilisation. De plus, la difficile compréhension, a priori, du comportement en performances de ce type d’applications complexifie également l’évaluation du niveau d’adéquation des architectures matérielles avec de tels algorithmes.

Dans ce contexte, ces travaux de thèses portent sur l’exploration d’algorithmes de traitement de graphes sur architectures distribuées en utilisant GraphLab, un framework de l’état de l’art dédié à la programmation parallèle de tels algorithmes. En particulier, deux cas d’applications réelles ont été étudiées en détails et déployées sur différentes architectures à mémoire distribuée, l’un venant de l’analyse de trace d’exécution et l’autre du domaine du traitement de données génomiques. Ces études ont permis de mettre en évidence l’existence de régimes de fonctionnement permettant d’identifier des points de fonctionnements pertinents dans lesquels on souhaitera placer un système pour maximiser son efficacité.

Dans un deuxième temps, une étude a permis de comparer l’efficacité d’architectures généralistes (type commodity cluster) et d’architectures plus spécialisées (type serveur de calcul hautes performances) pour le traitement de graphes distribué. Cette étude a démontré que les architectures composées de grappes de machines de type workstation, moins onéreuses et plus simples, permettaient d’obtenir des performances plus élevées. Cet écart est d’avantage accentué quand les performances sont pondérées par les coûts d’achats et opérationnels. L’étude du comportement en performance des ces architectures a également permis de proposer *in fine* des règles de dimensionnement et de conception des architectures distribuées, dans ce contexte. En particulier, nous montrons comment l’étude des performances fait apparaître les axes d’amélioration du matériel et comment il est possible de dimensionner un cluster pour traiter efficacement une instance donnée.

Finalement, des propositions matérielles pour la conception de serveurs de calculs plus performants pour le traitement de graphes sont formulées. Premièrement, un mécanisme est proposé afin de tempérer la baisse significative de performance observée quand le cluster opère dans un point de fonctionnement où la mémoire vive est saturée. Enfin, les deux applications développées ont été évaluées sur une architecture à base de processeurs basse-consommation afin d’étudier la pertinence de telles architectures pour le traitement de graphes. Les performances mesurés en utilisant de telles plateformes sont encourageantes et montrent en particulier que la diminution des performances brutes par rapport aux architectures existantes est compensée par une efficacité énergétique bien supérieure.

# Introduction

## Contents

---

<b>Context of the thesis . . . . .</b>	<b>3</b>
<b>Research focus and problematic . . . . .</b>	<b>5</b>
<b>Outline of this manuscript . . . . .</b>	<b>7</b>

---

## Context of the thesis

In 1964 was released what is considered to be the very first supercomputer ever, the CDC 6600 [1], sold about \$8 million each and outperforming every other existing computing platforms. The handcrafted machine was able to execute an outstanding 500 thousand Floating Point Operations Per Seconds (FLOPS) and even reach approximately 1 mega-FLOPS. The core architecture was designed around a 10MHz 60-bit processor with 10 so-called parallel functional units, a very early superscalar-like design. This design also allowed the use of a reduced instruction set, which can be seen as a precursor to the RISC processors that followed.

Since the 6600, other supercomputers have been designed, notably to address the growing demand for computing power. In particular, the Cray 1 [2], released in 1976, brought numerous improvements which quickly promoted it as one of the most successful platform in supercomputer history. With the slightly slowing pace of frequency-related and microarchitectural improvements in processors, new orientations where explored to face the continuous increase in processing power requirements and eventually lead to the design of the first massively parallel computers. In particular, Fujitsu released in 1993 its Numerical Wind Tunnel [6] embedding 166 vector processors, later followed by the 2048-core Hitachi SR2201 [5] in early 1996. It was a dramatic contrast with previous platforms, whose core numbers were kept at a limited four to eight configuration, and a first step toward massively parallel computers.

ASCI Red [7], the first supercomputer to be built around off-the-shelf processors was released in 1996. This platform was the first supercomputer to overcome both the 1 TFLOPS and 1 Megawatt marks. This breakthrough set up the commoditization of supercomputers, with ordinary workstations beginning to be assembled in compute clusters to perform large-scale processing. In the early 2000's, thousand-core supercomputers were designed and eventually broke the 1-PFLOPS barrier, notably with the IBM Roadrunner [31]. Great focus was put at the time to improve networking, power consumption and heat management. Roughly ten years later, supercomputer performances reached

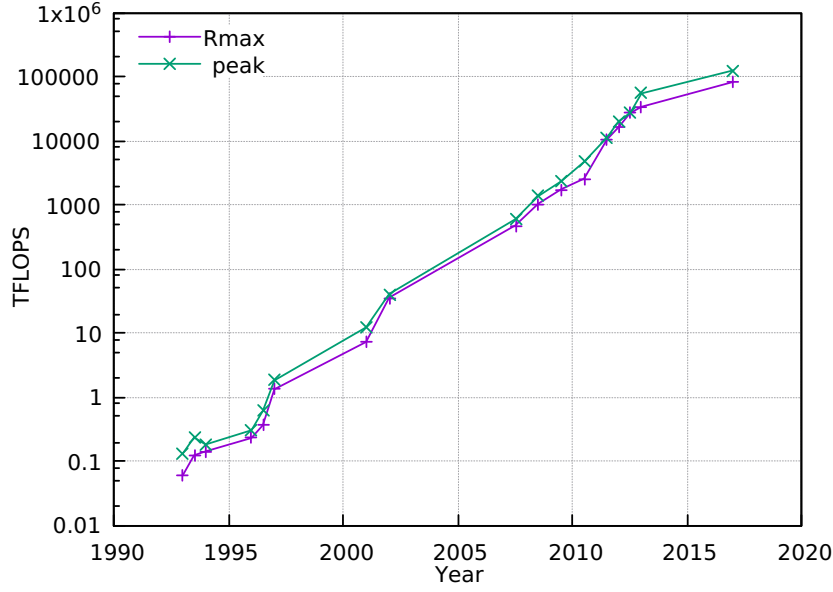


Figure 1: *Peak and achieved performances (logarithmic scale) of TOP500's most powerful supercomputer over time.*

more than 10 PFLOPS for an estimated consumption of 10MW, as visible in Fig. 1, showing the exponential growth in peak performances of the TOP500 [4] list leading supercomputers. Many of the supercomputers previously presented contributed to major advances in scientific computing, modeling and simulations. Indeed, a primary task of such platforms, especially in the 1960-1970 era, was modeling and simulation for the design of nuclear weapons. The range of applications was quickly extended to weather forecasting, aerodynamic computations and molecular dynamics modeling. However, primary customers were also limited in number due to the extensive initial and operating costs, the limited number of relevant applications at the time and the required programming effort. Indeed, to fully exploit such large installations of cores and memory, parallel programming techniques are needed. Hence, efficient development of applications over such platforms is achieved at the cost of a greater complexity.

Meanwhile, new competitors are entering the field of High-Performance Computing (HPC) with the emergence of large-scale data-mining, often referred to as High-Performance Data Analytics (HPDA) or Big Data computing. Indeed, though traditional scientific computing applications have matured along with supercomputers, new applications are joining in from the data-mining area. Data mining is a not so young scientific domain which mostly involves the processing of datasets using statistical or machine learning techniques to discover patterns or knowledge in it — finding a needle (the knowledge) in a haystack (the data). A particularly concrete example of the spreading of data analytics applications is shown by the fact that a growing number of companies nowadays leverage intensive data-mining algorithms to exploit customer data. As an example, exploiting business-related data can help companies to *e.g.* adapt their business to their customer base, improve their internal processes or plan a marketing campaign. Finally,



---

though data analytics is considered a rather mature domain, the term Big Data has only been coined in the late nineties with the growing scale of datasets.

More recently, applications processing data modeled under the shape of graphs have been witnessed. This particular area of data-mining, more formally known as graph-processing or graph analytics, has gained much interest in recent years, with the massive emergence of social networks. More broadly, graph algorithms have proven useful in an increasing number and variety of domains, such as web analysis, security or biomedical applications.

In particular, the ever-accelerating pace of data production has lately transformed data-mining applications into strong peers of legacy HPC applications. These new applications are challenging current architectures, notably due to the required memory amounts and the changing nature of their execution behaviors, which is contrasting with more traditional HPC workloads. Data mining applications — and even more strikingly, graph analytics — have long been known for being data-dependent, irregular and hardly predictable. This becomes a major issue when it is required to deploy them efficiently on distributed architectures at a large-scale.

A consequence of this shift in hardware architectures used in data-mining is the increased programming challenge. Indeed, it requires the end-user, presumably a data practitioner, to endorse multiple responsibilities at the same time, in addition to being a specialist in its field. In order to address the memory requirement, the user must be able to appropriately size its hardware platform, that is, choosing the appropriate amount of compute nodes and per-machine available memory. Then, in order to deploy the algorithm successfully, he must develop system administrator skills to be able to cope with tasks related to cluster management (*e.g.* handling node failure, network congestion, system update or process distribution), and finally, to fully benefit from the cluster, he should also be a parallel and distributed programming expert. Moreover, data analytic applications evolve at a fast pace and as a consequence, programmers seldom allocate time for the development of an "optimal parallelization" as lifecycles of such applications are shorter in comparison to those of the scientific computing domain. Indeed, data practitioners require a way to quickly implement their algorithms and deploy them seamlessly on a distributed architecture, while benefiting from the performance brought by ever-scaling cluster architectures.

## Research focus and problematic

In this thesis work, we focus on large-scale graph analytics workloads deployed over distributed-memory architectures. With the advent of Big Data, HPC tools and architectures are more and more often necessary to match the processing/memory requirements of data-mining tasks, hence the term of High-Performance Data Analytics. As graph analytics algorithms are known for being particularly data-dependent, irregular and un-

structured, achieving a scalable parallel implementation on distributed architectures is a particularly challenging task. Though programming frameworks have been proposed to address this programming issue, their scalability and performance behaviors are non-trivially assessed. Moreover, the fast pace at which data-mining applications evolve makes traditional application/dataset benchmarking approaches hardly applicable. In such a situation, choosing an adequate hardware resource becomes challenging. However, being able to properly design a cluster architecture is extremely important to enable the efficient processing of large volume of data. Such ability can help as well, for a given problem, to optimize a cluster setting for maximum compute performances or energy efficiency. Thus, it is also crucial to have means to understand how performances evolve with increasing dataset and cluster sizes, as it gives user hints to appropriately design its cluster for its application needs. In view of those considerations, we hence address the points of understanding performance behaviors of distributed graph analytics workloads implemented using a dedicated library and propose an approach toward scalability/performance evaluation of such applications. The aim of this work is eventually to adumbrate guidelines for the appropriate design of clusters in the context of HPDA applications.

To achieve the aforementioned objectives, we decided to focus on two real-life graph-related problems rather than numerous synthetic applications usually used for benchmarking purposes. Indeed, in the context of data-dependent performances (*i.e.* performances vary with data size and properties) leveraging well-known algorithms on random scale-free graphs may bring less genuine insights on the real-life performances of a system. For each problem, we provide an in-depth and practical presentation of the novel algorithm we propose, its distributed implementation and the measured performances using a dedicated benchmarking approach. We argue the contrasting natures of the considered problems make them relevant and enable, *in fine*, the gathering of more global insights on performances with respect to software aspects.

We also conducted a performance-oriented comparison of three different hardware platforms based on the use-cases we worked on. These studies highlight inefficiencies in current high-performance architectures, questioning their usage for such purposes notably when put in perspective with their cost. Finally, we propose some cluster design guidelines and architectural considerations to conceive the next generation of high-performance graph-processing platforms. In particular, we evaluate the relevance of emerging low-power embedded computing platforms in such a context.

To conclude, we argue this work is of much relevance to the following audiences. Data practitioners, and in particular, programmers in the field of graph analytics may be interested by practical aspects of deploying GraphLab implementations, as presented in Chapters 2 and 3. We consider that the benchmarking approaches introduced in Chapter 2 and leveraged throughout this work in particular in Chapters 3 and 4, can be useful as well. Indeed, understanding the behavior of graph-processing applications is of much importance in a field such as graph-mining, where dataset sizes are increasing.

---

Additionally, being able to adequately design a cluster for a dedicated workload can bring numerous improvements in terms of compute resources availability or energy efficiency. Finally, we argue that the outcomes of the work provided in Chapters 4 and 5 can lead to visible improvements in the hardware architecture design of more efficient distributed clusters. On a more domain-specific point of view, both use-cases studied in this work constitute relevant and novel contributions to their relative fields and thus can be of interest for their applicative aspects.

## Outline of this manuscript

This manuscript is constituted as follows. Chapter 1 starts with a state of the art in current architectural trends in High-Performance Computing. High-end compute servers, derived from large supercomputing installations seem a weapon of choice for any kind of large computations. However, their cost of ownership and operation may be out of reach, not mentioning the large programming gap required to fully exploit the complex hardware provided. On the other side, many-cores with hundreds of cores in a chip seem promising considering their processing power to energy figure, but they usually come with limited amount of memory and/or a high programming gap. In between, assembling clusters of inexpensive workstations is an interesting trend exhibiting promising performances with respect to its price. Mainstream programming models of interests are discussed and reviewed, as with parallel software tools for data-mining applications. A striking inadequacy between traditional HPC tools (complex, general-purpose) and data-mining requirements (shortened development time, domain-specific semantic) led to the emergence of new domain-specific programming paradigms of many forms. In the particular case of graph analytics, a large variety of approaches has been proposed and, amongst them, domain-specific libraries seem a fairly interesting compromise between ease of use and relevant performances.

As we focus on graph-mining tasks, Chapter 2 introduces mandatory notions in parallel graph-processing programming and details the experimental environment leveraged in this thesis work. In particular, vertex-centric frameworks are introduced and GraphLab, the library we use, is presented in details along with our experimental hardware platforms. This chapter addresses as well the critical issue of assessing and measuring performances of graph-processing systems. More precisely, metrics used in this work and benchmarking aspects of the experiments are discussed in this context. Indeed, graph-processing benchmarks are often constructed around a synthetic scale-free graph generator and a graph algorithm. Though this approach enables a performance comparison between architectures, it hardly gives hints on the behavior of real-life use-cases.

Thus, we decided to build our study on two real-life applications as we argue that the practical experience of implementing and deploying them is valuable. Then, Chapter 3 details the two real-life use-cases we developed and deployed on our experimental systems

using GraphLab. The first application presented is a program trace analysis algorithm, while the second application is a graph-filtering algorithm for genomic data. In particular, we study operating performances and scalability aspects, leveraging the experimental context and metrics previously presented.

Having presented the algorithms and compared their individual performance properties on relevant architectures, gathering software-related insights on performance behaviors, we conduct a comparison of all available hardware systems in Chapter 4. The goal of this chapter is to highlight the efficiency of the available platforms in order to provide more hardware-related insights on the assessed performances. Using performance figures and traditional metrics, we show how such analysis can help in tailoring cluster architectures to a particular workload, which is strikingly relevant in a domain where datasets vary in size and properties, hence affecting the system’s throughput.

Chapter 5 discusses more general aspects and perspectives regarding distributed memory cluster design in the context of graph-mining. We propose in this chapter a hardware approach called victim-swap, towards the mitigation of the performance decrease observed when all the memory available in the node is saturated. Indeed, we observed notably that often, a particularly interesting operating point is set near the saturation point of the system memory, increasing the risk of observing swap in/out operations slowing down computations abruptly. We show that by leveraging such cost-effective approach, this risk can be mitigated, hence making it an interesting feature for the next-generation of servers for graph-analytics. Then, having previously studied the performances of well-established platforms and going further in hardware propositions, we provide a study of the relevance of emerging embedded computing platforms or microservers. Indeed, though we expect a performance degradation with respect to high-end processors, it is of particular relevance to assess if it can be mitigated by an unmatched energy efficiency in the context of graph analytics — a domain known to be rather memory-bound than compute-bound.

Finally, research perspectives of this work are detailed and presented after a general conclusion is provided.

# High performance computers and parallel programming paradigms

## Contents

---

<b>1.1</b>	<b>Landscape of high performance parallel machines . . . . .</b>	<b>11</b>
1.1.1	Supercomputers and high-end architectures . . . . .	11
1.1.2	Data-crunching many-cores . . . . .	13
1.1.3	Commodity clusters . . . . .	14
1.1.4	Summary . . . . .	14
<b>1.2</b>	<b>Abstractions and design models for parallel programming . .</b>	<b>15</b>
1.2.1	Execution models . . . . .	16
1.2.2	Memory models . . . . .	18
<b>1.3</b>	<b>Towards parallel computing for data-mining . . . . .</b>	<b>20</b>
1.3.1	Integrated Development Environment for data-mining . . . . .	20
1.3.2	Language extension approaches . . . . .	21
1.3.3	Library-based approaches . . . . .	23
<b>1.4</b>	<b>Concluding remarks . . . . .</b>	<b>24</b>

---

The race for processing performances ignited the development of parallel machines and highly efficient parallel implementations. High-Performance Computing (HPC) has been gathering applications requiring large-scale resources such as clusters of computers since the sixties. The field of scientific computing has provided historical HPC applications, as for example physics simulations or partial differential equation solvers. Since, these optimized implementations have been developed and successfully deployed across large-scale distributed architectures — a required burden in order to fulfill the compute power requirements of these applications.

Programming a distributed computer raises substantial programming challenges, such as communication management or data placement which are parallelization-related issues. In particular, the efficient design of a parallel program requires an in-depth understanding of the underlying hardware and software architectures. The HPC community provided numerous approaches to address these parallelization problems. Amongst them,

the Message-Passing Interface (MPI) [62] has emerged as the standard for parallel applications on distributed architectures using explicit message passing. An MPI program is usually constituted of many processes distributed across compute nodes performing computations on their private data, only communicating through explicit messages as needed. Though MPI grants the programmer full control over communications, manually handling communications within a program is a non-trivial task and requires an extensive programming effort.

Orthogonal to MPI, another standard has appeared for the parallelization of programs, namely OpenMP [8]. OpenMP allows a fast, user-friendly parallelization of sequential programs by leveraging multithreading over a shared memory view. Indeed, a small set of pragmas are used to automatically parallelize loops in the code, thus hiding the complexity of thread management and allowing an incremental parallelization of sequential sources. An OpenMP program is organized around sequential and parallel regions. Within sequential regions, only the master thread is active. This thread is then forked in parallel region, where spawned workers communicate implicitly through shared objects. Despite being more user-friendly, OpenMP does not scale-out well over non-shared memory architectures due to the high cost of maintaining data consistency across physically separated memory spaces.

More recently, data-mining applications started to claim the title of HPC use-cases, as they require more and more frequently high-performance architectures to manage their ever-increasing dataset size, hence coining the term of High-Performance Data-Analytics (HPDA). In particular, with the rise of graph data in many domains including social networks, security or bioinformatics, graph analytics algorithms gained interests. Though these applications are considered being particularly difficult to deploy efficiently on parallel architectures — notably due to the irregular nature of graph data — data practitioners need now to scale-out their graph analytics workloads. Contrary to the scientific computing, which is now a relatively mature and resourceful domain having deployed fine-tuned distributed MPI applications at scale, the need for large-scale graph-processing implementations is more recent. Unfortunately, none of the aforementioned parallelization approaches (multithreading with OpenMP or message-passing with MPI) is a completely satisfying answer for HPC newcomers because of the scalability or the complexity issues. Indeed, data scientists need now to work at multiple levels to develop an efficient implementation of their algorithm: they need to understand the matching between their algorithm and their data but also, and perhaps more importantly, the underlying hardware. Moreover, these issues are emphasized by the fast pace at which data-mining applications evolve, which is in contrast with decade-long lifecycles of most scientific codes.

This chapter is organized as follows. In the first section, current architectural trends in the field of High-Performance Computing are discussed in order to understand to which extent they can be seen as a hardware platform of choice in the context of High Performance Data Analytics (HPDA) and, in particular, large-scale graph analytics. Then, in

Section 1.2, mainstream parallel execution models are detailed, followed by a description of associated relevant memory models. Finally, programming frameworks for data-mining applications are introduced and discussed, before a conclusion is drawn at the end of the chapter.

## 1.1 Landscape of high performance parallel machines

In this section, we review the main hardware trends in distributed parallel architectures. First, we explore architectures of extreme scale installations seen in HPC centers and high-end compute servers inspired by such machines. Then, massively parallel many-cores are presented. Finally, the use of commodity desktop workstations linked in cluster is discussed with respect to large-scale distributed computations.

We focus particularly on node architecture in terms of processor micro-architecture, memory amount or hardware accelerators as it has fueled considerable amounts of specializations that impacted programming. In comparison, networks and interconnects are quite transparently interoperable from the point of view of the user. Hence, we take the assumption that the network is a uniform, reliable crossbar communication network.

### 1.1.1 Supercomputers and high-end architectures

The study of the state of the art in the field of high-performance computer architecture shows a great deal of variety. Historically, High-Performance Computing started with the emergence of supercomputers, or large powerful computer infrastructures exhibiting astonishing performances at the cost of a staggering complexity and specificity of each architectural solutions. Hence, efficiently exploiting such installations implies a great programming challenge. Nowadays, most supercomputers are built with high-end general-purpose multiprocessors featuring heterogeneous accelerators (such as co-processors [63] or GPU [86]). Still, such hardware accelerators have already proven useful with existing scientific computing applications, *e.g.* Monte-Carlo numerical simulations for stratospheric balloon envelope drift descent analysis [Plazolles2017], where the massive available hardware parallelism could be fruitfully wielded by the application. More rarely, some supercomputers rely on specific, off-the-track, computer hardware such as the Cray XMT system [29] with its massively multithreaded processors providing 128 hardware instruction streams where usual simultaneous multithreaded (SMT) cores usually provide up to 16 hardware threads each [21, 82, 86, 110].

IBM Bluegene’s architectures — a successful example of a high-end architecture — often top supercomputing ranks on various lists such as Top500 [4], Green500 [28] and Graph500 [45], which evaluate respectively floating point, power-efficiency and graph-processing performances. Architecture-wise, a typical BlueGene/L node embeds two PowerPC440 running at 700MHz [21] and a more recent BlueGene/Q has a 16-core

System	Nodes	Cores	Node architecture	Memory (per-node)	Rank
TaihuLight (2016)	40,960	10,649,600	1x SW26010 260C	32GB	Top500: 1 Graph500: 2 Green500: 17
Tianhe-2 (2013)	16,000	3,120,000	2x 12-core Intel Xeon Ivy Bridge 2.2GHz 3x Xeon Phi	64GB 24GB (Xeon Phi)	Top500: 2 Graph500: 8 Green500: 147
Titan (Cray) (2012)	18,688	299,008	1x 16-core AMD Opteron 6274 2.2GHz 1x Nvidia K20x	32GB 6GB (K20x)	Top500: 4 Graph500: N/A Green500: 109
Sequoia BlueGene/Q (2012)	98,304	1,572,864	1x 16-core PowerPC A2 1.6GHz	16GB	Top500: 5 Graph500: 3 Green500: 100
K Computer (2011)	88,128	705,024	1x 8-core SPARC64 VIIIfx 2.0GHz	16GB	Top500: 8 Graph500: 1 Green500: 277

Table 1.1: *Evolution of supercomputer architectures. Though recent installations have shown complex node architectures embedding co-processors and GPUs around a large-many cores, the leading Graph500's architecture shows a much simpler architecture, without hardware accelerators. Top500, Graph500 and Green500 lists as of June 2017.*

PowerPC A2 running at 1.6 GHz. In comparison, a typical node from Tianhe-2 [82], one of China's largest supercomputers, embeds two Intel Xeon Ivy Bridge processors running at 2.2GHz and three Xeon Phi co-processors [82], a state of the art co-processor [63, 111] embedding dozens of x86 processors. This results in a grand total of 3, 120, 000 available cores in the full-scale Tianhe-2 installation. The Titan supercomputer from Cray in the National Oak Ridge Laboratory, another state of the art supercomputer, features a 16-core AMD Opteron plus a Nvidia Tesla K20 GPU per node [86].

More recently, for the first time, the TaihuLight [101], the June'17 TOP500 leading supercomputer, embeds chinese-design processor. Indeed, it features a considerably simpler architecture with a single 1.45GHz, 260-core SW26010 processor and 32GB of memory per node. Interestingly enough, supercomputer nodes seem to evolve towards such simpler architectures, a trend anticipated by IBM with its BlueGene machines or by the K-computer, as visible in Tab. 1.1. Such design choice is particularly relevant in the context of graph-processing, as highlighted by their Graph500 results, where the top three machines have straightforward architectures with no hardware accelerators or GPU.

Running and maintaining such heavy infrastructures involve high operational expenditures and may be out of reach for many. However other options are available such as renting compute time slots on open supercomputing centers or through research frameworks. PRACE [98], the European program for advanced computing is an example of such supercomputing center association granting access to high performance platforms.

Such supercomputers have also driven the evolution of high-end server architectures at a more moderate scale. Typical racks embed 2 to 16 high-end processors with terabytes of storage memory [103, 105] with, optionally, hardware accelerators or GPUs. Although more affordable than extreme-scale HPC installations, they usually come at a high cost



of ownership, making them unsuitable if expected performances are not guaranteed as in data-dependent and irregular workloads. However, historical web hosting services also offer such dedicated HPC architectures for rent [97] at a more affordable cost, enabling performance benchmarking prior to any longer-term investment.

Even though these architectures can be accessed or built at a more moderate scale matching a smaller demand, the complexity of such clustered architecture is rather high. This complexity greatly increases the programming challenge that users are dealing with. Indeed, extracting the highest performances on such high-end nodes embedding accelerators is not easily reached in the context of data or graph analytics.

### 1.1.2 Data-crunching many-cores

On the other side of this architectural spectrum, embedded distributed platforms are also available, featuring dozens of processing elements gathered on a single-chip distributed computer. Such platforms can be of huge interests for the processing of large datasets when taking into considerations their low power capabilities. Examples of such architectures include the MPPA many-core, Epiphany-IV and TileGx, detailed in Tab. 1.2.

The MPPA-256 [65] many-core from Kalray is dedicated to massively parallel embedded applications and features up to 256 32-bit cores in a chip, clustered in bulk of 16 cores. The MPPA2-256 (Bostan) [106] was later introduced, with up to 288 64-bit programmable cores. However, though promising, both architectures are more suitable for time-critical or networking applications [92] rather than large-scale data applications.

Adapteva’s Epiphany IV [88] can scale from 64 to 4096 cores interconnected by a 2D grid network but processing elements are limited in terms of arithmetic operations and memory per core. Indeed, systems presented in [88] do not exceed 1GB per board, which shall be extended for larger scale data analytic workloads. Moreover, in this purely distributed architecture, the on-chip memory management is leaved entirely to the programmer, increasing the programming burden.

The TileGx processor family is another example of embedded architecture ranging from 9 to 72 processing elements. Conversely to the Kalray and Epiphany architectures, the TileGx provides full cache-coherency across the entire chip [77], at the expense of a more limited scalability.

System	Number of cores	Cluster size	Core architecture	Memory management
Kalray MPPA2	256	16 cores	VLIW core, 800MHz	Explicit
Tilera TileGx	9-72	N/A	VLIW core, 1.2GHz	Cache coherence across the chip
Adapteva Epiphany-IV	up-to 4096	N/A	RISC core, 800MHz	Explicit

Table 1.2: *Architectural trends in embedded high-performance manycores. The TileGx architecture, contrary to the others, provides cache coherence across the whole chip, but this capability explains its limited scalability with regards to the MPPA and Epiphany architectures.*

Though all these architectures foster energy efficient parallel computing, they require dedicated development suites in order for programmers to be able to successfully deploy their implementations. Moreover, notwithstanding the effort required to master such specialized toolchains, it is also a tough task to understand how to adequately implement an application in order to fully exploit the architectural specificities of these platforms. Finally, not all libraries and software tools are fully supported by other processors than the traditional architectures encountered in servers or workstations and the porting task can be demanding and time-consuming.

### 1.1.3 Commodity clusters

The previously presented architectural trends show that the high-end HPC world is quite heterogeneous and programming at a large-scale on these machines is a cumbersome task [70, 80]. Moreover, previously outlined architectures are either lacking library or compilation support (embedded-oriented many-cores) or unreachable for many due to their high cost of ownership (HPC supercomputers). Acknowledging this context, commodity cluster architectures are emerging in between these two trends, with in mind to leverage more processing power and memory at a more affordable cost. Indeed, desktop workstations nowadays embed 4 or 8 cores and sometimes up to 16 or even 32 Gigabytes of memory. Yet, some datasets may still not fit in their memory, thus requiring the clustering of workstations around, *e.g.* an Ethernet network to further expand memory and processing capabilities. In this context, commodity clusters can be seen as an intermediate solution of interest notably because their performance/price ratio can be significantly higher than traditional high-end clusters.

Such approach has been successfully adopted by Google [12] with an architecture combining thousands of desktop workstations linked across Ethernet networks. As explained by Barroso *et al* in [12], commodity clusters have many advantages such as *e.g.* the initial cost of ownership of compute nodes, which is orders of magnitude lower than high-end systems. Though reliability in commodity clusters is an issue compared with high-end architectures, it can be mitigated in software using dedicated monitoring systems. Moreover, at the scale of hundreds or thousands of machines, the burden of maintaining an architecture is equivalent regardless of the kind of architecture used (high-end or commodity). In the particular case illustrated in [12], reliability is actually software-managed and repair operations are batched in order to keep a low operating cost.

### 1.1.4 Summary

In this section, we detailed the three main architectural trends observed in large-scale distributed processing. Historically, large-scale systems based on dense high performance nodes have proven, despite striking peak performances, to be out of reach for many data-mining applications with respect to the programming gap or the operating/acquisition

cost wall. On the opposite side, clusters of embedded manycores can be a path of interest thanks to their relatively high energy efficiency which can make acceptable their relative computing performance decrease compared with high-end machines. However, a sufficient amount of memory as with a fair share of mainstream programming tools and libraries should be provided in order to make them hardware platforms of choice as large data analytics platforms.

Finally, distributed systems based on inexpensive desktop workstations have raised some interests, notably in the context of extreme scale web-related applications. They seem an interesting path for the efficient deployment of applications with hardly predictable, data-dependent behaviors as they provide an architecture simple to exploit with a great amount of available programming toolsets. However, though commodity clusters may be significantly easier to leverage as they rarely features complex hardware accelerators (*e.g.* GPU or Co-processor), programmers still face an important challenge materialized by parallel programming issues such as data placement or communication management. Indeed, an in-depth understanding of parallel programming models is required to overcome such programming issues and fully exploit the provided hardware. Hence, the following section addresses the state of the art of relevant parallel programming models in the context of data analytics.

## 1.2 Abstractions and design models for parallel programming

Usually, parallel programs are organized around multiple threads of execution processing data stored in memory and possibly communicating. In this context, the way computations and communications are handled is a concern of high relevance, as with the memory abstraction offered to the programmer.

The design of a parallel program is made even more difficult by the variety of programming models available in the field. In particular, though some approaches may only differ by the syntax provided by the implementation, many approaches differ in fundamental parallelism concepts which have to be well understood for the developer to make sound implementation choices. These choices of paradigms may as well have direct impacts on the degree of freedom of parallel execution given to the programmer, impacting scheduling or memory use.

Additionally, the choice of a particular model is resulting from a trade-off between aspects taken care of by the programmer, by the compilation toolchain and by the runtime. Such trade-off is managed through programming abstractions exposing or hiding mechanisms to/from the programmer (*i.e.* increasing or decreasing the programming challenge at a cost) and through implementations sharing tasks between compilers and runtime systems.

In the following sections, we explore two crucial aspects of parallel programming abstractions : execution and memory models.

### 1.2.1 Execution models

Execution models state how the execution of a program takes place. In the context of parallel programming, we introduce mainstream execution models in the field, in the following section. These models describe how multiple workers or threads are orchestrated and how the communications are handled in parallel programs. These are aspects of parallel programming of much relevance as they impact both programmability (programming language syntax or constructs) and performances (appropriateness between models and algorithms).

Fork-join is an execution model notably implemented by OpenMP. In a typical fork-join program, the execution of an application can be represented by a master thread forked inside parallel regions where data are processed concurrently by independent workers or lightweight userspace threads who join at the end of the parallel region, as shown in Fig. 1.1. Though this model has proven to be efficient and easy to use with OpenMP, a specific and non-trivial investigation should be held while taking care of shared variables such as reduction objects in order to extract relevant performances. A typical pathological example is an OpenMP parallelization of the histogram problem [57].

Single-Program Multiple-Data (SPMD) is another model in which multiple copies of the same program are independently launched in parallel and communicate through messages. In particular, MPI programs are often based on a SPMD model, with independent processes being launched on different machines, as illustrated in Fig. 1.2. In fact, the flexibility of the SPMD model is a major advantage: as it does not require centralized control (*i.e.* a master thread in the sense of the Fork-Join model), it can be implemented on both shared and distributed memory in an efficient and scalable way. Nonetheless, the downside of this flexibility lies in the fact that, when complex communication behaviors are expected, programming an MPI application can become cumbersome and error-prone

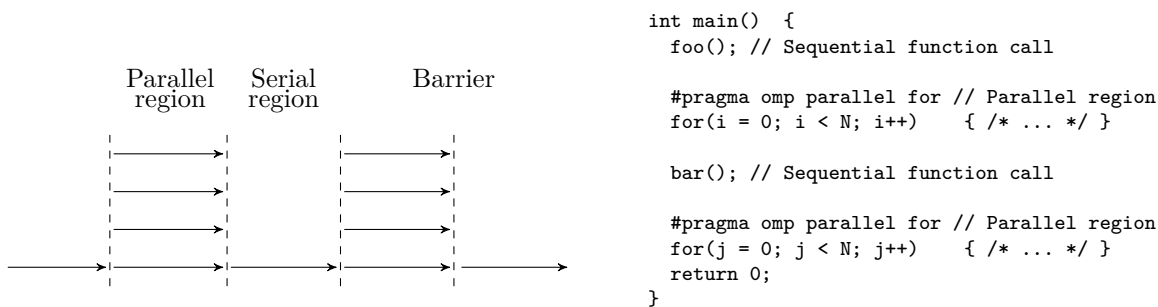


Figure 1.1: The Fork-Join multithreaded execution model (l.) and associated OpenMP pseudo-code (r.). The execution of the program is separated in sequential and parallel regions, delimited by pragmas in the code. Communications between threads are performed through the use of shared variables in the global memory space. A synchronization barrier is used at the end of parallel regions when threads join.

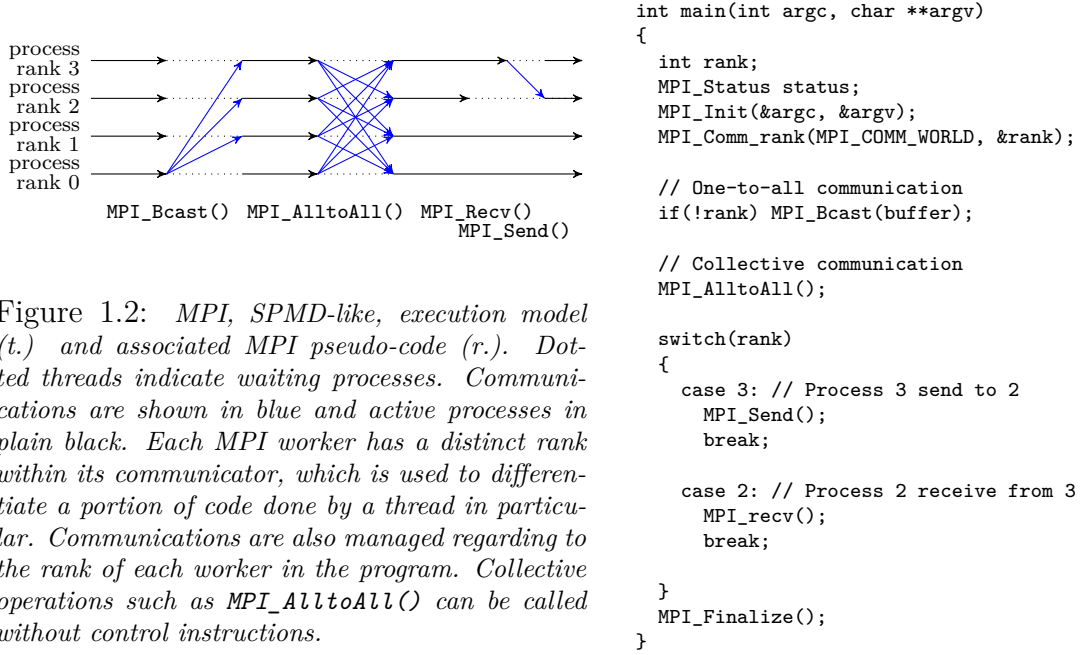


Figure 1.2: *MPI, SPMD-like, execution model (t.) and associated MPI pseudo-code (r.). Dotted threads indicate waiting processes. Communications are shown in blue and active processes in plain black. Each MPI worker has a distinct rank within its communicator, which is used to differentiate a portion of code done by a thread in particular. Communications are also managed regarding to the rank of each worker in the program. Collective operations such as `MPI_AlltoAll()` can be called without control instructions.*

as every transaction should be manually handled. Yet, the fine grain control over communication leaves room for optimizations.

MapReduce is a programming model in which data are processed in two steps using the following operators: A Map operation which splits the data and maps them to the workers and a Reduce operation where results are gathered into a more condensed form [83]. More precisely, the Map function takes input points and outputs a set of  $\{key, value\}$ . The MapReduce runtime then groups values by keys (“Shuffle” part in Fig. 1.3 (l.)) and forward them to the Reduce function which takes a key and the set constituted of all associated values. It then applies a kernel function to obtain an usually smaller set of data.

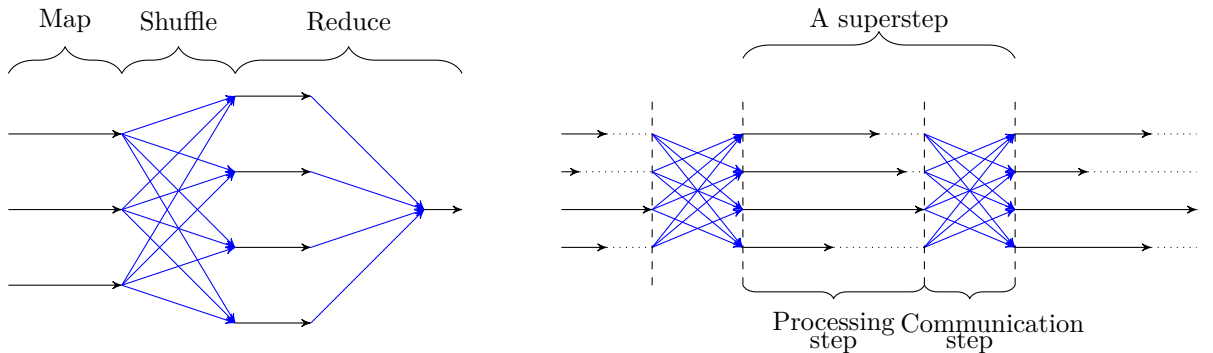


Figure 1.3: *The MapReduce (l.) and BSP (r.) model. The first part of a MapReduce program is a sort of the items to find an appropriate mapping. Then a shuffling phase starts and data are mapped to their reducer. Finally, the reduce step condense the data into a compact form. In the iterative BSP model, each superstep includes distinct processing and communication steps. Workers may be starving while waiting to be allowed to communicate.*

MapReduce is well-suited for embarrassingly parallel programs (*e.g.* problems which can be easily decomposed onto many workers and processing can be done independently). However, it appears to be inefficient when dependencies between computations arise, as for example with machine learning algorithms which satisfy the Statistical-Query model [43] or when multiple iterations over a dataset are required. Famous implementations of MapReduce include Google’s MapReduce or Apache Hadoop, and were designed to be fault-tolerant and scalable. Yet, the fault tolerance mechanisms embedded in the aforementioned implementations are costly and safety is here obtained in spite of pure compute performances.

In 1990, Valiant [3] introduced the Bulk-Synchronous Parallel execution model (BSP). In this model, concurrent threads process data over so-called supersteps, as shown in Fig. 1.3 (r.). One superstep is divided into two phases: a processing step where threads are independently executed in parallel and a communication step where threads are synchronized before being allowed to communicate altogether. A critical issue with this model is load-balancing. If some threads run faster than others, they will be stalled and waiting to be allowed to communicate — hence resulting in a waste of compute resources. A program implemented in the BSP model could benefit from load-balancing policies such as work-stealing, allowing starving threads to help other threads to complete their tasks faster. Additionally, some domain-specific models inspired by BSP were created to match the requirements of the applicative domain, such as for example with the Gather-Apply-Scatter model [61] which is a specialized BSP model for graph-processing applications.

Finally, the data-parallel model is a perfect match for parallel problem in which every unit of data can be process in total isolation from the other data (*e.g.* in some image processing algorithms). In this model, which can be seen as a communication-less SPMD model, data partitioning is static and consists only in slicing the input dataset into chunks mapped onto workers. Then, attention must be paid to work-balancing by providing equally-sized chunk to workers in a performance-homogeneous environment.

Though understanding the execution behavior of a parallel program is of much importance, the knowledge of how the memory is exposed to the programmer is of equal interest. Indeed, except in data-parallel programs, most parallel applications have to handle communications of data between threads using direct memory sharing or messages. In the following subsection, we review mainstream memory models in the context of parallel programming.

### 1.2.2 Memory models

In parallel programs, threads or processes communicate mostly by two means: explicit messages through networks or by using shared spaces of memory, hence requiring explicit synchronization mechanisms. Moreover, though it has a great impact on programmability, the choice of a memory model may also impact performances and requires additional software layers or runtimes in cases where the selected model does not match the under-

lying architecture (*e.g.* a shared memory over a distributed memory architecture). In the following, the mainstream memory abstractions, summarized in Fig. 1.4, are presented.

In terms of memory models, the landscape is bounded by two opposite views: A fully distributed view where processes all have independent and private memory spaces, enforcing message-based communications and a flat view where every thread works in the same, shared address space.

The shared memory model matches perfectly multithreaded applications executed on a multi-core flat-memory machine. In such view, every threads of the application have access to the whole address space, hence relieving the need for explicitly managed communications, facilitating the programming. Indeed, communications are performed through shared objects directly available to every thread. However, the programmer must manage potential conflicts using dedicated semantics to ensure hazard-free variable accesses. Oppositely, the distributed model prevents the need for such mechanisms as communications between threads or processes is fully made explicit by the use of a messaging system.

Between these bounds have emerged other models such as the Partitioned Global Address Space (PGAS) where each thread has a private local space and a partition of the shared space. The collection of all shared partition constitutes the shared globally-addressable memory space. This approach tends to balance the best aspects of both shared and distributed memory. It is for example implemented by high-productivity languages such as Co-Array Fortran, X10, Titanium and UPC. An extension to the PGAS model,

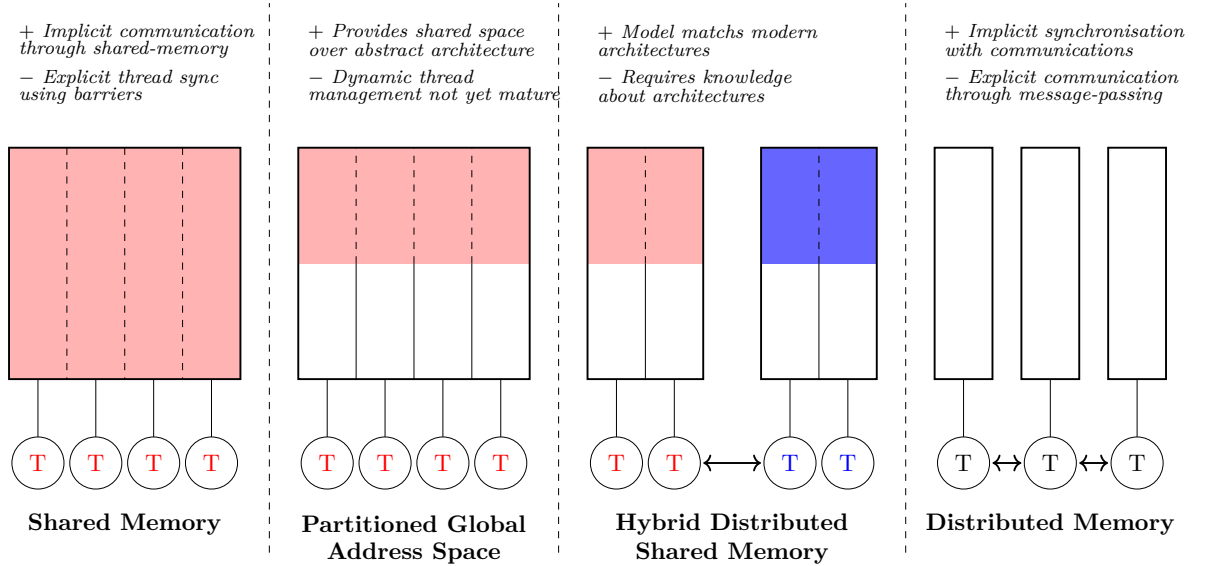


Figure 1.4: Comparison between main memory models in parallel processing applications. Colored areas show Uniform Memory Access (UMA) zones. White areas indicate private memory spaces. Arrows indicate message-passing channels. The leftmost model describes a shared memory model in which every thread has access to the whole address space. The Partitioned Global Address Space (PGAS) model illustrates a model in which threads have a local memory divided into a private space and a shared partition of a global address space. The third model exhibits an hybrid distributed-shared memory where the memory space is distributed at a coarse grain, but shared between threads at a finer grain. Finally, the rightmost model shows a purely distributed memory model. In this model, each thread has a private local memory and share data using explicit messaging.

namely the Asynchronous PGAS (APGAS), adds the notion of “places” which is a movable and coherent collection of data and associated processing routines.

Finally, to address the hierarchical nature of NUMA-based clusters, so-called hybrid programming approaches try to mix shared-memory parallelism within a node with message-passing for inter-node communications [15, 17]. These projects used mostly a mixed OpenMP+MPI approach, which is a complex programming style as these two standards were not developed to be intertwined. Another study [67] leveraged new shared-memory possibilities integrated in MPI-3 in lieu of OpenMP to address this issue. However, the programming challenge provided to programmers remains important and cannot be neglected, especially in the context of data-mining applications calling for shortened development times.

This programming challenge is an issue of particular interest for HPC-newcomers as data-mining algorithms are often irregular and data-dependent. Moreover, the rapidly changing nature of data analytics algorithms cannot afford time-consuming implementations. In the following section, data-mining programming tools aiming at binding programming models with parallel architectures are discussed with respect to parallel programming aspects and their scalability properties.

## 1.3 Towards parallel computing for data-mining

While data- and graph-mining applications can be considered as recent with respect to today’s increasing dataset sizes, a fair amount of programming options exists in the field to facilitate the development of such applications. Indeed, lifecycles of applications from these domains are way shorter than *e.g.* simulation codes that can sustain long-term programming effort and extensive fine-tuning optimizations. In order to shorten the development time and improve the productivity for data analytics algorithms, many programming tools have been developed with varying degrees of domain-specificity and various philosophy.

In this section, we present some existing approaches, such as fully integrated development environments enabling visual programming for easy application design. Then, domain-specific libraries and programming languages are introduced.

### 1.3.1 Integrated Development Environment for data-mining

A great amount of tools was developed to enable easy and fast development of data-mining applications. Amongst them, some of these frameworks provide tools for visual programming as with for example RapidMiner [26], KNIME [27], WEKA [37] and more recently Orange [64]. All previous frameworks come with a wide range of built-in applications, algorithms or toolboxes from the domains of machine learning algorithms, *e.g.* rule learners, decision trees, clustering, k-NN or Naive Bayes. These three frameworks have built-in



data structures and algorithms that can be extended through an XML or Java API. They are able to process data located in memory and in databases. They also enable clustered computing of their algorithms based on Hadoop (mostly using MapReduce<sup>1</sup>) but often in a limited way in terms of programming support or library availability.

Orange [64] provides also a large set (100+) of data-mining and machine learning function implementations. It can be used in a visual programming fashion or through library calls within Python scripts. Despite a rich set of libraries, Orange was designed for single-computer use and the amount of data that could be processed within this software is thus limited by the hardware architecture used [96].

Related work includes Torch7 [99], a LuaJIT computing framework for (but not limited to) machine learning, and SIPINA/TANAGRA [23], two closely related data-mining educational frameworks. Other sequential tools include Java-ML [36], a java machine-learning library, and ELKI [30], a complete data-mining framework turned towards knowledge discovery in database (KDD Applications). Though these frameworks allow the fast development or processing of datasets, they often fail to address the scaling issue, critical in nowadays data analytics. Additionally, many tools in this category have an *infrastructure* approach which may be cumbersome to leverage on moderate-size clusters.

### 1.3.2 Language extension approaches

Many parallel programming approaches have taken the form of language extensions, that is, extending a language's set of keywords in order to add *e.g.* parallel execution features. Such approaches have the benefit of leveraging tools usually well accepted and known, such as mainstream programming languages, with improved performances/usability for a given domain. However, the need for new compilers supporting the added syntactical constructions is a drawback to such approach.

As previously explained, mainstream parallel programming approaches such as MPI or OpenMP may not be absolutely suitable as-is for distributed data-mining because of their complex use or lack of scalability. In between these approaches, XMP is an OpenMP-like, pragma-based, C/C++/Fortran language-extension which aims at gathering the ease of use of OpenMP programming schemes for distributed architectures. Although it as scaled up to thousands of nodes on the K-Computer supercomputer in Japan on regular applications, its performances are unclear for irregular, data-dependent algorithms such as data or graph-processing workloads. Moreover, the time-consuming and cumbersome task of partitioning the data is left to the programmer. Finally, as a general-purpose programming tools, XMP is deprived of potential domain-specific optimizations.

With the emergence of the Partitioned Global Address Space (PGAS) memory model, some options were added to the panel of parallel programming frameworks, including new languages and libraries. Unified Parallel C (UPC) [20], X10 [48], Co-Array Fortran [14]

---

<sup>1</sup>see in 1.2 Execution models

and Titanium [25] are four high-productivity languages which emerged at the beginning of the 2000's. They all implement the PGAS model, except X10 which supports also dynamic threading for Asynchronous PGAS capabilities.

UPC [20] is an extension to the C language which supports declaration and manual mapping of shared variables. It is based on the SPMD execution model, as most MPI programs, and communications are managed implicitly through shared objects and constructs. UPC, as other PGAS languages, uses fat-pointers for remote references. This process, managed by the runtime system, can be improved by enabling hardware support as in [85] or using the Remote Direct Memory Access (RDMA) features [76] available on some HPC environments but mostly unavailable on commodity desktop workstations or even higher-end compute installations. A drawback of the UPC approach is that the language can be as verbose and complex as a plain C-MPI distributed program when it comes to implement complex algorithms where data should be carefully placed to improve locality. Furthermore, there is no support for online data re-partitioning or dynamic multithreading, not mentioning reliability management.

Related works include Co-Array Fortran [14], an extension of the Fortran programming language with support for shared array over shared or distributed architecture. Programs based on Co-Array Fortran are inherently SPMD and well-suited for regular applications such as the processing of large arrays of variables. This favourable characteristic make Co-Array Fortran hardly a candidate of choice for unstructured applications such as those of the graph analytics domain.

X10 [48] and Titanium [25] are two programming languages for parallel programming. They differ in X10 enabling dynamic multithreading and compiling to native code whereas Titanium is purely executed in a Java Virtual Machine. X10 is also pushing further the PGAS model by providing *places* which enable dynamic creations of new threads for concurrent processing of data [60]. This language also provides promising domain-specific libraries [55].

Green-Marl [58] relates to these general purpose programming languages as it is a domain-specific language dedicated to graph-mining. It improves the productivity of a programmer as the algorithm can be expressed in a really natural way by providing graph-related constructs within the language keywords. A dedicated compiler is in charge of transforming it into C++ and is able — as it is domain-specific — to apply optimization specific to the graph domain that general-purpose compilers would not be able to foresee. At the time of writing, no distributed cluster back-end is available, although plans were announced to implement it through source-to-source compilation to a domain-specific C++ library. Advantages of such tools are plenty: expressive and natural abstraction for graph-processing algorithms, domain-specific implementations. However, as Green-Marl is a rather recent dedicated language, compiler tool support and end-user acceptance must be improved.

### 1.3.3 Library-based approaches

Some different approaches have also emerged, mostly under the shape of more or less domain-specific libraries or languages for parallel architectures. The advantages of such approach lie mostly in the fact that, by leveraging libraries based on mainstream languages, acceptance is greatly facilitated and compilation toolchains are available. This section discusses approaches from general data-mining systems to domain-specific libraries targeting a precise applicative domain within the data-mining field.

FREERIDE [11], and its extension for grid-computing, FREERIDE-G [32], are two C++ middlewares for parallel execution of data-mining applications. Though the former targets shared-memory computers, the latter is oriented toward moderate-size grid computers. In terms of execution models, both frameworks implement a BSP model which closely resembles an iterative MapReduce model. Though FREERIDE-G addresses the issues related to data-mining programming in a quite domain-agnostic way for distributed architectures, the programming model enforced is too limiting and can thus be counter-productive, despite interesting performances [38].

The Galois system [33] can be compared to OpenMP as it implements a fork-join programming model. This library provides tools and runtime for parallelization of iterations over partially ordered or unordered sets using speculative execution, tackling the issue of *data-amorphous parallelism* [39]. However, Galois sets are available only on shared-memory systems, making them hardly a choice for large-scale computations requiring distributed architectures. Related works to Galois and FREERIDE include the first version of GraphLab [43], a domain-specific alternative for the parallelization of graph analytics applications for shared memory computers, with however no support for speculative execution. The work of Jin *et al* [22] on an optimizing runtime for locking mechanisms over shared variables relates to these previous works, tackling shared-memory parallelization. Though these works propose interesting approaches for the processing of data- and graph-analytics workloads, the fact they target shared-memory computers makes them hardly suitable in the context of large-scale processing over distributed architectures.

DyDSM [69] is a distributed shared memory system for speculative parallel execution of graph-processing programs. The distinctiveness of DyDSM lies in the use of idling cores in hierarchical distributed clusters for managing communications between workers, using dedicated communication threads. With this approach, communication procedures are removed from critical execution paths of working threads and are executed apart. DyDSM also features a prefetch predictor which has to be user-defined and can be cumbersome for the user to program. A related work which seems to address the issue of programming productivity is NIMBLE [51]. NIMBLE enables the execution of data-mining algorithms expressed using a Java API on top of a distributed framework (currently, Hadoop), implementing an APGAS model.

As promising as seem those projects, they still exhibit a rather generic programming approach with complex underlying mechanisms to handle parallelism, failing to provide an

applicative domain-specific API which could help in improving programmers' productivity on top of commonly used tools. This issue is however somehow addressed by the following domain-specific libraries. GraphLab PowerGraph [56], Pregel [44] and Giraph [74] are three libraries for graph-processing on distributed architectures implementing a vertex-centric programming model. PowerGraph (the distributed branch of GraphLab) differs slightly by providing released consistency models [94] and providing a slightly derived vertex-centric model, called the Gather-Apply-Scatter (GAS) model. PowerGraph also provides semantic constructions acting on the whole graph structure and asynchronous capabilities, while Pregel and Giraph are purely synchronous vertex-centric frameworks. GPS [73] (Graph-Processing System) and Mizan [68] are related to these previous graph libraries but diverge in their slightly more general execution model. Both of them provide a vertex migration feature for dynamic re-partitioning of the graph, with however limited gains as suggested in [79].

Such approaches are particularly relevant as they provide an expressive programming model under the form of a programming library compatible with most standard compiler toolchains. Moreover, their ability to hide parallelism details and to be deployed on distributed architectures make them natural candidates of choice for scalable graph-processing algorithms.

## 1.4 Concluding remarks

In the field of data-mining, and more precisely in the context of graph analytics, applications require more and more memory and processing power. Having reviewed three aspects of High-Performance Computing with respect to graph analytics, in particular, hardware trends, programming models and software approaches, we can gather the following remarks.

Though supercomputers exhibit characteristics able to tackle both the issue of memory and compute power, their operating costs and their programming complexity hardly make them candidates of choice in such a fast-paced environment that is the data analytics domain. In contrast, clusters of inexpensive commodity workstations seem a fairly decent choice in this context, notably by providing amenable price to performance figures with an accessible architecture, programming-wise. Indeed, commodity clusters may be more easily exploited compared to data-crunching many-cores exhibiting a specialized hardware. In order to gain more insights on these statements, we set-up and evaluate in this thesis work different hardware architectures presented in Chapter 2, matching both the commodity cluster and the high-end server trends.

To exploit large amount of clustered computers, a scalable implementation must be deployed. Library-based, domain-specific approaches sparked interests as they are addressing many issues encountered by data practitioners requiring to process graphs on distributed architectures. Providing a natural and expressive abstraction is indeed key to

leverage an increased programmer productivity and a quick prototyping of algorithms, as required by the data-mining domain. In particular, by hiding parallelization management details or data partitioning, they allow programmers to seamlessly deploy their algorithms over distributed clusters. Moreover, the ability to build implementations leveraging such libraries on top of mainstream tools enables a facilitated acceptance by the community, hence ensuring a greater support. Such possibility removes as well the requirement for new compiler toolchains. Hence, we describe in Chapter 2 the advantages and drawbacks of vertex-centric graph-processing libraries which satisfy the aforementioned characteristics. Then, we introduce the details of GraphLab PowerGraph, a framework of much interest in distributed graph-processing.

In order to validate the relevance of such frameworks for graph analytics on distributed architectures, it is mandatory to be able to assess their performances with respect to computations and scalability. Though benchmarks have been traditionally widespread in High-Performance Computing, evaluating performances in the context of High-Performance Data Analytics — let alone graph analytics — solely relying on FLOPS may not be sufficient to predict performance behaviors with increasing problem sizes and cluster scales. The issue of benchmarking in the concerned applicative domain is addressed in Chapter 2 in which are discussed relevant metrics for the assessment of graph-processing framework performances.

Then, we demonstrate on two real-life use-cases how such graph-processing libraries are particularly suitable for a large variety of graph problems. In particular, we study in details and propose two novel algorithms and their distributed implementations to address graph analytics problematics from unrelated applicative domains — which are introduced in details in Chapter 3 — and evaluate their performances.

Having studied software-related aspects, we investigate, in details, the impact of the architecture on performance behaviors and compare different hardware trends in Chapter 4 using the two use-cases previously presented. Then, we show how such performance analysis is of particular relevance to adequately size a cluster for a given workload. Finally, hardware propositions towards more efficient graph-processing platforms are presented in Chapter 5.



# Graph-processing at scale on distributed architectures

## Contents

---

<b>2.1</b>	<b>Graph-processing: Programming models and state of the art</b>	<b>29</b>
2.1.1	Programming models: from BSP to vertex-centric programming	29
2.1.2	Landscape of vertex-centric frameworks . . . . .	30
<b>2.2</b>	<b>Anatomy of a GraphLab program . . . . .</b>	<b>31</b>
2.2.1	Initialization and input file parsing . . . . .	32
2.2.2	Partitioning of the graph structure . . . . .	32
2.2.3	Graph-processing using the GAS execution engine . . . . .	33
2.2.4	Result output and execution termination . . . . .	36
2.2.5	Synthesis . . . . .	37
<b>2.3</b>	<b>Performance evaluation of GraphLab programs . . . . .</b>	<b>37</b>
2.3.1	Benchmarking off-the-shelf application against real-life performance tuning . . . . .	38
2.3.2	Throughput metrics for operating performance benchmarking .	39
<b>2.4</b>	<b>Available distributed memory architectures . . . . .</b>	<b>40</b>
2.4.1	Presentation of the compute clusters . . . . .	40
2.4.2	Comparing hardware approaches . . . . .	42
<b>2.5</b>	<b>Synthesis . . . . .</b>	<b>43</b>

---

The graph-mining area is a data-mining domain in which applications process large graphs to extract higher-level information. In particular, graph-processing is key in a wide range of fields such as biology, web analysis, security or social networks. With the advent of Big Data, processing power and memory are more and more needed, requiring larger and resourceful machines. As illustrated in Chapter 1, parallel programming techniques leveraging large-scale distributed architectures emerged to tackle these challenges. However, mainstream parallelization approaches are unsuitable as-is for irregular and unstructured applications such as graph-processing. As a matter of fact, classic High-Performance Computing (HPC) applications included regular algorithms such as physics simulations for which pure-MPI implementations are suitable. However, recent irregular

graph-processing applications (*e.g.* graph-traversal) would be costly to implement using explicit communication semantics as proposed in MPI. This is reinforced by the differences in data-mining application requirements, where programmers need to be able to quickly implement and deploy their algorithms without having to manage time-consuming low-level aspects of parallel programming or optimizations.

To bridge the gap between ease of use and scalability for irregular applications, many dedicated frameworks appeared, aiming at the scalable deployment of graph-related workloads. These graph-processing tools are available in many different flavors with specific programming models, memory views, languages or associated toolsets [95]. Thus, considering a novel graph application, it is challenging to choose the most appropriate software tool as literature exhibits global trends, but no clear winner [78, 79]. Amongst the landscape of eligible tools for the implementation of graph-processing algorithms, vertex-centric frameworks have raised interests for associating an expressive abstraction with sound performances. In order to get a better insight on such tools, we propose a review of the state of the art on vertex-centric graph-processing tools, detailed in Section 2.1. In particular, the specific vertex-centric programming model is presented in details before mainstream implementations are discussed. Finally, we introduce in Section 2.2, GraphLab, the library used to implement graph algorithms studied in Chapter 3. As this framework provides an interesting compromise between relevant performances and ease of use, we then present in details — albeit from a practical point of view — the GraphLab library used throughout this work.

Having selected a software tool to implement an algorithm and understood its programming model is only halfway towards an efficient execution with relevant performances. The remaining path to explore leads towards hardware aspects of the system on which the application is to be deployed. Indeed, adequately sizing a cluster architecture for the efficient processing of distributed graph analytic workloads is a tough task. Part of the difficulty of this task lies in the difficulty to assess and understand the performance behavior of the software, the hardware and the dataset. Indeed, in the context of data-analytics algorithms, the dataset properties often impact compute performances of the implementation. The Scalable Synthetic Compact Applications [18], and later Graph500 [45], two graph-processing benchmarking suites, have been designed as an attempt to tackle this challenge and introduced a performance metric called TEPS (Traversed Edges Per Seconds). Though TEPS inspired other throughput related metrics such as Edge (respectively Vertex) per seconds, or shortly EPS (respectively VPS) [78], a single throughput measurement is often not enough, and traditional metrics are still widely used (resource usage, communication volume, timing) [66, 75, 79, 89, 93] along cost or energy metrics [84]. Performance measuring and benchmarking aspects of this thesis work are discussed in Section 2.3.

Finally, Section 2.4 presents different hardware architectures that were used in this thesis for the study of our algorithms detailed in Chapter 3. In particular, we present



three architectures, divided in two categories: commodity clusters and high-end servers. These architectures are later compared in terms of operating performances and overall efficiency for graph-processing in Chapter 4.

## 2.1 Graph-processing: Programming models and state of the art

As data-mining applications processing large graphs became widespread, the ability of processing data on large-scale systems have become a necessity to tackle such dataset growth. Though many kind of generic tools were developed in the field, as reviewed in Chapter 1, vertex-centric programming has been raising significant interests recently, as it provides a natural and expressive abstraction for the implementation of graph algorithms.

In this section, we present the vertex-centric programming model and how it relates to the Bulk Synchronous Programming model (BSP) from which it originates. Then, mainstream implementations of the model are reviewed and discussed.

### 2.1.1 Programming models: from BSP to vertex-centric programming

The Bulk-Synchronous-Parallel (BSP) model was introduced by L. Valiant as a bridging model between hardware and software for parallel programming [3]. In the original work, a BSP computer model is introduced, composed of a certain number of processing units, a router organizing communications and a set of synchronization primitives. A BSP program is composed of concurrent threads executing synchronized parallel *supersteps*. A *superstep* can be divided into two stages: a processing step where threads are independently executed in parallel, and a communication step where threads are allowed to communicate. Finally, a synchronization barrier ensures that all workers have finished their iteration (communication and processing) before starting a new one. A critical issue in BSP programs is load-balancing as the makespan of each superstep is determined by the longest thread [95].

The BSP model was later extended into the vertex-centric programming model, to better address the graph-mining domain. In such a programming model, every vertex of the graph is seen as a concurrent thread of execution, performing an *update function*, also known as a *vertex program*. In an *update function*, the current vertex reads its received messages, updates its internal data and either sends messages to other vertices or votes to halt. If a vertex has voted to halt, it will remain inactive during the next supersteps, until it receives a message. Convergence is reached when every vertex in the graph has voted to halt — that is, when there is no more remaining active vertex — or when a user-defined convergence criterion is met.

	Open source	Prog. model	Graph part.	Language	Support
Pregel [44]	No	BSP/Mes.-Passing	Edge-cut	C++	N/A
Giraph [74]	Yes	BSP/Mes.-Passing	Edge-cut	Java	Yes
GPS [73]	Yes	BSP/Mes.-Passing	Edge-cut	Java	Yes
Mizan [68]	Yes	BSP/Mes.-Passing	Edge-cut	C++	Limited
GraphLab v1 [43]	Yes	BSP/Shared Mem.	N/A	C++	Yes
GraphLab v2 [56]	Yes	GAS model	Vertex-cut	C++	Yes

Table 2.1: *Comparison of vertex-centric frameworks. Most frameworks implement a programming model based on the BSP model with explicit message passing between vertices, except GraphLab which implements a more specific GAS model in its distributed version. GraphLab v2 also differs in its graph partitioning policy by implementing an edge-cut policy instead of the more commonly used vertex-cut.*

Notably, in most implementations of the so-called vertex-centric programming model, the programmer is provided a somehow restrictive abstraction, where vertices executes local-only computations. This local-enforced policy has two impacts on the programming. The first effect is a narrower application range, as not all graph computations are well expressed in such a model and the second, a facilitated graph distribution and a reduction of the amount of required communications.

Having introduced the abstraction behind the vertex-centric paradigm, we present in the remainder of this section, the main programming frameworks for vertex-centric graph-processing, as summarized in Tab. 2.1.

### 2.1.2 Landscape of vertex-centric frameworks

Google’s Pregel [44] and Apache’s Giraph [74] are two similar libraries for distributed graph-processing which implement the BSP programming model with a vertex-centric view. While Pregel is an undisclosed C++ library, Giraph is provided as an open-source Java API built on top of Hadoop. Giraph shows remarkable performances amongst comparable frameworks in various studies, however, its memory overhead is high [78, 79] which may be acceptable at the scale of a thousand or more high-end machines, but may be prohibitive in the context of a more moderate commodity cluster.

GPS (*Graph Processing System*) [73] and Mizan [68] are two open-source graph-processing implementations of the BSP model. As in Giraph/Pregel programs, during each superstep, nodes synchronously execute an *update function* in parallel. Contrary to Pregel, GPS also provides global computation semantics such as a `master.compute()` function that can be called at the beginning of every superstep. The master has access to all global objects and can update them before broadcasting changes to the workers. Both frameworks feature a built-in graph partitioning engine allowing static and dynamic partitioning of the graph. However, though they differ in maturity and performances [79], they are both outperformed by Giraph or GraphLab PowerGraph. Finally, despite its overall memory efficiency, the fact that GPS uses additional threads to poll for messages is penalizing for supersteps with light processing loads.

GraphLab [43, 61], is originally a shared-memory C++ framework for in-memory parallel graph-processing implemented using pThreads. Contrary to Mizan, GPS and Giraph, the graph structure cannot be modified once the processing have started. Graph computations within GraphLab are expressed through the implementation of a so-called *update function*. Iterations of such *update functions*, or supersteps, are orchestrated using a synchronization mechanism. An asynchronous execution mode is also provided, in which vertex iterations can be concurrently processed upon scheduling allowance.

GraphLab PowerGraph [56] can be seen as the distributed version of GraphLab. In this iteration of the GraphLab framework, the vertex-centric model is extended to the subtly more restrictive Gather-Apply-Scatter (GAS) model in order to provide an efficient distributed implementation matching the programming philosophy of the original tool. Distributed computing programming aspects such as graph partitioning or process management are hidden from the programmer in order to increase productivity and provide a seamless deployment of the implementation. Still, many parameters can be easily tailored *e.g.* using command line options and left at the programmer's discretion. The backbone of PowerGraph is based on an MPI layer for the management of inter-process communications and multithreading is leveraged at the compute node level to increase performances by exploiting every available cores in the processor.

In conclusion, having reviewed the state of the art in vertex-centric programming frameworks, GraphLab<sup>1</sup> appears to be a compromise of choice between expressive abstraction and distributed processing performances. Moreover, GraphLab exhibits a high level of maturity and the fact it is based on a mainstream programming language (C++) eases its acceptance. Indeed, using a programming library of a widespread language with a large set of debugging and compiling tools available makes for a reduced programming burden compared to a new language. Moreover, in terms of pure performances, GraphLab often compares favourably to other frameworks [79]. For all these reasons, we decided to make GraphLab our framework of choice for graph-processing over distributed architectures. In the following section, the anatomy of a GraphLab program is detailed in order to provide a detailed yet practical understanding of the framework's usage.

## 2.2 Anatomy of a GraphLab program

A typical data analysis program can be divided in several parts often including: parsing of the input files, execution of the algorithm and output of the results. GraphLab is no different and most GraphLab programs can be divided similarly in four consecutive stages. At first, the initialization of the main GraphLab structures and the MPI layer is performed, followed by parsing of the input files. Then, the final graph structure is instantiated and partitioned across the cluster using GraphLab built-in heuristics. Con-

---

<sup>1</sup>Note: In the remainder of this manuscript, every reference to GraphLab would refer to PowerGraph, the distributed version of GraphLab.

sequently, the execution engine can be instantiated and started, to begin the distributed processing of the previously committed graph. After convergence of the algorithm, results are produced and written to output files. In the remainder of this section, we present in details these four steps.

### 2.2.1 Initialization and input file parsing

Before calling the GraphLab program, input files must be prepared. Indeed, as the parsing step is parallel, based on the number of files to parse and performed on every nodes of the cluster being used, the input dataset has to be split among cluster nodes. To such extent, the use of a distributed filesystem, such as NFS or HDFS, can be relatively helpful.

At the very beginning of the execution, a mandatory preamble is required to start the MPI layer, as GraphLab relies on MPI for inter-node communication. This initialization is performed through a necessary call to `MPI_Init()` as in regular MPI programs. The distributed control structure orchestrating communications is instantiated at the same time. This structure is notably handling both the graph and the execution engines.

Once the initialization preamble is done, the parser is called to analyze the input files. A call to the `load()` member function of the graph object is performed, with two arguments, respectively a pattern matching the input file paths and a pointer to the parser function. When using datasets following standard conventions (such as SNAP datasets [81]), the graph parser function can be omitted and a default parser can be passed as a command line argument, otherwise, a parsing function must be implemented by the user.

At parsing, each input file is independently parsed in parallel and each call to the parser function processes a single line of input at a time. When the parsing ends, every compute node in the cluster has a local edge list, turned into a local subgraph. Possible conflicts (*e.g.* creation of two vertices of same identifier) occurring at parsing time are resolved at a later stage, during the commit step. Indeed, a partitioning heuristic is executed to balance the graph so that it minimizes the graph vertex replication factor.

### 2.2.2 Partitioning of the graph structure

The main data structure of a GraphLab application is the distributed graph object being processed. As both vertices and edges can hold data, their types are user-defined and can be a standard C++ type/class or more conveniently a class defined by the user. However, in that latter case, the programmer must follow a template enforcing the strict serialization requirements of GraphLab. In particular, for custom edge or vertex class, serialization and deserialization methods have to be explicitly implemented by the programmer.

Once the parsing of the input file is done, the graph structure must be *finalized* using the `graph.finalize()` method. The finalization of the graph structure — also called the *commit* of the graph — corresponds to the instantiation and the distribution of the

graph data-structure. This main graph structure is immutable after the finalization step, implying that past the `finalize()` function call, any change in the graph structure (*e.g.* the addition or deletion of a vertex/edge) will not be visible on the processed graph. Indeed, for these modifications to be accounted for, the graph must be committed again after the addition or deletion of any edge or vertex, *i.e.* a call to the `finalize()` routine must be performed.

Traditionally, vertex-centric frameworks have used an *edge-cut* graph partitioning policy, that is, they break the graph in smaller subgraphs and place these subgraphs in different compute nodes so that the number of edges spanning processes is minimal. GraphLab however, provides a different approach, leveraging a so-called *vertex-cut* policy. Using the *vertex-cut* partitioning method, graph edges are not allowed to span over two different machines, whereas a graph vertex can be split across more than one machine. This strategy achieves better load-balancing for the processing of power-law graphs by splitting high degree vertices into smaller replicas placed on different machines [71].

The user can choose between the following provided ingress methods for graph partitioning, namely *pds*, *grid*, *random* and *oblivious*. If no ingress method is passed by the user, an automatic ingress selection routine, *auto*, is executed to select one of the method, using the cluster size as a basis for the decision. Though some of these ingress methods can be used with an arbitrary number of GraphLab processes (*e.g.* *random* or *oblivious*), the *grid* and *pds* ingress methods are more restrictive. Indeed, the *grid* ingress policy requires  $N \times M$  processes satisfying  $|N - M| < 2$ , and the *pds* method requires a number of compute nodes equal to  $P^2 + P + 1$ . As GraphLab states preferring strongly one process per machines,  $N$ ,  $M$  and  $P$  represent here the number of machines used in the cluster when launching the GraphLab instance. We mostly use the two first ingress methods (*random* and *oblivious*) in our experiments as some of the operated cluster configurations do not match the previous requirements of *pds* or *grid*. In brief, the *random* method distributes edges across the cluster using a random hash strategy while the *oblivious* method uses a greedy heuristic launched independently by each GraphLab process [71].

### 2.2.3 Graph-processing using the GAS execution engine

GraphLab PowerGraph introduces a programming model derived from the BSP-inspired, vertex-centric model, called the Gather-Apply-Scatter (GAS) model. Though the GAS model may be seen as a more restrictive abstraction with respect to the BSP model from which it is inspired, it is particularly well suited for iterative vertex-centric graph computations (*i.e.* such as PageRank). However, expressing algorithms with multiple graph traversals such as Betweenness Centrality Score computation [58] may require a slightly greater programming effort. In the GAS programming model, computations are implemented as successive update functions constituted of three successive *minor steps*, as shown in Fig 2.1.

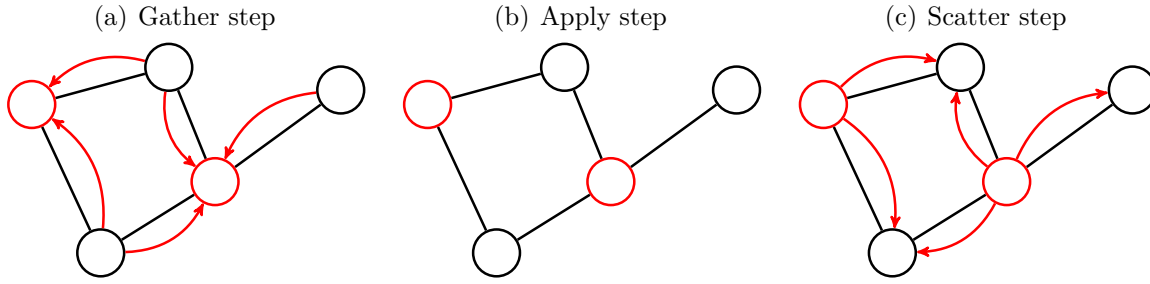


Figure 2.1: GraphLab’s GAS model is an iterative model in which every vertex in the graph performs successively three minor steps, namely Gather, Apply and Scatter. Active vertices are shown in bold red, inter-vertex communications are displayed using red arrows. (a) In the Gather step, active vertices collect information in their 1-hop neighborhood (edge: mutable, vertex: const). (b) Then, active vertices can modify internal data, possibly using previously gathered information (edge: const, vertex: mutable). (c) Finally, in the scatter step, active vertices push changes to neighbours (edge: mutable, vertex: const).

- *Gather*: This step is a parallel reduction over data held by edges and vertices directly connected to the current vertex (1-hop neighborhood), as visible in Fig 2.1(a). In practice, this minor step is performed in two times. Firstly, a set of edges must be selected, between respectively *no e.g. in edges*, *out edges* and *all edges* to define over which edges the reduction is performed. Then, the previously selected set of edges is processed independently and the partial result of the Gather operation is aggregated in the return value and passed to the Apply function. In order to be flexible, the Gather step can return an arbitrary type/class tailored to the following apply function’s need. During this minor step, the vertex cannot be modified, contrary to data held by edges.
- *Apply*: The vertex state is now mutable and can be modified according to the previously gathered data (Fig. 2.1(b)), in isolation from the graph. This local processing is done over the vertex-local data and the returned Gather value.
- *Scatter*: As in the Gather step, a subset of bound edges must be selected beforehand, from *no e.g. in edges*, *out edges* and *all edges*. Once the subset of edges selected, the vertex can decide to vote to halt, if it has satisfied its convergence criterion, or reschedule itself otherwise. It also has the possibility to signal (*i.e.* reschedule) vertices connected to the selected subset of edges for the next iteration. Starting at the scatter step, changes made to the vertex state become visible to other vertices (Fig. 2.1(c)).

The three-step GAS model, though restrictive, allows the programmer to express in a rather natural way local computations seen from the point of view of the graph vertices. This model is further restricted in the distributed execution context, as illustrated in Fig. 2.2. In particular, as vertices can be split across compute nodes, the reduction operation during the Gather step must be associative and commutative to ensure a correct result is forwarded to the apply function, regardless of how a vertex is split. This design

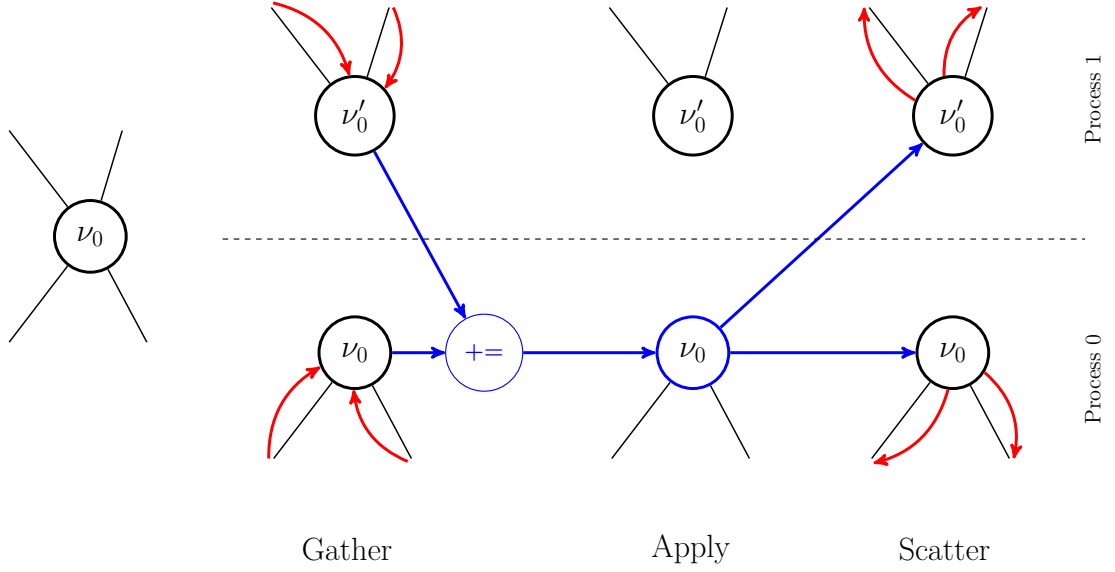


Figure 2.2: Illustration of the Gather-Apply-Scatter model in a distributed context using the vertex-cut graph-partitioning policy. Vertex  $\nu_0$  is a shared vertex, meaning a replica of  $\nu_0$ —named  $\nu'_0$ —is located on a remote machine. Thin black edges represent real graph edges, whereas thick blue arrows show data-path. In the Gather step, each replica ( $\nu_0$  and  $\nu'_0$ ) does a local gather operation on its local set of edges. Then, partial results are forwarded to the master replica ( $\nu_0$ ) which produces the final results using a commutative and associative operator. The Apply step is performed by the master replica. Finally, the Scatter operation is, much like the Gather step, executed locally by each replicas, once updated by the result of the apply step.

choice is necessary to comply with the *vertex-cut* policy offered by GraphLab and has consequences on programmability.

From the point of view of the implementation, once the programmer has described its algorithm using a dedicated execution engine class, the execution engine can be instantiated and launched. The set of initially active vertices can be tailored to the algorithm requirements (*e.g.* every vertex, no vertex or any subset of vertices). Once launched, the execution engine runs until convergence is reached, *i.e.* when no active vertices are scheduled for the next iteration or when the user-defined maximum number of executed iterations has been reached. In addition to these criteria, we implemented another feature to the GraphLab core in order for the execution engine to be stopped when the number of active vertices has converged, hence the number of active vertices stopped evolving over iterations. This convergence criterion can be activated by defining a macro and passing the relevant option to the execution engine.

Two kinds of execution engines can be used, namely the asynchronous and the synchronous engines. When using the synchronous engine, a superstep ends when every vertex completes its superstep and the graph is synchronized. Otherwise, supersteps are executed asynchronously with a large variety of scheduling options available. The framework proposes three different levels of data consistency policy to address the possible consistency issues, respectively the *vertex*, *e.g.* and *full consistency models*, as depicted in Fig. 2.3:

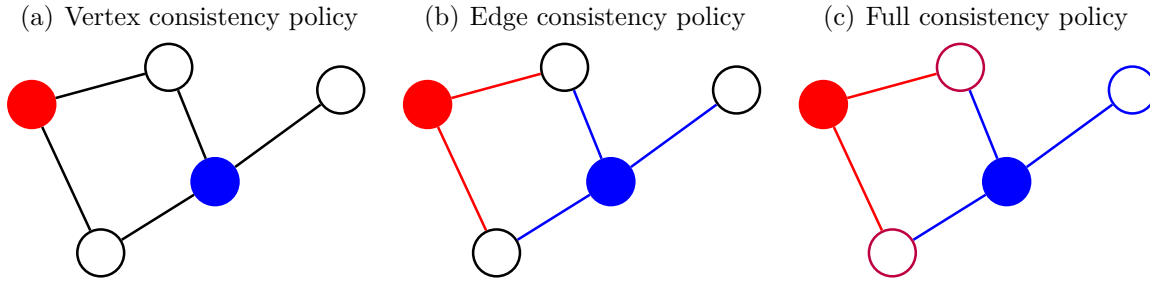


Figure 2.3: GraphLab’s consistency policies for two active vertices, respectively in blue and red. Colored edges/vertices are in the same consistency area. (a) Using the vertex consistency policy, vertices are in isolation and cannot access other vertices data. (b) Using the edge consistency policy, vertices and their connected edges cannot be modified during their update. (c) Using the full consistency policy, vertices and their 1-hop neighbourhood cannot be accessed during their update. In the considered example, two vertices (in purple) span the same consistency area, thus implying the blue and red vertices cannot execute their update function in parallel.

- The *vertex consistency model* (Fig. 2.3(a)) ensures that the current vertex is not read nor modified by other vertices during the execution of its update function.
- The *edge consistency model* (Fig. 2.3(b)) enforces that the current vertex and its edges are neither modified nor accessed during its update function.
- Finally, the *full consistency model* (Fig. 2.3(c)) extends these rules to the full neighborhood of the vertex (*i.e.* all the vertices connected to it cannot be accessed nor modified).

Then, it is the responsibility of the user to choose the appropriate consistency model with respect to the implemented algorithm.

Additionally, GraphLab provides some other processing-related capabilities, such as the possibility to execute a reduction over the whole graph, outside the scope of the execution engine *e.g.* for more complex initialization purposes. Such function can be seen as a lightweight, unique, *Apply* minor step.

## 2.2.4 Result output and execution termination

After processing, GraphLab can output some information related to the execution (*i.e.* the number of updates having been issued as with the completion time of the execution engine). However, to save the results of the execution, a `writer` class must be implemented, describing how each vertex and edge should behave when saving the graph at the end of the execution. For instance, in a PageRank algorithm, where the rank of a vertex is computed iteratively, the desired output of the algorithm is the rank of every vertex. In that context, the end-user may implement the graph `writer` class so that vertices (respectively edges) return their rank only (respectively nothing). This is particularly useful when vertex/edge classes become complex as the default behavior is to save every attribute of a vertex.



The saving operation is performed through a call to `graph.save()`. This method takes as arguments the path and prefix for the output files and the graph writer class. By default, the number of output files written is 4 per process, but it can be adapted further to take benefits from the parallelism of the underlying system architecture, using dedicated command line options. Moreover, output files are produced independently by each cluster node, thus requiring a mechanism to copy the files back on one node if no distributed filesystem is used. Finally, GraphLab embeds built-in file compression routines, enabling the direct output of compressed files.

### 2.2.5 Synthesis

This section has shown that a GraphLab program can be seen, from a high-level point of view, as a parser, a graph structure and some execution engines.

To summarize, from a user perspective, writing a GraphLab program requires the implementation of at least a parser and an execution engine. If the edges and vertices are to hold more complex values than basic types, the user must then implement dedicated classes and their corresponding writers. The parser is typically implemented as an embarrassingly parallel step to be executed concurrently within and by cluster nodes. Moreover, though the programming model of the execution engines is restrictive, it is hence simpler to leverage. In opposition, the graph structure is committed and dispatched by a built-in GraphLab heuristic, as selected by the user.

However, as simple to handle as GraphLab can be, the issue of measuring the performances is relevant and non-trivial in the context of such complex programs. Thus, we present in the following section we address the issue of evaluating the performances of GraphLab applications.

## 2.3 Performance evaluation of GraphLab programs

Starting with the very first programs, benchmarking and performance evaluation methods have been provided. Though measuring efficiency of a sequential program has been widely discussed, evaluating performances of a parallel program can be a cumbersome process. Indeed, one of the simplest form of benchmarking approach can be to evaluate the wall-clock execution time of a program and assess its evolution with *e.g.* another architecture or after a careful optimization. However, this simple metric — though being a merciless element of performance evaluation — does not reflect performances in their overall complexity. In particular, it hardly gives an appreciation of the degree of efficiency a system reaches with a particular implementation and dataset.

In the following section, we discuss the issue of assessing operating performances in the context of graph-processing and present the metrics and methodology used throughout this thesis work.

### 2.3.1 Benchmarking off-the-shelf application against real-life performance tuning

In the high-performance computing domain, hardware platform benchmarking studies have long been performed. The Top500 bi-annual list [4] of the most powerful supercomputers is such a performance evaluation approach based on the evaluation of floating point instruction throughput. It has inspired Graph500 [18, 45], a similar graph-related initiative. Similarly to Top500, Graph500 provides a defined code base of a graph-processing application, tunable to some extent, and ranks hardware platforms by decreasing peak performance, expressed in Traversed Edges Per Seconds (TEPS).

The TEPS metric is based on a similar idea as the more widespread FLOPS (floating point operations per second) which is a metric expressing a system’s throughput in terms of executed instructions. This metric aims at enabling a comparison of (super)computers on a consistent basis using a graph-processing kernel. The idea behind TEPS and FLOPS is rather similar — expressing a throughput of respectively processed edges and committed micro-operations. However, though TEPS is the main ranking metric of the Graph500 list, the scale of the processed random graph and the total power consumption may also be comparison points to be taken into account — the bigger the scale, the better, with an as reduced as possible consumption.

Though this benchmarking approach is obviously of much relevance, the sole TEPS value hardly gives precise hints on the efficiency of an execution. Moreover, as Graph500 is mostly based on a graph-traversal algorithm, it fails to be representative of other graph-mining tasks not involving walks in a graph, *e.g.* vertex-centric operations. Graph500 encourages also end-users to fine-tune and optimize (under guidance) the code for their architectures, which leads to non-portable, high-performance implementation in striking contrast with trends such as the use of Hadoop or GraphLab which aims at shortening development time to achieve an acceptable compromise between productivity and implementation performances.

Finally, though Graph500 evaluates TEPS on the graph traversal part of the benchmark, graph structure commit/distribution, parsing — or for Graph500, random graph generation — are not taken into account into the single metric based result. This is especially a caveat in a domain in which processing time is not always dominating largely non-processing task times (*e.g.* parsing or preprocessing). Thus, results obtained through such benchmarking on synthetic datasets can hardly give clues on how adequate the underlying hardware system could be on a novel use-case or dataset. However, being able to assess such appropriateness is of particular relevance as graph-mining is data-dependent and has applications in many different fields (biology, security, social mining) having datasets of different properties.

With this in mind, the next subsection discusses the metrics used for our performance behavior studies of graph applications and shows in particular for what reasons we argue they are of particular relevance in this context.

### 2.3.2 Throughput metrics for operating performance benchmarking

As GraphLab programs are composed of multiple steps exhibiting different forms of parallelism (*e.g.* parsing, execution engines or result saving), the use of a single metric is not sufficient to grasp such complexity and help understand the operating performance behavior. Moreover, being able to understand and even predict how performances evolve with respect to dataset sizes and cluster scale is important in a domain which shows ever-increasing dataset sizes.

To tackle these two aforementioned aspects, we used in particular two throughput metrics aside traditional resource usage measurements to assess and understand the performances behaviors of our implementations. In GraphLab, an update (*i.e.* a call to an *update function*) can be seen as the processing quantum of an algorithm, we therefore measure:

- The *update rate* ( $upr$ ), defined as the total number of updates ( $N_{update}$ ) performed per second during the GraphLab execution engine ( $t_{proc}$ ).

$$upr = \frac{N_{update}}{t_{proc}}$$

- The *whole-process update rate* ( $upr_{wp}$ ), defined as the total number of updates ( $N_{update}$ ) performed during the wall-clock execution time ( $t_{wall}$ ), including parsing, partitioning and result dumping.

$$upr_{wp} = \frac{N_{update}}{t_{wall}}$$

The *update rate* gives insights on the raw processing performances of the system for a given execution while it only accounts for the processing part of the program (*i.e.* the time spent in the execution engine). It is of particular importance as it enables the comparison of the graph-processing efficiency of different hardware architectures and the assessment of performance evolution with respect to dataset scale.

Compared to the previous metric, the *whole-process update rate* weighs raw performances by taking into account the time spent in non-processing tasks such as *e.g.* parsing. This gives a more global appreciation of the execution, which is especially relevant when processing/non-processing tasks are of the same order of duration.

We argue that both metrics are equally useful. The *update rate* gives raw performance figures of GraphLab whereas the *whole-process update rate* gives higher level hints such as decreased performances due to a low computing/parsing ratio. However, though throughput metrics cannot be used to compare two different algorithms as their *update function* may differ in computational complexity, they are particularly helpful in assessing scalability-related performance behaviors, a key requirement in the context of evaluating

ever-increasing processing workloads. These two metrics were particularly helpful in observing operating performance behaviors as presented in the following Chapters. Additionally, as many other graph-processing libraries are based on the vertex-centric scheme, we argue that such metrics can be used more generally to evaluate performances of other related vertex-centric frameworks.

Finally, though these two metrics relate to TEPS as they are throughput-based graph-related metrics, they differ in the following. TEPS is a rather algorithm-agnostic metric as performances is computed using data-based figures (output edges processed per unit of time). Hence, raw performances of many different (graph-traversal) algorithms can be compared on a same basis. However, though TEPS is an adequate metric for graph-traversal algorithms, it is unsuitable for vertex-centric graph-algorithms. In comparison the update rate can be used for every algorithm implemented with the vertex-centric model (including, graph-traversal).

The FLOPS metric, though it can be seen as TEPS as an algorithm-agnostic metric, relates to the update rates as they both use a number of operation performed per second as a basis for performance measuring — micro-instructions for FLOPS and vertex-functions for the update rates. Moreover, in our experiments we do not only consider update rates as isolated measurements which would lack of precious insights on the performance behaviors. We rather observe how this measurement evolves when accounting for varying parameters such as problem size and cluster scale. Comparing curves is a strength of the current work because it not only tells the system performances but also highlights the flaws and opportunities.

## 2.4 Available distributed memory architectures

In the experiments performed within this thesis, three different hardware targets were used, each belonging to a relevant architectural trend in the field of distributed computing and described in Tab. 2.2. In particular, two commodity clusters and a high-performance compute server installation were used and evaluated with respect to operating performances and scalability. In the following section, we present these platforms and discuss them with respect to the hardware design considerations addressed in Chapter 1.

### 2.4.1 Presentation of the compute clusters

#### Low-end commodity cluster (LECC)

Early in the thesis work, a distributed memory platform was required. To address this issue, a first experimental architecture was set-up, composed of 7 desktop workstations embedding an Intel Core 2 processor, 4GB of physical memory and 4GB of swap space each. The workstations run Ubuntu 14.10 LTS and are linked through a 1GB/second Ethernet network.

	HC	LSCC	LECC
System	High-performance cluster	Larger-scale commodity cluster	Low-end commodity cluster
Base configuration	Dell PowerEdge R730xd	Lenovo ThinkCentre	Dell Precision 340
Per-node base price	EUR6000	EUR700-800	EUR290-390
Total Nodes	9	16	7
Total Physical Cores	144	64	14
Total Threads	288	64	14
Total Memory	1152GB	128GB	28GB
Processor	Intel Xeon E5-2640	Intel i5-4430	Intel Core 2-X6800
Processor number	2	1	1
Launch date	Q3'14	Q2'13	Q3'06
RCP	USD939-944	USD182-187	N/A
Core number	8	4	2
Thread number	16	4	2
Cache	20Mb SmartCache	6Mb SmartCache	4Mb L2
Base frequency	2.4GHz	3.0GHz	2.93GHz
Turbo frequency	3.4GHz	3.2GHz	N/A
TDP	90W	84W	75W
Single thread rating	1950	1825	1110
CPU Mark	14005	6279	1885
SPEC int* rate base 2006	714 (2-socket)	152 (1-socket)	31.1 (1-socket)
SPEC fp* rate base 2006	589 (2-socket)	127 (1-socket)	26.8 (1-socket)
Node memory	128GB	8GB	4GB
Node swap	4GB	8GB	4GB
Memory type	DDR4 RDIMM	DDR3 DIMM	DDR2 DIMM
Memory speed	2133MHz	N/A	667MHz
Network	1Gbps/Ethernet	1Gbps/Ethernet	1Gbps/Ethernet
OS	CentOS Linux 7.2.1511	Debian 3.2.88-1	Ubuntu 14.04
GraphLab version	v2.2 PowerGraph	v2.2 PowerGraph	v2.2 PowerGraph
MPI layer	OpenRTE 1.10.0	OpenRTE 1.4.5	OpenRTE 1.6.5

Table 2.2: Detailed description of the three distributed architectures involved in the thesis work. The High-performance cluster (HC) shows an up-to-date, performance-oriented configuration with a considerable amount of RAM available to the two-socket Intel manycore. The two remaining architectures, the larger-scale commodity cluster (LSCC) and the low-end commodity cluster (LECC), are two commodity cluster built from desktop workstations. Benchmarking figures: Single thread rating/CPU Mark are provided by CPUbenchmark [112], SPEC rate are provided by Intel [108–110]. Listed Processor Recommended Customer Prices (RCP) are provided by Intel [108–110]. Per-node base price of the Lenovo ThinkCentre and the Dell Precision workstations were not available at the time of writing, hence the price given is for a similar workstation configuration of the updated product range from these manufacturers.

This architecture enabled the deployment and initial testing of a first experimental setup with benchmarking support on a true distributed memory testbed. The system can be seen as a somewhat representative platform from the commodity computing area, a trend seemingly rising in recent years.

### Larger scale commodity cluster (LSCC)

The larger scale commodity cluster (LSCC) is a commodity cluster architecture. Each LSCC node embeds a more up to date hardware compared to the previously presented LECC platform. Access to the cluster was provided by the *Ecole Nationale Supérieure des Techniques Avancées*, located on the Polytechnique campus in Saclay (FR).

In this cluster, each node is composed of an Intel i5 processor, 8GB of RAM and 8GB of swap space, and is connected through an Ethernet network to other nodes. Contrary to the LECC system, the LSCC uses a distributed filesystem (NFS) between every node.

Initially, up to 336 machines were available, grouped in different rooms and associated subnets. Eventually, only 128 compute nodes could be gathered in a consistent cluster (mostly for OpenRTE [19] version compatibility). However, as machines were dispatched on various subnetworks with varying latency, we decided to focus on a up to 16-node configuration for our studies, ensuring the 16 machines are on a single subnetwork with identical hardware and software configurations. Indeed, we observed that having such layered networks had a significant negative impact on performances.

### High-end server cluster (HC)

The High-performance cluster (HC) used is 9-node distributed architecture located at CEA Saclay. Each node is composed of a dual socket Intel Xeon processor embedding a total of 16 cores (32 threads) for an available memory of 128GB plus 4GB of swap space. The cluster is composed of nodes running CentOS Linux, release 7.2.1511 (Core) and has a distributed filesystem available on each node. Additionally, the Hadoop distributed filesystem (HDFS) was set-up and available on the nodes.

This cluster can be seen as a hardware of choice for large-scale computations as it represents compute clusters from the high-end part of the hardware spectrum, contrary to the two previous platforms. In the following subsection, the specifications of the three architectures are compared.

#### 2.4.2 Comparing hardware approaches

Compared to the other introduced platforms, the high-performance cluster (HC) can be seen as a high-end architecture, with each node embedding a high-performance settings and costing an approximate 6000€ per node, as visible in Tab. 2.2. In contrast, the commodity cluster node architectures presented each cost less than a thousand euros, exhibit only general-purpose hardware and provide more moderate amount of memory. However, it is to be noted that the commodity systems differ in their production date, respectively 2006 and 2013, resulting in the LECC exhibiting a much lower-end hardware compared to the LSCC.

Available benchmarking figures of the CPUs in each machines show significant performance gaps between the three architectures. Indeed, sequential performances of the modern Intel processor architectures seen on the nodes of the HC and LSCC systems significantly outperform those of the LECC platform as exhibited in the benchmarking section of Tab. 2.2. Moreover, the HC, the LSCC and the LECC node architectures differs in the inner parallelism degree offered by each node: the HC proposes a large Xeon-based architecture with up to 32 threads available whereas both commodity clusters provide

much more frugal architectures, with respectively 2 and 4 threads. This translates notably into the LSCC's i5 processor matching single-thread performances of the HC's Xeon despite being largely overtaken in a parallel context, as shown by the SPEC2006 rate metrics which show parallel throughput performances of the CPUs.

These differences can be explained by their relative age (for the LECC cluster) and their product category (for the LSCC platform). Moreover, the high-end configuration's theoretical superiority must be put in perspectives with its higher price tag. Indeed, in comparison, the other clusters are composed of inexpensive desktop machines assembled across a cheap Ethernet network.

In particular, although it can be expected that the massive amount of memory provided by the HC architecture will outperform both other systems in terms of maximum manageable problem size, it cannot be stated prior to an in-depth study whether it would exhibit higher performances (throughput-wise) for smaller cases. Moreover, it is still unclear how these three architectures will behave with respect to scalability aspects. Such a study can be helpful in addressing *e.g.* the question of whether it is preferable to have a few high-performance compute servers rather than dozens of commodity desktop workstations.

Concerning software aspects, all systems execute the same version of GraphLab v2.2 and use OpenRTE for the MPI Layer on top of various Linux distributions. In order to extract some more in-depth insights on the processing, various instrumentations were implemented within GraphLab, such as fine-grain minor-step timers.

## 2.5 Synthesis

With the recent interest in deploying larger-scale graph-mining algorithms, new programming paradigms and their associated toolset have emerged. In particular, vertex-centric programming libraries raised interests, as such tools provide a more natural, yet restrictive, programming framework relieving the programmer from the burden of handling some parallelism issues such as communication management. Amongst such tools, GraphLab stands as a state of the art, mature and well-accepted compromise between ease of use and operating performances.

In this chapter, we detailed the Gather-Apply-Scatter programming model provided by the GraphLab framework. Though relatively restrictive, it enables a natural and expressive frame for the implementation of vertex-centric computations, addressing by design a large category of graph analytics algorithms. The very local nature of the computations expressed in the GraphLab GAS model makes their behavior clearly understandable and tractable with the use of domain-specific semantic constructs. Finally, by hiding parallelism management details and allowing the programmer to *think like a vertex* [74], the effort spent in the implementation concentrates mostly on carefully designing a graph structure, an adequate parser function and an efficient algorithm.

In order to gain an in-depth view and practical experience of GraphLab, we study and evaluate two use-cases, introduced in details in Chapter 3. Interestingly enough, these applications come from unrelated domains, relatively far from the social mining area from which GraphLab has already gained much popularity. The first use-case addresses the analysis of execution traces using a graph modelization to identify certain low-level properties of the execution of a program on a processor architecture. The second use-case comes from the domain of genomic data processing and addresses the issue of filtering De Bruijn graphs constructed from sequencer reads in order to facilitate the assembly of whole genomes. Indeed, most genome analysis methods are based on graph-processing methods and require large-scale machines due to the massive amount of data produced by next-generation sequencers.

However, though the programming model is well defined and easily understood, predicting performances of a GraphLab program and their evolution with respect to increasing problem instance sizes and cluster scale is hardly at hand. Moreover the impact of the cluster type used to perform such graph-mining tasks with GraphLab is unclear and careful benchmarking and profiling is thus required in order to assess operating performance behaviors. Hence, as we have three architectures at hand, of different kinds, it is interesting to compare which hardware trends is the most appropriate in the context of graph analytics at scale. In particular, investigating if commodity clusters, which seems a promising approach, are able to keep up in performances with considerably more expensive high-end systems is relevant. To this extent, a cross-architecture comparison is performed in Chapter 4 and performance behaviors of the systems are discussed with respect to the two previously introduced use-cases. We also show in this chapter how throughput analysis can be helpful for the adequate sizing of a cluster in the context of graph-processing. Then, we formulate in Chapter 5 hardware-related propositions towards more efficient graph-processing servers.



# Practical deployment of parallel graph applications

## Contents

---

<b>3.1</b>	<b>Program trace analysis . . . . .</b>	<b>46</b>
3.1.1	Algorithms and graph models of computation . . . . .	47
3.1.2	Vertex-centric implementation . . . . .	49
3.1.3	Experimental protocol, materials and methods . . . . .	52
3.1.4	Experimental results . . . . .	52
3.1.5	Synthesis . . . . .	58
<b>3.2</b>	<b>De Bruijn graph filtering for <i>de novo</i> assembly . . . . .</b>	<b>59</b>
3.2.1	De novo assembly of short reads using de Bruijn graphs . . . . .	60
3.2.2	Problem modelization . . . . .	64
3.2.3	DBG preprocessing algorithm . . . . .	67
3.2.4	Vertex-centric implementation of the algorithm . . . . .	70
3.2.5	Materials and methods . . . . .	71
3.2.6	Experimental results . . . . .	72
3.2.7	Synthesis . . . . .	76
<b>3.3</b>	<b>Concluding remarks . . . . .</b>	<b>78</b>

---

Chapter 1 has notably illustrated that, despite the variety of tools for distributed programming in the field of High-Performance Computing (HPC), it is a tough task to implement and efficiently deploy graph-processing applications. Recent years have seen some parallel programming frameworks appear in an attempt to find a compromise between expressive abstractions and relevant performances on distributed architectures. Amongst them, vertex-centric programming libraries have been proposed to tackle the challenge of deploying graph analytic algorithms on large-scale architectures, as seen in Chapter 2. GraphLab [43] is such a framework, facilitating — using PowerGraph [61], its distributed version — graph algorithms deployment over distributed systems using a restrictive yet natural programming abstraction.

Having selected GraphLab as a framework of choice for graph-processing, it is necessary to evaluate its benefits for applications outside of the social mining domain for

which it was designed. To this extent, two real-world applications leveraging GraphLab are studied in details in this Chapter. In particular, the motivations and context behind the development of a distributed implementation for each of them are addressed and detailed. Indeed, it was decided to investigate the whole process of implementing, deploying and evaluating a real-life algorithm rather than off-the-shelf, synthetic random benchmarks. These use-cases are extracted from two fairly unrelated fields, reaffirming (if needed) graph-processing as a cross-domain discipline.

The first application comes from the program analysis domain and considers an algorithm for the analysis of processor-level execution traces of a program. In particular, we developed an algorithm that processes a graph constructed from a program trace to extract read/write relationships between instructions. This first application has been particularly helpful in identifying relevant GraphLab performance metrics and operating performance behaviors.

Then, the second presented use-case comes from the genome assembly domain. In this context, we conducted an in-depth study of the software aspects of genome assembly, with a particular attention to data production and their characteristics. We then developed an algorithm to address the particular problem of mitigating the error rate of next generation sequencer (NGS) data, that hinders the assembly procedure, thus decreasing the benefits of using such technology. In practice, the algorithm processes de Bruijn graphs constructed from sequencer reads in order to remove erroneous edges and nodes, facilitating the later genome assembly step which then manipulates much smaller graphs.

The chapter is thus composed as follows. The two first sections detail the program trace-analysis and the de Bruijn graph filtering algorithms. Then, building on the outcomes of the two studies, a conclusion is drawn.

## 3.1 Program trace analysis

This section provides a detailed study on the use of GraphLab for the implementation of a trace analysis algorithm which is, to the best of our knowledge, a novel use-case for vertex-centric graph-processing libraries. Indeed, graphs are often used as models in the context of program analysis (*e.g.* Data Flow Graphs), making distributed graph-processing tools relevant for the implementation of large program trace analysis based on graph modelization.

In this study, the scalability of our GraphLab implementation is investigated on the moderate scale commodity cluster *LECC* described in Chapter 2. To this extent, several metrics were used, including execution times and throughput measurements such as *update rate* and *whole-process update rate*. This investigation led us to highlight different operating ranges, leveraging better understanding of our testbed's behavior for further performance predictions and platform tuning.

### 3.1.1 Algorithms and graph models of computation

#### Context

Program trace analysis gives fruitful code tuning opportunities. The analysis of read/write relationships between program instructions can help to highlight data transfers in order to identify possibly relevant parallel code transformations. As an example, if an instruction has a low read count per data production (or unique write) of another instruction, they are intertwined and thus cannot be parallelized. Conversely, a high read count for a few unique writes between two instructions might indicate that these instructions can be separated on different cores or processors. To this extent, we developed an algorithm that extracts these relationships from program traces issued by an instruction-set simulator. In practice, this information is hidden in the billion-instructions input trace and the algorithm turns it in a more exploitable form.

As program trace analysis is adequately modeled using graph semantics, the use of a graph programming model seems particularly relevant. From a user point of view, the development time of a graph algorithm is significantly reduced by leveraging such an expressive programming model. Moreover, the burden of communication management and parallelization is removed from the user and held by GraphLab, thus facilitating the implementation task.

#### Graph formulation of the algorithm

As described in Section 2.2, the main components of a GraphLab program are the data structures and vertex program. Vertices and edges are described using C++ classes and used as template parameters of the graph structure. The vertex program is also described using a template class which methods match the Gather-Apply-Scatter model (GAS). This section presents the implementation of the algorithm, starting with an overall presentation of the program and the underlying data structures. Then, an overview of the execution engine is proposed, followed by a description of the *update function* and a discussion of the implementation.

Considering the support example in Fig. 3.1, a GCD assembly kernel (Fig. 3.1(a)) is executed, producing an execution trace (Fig. 3.1(b)) used as the input of the algorithm. Formally, the input program trace is composed of successive instruction instances composed of a (unique) timestamp  $\mathbf{t}$ , a program address (or instruction)  $\mathbf{IX}$  and the list of input data creation timestamps. Each instance is a node of the input graph (Fig. 3.1(c)). For example, the node 3 of type  $\mathbf{I2}$  is built from the instance with timestamp  $\mathbf{3}$  (*i.e.* the time it was executed) and connected to nodes of timestamps 1 and 0 (*i.e.* these instances produced the data consumed by  $\mathbf{I2}$  at 3).

The input graph data structure (*e.g.* Fig. 3.1(c)) can be constructed by a user-defined or a built-in parallel parser (described in Algorithm 1), fed with input files. To match the need of the use-case, we implemented a parser which constructs the acyclic directed

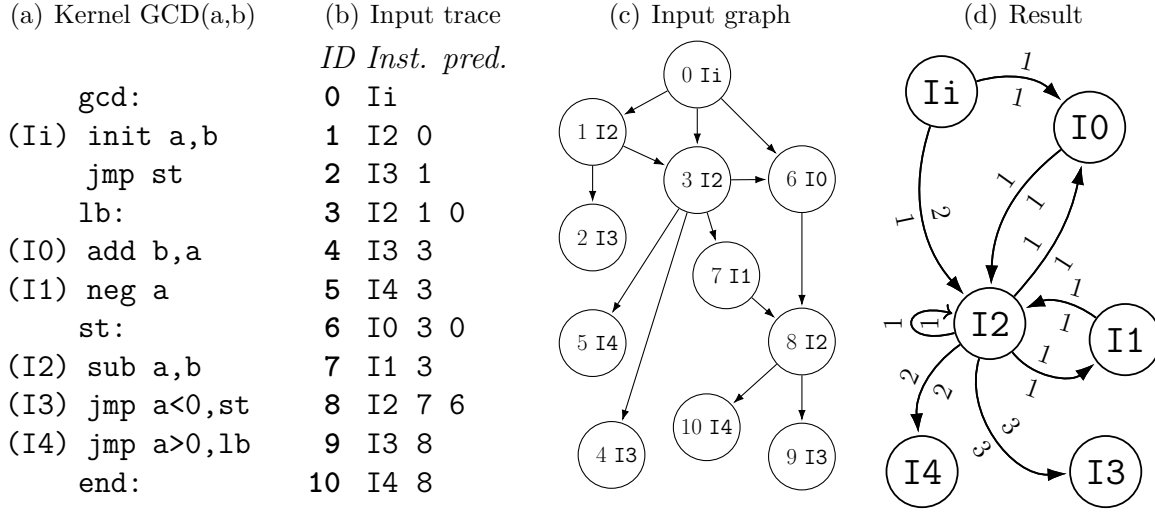


Figure 3.1: (a) Example kernel:  $GCD(a,b)$ . (b) Execution trace of  $GCD(12,8)$ . Each line is composed of a timestamp (used as a vertex identifier), an instruction and its predecessors identifiers. (c) Input trace modeled as a graph. (d) Output graph. Edge labels are readcount (up) writecount (down).

graph (Fig. 3.1(c)) from the execution trace. As each instance (*i.e.* node) is unique, the input trace can be parsed in parallel.

In the generated GraphLab input graph (Fig. 3.1(c)), an edge represents a data transfer, thus the origin (respectively the end) of an edge is connected to the producer (respectively the consumer) of a data. Each vertex of the graph represents an instruction instance and holds its program address as a public value. The algorithm is applied to this graph and produces the graph shown on Fig. 3.1(d) under the form of an adjacency list. Vertices of the output graph represent instructions of the kernel source code and edges express data transfers between them. In particular, each edge holds the number of unique reads and writes between two instructions.

Once the graph is parsed, the graph structure is committed: the chosen ingress method partitions and distributes the graph over the cluster. Finally, the execution engine can be allocated and initiated.

The execution engine executes supersteps (*i.e.* updates) implementing the algorithm described in Algorithm 2. During each update, the partial weight of a specific edge of the output graph is computed by a vertex: Considering the node 8, which is of type I2 (in the example on Fig. 3.1(a), it is the `sub a,b` instruction) connected to nodes of type I3 and I4. Vertex 8 produces in two supersteps the partial weights  $W_8(I2, I3)$  and  $W_8(I2, I4)$ , of output graph edges  $I2 \rightarrow I3$  and  $I2 \rightarrow I4$ . Formally, the overall weight of an output graph edge,  $W(I_{src}, I_{dst})$ , is defined as:

$$W(I_{src}, I_{dst}) = \sum_{\forall v(I_{src}) \in V} W_v(I_{src}, I_{dst})$$

That is, the sum of the partial edges weight  $W_v(I_{src}, I_{dst})$  as mined by the algorithm on nodes  $v$  (of program address  $I_{src}$ ) connected to nodes of program address  $I_{dst}$ .

---

```

Data: Graph data structure G
foreach input line inL parallel do
     $vId \leftarrow inL.getToken();$ 
     $cId \leftarrow inL.getToken();$ 
    if ( $vId = -1$ ) or ( $cId = -1$ ) then
        return false;
    end
    while inL.notEmpty() do
         $dId \leftarrow inL.getToken();$ 
         $G.add\_edge(dId, vId, 1);$ 
    end
     $G.add\_vertex(vId, cId);$ 
    return true;
end

```

**Algorithm 1:** *Parallel input file parser. Each line can be parsed in parallel, with conflicting vertex add resolved at commit time, with every machine in the cluster processing a subset of the input files.  $vId$ ,  $cId$  and  $dId$  stands respectively for vertex, instruction class and destination identifiers.*

Once the algorithm has converged (*i.e.* every node have no more edges to process), results are written out to disjoint files containing the adjacency list of the output graph.

### 3.1.2 Vertex-centric implementation

#### The vertex update function

In an update, a vertex evaluates one destination instruction per superstep. This means that if a vertex has neighbors with identical instruction types (respectively of  $N$  types), it will executes only one superstep (respectively  $N$  supersteps).

- *Gather*: The gather step counts the number of outgoing edges that connect the current vertex to vertices which instruction is currently being evaluated (as these edges represents a read from this target instruction). These edges are then masked while the other edges are kept for further evaluation in later iterations.
- *Apply*: The apply function receives the result of the gather step, which is the number of reads from the current target instruction. In the implementation, the apply function stores the partial weight of the output graph edge (*i.e.* the write/read count between the current node program address and the targeted one). The next target destination instruction is chosen before exiting the Apply function.
- *Scatter*: In the scatter step, the node will either reschedule itself, if there are remaining destination iteration to process, or vote to halt.

Formally, the complexity of an update function is mostly impacted by  $d_v$ , the out degree of  $v$ . The number of iterations is however more complicated to predict as it depends of the variety of instructions connected to the node.

```

foreach active vertex  $\nu$  parallel do
  Gather:
  foreach output edge  $e$  of  $\nu$  parallel do
    read_count  $\leftarrow$  0;
    next_class  $\leftarrow$  -1;
    if  $e$  is unmasked then
      if  $\nu$ .current_class  $\neq$  -1 then
        if  $e$ .target_vertex.cId =  $\nu$ .current_class then
           $e$ .mask();
          read_count  $\leftarrow$  read_count + 1;
        else
          next_class  $\leftarrow$   $e$ .target_vertex.class;
        end
      end
    end
  end
  total.read_count  $\leftarrow$   $\sum$  read_count;
  total.next_class  $\leftarrow$  Min(next_class);
  Apply:
  if total.read_count > 0 then
    output  $\leftarrow$   $\nu$ .cId  $\nu$ .current_class 1 total.read_count;
     $\nu$ .current_class  $\leftarrow$  total.next_class;
  end
  Scatter:
  if  $\nu$ .current_class  $\neq$  -1 then
    Scheduler  $\leftarrow$   $\nu$ ;
  end
end

```

**Algorithm 2:** Vertex function implemented for the program-trace analysis use-case. During the gather step, each node will count how many edges are connected to a node of the considered instruction class (*read\_count*). Accounted edges are removed (i.e. masked). The partial output edge corresponding to the edge between the node instruction class and the considered instruction class is output with the computed write and read counts (respectively 1 and *read\_count*), during the Apply step. The next considered instruction class (*next\_class*) is then decided and, during the Scatter step, the node votes to halt or reschedule itself if there are remaining unprocessed edges.

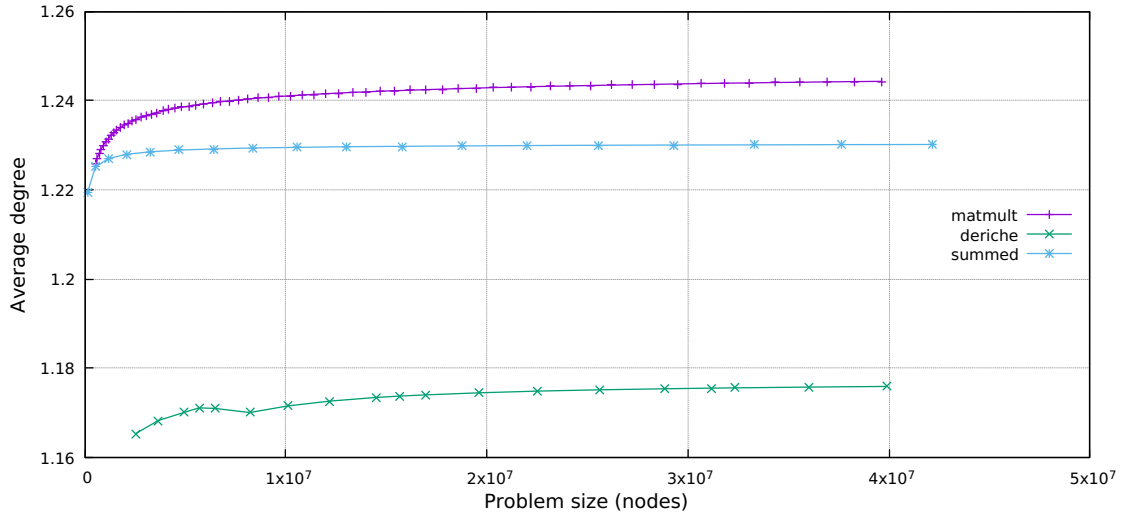
## Implementation, parallelism and synchronization

Parallelism of the algorithm is expressed naturally from the point of view of a node. As each node only writes a private value never read by any other node, every node can execute its program independently from the others. There is no dependency requiring synchronizations to maintain consistency. By allowing better load-balancing, asynchronous execution should be efficient for this use-case. For these reasons, the asynchronous engine was compared against the synchronous engine.

(a) Datasets

<i>Kernel</i>		<i>Vertices</i>	<i>Edges</i>	<i>Density</i> $10^{-6}$
deriche	<i>min</i>	2.5M	2.9M	0.459
	<i>max</i>	39.9M	46.9M	0.030
summed	<i>min</i>	132.5k	161.5k	9.205
	<i>max</i>	42.2M	51.9M	0.029
matmult	<i>min</i>	530.2k	650.0k	2.312
	<i>max</i>	39.6M	49.3M	0.031

(b) Average degree



(c) Degree distributions

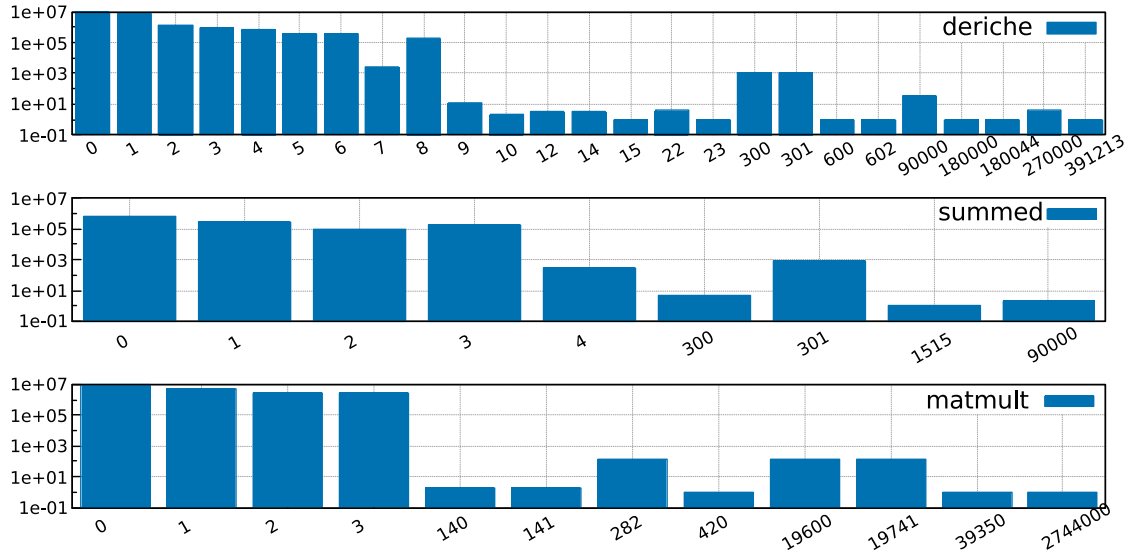


Figure 3.2: Input graph datasets characteristics. Table (a) presents the three kernels with respect to size and density. Each kernel min and max graph characteristics are presented. Figure (b) shows the evolution of the out average degree with growing graph sizes. Figure (c) shows the out degree distributions of the three considered kernels. Empty out degree classes are not displayed.

### 3.1.3 Experimental protocol, materials and methods

Different storage methods have been evaluated for the input files: a distributed filesystem spreading across the cluster or using the local filesystems of the cluster with manual file management. The distributed filesystem used is *Hadoop Distributed FileSystem* (HDFS) version 2.7.1. Notwithstanding its convenience, HDFS clients use a substantial share of the relatively limited cluster memory. Moreover, the time spent in read/write operations (parsing, result dumping) decreased the overall performances by around 30% when compared to a native filesystem. Though this cost can be mitigated for larger scale high-end systems, larger instances and more complex algorithms, it remains a prohibitive overhead with respect to our testbed. Hence, for these experiments, each input file was split in 32 parts and distributed across workstations. Then, each machine parses and loads randomly chosen splits of the input file.

The program traces used as an input of the framework comes from three compute kernels as shown in Fig. 3.2(a): a 3D matrix multiplication, a Deriche filter and a summed area table kernels (referred to as *matmult*, *deriche* and *summed* respectively). These applications were executed with different parameters to produce traces of various sizes. An interesting property of the execution traces is that their average degree — *i.e.* the average number of edges connected to each node in the graph, defined by the division of the number of edges by the number of nodes in the graph — tends to be constant with growing graph sizes (see Fig. 3.2(b)). This behavior can be explained by the fact that the kernel loop code weight grows with increasing input parameters. Additionally, the degree distributions of the kernels (Fig. 3.2(c)) are sparse, showing a few high degree nodes and a large set of low degree nodes.

### 3.1.4 Experimental results

Unless otherwise stated, all results were produced using the synchronous engine and the *oblivious* ingress method. We discuss in this section the throughput of the system and analyze the impact of the underlying memory system. Then, the executions are further analyzed through resource usages and timing aspects. Finally, additional insights on asynchronous execution and ingress methods are presented.

#### Throughput analysis

In Fig. 3.3 are shown the *update rates* from three kernels of varying sizes, for different cluster scales. This metric illustrates raw performances, independently from the rest of the program (*i.e.* the number of updates performed during the duration of the execution engine only).

For each cluster setting, three operating ranges can be observed. To begin with, a suboptimal performance area is visible on the left side of each plot in Fig. 3.3. We refer to this area as the *underloaded operating range* as the addition of compute nodes decreases



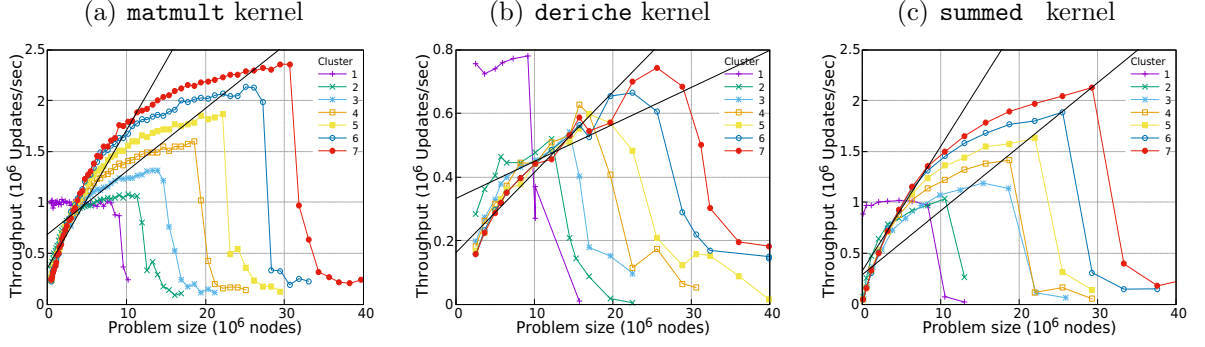


Figure 3.3: Update rate figures for (a) the *matmult*, (b) the *deriche* and (c) the *summed* input kernels. The update rate is the number of updates performed per second during the processing part of the program.

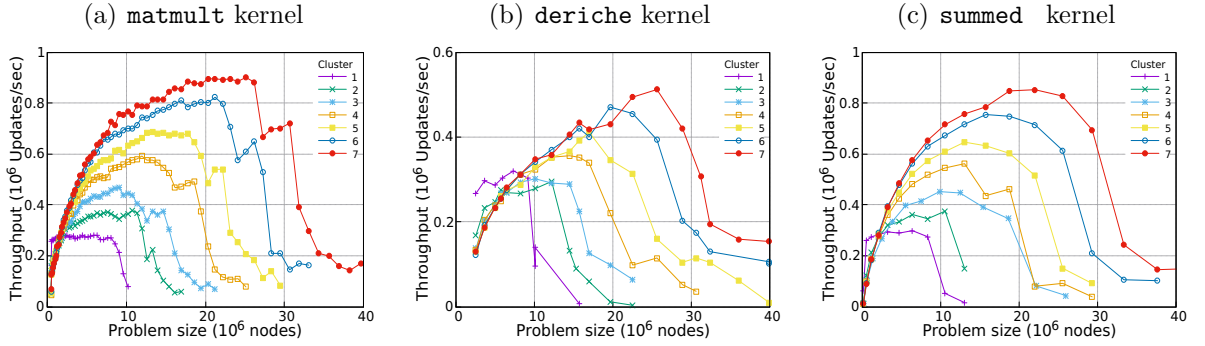


Figure 3.4: Whole-process update rate figures for (a) the *matmult*, (b) the *deriche* and (c) the *summed* input kernels. The whole-process update rate is the number of updates performed per second during the program execution time.

the per-node workload and deteriorates performances. A second operating range can be observed: the *nominal operating range*, in the middle area of the plots. In this area, performances reach a slightly increasing plateau, including a peak operating point. This is the operating range in which compute nodes are the most efficiently used, and GraphLab exhibits the most scalability. Adding machines to the cluster in this setting will increase significantly performances and will reduce the overall processing time. However, adding a large number of compute nodes with the same problem size eventually set the cluster in a suboptimal operating point, resulting in an inefficient use of resources. Finally, in the rightmost operating area, the *update rate* starts to abruptly decrease when the problem expands to the swap memory space: the system enters the *overloaded operating range*.

Figure 3.4 shows the *whole-process update rate* for the three input kernels. Even though the absolute values are around 60% lower compared to the *update rate*, the three aforementioned operating ranges can be similarly observed. However the whole-process peak operating range is reached for slightly smaller problem sizes in comparison with the update rate peak operating points.

In comparison with other kernels, performances of the *deriche* kernel are lower and the *nominal operating range* is reduced significantly. The *whole-process update rate* exhibits

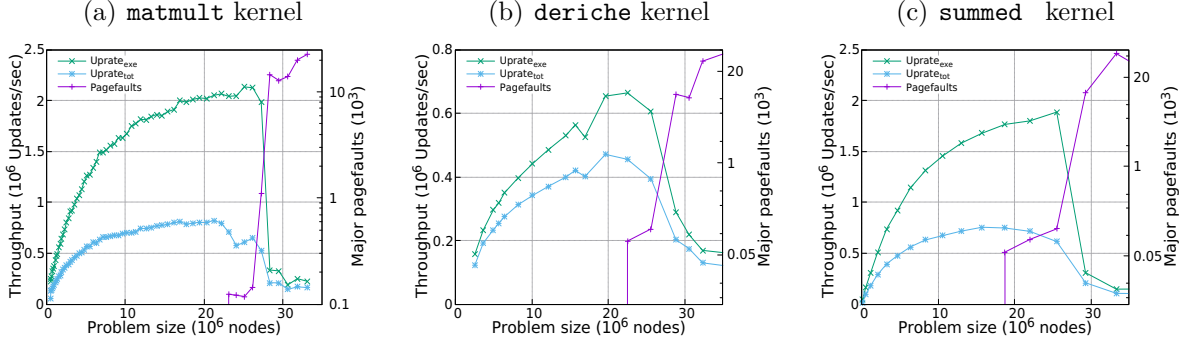


Figure 3.5: Performance and pagefaults figures for (a) the *matmult*, (b) the *deriche* and (c) the *summed* input kernels. PageFaults impact on throughput (right y-axis in log scale) on a 6-node cluster setting. Pagefaults are gathered using GNU time.

also a similar behavior, confirming that the *deriche*'s graph properties have an impact on performances. Further investigations showed that the *deriche* graph requires 126 iterations before convergence (respectively 21 and 15 for *matmult* and *summed*). This imbalance in terms of iterations might be explained by the fact that the *deriche* kernel is a larger and strongly connected kernel (about 4600 instructions in the kernel assembly code, around 30% more than *matmult* and *summed*). When comparing the execution of these three kernels in terms of active nodes, 99.7% of *deriche*'s nodes are inactive after 9 iterations. For *matmult* and *summed*, this happens after only 3 iterations.

### Overloaded range

Figure 3.5 shows, for a given cluster scale, the *update rate*, the *whole-process update rate* and the major PageFaults with respect to the problem size. Three regimes can be seen: until the *whole-process update rate* peaks, no pagefaults occur. Then, a small increase is seen as the *whole-process update rate* starts to decrease slightly. The amount of pagefaults in this area is kept quite low (a few hundreds) but is still non null. Finally, above a sufficiently large instance size, a steep rise in pagefaults is caused by numerous swap operations during the execution, thus significantly lowering performances. Taking this into account, increasing the memory per node available in the cluster should extend the *nominal operating range* by pushing the abrupt decrease towards larger graph instances.

Interestingly, the slight increase in major pagefaults visible before entering the overloaded range has no impact on the *update rate*, implying that these pagefaults occur outside the processing step. However, these pagefaults still have a cost, visible as the *whole-process update rate* starts to decrease slowly before processing-related swaps occur. When GraphLab allocates the execution engine after the commit of the graph structure and prior to the processing step, the OS is forced to swap out old data from the initialization part of the program. In other words, the physical memory limit is reached just before the execution step: the cluster is at the border of the *nominal* and *overloaded operating ranges*.

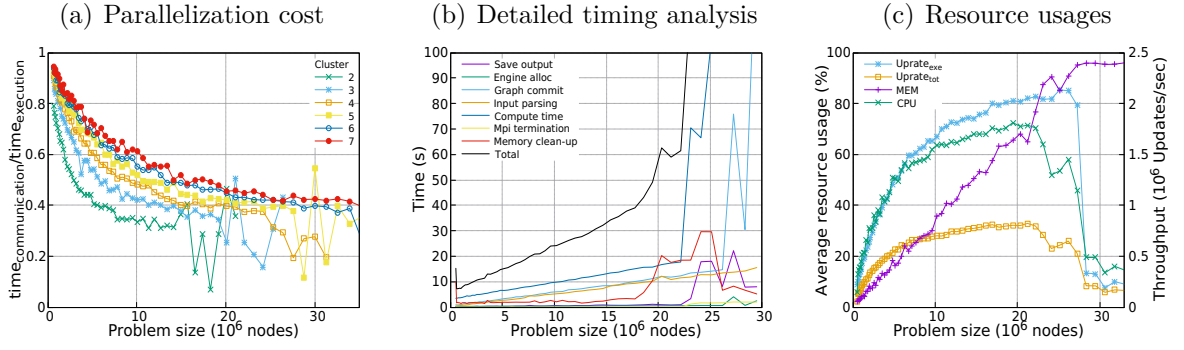


Figure 3.6: *Timing analysis for matmult kernel. (a) The parallelization overhead is the share of engine execution time not spent in processing. GraphLab was instrumented to extract the time overhead (be it synchronization or communication) in the execution engine. (b) Execution times of the program in a 4-node cluster setting. (c) Memory and CPU usages of the whole program (as given by GNU time), compared to throughput, for a 6-node cluster.*

## Underloaded range

The mitigated performance gain when adding computing nodes to the cluster in the *underloaded operating range*, is explained by the share of time spent in communications during the execution engine, as shown in Fig. 3.6(a). Indeed, with smaller input instances, a high ratio (close to 1) is observed, implying a high parallelization cost for the given input datasets. Conversely, with larger problem sizes, the parallelization cost decreases and tends to a constant 30 to 40% rate depending on the cluster size. Interestingly, this ratio limit slightly increases with the number of compute nodes. Extending this study to larger scale clusters might reveal scalability limits.

## Fine-grain timing analysis

The scalability of the distinct program steps was investigated with respect to the datasets size. Figure 3.6(b) shows execution times of the execution engine, the parsing step, the graph-commit, the engine allocation and the result saving, for various graph sizes, in a 4-node setting. Below the *overloaded operating range*, the program exhibits near linear scalability for every of its independent parts. In average, the pure compute time represents between 36.79% and 43.91% of the execution while the parsing/loading time represents between 21.08% and 24.41%, for the *matmult* kernel. When entering the *overloaded operating range*, the processing and saving times are rising steeply. This figure illustrates that when working after the *nominal operating range*, the bottleneck is reached during the processing time.

Throughput metrics and resource usage figures in a 6-compute node setting are shown in Fig. 3.6(c). The memory consumption is roughly linear with respect to the number of vertices, the number of edges impacting the slope of this curve. This figure also confirms that the decrease in performance happens when reaching physical memory saturation.

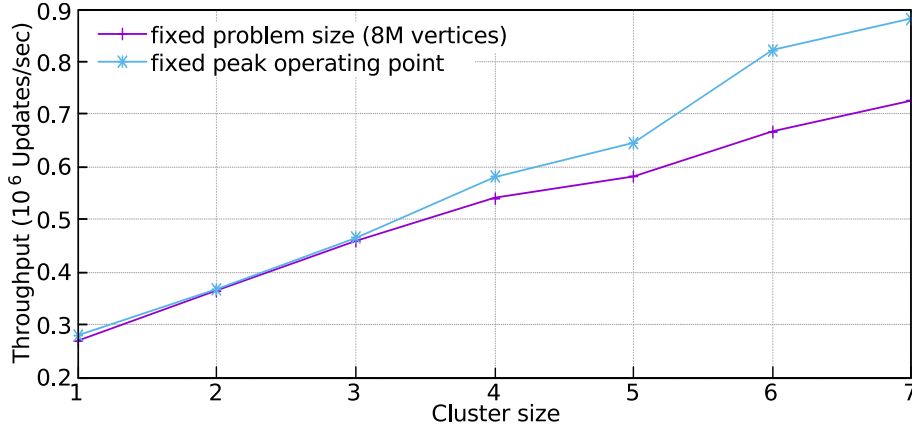


Figure 3.7: *Performance scalability.* The purple curve shows update rates for varying setup, at a given graph size in nominal and underloaded ranges. The blue curve shows the update rate in the peak operating point of each setup.

## Scalability

Performance scalability is shown Fig. 3.7. The *whole-process update rate* for a fixed graph size of around 8M nodes is plotted in purple. For this graph, a maximum performance gain of around  $\times 2.7$  is achieved with 7 compute nodes over the single machine GraphLab execution. However, this gain is starting to stall on the largest cluster configuration. Indeed, with 7 machines, the cluster leaves the *nominal operating range* for the *underloaded* one. These findings draw a scalability limitation in terms of performances: for a given problem size, the performance gain seems bounded and the addition of compute nodes does not improve performances significantly.

The blue curve in Fig. 3.7 shows the scalability at the peak operating point for a hardware setup. The *whole-process update rates* used are the peak values measured for each cluster sizes and correspond to a memory usage of approximately 70%. A gain up to  $\times 3.12$  can be observed over the single workstation configuration, indicating that performance scalability can be achieved by keeping the cluster around an optimal operating point, for increasing problem and cluster sizes.

## Asynchronous execution

Figure 3.8(a) and 3.8(b) show the *update rate* for the asynchronous engine (with the *vertex consistency model*), using traces from the *matmult* kernel as inputs. If a 3-area behavior can be observed as in synchronous execution, the asynchronous engine is outperformed in the *underloaded* and *nominal operating ranges*. On average, in the *nominal* (respectively *underload*) *operating range*, the performance overhead is of  $\times 1.51$  (respectively  $\times 1.21$ ). With growing input data, the asynchronous *update rate* rapidly tends to a constant value before decreasing abruptly when the cluster runs out of physical memory. The *nominal operating range* is wider than with the synchronous execution, despite being globally less efficient. This might indicate a better scalability with respect to the cluster size.

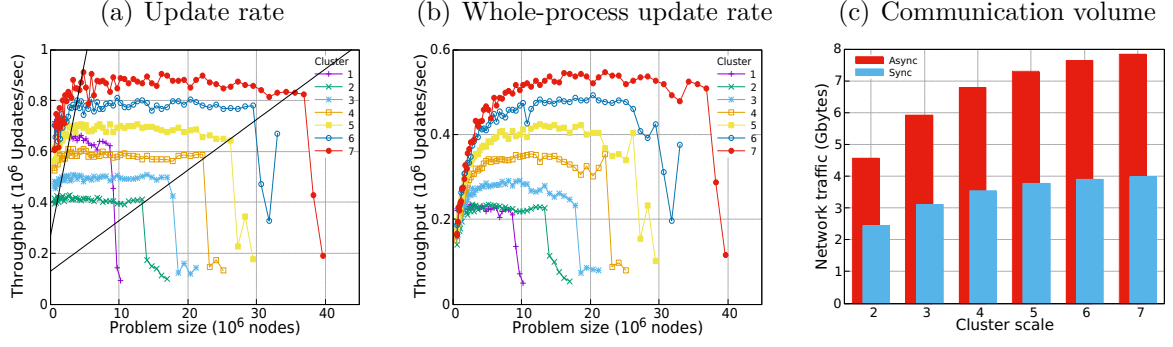


Figure 3.8: *Asynchronous execution performances.* (a) The asynchronous update rate is the ratio between the number of update functions completed and the execution engine time. This metrics illustrates raw distributed processing power. (b) The whole-process update rate is computed using the total number of update functions completed and the wall clock time of the program. (c) Evolution of communication volume sent over the network (14M nodes).

Figure 3.8(c) shows the communication volume when using the asynchronous and synchronous engine. The performance issue observed in Fig. 3.8(a) might be explained by a communication overhead generated when using the asynchronous engine, which is 1.93 times greater, on average. However, other studies usually exhibit a much higher communication overhead with state of the art algorithms and datasets [79].

Finally, tuning perspectives of the asynchronous engine remain unclear despite its appealing possibilities. For instance, parameters such as the number of lightweight threads and their stacksize can be tailored to suit the execution. However, they require a dedicated study in order to find to which extent tuning these specific parameters can improve performances. Further research should be made to identify optimization opportunities considering the suitability of asynchronous execution for this use-case. Moreover, mixed synchronous/asynchronous techniques as used in emerging frameworks [95] could be of interest.

### Ingress method and self-tuning

Though most of the experiments were conducted using the *oblivious* ingress method, different ingress strategies were evaluated: *random*, *oblivious* and the built-in automatic ingress method selection. The replication factor is the average number of compute nodes spanned by a vertice, it is thus a good metric to measure an ingress method quality.

In practice, *oblivious* always outperforms the *random* method, as shown in Fig. 3.9 with an average replication factor of 1.77 due to the low average degree of the graphs. A similarly low value can be observed in [91], on a graph of similar properties (RoadUS dataset, low average degree, comparable density). However the replication factor is lower in the use-case considered, possibly due to the smaller cluster used and the slightly lower average degree. In comparison, for a graph of much higher average degree (Twitter Graph, average degree of 35), the replication factor achieved is around 5 for an 8-node cluster [91].

The *automatic* strategy is at best optimal, for the 3-node and 5-node cluster configurations, and suboptimal in every other case. As described in the source code of GraphLab: when no ingress method is given, GraphLab — through its *automatic* strategy — successively checks if the number of processes spawned allows the selection of the *pds* and the *grid* methods. It then selects the *oblivious* method if both previous methods failed to be chosen. In the context of the considered use-case, this feature is not recommended and the use of the *oblivious* method should be preferred as it always experimentally achieves a better replication factor.

### 3.1.5 Synthesis

We evaluated the execution of GraphLab in the context of a trace-mining problem on a moderate scale commodity cluster for three types of input traces. We conducted experiments with varying cluster scales and input instance sizes to evaluate scalability under various conditions and observed the emergence of an optimal operating range which we have characterized. Despite its overhead, the provided GAS abstraction matches our algorithms and helps reducing the development time of the corresponding distributed graph application, leveraging improved productivity. Finally, though the experiments exhibit a bounded performance scalability for a fixed graph size, with increasing input problem and cluster sizes, performance scales linearly at a given optimal operating point.

Though our throughput and memory-related metrics facilitate *a posteriori* analysis, they can also serve as basis for the elaboration of performance prediction methods. However, without a thorough characterization of the graph input evolution, elaborating a performance model appears to be out of reach.

From the hardware point of view, these experiments confirmed that the limiting factor is not the processing power but the amount of memory and the network capacity. In this context, a similar study on more frugal architectures with comparable amount of memory

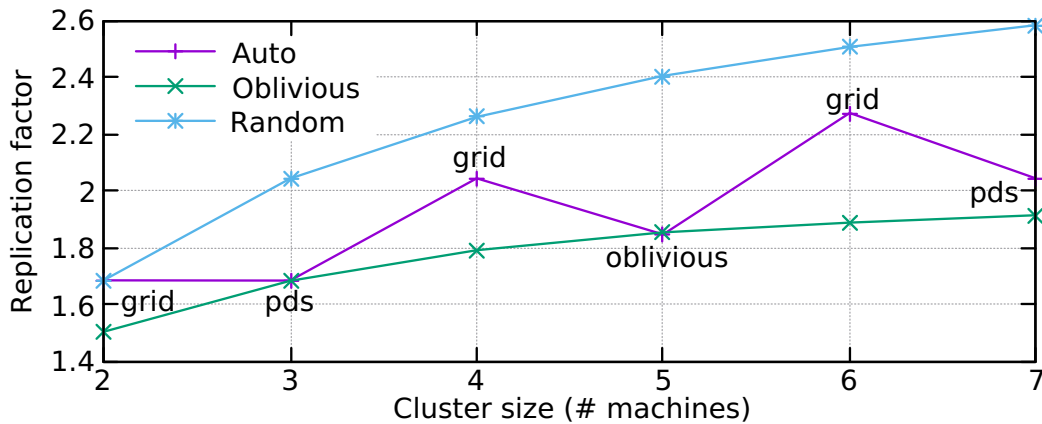


Figure 3.9: Performance comparison of ingress methods on a *matmult* kernel graph (14M edges, 10M nodes) processed on a 2- to 7-node cluster. A similar behavior is observed with the *deriche* and *summed* kernels.

is relevant to evaluate performances with respect to energy efficiency. With the growing need for processing power in the considered domain, the ability to design energy efficient infrastructures is relevant.

## 3.2 De Bruijn graph filtering for *de novo* assembly

The following section describes in details another use-case used in the thesis work. The algorithm presented belongs to the bioinformatics domain, and especially to the field of genome assembly. Thus, we introduce the context of the study, the problem modelization, the GraphLab implementation and some experimental results.

Traditional Sanger-based genome sequencing technology usually produce long, low error-rate data as basis for the assembly of whole genomes. However, only a limited number of genomes were fully assembled with this costly and lengthy technique. With recent advances in parallel sequencing technology, next-generation sequencers (NGS) have emerged, inexpensively producing vast amounts of shorter reads. This dramatic cut in sequencing costs has enabled many applications, including *Resequencing* of whole genome [24], *de novo* assembly [16] or metagenomic analysis [54].

Though generating large data samples from experimental runs was made easier, new challenges have emerged on the software side. Indeed, a major drawback of these new sequencers is their relatively high error rate [50] compared to previous slower technologies, requiring a greater algorithmic effort for the assembly step. With respect to computing aspects, the great increase in data volume is a key challenge in whole genome assembly. The non-negligible error rate combined with the short read length calls for preprocessing in the genome assembly software pipeline. In particular, to overcome read errors, greater coverages — *i.e.* increasing the average number of times a nucleotide is sequenced — are needed, thus increasing further the dataset sizes. Consequently, removing erroneous reads is key as it helps reducing memory consumption while improving result accuracy during further processing steps.

In this work, we focus on preprocessing aspects of genome assembly, and address in particular the following:

- We propose an extended study of de Bruijn graph properties in the context of genome assembly and introduce a definition of *operating coverage*.
- Based on a thorough analysis of the de Bruijn graph edge weight distribution, we propose a new filtering algorithm that relies on certainty propagation from reliable portions of the dataset.
- Finally, we propose an implementation of the algorithm leveraging GraphLab and demonstrating its scalability properties.



### 3.2.1 De novo assembly of short reads using de Bruijn graphs

#### Genome assembly

The assembly of a genome without a reference dataset to align sequencer reads with, is known as *de novo* assembly. Short reads are assembled in *contigs* – or long contiguous sequences – originally found in the genome being studied.

With the advent of NGS, researchers began considering *DNA* assembly from short reads, accounting for the low per-nucleotide cost and shortened experiment time. However, as reads get shorter and of poorer quality, greater software complexity is required to perform the assembly step, along with larger scale hardware infrastructures.

Although sequencing techniques have been improved, reads from NGS often come with a non-negligible, technology-dependent error rate [41]. This rate is usually of the order of 1% and often significantly less [52] before read filtering. To mitigate this effect, a deeper sequencing (or higher per-base coverage) is required. In the context of high-throughput sequencers, errors usually fall in the following categories:

- *Insertion*: A base is erroneously inserted between two others (Fig. 3.10(b)).
- *Deletion*: A base is skipped from the read (Fig. 3.10(c)).
- *Substitution*: A base is substituted for another or encoded with a 'N' when ambiguous (Fig. 3.10(d)).

Insertions and deletions — or *indels* — are however relatively rare compared to substitutions. A detailed review of errors in NGS technology was presented by Minoche *et al* [52].

Mainstream assembly paradigms include three different kinds of techniques, being greedy methods, Overlap-Layout-Consensus (*OLC*) and de Bruijn graph (DBG) based approaches. However, *OLC* and de Bruijn-based strategies are more common nowadays, as greedy methods are not designed to handle additional information (e.g. mate pairs) with ease [72]. Moreover, greedy implementations usually have a high memory footprint [53].

*OLC* methods rely on finding overlapping reads through a computationally intensive all vs. all comparison. Then, this step is followed by the search of Hamiltonian paths in the graph made from overlapping sequencing reads – a problem known to be NP-complete in its classic formulation. Both PCAP [13] and Celera [9] use this method.

Recently, a greater interest have been raised in de Bruijn-based methods [10] as *OLC* and greedy approaches have proven not to be a great match for short read assembly of large genomes. In the following, we introduce de Bruijn graph approaches, along with state of the art tools for parallel *de novo* assembly. Finally, read error mitigation and other challenging aspects are introduced.



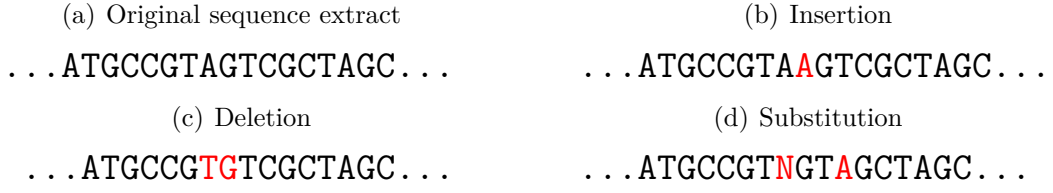


Figure 3.10: Read errors in sequencer data. (a) Extract from the genome being sequenced, before alteration. (b) insertion: a base (in red) has been erroneously inserted in the read. (c) deletion: a base has been deleted in the read between the two red characters. (d) substitution: two substitutions are visible (in red), an uncalled base, encoded 'N', and a miscalled base, reading an 'A' instead of the expected 'C'.

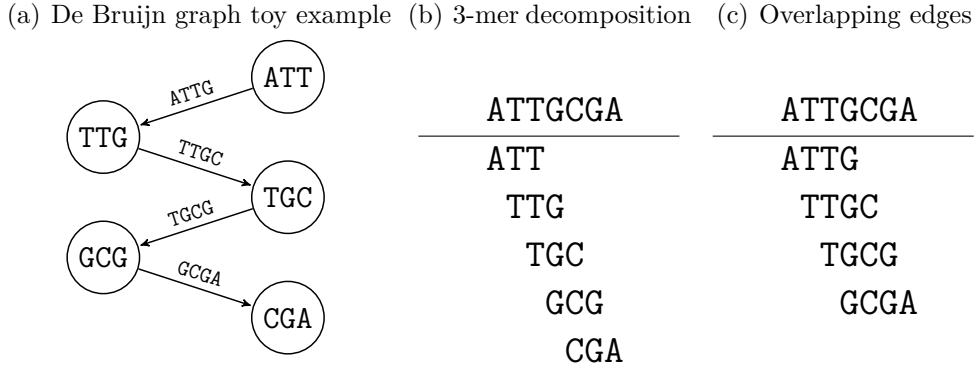


Figure 3.11:  $k$ -mer decomposition and associated de Bruijn graph of a sequence. 3.11(a) De Bruijn graph modeling the sequence **ATTGCGA**, for  $k = 3$ . 3.11(b) 3-mer decomposition of the sequence. 3.11(c) Associated 4-mer (edges) linking nodes in the graph.

## De Bruijn graphs

De Bruijn graphs (DBG) have been used to address the problem of short read assembly of large genomes. They have several advantages that makes them candidates of choice over *OLC* methods. In particular, they handle with a lower memory usage the high information redundancy caused by the high coverage of datasets generated by NGS. Moreover, the assembly is achieved by determining an Eulerian path in a de Bruijn graph which is intrinsically more tractable compared to finding Hamiltonian paths.

Let's consider a DNA sequencing model featuring a  $n$ -base (or nucleotide) long target genome. Each read of varying length contain  $l$  bases (A, T, G or C) of the genome and may be decomposed in  $(l - k + 1)$  overlapping  $k$ -mers. Formally, a  $k$ -mer refers to every  $k$ -base long substrings contained in a read. A graph is then constructed where nodes are  $k$ -mers and edges are  $(k + 1)$ -mers linking overlapping  $k$ -mers. As an example, the read **ATTGCGA** can be decomposed in 5 3-mers, as shown in Fig. 3.11(b), that constitutes nodes in the associated de Bruijn graph shown in Fig. 3.11(a). In NGS, reads are considered short ( $l < 250$ ) but this is compensated by producing enough reads to cover the whole sequence or targeted region.

However, read errors alter the graph by adding unnecessary edges and vertices, as shown in Fig. 3.12. Mid-read errors create *bubbles* – an additional  $k$ -long path between two nodes — in the graph. If the error appears closer to the end of the read, *dead-end*

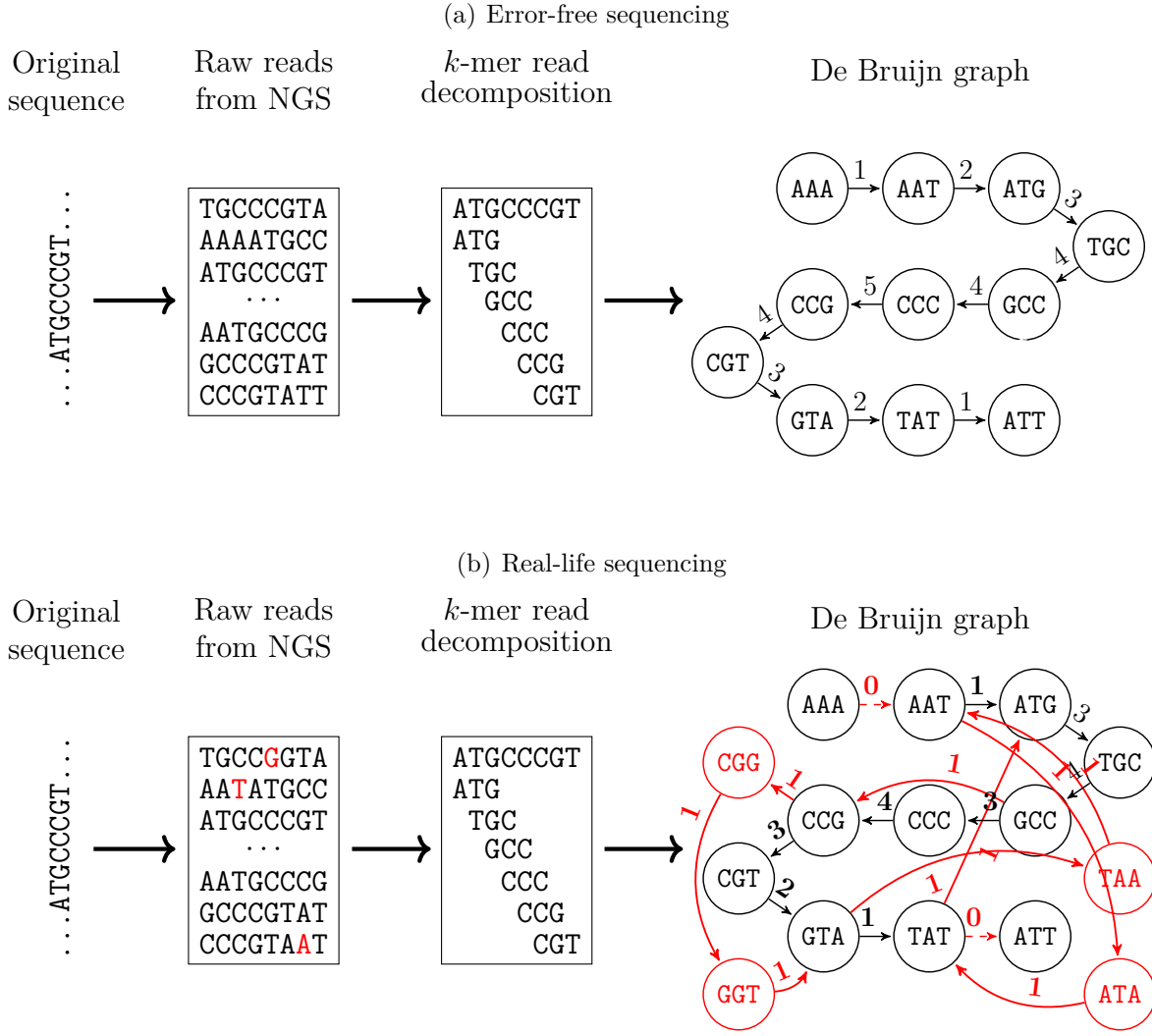


Figure 3.12: Typical sequencing experiment set-up. On the left side, the original targeted sequence is sequenced, producing raw reads. Each read is decomposed into  $k$ -mers. A De Bruijn graph is then constructed where each node is a  $k$ -mer and each edge, an overlapping  $(k + 1)$ -mer holding its occurrence count. (a) shows the idealized, error-free set up. (b) illustrates the impact of three substitutions in the input datasets. Erroneous edges and nodes in red. Changed edge weights are in bold.

branches or tips, can be observed in the DBG. Assemblers must be able to mitigate these alterations in order to reconstruct the targeted genome.

### Parallel softwares for *de novo* assembly

A wide range of DBG-based tools have been developed for *de novo* assembly, targeting different sequencers. Recently, parallel assemblers have been used to tackle the growth in datasets, in order to achieve the assembly of genome as large as the human one. All of these approaches are de Bruijn based, however they differ in their parallel programming models or their preprocessing routines for read error mitigation. While some rely on traditional shared-memory multithreading [35, 42] or distributed multiprocessing [40, 54, 113], some works try to get the best from emerging cloud infrastructures [49, 90].

The *Ray* [41] parallel assembler processes data from a mix of sequencing systems and is therefore not linked to a particular technology. However, though this assembler is DBG-based, its assembly method does not rely on Eulerian walks.

Another parallel assembler is *ABYSS* [40], which exhibits a low memory overhead albeit being relatively slow [46]. Moreover, *ABYSS* may be used for additional purposes, such as transcriptome assembly [72].

*MERmaid* [113] is an assembler based on *Meraculous*, a state-of-the-art whole genome assembler for NGS data. This assembler is able to generate contigs from short reads in parallel, but does not provide a parallel scaffolding step. An interesting feature of this framework is that it uses a Bloom filter to probabilistically filter out irrelevant  $k$ -mers.

While previous assemblers rely on traditional message-passing backends for parallelization, *Contrail* [49] is a MapReduce-based framework implemented on top of Hadoop. However, little public information is available on the framework at the time of writing. *Spaler* [90] relies as well on cloud infrastructures and leverages the Hadoop-based Spark/GraphX framework for the *de novo* assembly of short reads through sub-paths merging.

Finally, *Velvet* [35] and *SOAPdenovo* [42] are two legacy, DBG-based, multi-threaded assemblers. *Velvet* shows that high-quality assemblies could be produced with very short reads (30 base-pair per read) leveraging high coverage datasets, although it requires large amounts of RAM [72]. *SOAPdenovo* comes as a memory efficient assembler with robust error-correction and scaffolding possibilities. It is recommended for very short read assembly of large genomes, according to Zhang *et al* [53], and is also used for transcriptome and metagenome assembly [72].

A particularly challenging task faced by genome assemblers is the mitigation of sequencing errors. Errors hamper assembly performances in multiple ways, *e.g.* increase in memory overhead or decrease in result quality. Amongst regular assembly toolchains, a vast majority include error-correcting routines. For example, a widespread trend is to discard reads containing any uncalled bases (encoded with 'N' or a dot) and/or to remove *weak k*-mers (*i.e.*  $k$ -mers with low counts) using a user-defined threshold.

Standalone software tools were developed to address error-correction. *EDAR* [50] and *Hybrid-SHREC* [47] are examples of such tools developed aside mainstream toolchains. *EDAR* comes as a complex preprocessing algorithm suite. While using this suite gives more accurate assemblies, no clues on execution performances or scalability are given in [50]. *Hybrid-SHREC* provides a method for error-correction implemented in Java using parallel threads and supporting mixed read sets. Finally, some work also targets post-assembly [34] or long-read DBG [102] error-correction.

Numerous works have demonstrated that the assembly of large genome using short reads is computationally challenging. In particular, in the context of de Bruijn graph-based frameworks, large amounts of memory are required, hence calling for larger scale, distributed systems. While many parallel assemblers/preprocessors are based on estab-

lished parallel and distributed programming frameworks, few of them leverage emerging parallel libraries for graph-processing such as GraphLab, to the best of our knowledge.

### 3.2.2 Problem modelization

De Bruijn-based DNA assembly requires to compute Eulerian paths in a graph. However, the DBG is polluted by read errors which shall be removed prior to any attempt at building an Eulerian walk, as they increase the processing workload and hamper the quality of the results. However, being able to differentiate a correct  $k$ -mer from its erroneous counterpart is a difficult task. This section discusses the nature of the DBG and the impact of  $k$ -mer sizes and read errors on its distribution to identify appropriate filtering strategies.

#### Coverage and $k$ -mer spectrum

The *coverage*, or *sequencing depth*, represents how many times a base of the original genome has been *read*. A coverage of 40 means that, in average, every nucleotide in the genome is read 40 times and, in practice, the nucleotide coverage distribution is a Poisson law [87].

Due to the  $k$ -mer decomposition of sequencing reads, DBG edges follow a similar distribution. In the context of our experiments, we define the *average theoretical coverage* (noted  $c_t$ ) of a  $k$ -mer as follows, with  $c$  being the sequencing depth and  $r_l$ , the average read length.

$$c_t = c \frac{r_l - k}{r_l}$$

As one can observe, for a given sequencing configuration  $(c, r_l)$ ,  $c_t$  decreases linearly with increasing values of  $k$ .

The  $k$ -mer spectrum – or  $k$ -mer frequency distribution – of input data can be useful to infer information from the dataset. On real data, the expected distribution can be analyzed along three zones:

- A decreasing exponential at very low frequencies composed of numerous unique sequences obtained by alterations during sequence reads. Usually these weak  $k$ -mers have no biological significance and are akin to noise in the dataset. Most assemblers overcome this noise by discarding  $k$ -mers with a frequency lower than a user-defined threshold
- A Poisson-law distribution around the operating coverage value, following the base coverage distribution, although altered by the  $k$ -mer decomposition [10, 41].
- Harmonics of the coverage peak may indicate repeats in the sequence that were greater than a  $k$ -mer. The smaller  $k$  is, the higher the probability of having harmonics.

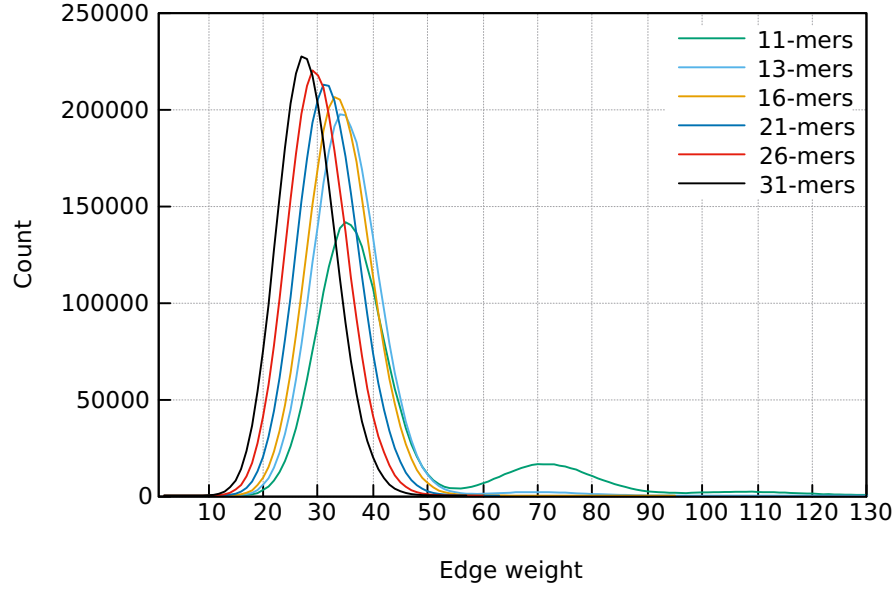


Figure 3.13: *De Bruijn graph edge weight distribution of the 3M-base synthetic dataset ( $r_l = 100$ ), for varying  $k$  decompositions. With smaller  $k$ , an harmonic peak is rising around twice  $c_t$ . As the dataset is error-free, no weak, low-frequency errors are visible in the spectrum.*

The analysis of the  $k$ -mer spectrum can help in determining experimental parameters. As an example,  $k$  can be adjusted to remove or reduce the amount of peak harmonics in the spectrum. Similarly, the threshold value filtering out weak  $k$ -mers caused by read errors can be set by analyzing the  $k$ -mer distribution.

The edge weight distribution of a dataset is shown in Fig. 3.13, for various  $k$  values. As this dataset is error-free, no low-frequency peak is visible. With  $k$  ranging from 16 to 31, no harmonics can be seen, contrary to lower  $k$  value, where a peak is observed around twice the peak operating coverage. Harmonics appear for smaller  $k$ -mers not selective enough, thus seen twice or more in the genome, where longer  $k$ -mers would have discriminated the two sequences in two different  $k$ -mers. Such repeats create several candidate Eulerian paths that have to be discriminated after the assembly to select the solution representing the target genome.

### Impact of the sequencing depth on the $k$ -mer spectrum

The increase in error-rate has a visible impact on the  $k$ -mer spectrum, as visible in Fig. 3.14. At the fixed sequencing depth of 40 used in the experiment, it becomes almost impossible to process datasets with noise levels of above 3.75%. Indeed, in noisier dataset distributions (such as 7.50 and 15.00%), the peak of true edges are merged in the low frequency error-induced peak.

This effect can however be mitigated by increasing the coverage, as shown in Fig. 3.15, where the edge distribution can be improved by leveraging deeper sequencing. However, this higher coverage is performed at the expense of an increased memory consumption, calling for larger scale machines.

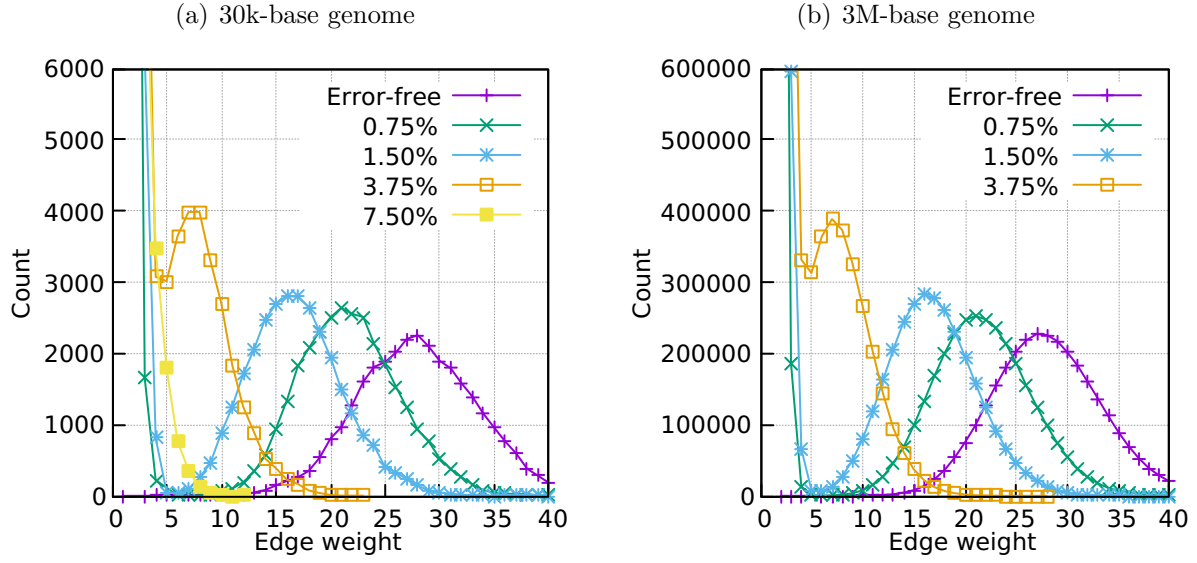


Figure 3.14:  $k$ -mer spectrum of the (a) ID0\_40X dataset (30k-nucleotide, 31-mers, 40X coverage) and (b) IDN\_40X dataset (3M-nucleotide, 31-mers, 40X coverage) with increasing error rates. The distribution peak is pushed toward lower edge weights until noise and true edges become indistinguishable.

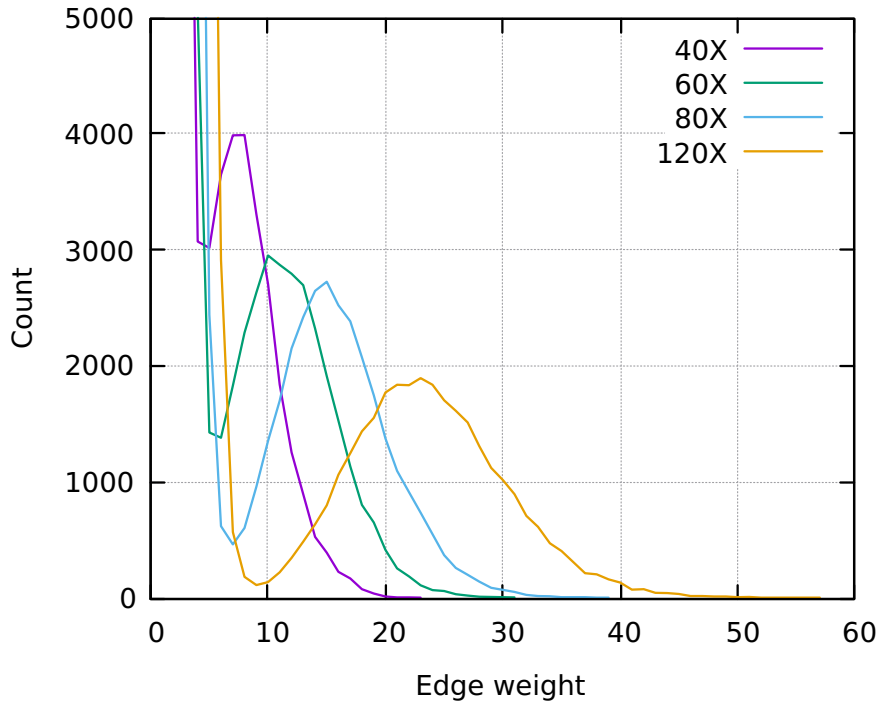


Figure 3.15:  $k$ -mer spectrum of the ID0\_40X dataset (30k-nucleotide, 31-mers, 3.75% error per read) with increasing sequencing depths. Increasing the coverage help in mitigating the effect of the noise, despite an increase in memory consumption due to redundancy.

### Corrected operating coverage

As our preprocessing strategy targets the identification of correct  $k$ -mers in the graph based on their weight, having a good estimate of the average weight for a valid  $k$ -mer is key.

Interestingly enough, we observe that the *operational coverage* decreases with increasing error rates on  $k$ -mer spectrums visible in Fig. 3.14. As the synthetic dataset generator used performs random base substitutions, each base has an alteration probability of  $e$ . Thus, the probability that each base in a  $k$ -mer is correct is  $(1 - e)^k$ . Consequently, the operating coverage is altered in a similar fashion, hence we calculate the *operational coverage*  $c_{op}$  using  $c_t$  and the error rate  $e$ , with the following formulation:

$$c_{op} = c_t \times (1 - e)^k$$

Conversely, the above formula helps assessing the minimum required sequencing depth to achieve a defined  $c_{op}$  for a given configuration  $(k, r_l, e)$ .

### 3.2.3 DBG preprocessing algorithm

Having analyzed the nature of the DBG and its  $k$ -mer weight distribution, this section presents a novel DBG preprocessing algorithm for *de novo* assembly. This section details the rationale behind this algorithm, introduces the GraphLab graph-processing framework and how it was leveraged to implement this preprocessing.

#### Edge capacity propagation algorithm

The target of our DBG preprocessing algorithm is to remove invalid  $k$ -mers and to identify for each  $k$ -mer the number of times it appears in the source genome. We call this  $k$ -mer occurrence count a *capacity* as it will be used for DBG Eulerian-walks during assembly stages. Edge capacities can only be derived from their DBG weight counterpart. However, as previously stated, DBG weights follow a Poisson distribution which means that there is no straightforward link from a given weight to a given capacity.

The rationale behind the proposed method is to rely on edges whose capacities can be determined at initialization and propagate their capacities to yet-unvaluated neighboring edges, as pictured in Fig. 3.16. Such edges are the ones which exhibit a weight close to the determined operation coverage  $c_{op}$ . Hence, we define a tolerance range  $tol$  as the interval around  $c_{op}$  for which a  $k$ -mer weight is close enough to  $c_{op}$  to be considered as valid. Consequently, an edge is assigned a capacity of 1 if its weight is within  $c_{op} \pm tol$ . Such edges are called *seeds* as they serve as source to the capacity propagation. The tolerance can be tuned to generate more or less *seeds* at the beginning of the algorithm. If the tolerance is too high, false positives may rise in the resulting graph, *i.e.* erroneous  $k$ -mer may be considered valid. On the other hand, a too selective tolerance may slow down the execution or block capacity propagation.

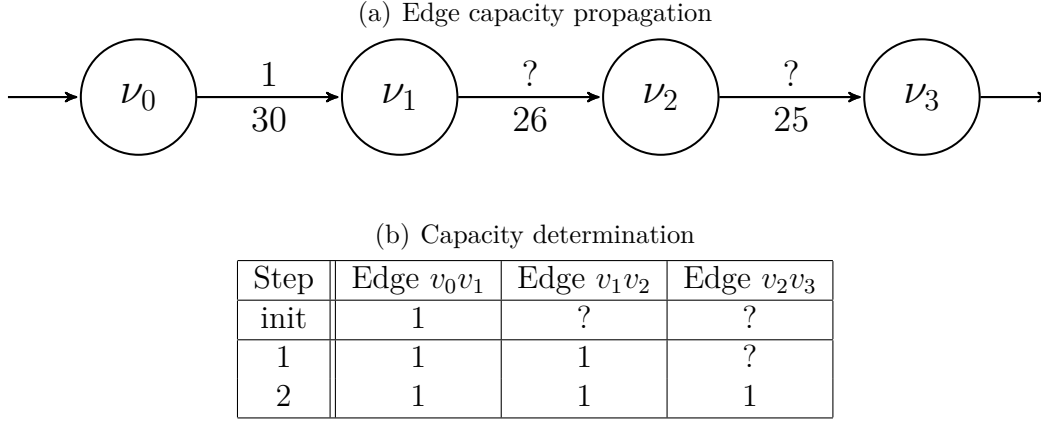


Figure 3.16: *Edge determination using capacity propagation.* Figure (a) shows a graph part over which is executed capacity propagation. Edges have capacities (up, '?' is undetermined) and weight (down). Table (b) details the execution of the algorithm. Edges  $e_{12}$  and  $e_{23}$  are valued using the initially determined capacity of the edge  $e_{01}$ .

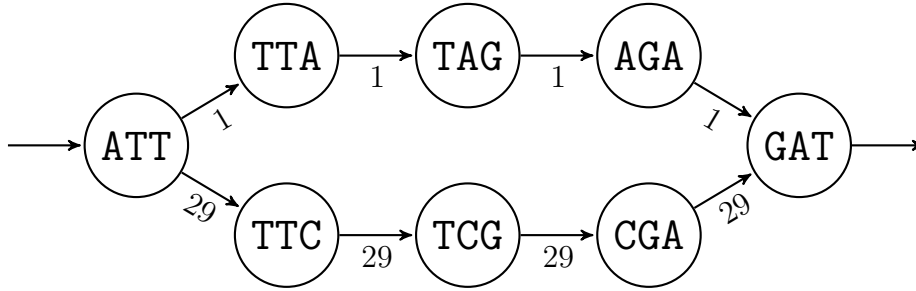
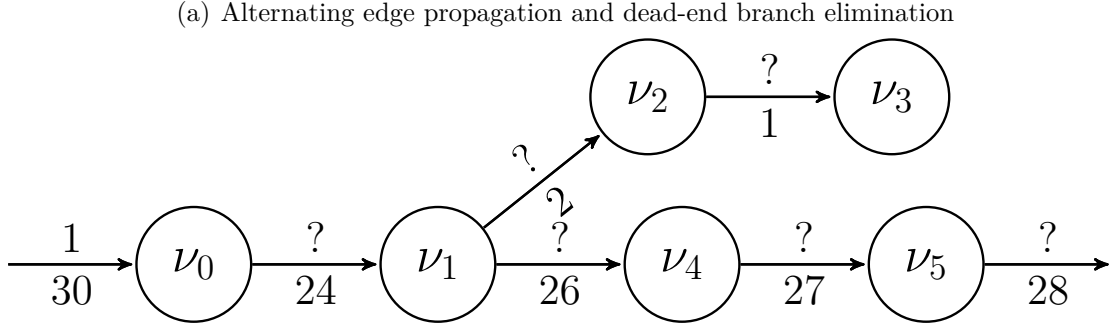


Figure 3.17: *Error-induced bubble in a 3-mer de Bruijn graph.* The expected sequence **ATTTCGAT** is sequenced using a 30x sequencing depth. Considering an error in a read, where the **C** is substituted for a **A** (e.g. **ATTAGAT** is read instead, in one read), a  $k$ -long bubble in the graph can be observed. This bubble can be mitigated using an adequate threshold value (low weight edges would be removed) or by allowing a greater tolerance around the coverage (edges of weight 29 would be considered determinable).

As previously mentioned, the DBG contains *weak*  $k$ -mers, whose weight is insignificant compared to the expected  $c_{op}$  weight. To quickly remove such nodes/edges in the DBG we use a threshold  $th$  under which edges are assumed to be invalid. A too high threshold value may filter out valid  $k$ -mers that were not well covered by the sequencing. Conversely, a too low threshold value will keep many invalid  $k$ -mers that will hamper execution speed and increase memory requirements. Figure 3.17 illustrates how *bubbles* can be mitigated by adequately setting such a threshold.

The proposed algorithm is structured in four steps. During the *parsing* step, we create the DBG from source NGS sequence reads. Then, the *pre-filtering* step removes all edges which weight is lower than  $th$ . In the *capacity initialization* step, we assign a capacity of 1 to edges which weight are within  $c_{op} \pm tol$ . All other edge capacities are left uninitialized. Then an iterative *propagation* step starts, during which every node checks its input and output edge capacities to determine its own capacity. If it succeeds and only one of its edges is undetermined, the node can update this edge to balance its input/output edge





(b) Alternating edge capacity determination

Step	Dir.	$e_{-0}$	$e_{01}$	$e_{12}$	$e_{23}$	$e_{14}$	$e_{45}$	$e_{5+}$
init		1	?	?	?	?	?	?
1	out	1	1	?	?	?	?	?
2	in	1	1	?	0	?	?	?
3	out	1	1	?	0	?	?	?
4	in	1	1	0	0	?	?	?
5	out	1	1	0	0	1	?	?
6	in	1	1	0	0	1	1	?
7	out	1	1	0	0	1	1	1

Figure 3.18: Determination of edge capacities over a graph containing a dead-end branch eliminated by zero-propagation. Figure (a) shows the graph processed, with only a single determined edge ( $e_{-0}$ ) and a dead-end branch (nodes  $\nu_2$  and  $\nu_3$ ). Tab. (b) details the edge determination in the graph along iterations. The alternating mechanism enables the processing of out- (respectively in-) edges during odd (respectively even) iterations as shown by the *Dir.* column. At the end of the execution, the erroneous tip ( $e_{12}$  and  $e_{23}$ ) is removed (i.e. capacities are set to 0).

capacities. Indeed, valid  $k$ -mers shall be connected to their predecessor/successor in the genome as soon as an error free  $(k + 2)$ -mer read sequence covers it. The *propagation* step ends when no edges are left undetermined or if the number of undetermined edges is constant between two successive iterations.

Source (respectively sink) nodes of the DBG have no input (respectively output) edges, therefore they always get assigned a capacity of 0. Such nodes are mostly generated by read errors at the beginning/end of a read. As a result, propagation of 0 capacities helps removing *dead-end branches* in the graph<sup>1</sup>. An example of *dead-end branches* removal from a graph using zero-capacity propagation is shown in Fig. 3.18.

As one can see this algorithm can be extensively tuned by setting the threshold  $th$ , and the tolerance  $tol$ , as illustrated in Fig. 3.19. It is also impacted by the  $c_{op}$  which depends on the NGS technology (through the read length and the read error rate) but also on the  $k$ -mer size which is user defined. In the following, we present the GraphLab implementation of our algorithm.

<sup>1</sup> However, this also inserts 0 capacities at the real source/sink of the genome which can trim the genome boundaries. This has a very limited impact on genome assembly, as relevant materials are usually not located on boundaries.

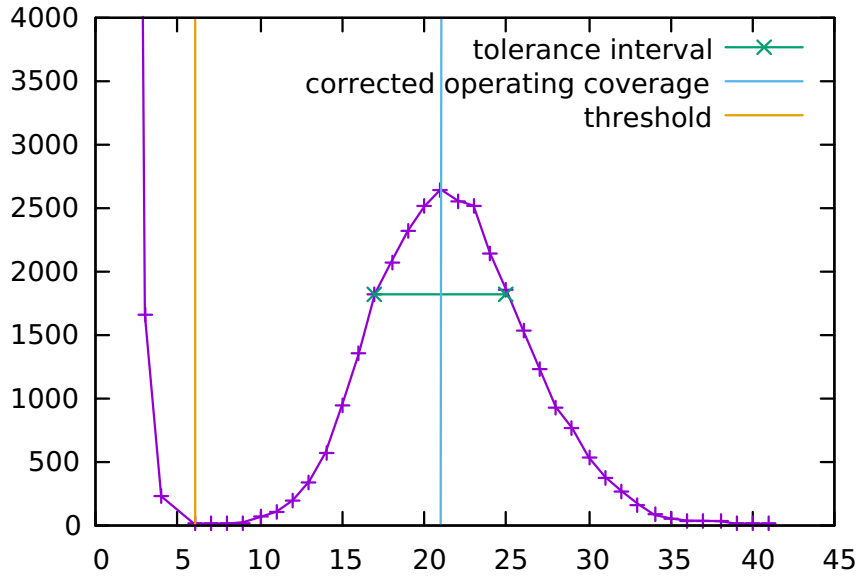


Figure 3.19: 31-mer spectrum of the 3M-base dataset illustrating the impact of parameters. A threshold value of 6 is appropriate to remove weak  $k$ -mers. An example of tolerance interval is depicted as well as the operating coverage.

### 3.2.4 Vertex-centric implementation of the algorithm

GraphLab programs are usually constituted by a parser, a graph structure and several execution engines driving the execution of *update functions*. Prior to parsing, the input file containing the reads is split across cluster nodes. This step is not mandatory but allows to execute the parsing step in parallel. At parsing, each read is decomposed in overlapping  $k$ -mers in order to create graph nodes and edges.

Concretely, a graph node is a class describing a  $k$ -mer and embedding information such as an ID, and the  $k$ -mer meta-information such as inbound/outbound capacities. Graph edges hold a pair of values, being the edge weight and its capacity. Once the parsing step is done, the data structure holding the graph is committed. Then, a folding operation is executed over edges to merge duplicates into single edges thus assigning their  $(k + 1)$ -mer count to their weight. At the end of this folding step, both the *pre-filtering* and the *capacity initialization* steps are performed, resulting in a cleaned-up graph with assigned seed capacities.

The main execution engine is then allocated and *update function* can start their iterations. To comply with GraphLab's programming model, the *update function* is divided into three minor steps described hereafter.

- *Gather* step: every vertex independently gathers its input and output edge capacities.
- *Apply* step: every node determines its own capacity and attempts to determine its edges' ones. This function is executed following an *alternating* scheme. On odd

(respectively even) iterations, every node attempts to compute its own capacity using their output (respectively input) edge capacities, and to determine their input (respectively output) edges' ones whenever applicable. This *alternating* behavior ensures that no write conflicts occur on edges, as no concurrent nodes attempt to write on a same edge.

- *Scatter* step : Any edge determination is committed (if applicable). Then, every node is either rescheduled or halted if it has converged (*i.e.* when every neighbour edge capacities are set).

The algorithm is executed using the synchronous execution engine, that is, supersteps are synchronized to ensure data consistency. This implies that no vertice can start an iteration until every active vertices have terminated the previous one. The engine then considers to have converged when there is no more active node, implying that no undetermined edge remains. Convergence is also reached when it is impossible to determine any more edges, *i.e.* when the number of determined edges is constant, the execution engine is then stopped.

### 3.2.5 Materials and methods

#### Accuracy metrics

In order to assess the accuracy of the introduced filtering algorithm, we have defined a set of metrics, defined in the following. The misvaluation rate  $\mu_m$  is the share of edges which capacities have been erroneously set. As synthetic datasets were used, the reference genome is available to measure the three following errors for each run:

- *False negative*: We define *false negatives* as edge capacities erroneously determined to a lower value than expected. A particular case is when 1-valued edge capacities are set to 0. The amount of *false negatives* is evaluated by counting the share of edges with a null capacity that do belong to the original genomes.
- *False positive*: Conversely, we define any edge which capacity has been over-defined as a *false positive*, and specifically, erroneous edge capacities set to 1.

The misvaluation rate  $\mu_m$  is formally defined as follows, where  $E_{\text{DBG}}$ ,  $E_{\text{fp}}$  and  $E_{\text{fn}}$  are respectively the edge count of the graph, of the false-positives and the false-negatives.

$$\mu_m = \frac{E_{\text{fp}} + E_{\text{fn}}}{E_{\text{DBG}}}$$

Another useful metric is the *undetermination rate*  $\mu_u$ , which is the ratio of the remaining undetermined  $(k + 1)$ -mers over the total number of  $(k + 1)$ -mers:

$$\mu_u = \frac{E_{-1}}{E_{\text{DBG}}}$$

Similarly the *correct determination rate*  $\mu$  can be defined using the previous metrics:

$$\mu = 1 - (\mu_u + \mu_m)$$

This metric is of interest as it assesses the rate of good decisions, *e.g.* erroneous  $k$ -mer capacities cleared and correct  $k$ -mer capacities set to the appropriate value.

Finally, the *reconstruction rate*  $\tau_r$  is defined as the ratio of genome edges correctly determined over the total genome edge number, with  $E_{co}$  being the set of edges correctly determined belonging to the genome and  $N$ , the genome length in base:

$$\tau_r = \frac{E_{co}}{N - k + 1}$$

Though this last metric is, as  $\mu$ , an accuracy metric, it differs in that it represents the proportion of genome that can be reconstructed — it does not account for error-removal efficiency. These metrics enable a standalone validation of the method on synthetic benchmarks and were used to evaluate the impact of noise over synthetic benchmarks and infer the robustness of the method.

### Synthetic dataset generator

In order to validate the implemented algorithm and for development purposes, a two-stage parallel dataset generator was implemented and used. In a first step, a synthetic genome sequence is generated using a random number generator. In a second step, reads are performed in parallel over this sequence, aiming to achieve a given sequencing depth. Specifically, every time a read is performed, a random number is used as a start position for the read.

In order to measure up to which error rate the proposed algorithm performs correctly, we implemented a tunable error function in the generator. Hence, each read is individually altered according to a given error rate. When a dataset is referred to as having a 1% error rate, it implies that on average, 1% of the bases have been altered in every read. In the error model used in these experiments, only substitutions are performed, at random positions within the reads. Despite its apparent simplicity, this model remains practically relevant, as base substitution is by far the most common kind of detected errors in reads [52].

The main benefit of using synthetic datasets lies in the possibility to check and validate the output of the algorithm without requiring a proper assembly stage. It also helps to understand how the algorithm would perform for various NGS.

### 3.2.6 Experimental results

Experimental results are presented in this section. Unless stated otherwise, results are obtained on the HC platform with a 4-node cluster configuration, datasets have a se-

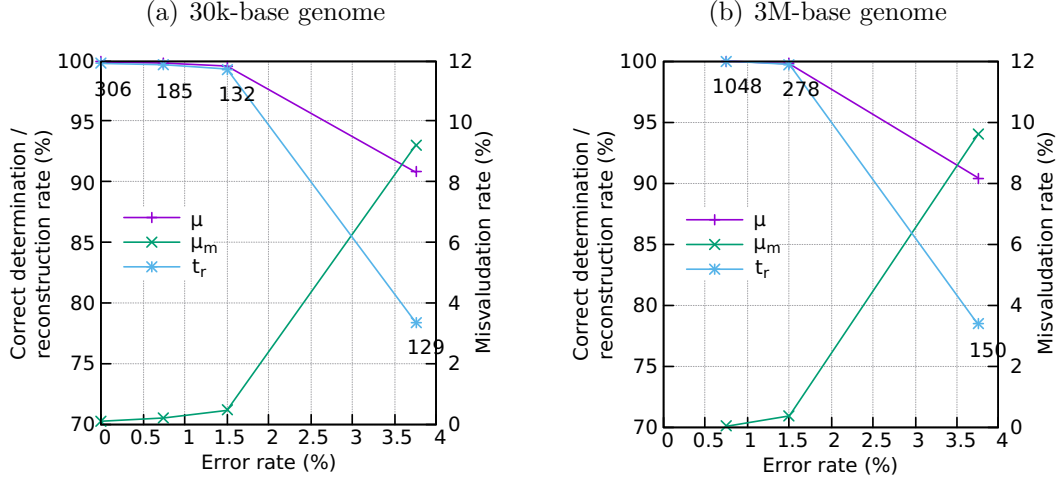


Figure 3.20: Algorithm accuracy with varying error rates for (a) 30K-base and (b) 30M-base genomes (tolerance set to 1, threshold set to 6, coverage is 40). Required iterations to convergence is reported on the  $\tau_r$  plot for each dataset.

quencing depth of 40 and a value of 31 is used for  $k$ . First, a review of the efficiency of the algorithm with increasing error rates is presented. Then, the impact of the tolerance parameter is investigated. Finally, the operating performance and scalability of the algorithm is discussed with respect to increasing cluster sizes.

### Increasing NGS error rate: impact on accuracy and algorithmic limits

In order to assess the algorithm accuracy, experiments were performed using datasets with increasing error rates at a modest sequencing depth of 40. As shown in Fig. 3.20, the algorithm is able to process datasets up to an error level of 3.75% with a correct determination rate of more than 90% with both datasets. This rate reaches more than 99% up to a 1.5% error rate, with a  $\tau_r$  similarly high in both datasets. This is compatible with real-world datasets, as NGS usually produces sequencing data with an error rate around 0.5–1.5% [52]. As a comparison, for a target genome of 3M base, the unfiltered, raw DBG with a 3.75% error-rate has around 40M edges, hence 92% of the edges are erroneous.

When increasing tolerance, an improvement in terms of convergence and accuracy is observed in Fig. 3.21(a). The convergence improvement can be explained by the fact that more seeds are initially determined, thus propagating edge capacities faster. The accuracy improvement lies in both an appropriate thresholding reducing the possibility of misdetermination by removing weak  $k$ -mers, and a better capacity propagation hampering misdetermination due to zero-propagation at boundaries. However, when tolerance is too high, a rising false positive rate is observed, lowering accuracy. This fact is further increased with low operating coverage (*e.g.* high error rate and low sequencing depth).

Finally, in Fig. 3.21(b) is depicted accuracy for the 30k-base genome for an error-rate of 3.75% with varying sequencing depths. The increase in sequencing depth improves  $c_{op}$ ,

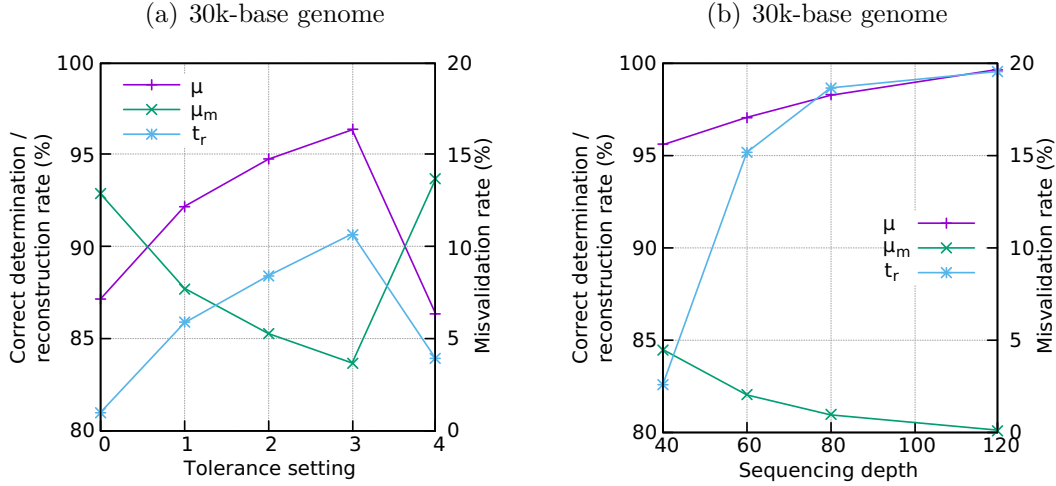


Figure 3.21: (a) Impact of the tolerance parameter on accuracy of the processing of the 30k-base genome (threshold set to 4, error-rate: 3.75%). (b) Impact of the sequencing depth on accuracy for the 30k-base genome (threshold set 5, error-rate: 3.75%).

as visible in Fig. 3.15, and improves as well the accuracy. In particular, a  $\tau_r$  of 82.57% is observed with  $c = 40$  while  $\tau_r$  of at least 95% are observed for  $c > 60$ . However, though result improvements are interesting, it should be noted that the coverage increase required for such purpose causes an important growth in input data volume.

### Algorithm behavior and scalability performances

The execution is guaranteed to converge as the number of active nodes can only decrease or remain constant over time, hence triggering the convergence condition in both cases. The number of active vertices over iterations of various relevant algorithm runs is depicted in Fig. 3.22. It is visible that the number of active vertices follows a decreasing exponential, impacted by the error-rate.

Notably, it was observed that higher error rates require fewer iterations to converge in comparable settings. This can be explained by the  $k$ -mer distribution: with increasing error-rates, the average operating coverage decreases, but the associated edge count rises (see Fig. 3.14), implying a rise in the number of initialized seeds. Indeed, with a greater number of edges initialized at the beginning, more edges can be determined in an iteration hence accelerating convergence.

Finally, a GraphLab implementation is evaluated in terms of execution performances and scalability. The scalability of the method is of particular importance as real-world datasets can be relatively large. As an example, though the genome of the *Escherichia coli* bacteria is a few million nucleotides long, the human genome is more than 3G-base long, making scalability key for *e.g.* personalized healthcare applications.

To assess the scalability properties of the implementation, the algorithm was benchmarked with constant parameters over error-free datasets of increased lengths with in-

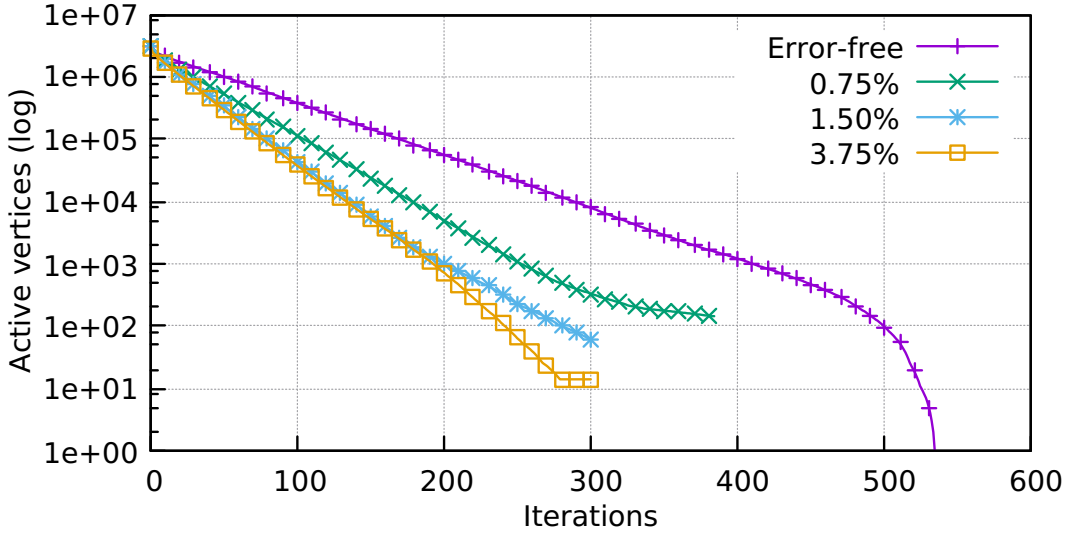


Figure 3.22: The figure shows the number of active vertices in the graph during the execution, for four different runs of varying of error-rates of the 3M-base dataset. Active vertex count axis in log scale.

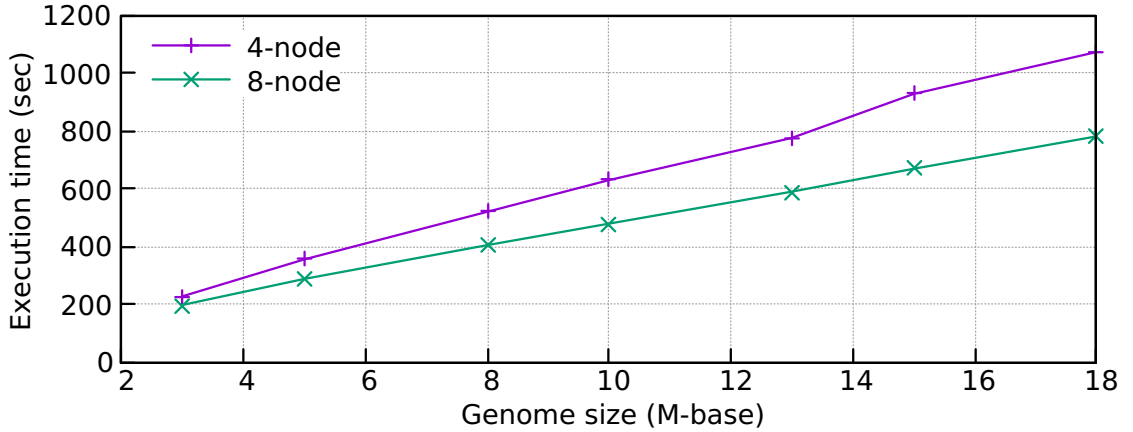


Figure 3.23: Evolution of wall execution time with increasing input genome sizes for two cluster configurations.

creasing cluster sizes. At first we compared the evolution of the wall-clock execution time with increasing genome sizes, depicted in Fig. 3.23, for two cluster configurations. The wall execution time scales linearly with respect to the input genome sizes, with a slope of 0.9 and 0.65 for respectively the 4-node and 8-node cluster configurations.

When analyzing in details the behavior in Fig. 3.24, it can be observed that, with respect to fixed size datasets, parsing and graph commit show a nearly linear speedup on even the 18M-base genome. However, the execution engine speedup is linear only up to the 7-node configuration as the dataset becomes too small to benefit from such architectures.

Figure 3.25 shows that increasing the amount of compute resources improves the total execution time. When distributing smaller datasets on increasing scale clusters, the compute part becomes less efficient. However, the speedup provided to parsing and other

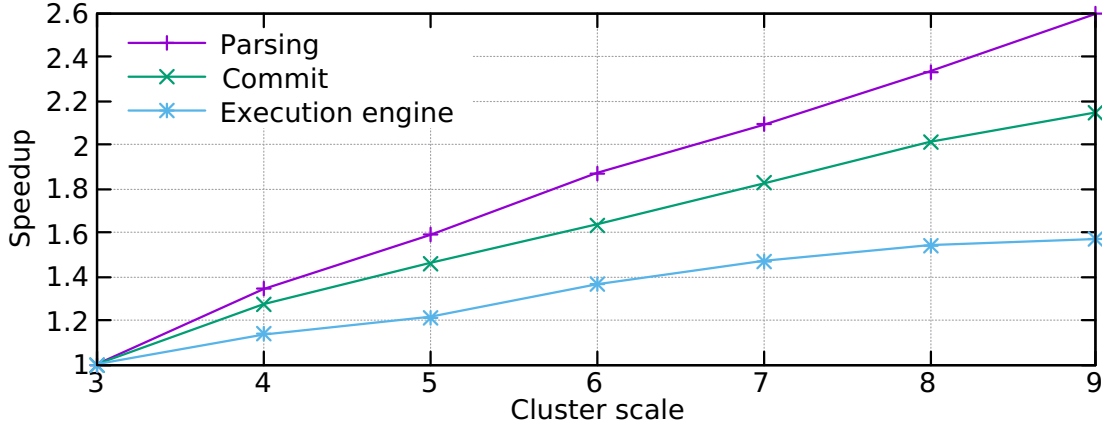


Figure 3.24: Speedups observed on parsing, graph commit and execution, for a 18M-base genome on varying cluster sizes. Though parsing and graph commit speedups are linear, execution speedup is linear up to the 7-node setting only, indicating an inefficient execution for larger scale settings due to underloading.

parts is able to compensate for this inefficiency, hence resulting in a globally accelerated execution.

The performances of the implementation in terms of *update rate* are presented in Fig. 3.26. In particular, higher execution performances are achieved when nodes are loaded with large enough graphs. This effect is particularly visible for larger clusters and relatively smaller instances (*e.g.* 9-node configuration and 3M-base datasets). In such disadvantageous case, the cost of managing the distributed architecture is high with respect to the problem being solved and becomes only bearable with growing instances. In other words, the performance benefits of adding machines are more significant for larger datasets (*i.e.* 13-15M-base), hence, scalability in performance is achieved with increasing datasets and cluster sizes.

### 3.2.7 Synthesis

*De novo* assembly of short reads shows promising perspectives for personalized healthcare applications, even though it faces great challenges in terms of data volume and complexity. Read error mitigation is still an important challenge and an active field of research and emerging parallel and distributed software libraries can help in tackling this challenge.

We presented in this section an exploratory study of a novel preprocessing algorithm for de Bruijn graph filtering in the context of *de novo* assembly of short reads. The experiments on synthetic datasets showed good determination rates even for high read error rate compared to the precision standards of real-world next generation sequencers. This unveils encouraging opportunities for this method as we plan to integrate this preprocessing module into a greater whole-genome assembly framework. With this complete assembly toolchain, a more thorough comparison with state-of-art solutions, in terms of both reconstruction quality and speed, will be conducted.



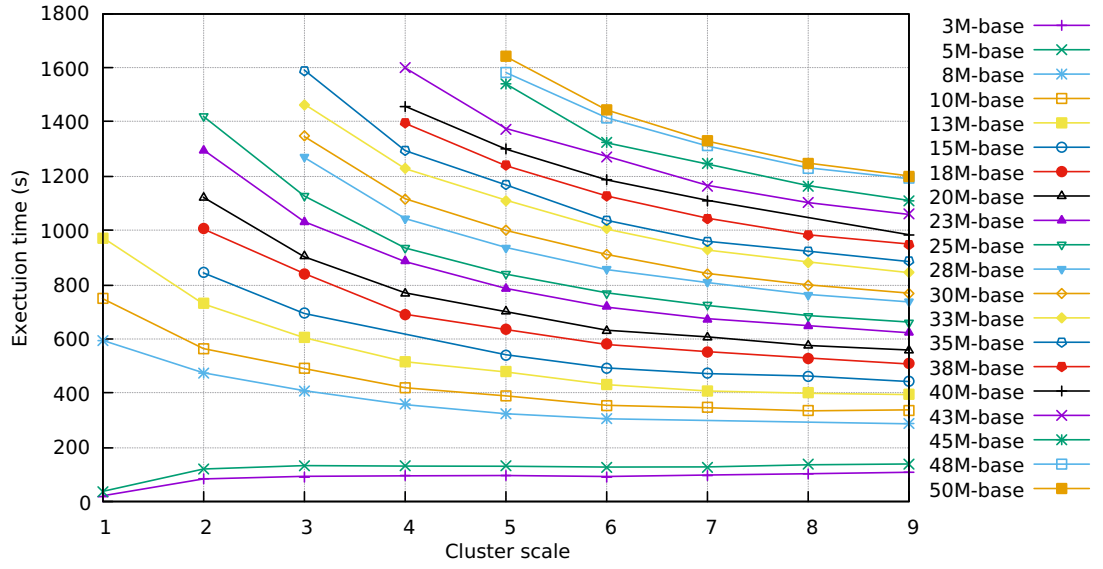


Figure 3.25: Execution time with varying cluster configurations, for different datasets. Larger datasets notably benefits from higher speed-up when scaling out on larger cluster configurations, indicating a possible under-load on smaller datasets.

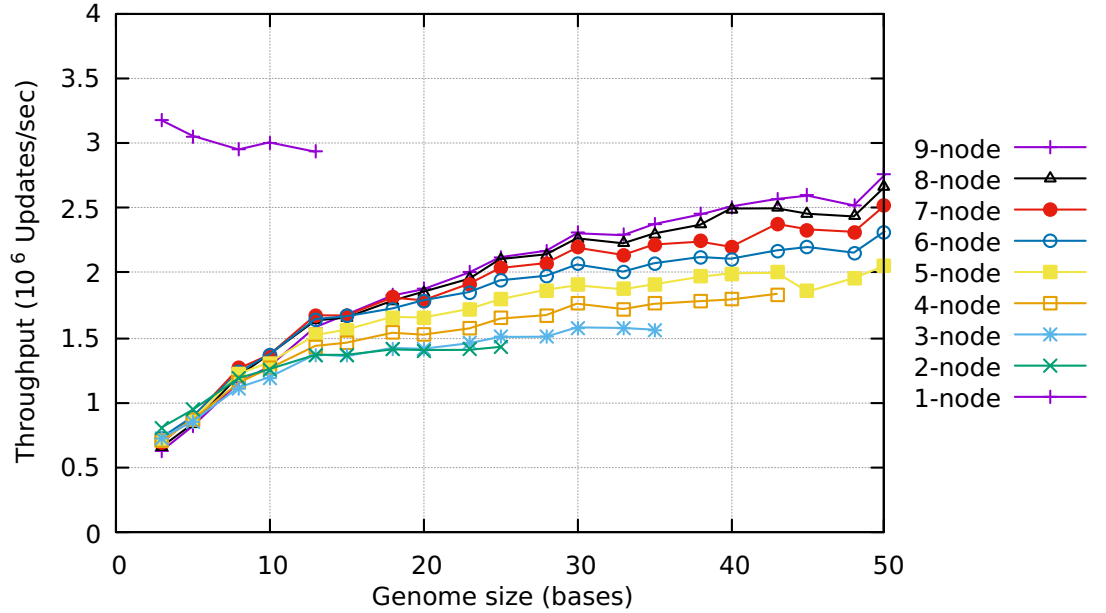


Figure 3.26: Performances of various distributed cluster configurations. The update rate — or the number of updates executed during the pure processing time — is plotted with respect to genome sizes, for error-free datasets ( $\text{tol} = 2$ ). Larger cluster configuration exhibits poor operating performances on small datasets due to a high parallelization cost, mitigated with larger instances.

Further validations are expected using more elaborated dataset generators [59] and optimizations such as the use of read *Quality Value* to filter low accuracy reads could be implemented to further improve results. Future work will also target the automated identification of the most adapted filtering parameters values (coverage, threshold and tolerance) for a given sequencing technology. We also plan to identify, for any given NGS technology, the optimal coverage required to produce high quality sequences with our filtering algorithm. In this work, we demonstrated the relevance of GraphLab as a parallel middleware for graph-processing in the context of DBG-based computations. The presented algorithm showed interesting operational performance properties. In particular, the method exhibits performance scalability with increasing dataset and cluster sizes. However, a number of tunable GraphLab parameters could be explored to further understand execution behaviors and achieve the best performances.

In a future work, we plan to measure the accuracy of our method on real datasets. In particular, it would be of interest to validate the algorithm accuracy when dealing with genome regions known to exhibit many repetitions such as the *Human Leukocyte Antigen* (HLA) region. Indeed, such repeats will result in DBG edges with capacities exceeding 1. We expect our capacity propagation method to help attribute correct capacities to highly repeated edges, where ambiguity is the highest.

### 3.3 Concluding remarks

In this chapter, we introduced two real-life use-cases from different applicative domains, relatively distant from the social mining area from which GraphLab originates. In both cases, the use of GraphLab enabled the seamless deployment of both algorithms on various machine scales and for different problem sizes. During the implementation process, we were able to appreciate the appropriateness of the vertex-centric programming model for such graph analytic tasks. This expressive model enabled a relatively fast development time and concise source code base.

Even though GraphLab greatly facilitates the development of graph applications, many tuning knobs are at hands. In particular, we notice that the choice of the ingress method, responsible for the partitioning of the graph, is of particular importance with respect to operating performances. Indeed, while GraphLab features many different methods for partitioning graphs, we found that the *oblivious* ingress method is superior to the other in every case, for the graphs we processed. However, as GraphLab displays replication factor after having committed the graph, quickly evaluating the best ingress method is possible.

The scalability of a GraphLab program must be seen along mainly three different axis, namely input parsing, graph commit and execution. Parsing and graph commit seem fairly linear for each application, with respect to both problem size and machine scale. About scalability of the execution engine, we observed that for insufficiently loaded

configurations, poor operating performances are extracted as the parallelism management cost is rising with respect to the problem size. Notably, we observed that the use of throughput metrics, such as the update rate and the whole process update rate, is of particular relevance to assess the execution behavior with respect to scalability. In particular, we observed operating points of interests, gathered in three operating ranges, for both use-cases in which we would like to set the system in.

Finally, even though the de Bruijn graph filtering algorithm is not suitable for an asynchronous execution as is, we noticed that this execution mode exhibits lower performances when compared to synchronous execution on the program trace analysis use-case. This observed decreased performances seemed to be due to a steep increase in network traffic, however many tuning knobs specific to the asynchronous execution remain to be explored through further research.

Having reviewed the software-related aspects of vertex-centric programming, we focus, in Chapter 4, on understanding how the hardware architecture impacts performances. We will aim at generalizing the observations made in the course of this chapter for both use-cases, using the variety of hardware systems at hand, described in Chapter 2. The proposed performance comparison highlights advantages and drawbacks of current distributed architectures for graph-processing. Eventually, we present in this chapter a method based on throughput analysis can for the adequate sizing of compute clusters.

Finally, in Chapter 5, we formulate hardware propositions for the design of cluster in the context of graph analytics algorithms. In particular, we evaluate the relevance of an emerging HPC platform based on embedded computer technology using the two aforementioned use-cases.



# Distributed architecture exploration for efficient graph-processing

## Contents

---

<b>4.1</b>	<b>Assessing performance and operating ranges . . . . .</b>	<b>82</b>
4.1.1	Operating ranges . . . . .	82
4.1.2	Targeting an efficient operating point . . . . .	83
4.1.3	Experimental parameters . . . . .	84
<b>4.2</b>	<b>Comparison of distributed architectures for graph-processing</b>	<b>84</b>
4.2.1	Analysis of GraphLab program profiles . . . . .	84
4.2.2	Throughput-based analysis . . . . .	87
4.2.3	Whole-process update rates . . . . .	90
4.2.4	Asynchronous execution performances . . . . .	92
4.2.5	Performance scalability . . . . .	93
4.2.6	Synthesis . . . . .	95
<b>4.3</b>	<b>Throughput-oriented dimensioning of a cluster . . . . .</b>	<b>96</b>
4.3.1	From operating ranges to machine capacity . . . . .	96
4.3.2	Replication factor . . . . .	99
4.3.3	Throughput-based methods for cluster dimensioning . . . . .	99
<b>4.4</b>	<b>Conclusion and perspectives . . . . .</b>	<b>101</b>

---

As highlighted in Chapter 1, graph-processing libraries based on the vertex-centric programming model have raised interests in the context of High-Performance Data Analytics (HPDA). In particular, such libraries claim to provide a fairly good compromise between programming productivity (*i.e.* shortened development time) and operating performances (*i.e.* efficient and scalable execution behavior) by leveraging large-scale distributed architecture while reducing the burden of handling parallel aspects of the programming.

Amongst such tools, we have chosen to focus on GraphLab as it exhibits a great level of maturity and compares favourably to other related frameworks for vertex-centric graph-processing, as discussed in Chapter 2. However, going more into the details of the implementation of GraphLab programs shows that they are composed of different parts

whose parallelism degree and execution time vary. In fact, the algorithm itself may exhibit a varying degree of parallelism during the course of its execution — not mentioning the impact of the graph data, as seen on the trace analysis use-case in Chapter 3. This makes more complex the assessment of the performances of such workloads solely relying on traditional benchmarking metrics such as floating point operations per second (FLOPS) or wall-clock execution time.

In Chapter 3, we showed that GraphLab is a tool of much relevance in order to solve realistic graph-related problems on two real-life use-cases. These experiments enabled to gain practical knowhow about operating and evaluating GraphLab on distributed architectures. In particular, using the dedicated metrics introduced in Chapter 2 and the throughput charts we proposed in Chapter 3, we observed that while executed and deployed on different hardware architectures, both use-cases exhibited a similar threefold performance behavior.

However, though we identified some software-related insights about performances (*e.g.* asynchronous execution or dataset properties), the impact of the underlying hardware architecture on performances was not addressed in the previous chapters. Hence, this chapter aims at investigating this issue by comparing performances of both algorithms on the architectures available in this work. In particular, we investigate in details the question of whether high-end architectures, such as the high-performance cluster (HC), or desktop workstation installations, such as the low-end commodity-cluster (LECC) and the larger-scale commodity cluster (LSCC), are the most adequate choices for graph analytics using vertex-centric libraries. Finally, having identified adapted hardware for HPDA applications, the course of this chapter leads towards an approach for appropriate cluster dimensioning with respect to a given instance size of a problem.

## 4.1 Assessing performance and operating ranges

In the following section, we gather previously observed operating ranges. Then, we details relevant cluster configurations and operating points. Finally, we introduces the experimental parameters of the performance study with respect to each of the two use-cases and general GraphLab parameters.

### 4.1.1 Operating ranges

#### Underloaded range

In this operating range, adding compute nodes to the cluster results in decreased throughput performances. In particular, although it may slightly improves global performance by accelerating *e.g.* the parsing step outstandingly, it does not improve the update rate at all. Indeed, when the cluster size increases, the per-node workload decreases, implying a rise in the relative cost of parallelism management (communication and scheduling).

### Nominal range

In the nominal range, throughput is increased significantly when compute nodes are added, hence a certain form of scalability is observed. In such operating regime, compute nodes are loaded enough to benefit from the added compute nodes. To stay in this operating range, the input dataset size must increase along with cluster scale.

### Overloaded range

When the size of the graph being processed becomes larger than the available memory, the execution begins to require the use of swap space — that is, virtually extending memory using permanent storage, *e.g.* a hard-disk drive. In this operating area, performances are significantly decreased due to the costly swap-in/swap-out operations.

Due to the configuration of the HC system, it is hardly possible to observe this range of operation, hence the following HC figures are plotted until the last operating point observed. Indeed, each HC node is configured so that it provides 128GB of RAM and only a moderate 4GB of swap partition — a memory space instantly consumed once the vast available memory is already filled.

#### 4.1.2 Targeting an efficient operating point

We argue that using operating performance charts (update rate and whole-process throughput) with respect to cluster scale and problem size, it is possible to select appropriate resource configurations for efficient processing of a dataset. In particular, we define the following two cluster configurations of interest with respect to the aforementioned operating ranges, for a given problem size:

- The *minimum cluster scale* is the minimum number of required machines to process a graph in the nominal operating range. In this operating point, the system is closer to the limit with the overloading range as the per-node workload is the highest.
- The *maximum throughput cluster scale* is the cluster configuration at which the highest throughput is measured in the nominal operating range for a given problem size. Operating under such conditions implies that the system may reach the interface between the nominal and underloaded ranges, as the per-node workload decreases with increasing cluster scale.

The throughput performance chart is essential to identify these operating points and allocate resources accordingly. Furthermore, we define also:

- The *peak performance point*, which is the operating point where the cluster configuration gives its highest throughput for a given cluster scale.
- The *system peak performance point*, which is the *peak performance point* of the largest available cluster configuration.

### 4.1.3 Experimental parameters

Unless stated otherwise, all the following performance figures were obtained using the parameters described hereafter. The `oblivious` ingress method was used for each experiment as it was previously proven to be the best partitioning policy in terms of usage flexibility and graph replication factor<sup>1</sup>. The synchronous engine was used consistently throughout these experiments and GraphLab was configured so that one thread per available core and one process per machine were used. The results were written to four output files per nodes, a default value provided by the framework.

The study was conducted on the two algorithms presented in Chapter 3. For the trace analysis use-case, the matrix-multiply kernel was scaled up to graphs larger than  $1.4 \cdot 10^9$  vertices. Input files were split in 32 parts located on a distributed filesystem for the LSCC and HC platforms.

Concerning the De Bruijn graph (DBG) filtering algorithm, synthetic datasets were processed. The synthetic datasets all exhibited the same sequencer properties (read length: 100, sequencing depth: 40, error-rate: 0.75%) with increasing genome scales (from a 30k-base to 40M-base genome). Input files were split in 144 parts<sup>2</sup> and issued using the synthetic dataset generator previously introduced in Chapter 3. Unless stated otherwise, a  $k$ -value of 31, a tolerance interval of  $\pm 2$  and a threshold set to 0 were used.

## 4.2 Comparison of distributed architectures for graph-processing

In this section, we compare the operating performances of two competing systems, the LSCC and the HC platforms. However, the LECC performances are shown as a reference baseline in some cases. First, a profiling is performed with each system in order to gain insights on how the wall-clock time is spread among parsing, processing, graph commit and save. Then, throughput-based figures are compared, and aspects of the execution related to scalability and asynchronous performances are investigated. Finally, a synthesis is presented in order to eventually conclude on the relevance of these platforms for vertex-centric graph-processing.

### 4.2.1 Analysis of GraphLab program profiles

In order to gain a first understanding of how each architecture behaves with the provided implementations, we performed a coarse-grain profiling of the use-cases. In particular, we investigated the share of wall-clock time spent respectively in parsing, graph commit, processing and saving, as shown in Fig 4.1 for a 6-node configuration of each system.

---

<sup>1</sup>See Sec. 3.1.4

<sup>2</sup>Indeed, the parallel machine used to generate datasets featured 144 cores, fully exploited by the parallel dataset generator to reduce the generation time.



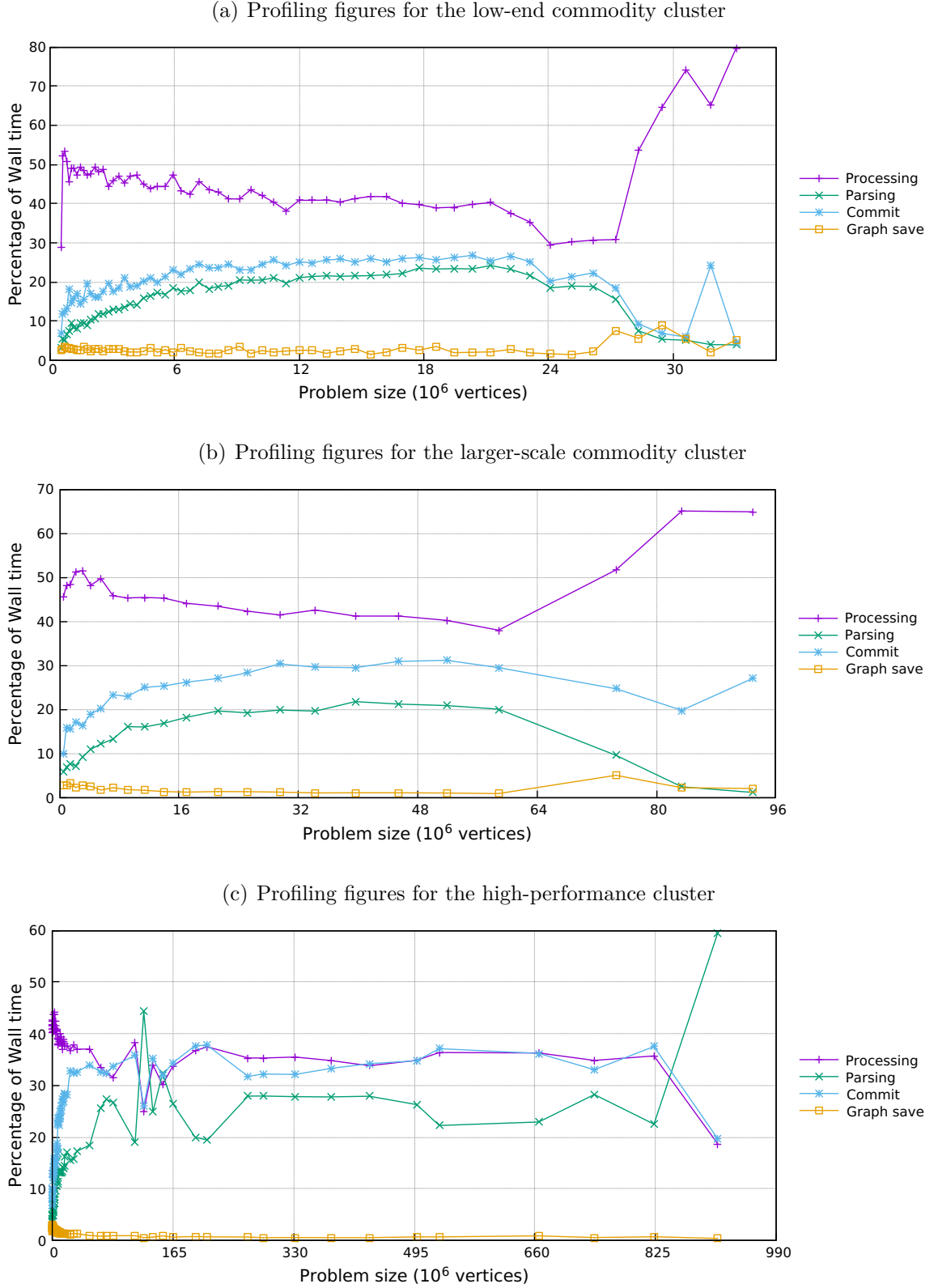


Figure 4.1: Comparison of profiling figures of the program trace analysis use-case on various architectures, expressed as share of time spent in different part of the program. Each system is profiled in a 6-node configuration and timings are obtained using embedded timers in the code.

Firstly, the two commodity clusters exhibit a similar behavior where the processing time dominates fairly (Fig. 4.1(a)-4.1(b)). Both architectures show a low graph saving time share, while parsing and commit represent about a half of the total execution time. In comparison, the HC exhibits a wall-clock time almost equally shared between processing, parsing and commit (Fig. 4.1(c)).

Since most of the commit step is spent in memory allocation and graph partitioning, no particular optimization can be leveraged from an end-user perspective. On the contrary, attention must be paid to parsing (at least 20% of the wall-clock time), as it is an embarrassingly parallel step. In such case, adequately splitting files — on a per-machine basis — to match the number of cores available in a cluster node can be advantageous.

It is to be noted that the aforementioned operating ranges can be glimpsed on the profiling figures. By way of example, the overloaded range can be seen clearly on the rightmost part of the two commodity cluster plots. In this area, the execution engine share of time is greatly increasing as the swap operations slow down the computations. By going further into the overloaded range, the commit share of wall clock time is as well increasing as the operating system begins to swap as early as the commit step, as visible in Fig. 4.1(b). Similarly, the underloaded range can be guessed as the wall-clock time share of the execution engine starts at a higher value, before lowering to a more moderate value in the nominal range.

The profiling figures (Fig. 4.2) differ slightly when considering the DBG algorithm. First, the processing dominates more clearly the wall-clock time in both cases. Moreover, the LSCC profile exhibits an interesting behavior: past a certain graph size, the processing time of the DBG filtering algorithm is decreasing in time-share with increasing graph sizes. This results from a two-stage commit and a folding step which are performed on the graph requiring more memory than the following processing step. Hence, the system actually swaps during the commit step (resulting in a way longer commit time). Then, as the processed graph is smaller, the execution can be performed with no swap-induced slowdown. However, when clearly entering the overloaded range 4.2(a) (around 22M vertices), the commit time share starts to abruptly increase, similarly as in Fig. 4.1(b).

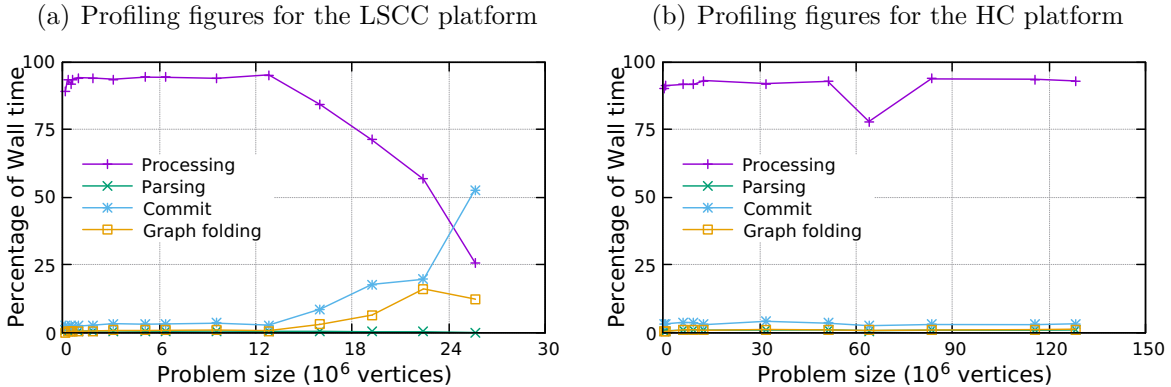


Figure 4.2: Comparison of profiling figures of the DBG filtering use-case on the (a) LSCC and (b) HC architectures (6-node configuration), expressed as share of time spent in different part of the program.

## 4.2.2 Throughput-based analysis

The *update rate* — that is, the total number of completed *update functions* divided by the time spent in the execution engine — is a throughput performance metric defined in Chapter 2. In the following, we investigate and compare the raw graph-processing performances of the surveyed distributed architectures.

Figure 4.3 shows the performance charts of the platforms for the program trace analysis use-case. Though they greatly vary in span due to the different amount of available memory, the same operating behaviors can be observed, regardless of the target machine. A common observation is that the 2-node configurations are consistently unable to provide a significant performance gain when compared to single-node baselines. Yet, they enable the processing of larger graphs. However, this can be seen as a somehow unfair comparison, as the performance penalty between single- and multiple-node executions is the highest for the 2-node setup — Indeed, it is then compensated with larger cluster configurations.

The largest processed graphs by each of the LSCC configurations in the nominal operating range is, on average, 2.27 times larger than those of the LECC platform, despite having only twice as much available memory. In contrast, the HC platform can sustain much larger graphs, as it provides 16 times more per-node memory than the LSCC.

We then compare the commodity clusters at the same operating points, *i.e.* the *peak performance points*. The performance improvement shown by the LSCC over the LECC is, in average, of 1.68, ranging from a  $2\times$  factor for a 1-node setup, to a  $1.63\times$  factor for a 7-node configuration. This throughput increase must be put in perspective with the greater processor performances and core number provided by the LSCC system compared to the LECC platform. However, when comparing how the LSCC is performing at the peak performance points of the LECC system<sup>3</sup> using comparable cluster configurations, the average performance improvement is slightly lower, yielding a gain of only 1.48. In similar conditions, the HC system exhibits an average throughput improvement of 2.08 over the LECC platform, ranging from a  $2.5\times$  factor for a single-node setup, to a  $1.76\times$  factor for a 7-node configuration.

For a given cluster configuration and dataset kernel, we observe that performances seem bounded with respect to throughput. That is, performances increase with problem instance (for a given cluster scale), then reach a plateau of peak configuration throughput, and decrease in the overloaded operating range. This performance plateau is particularly visible for the HC (Fig. 4.3(c)) system. In comparison, the performances of the LSCC system (Fig. 4.3(b)) — and of the LECC system (Fig. 4.3(a)), to a lesser extent — seem to never durably reach such a plateau. This is due to the fact that the memory amount per compute node, hence the amount of data to process, is too limited to achieve the peak performance of the cluster configuration. Thus, it could be interesting to increase

<sup>3</sup>*i.e.* problem sizes of 9, 14, 20, 24, 28, 32 and 38 million vertices, respectively for configurations ranging from 1 to 7 nodes.

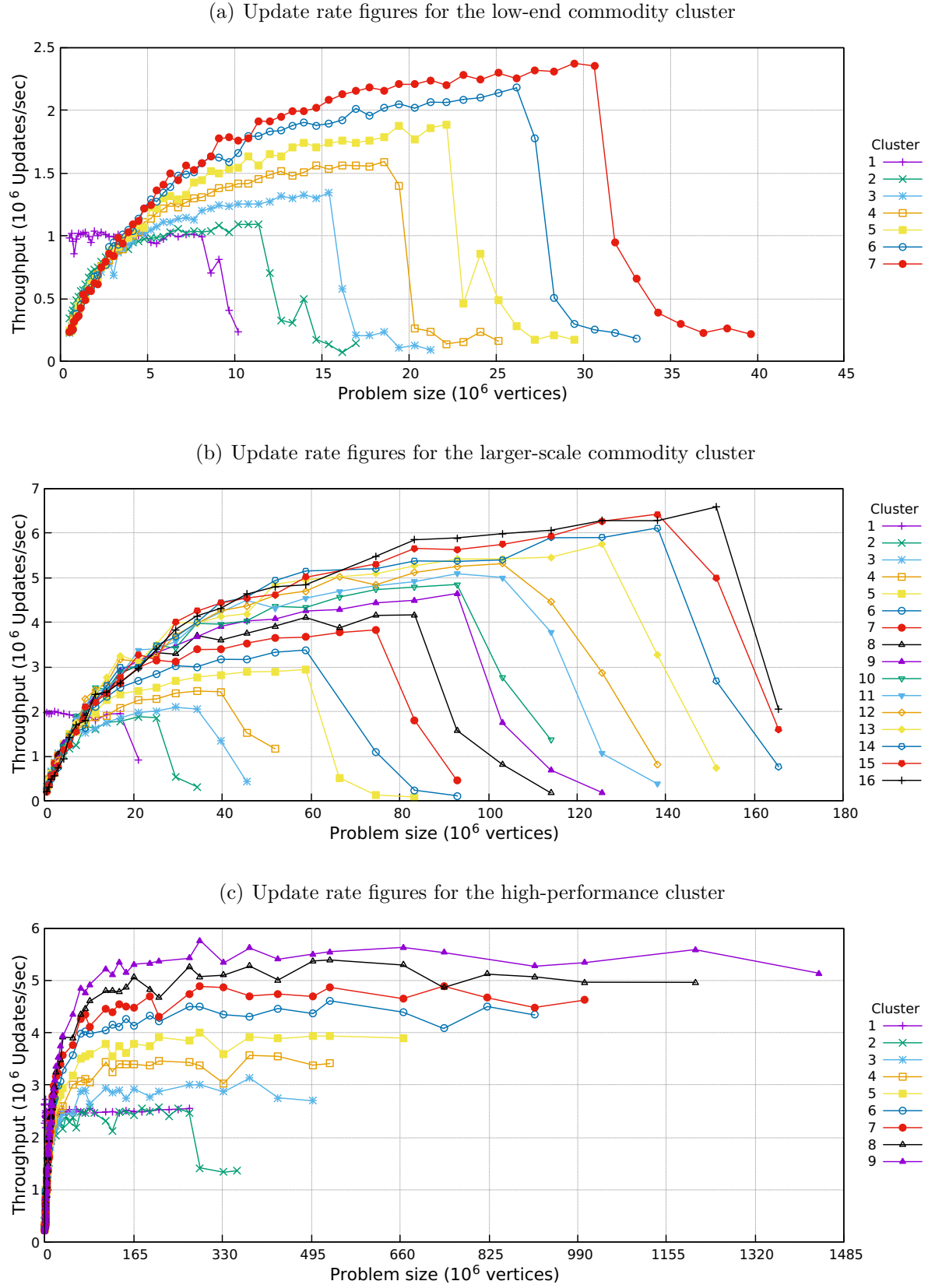


Figure 4.3: Performance figures in terms of throughput of the available distributed memory systems for the program trace analysis use-case. The clusters exhibit a similar threefold operating behaviors comprising the underloaded, nominal and overloaded ranges.

the memory amount on LSCC nodes to increase the maximum manageable problem sizes and also to reach higher performances at a moderate per-node cost. In contrast, the HC performances converge long before reaching memory saturation, hence indicating that the per-node memory amount could have been more efficiently used if shared with an additional node. This would have implied reaching higher system performances, although at a non-negligible cost.

To investigate further the performance of the HC system, we studied the throughput obtained on the trace analysis use-case using only 8 instead of the 32 threads recommended by GraphLab for the featured 32-core bi-processor. Interestingly enough, the throughput performances do not decrease significantly on such a configuration, reinforcing the idea that processing such datasets prevents the fruitful leveraging of the massively parallel architecture of the HC nodes.

When considering the sole update rate metrics for both commodity clusters, it appears that an interesting point to operate the system in is when the memory is close to saturation. In such an operating point, the cluster configuration delivers its peak performance. Interestingly enough, when comparing the LSCC and the HC systems, we observed that, though the HC can address a considerably broader range of problem sizes due to its available memory, it is outperformed by the LSCC installation in terms of raw performances. Indeed, the LSCC exhibits a peak performance of 6.61 MUPS (million updates per second) whereas the HC only reaches 5.57 MUPS on the trace analysis use-case. This is even more interesting when considering these system-wide peak performance results with respect to additional metrics such as the price (UPS/€, or update per second per euro) or the processor power (UPS/W, or update per second per Watt). Using such metrics, the LSCC would even more favourably compare, with a 550.8 UPS/€ while the HC only exhibits a moderate 103.2 UPS/€ at peak performance. The power-related metric confirms this statement, with a 4.92 kUPS/W for the LSCC, compared to the 3.44 kUPS/W of the HC.

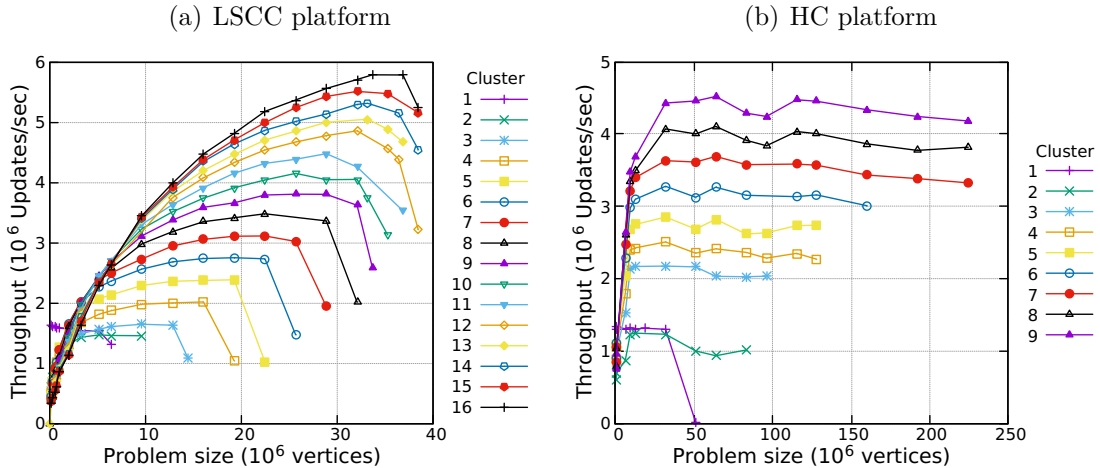


Figure 4.4: Throughput performance figures of the De Bruijn graphs filtering algorithm on the (a) LSCC and (b) HC platforms.

However, these derived metrics provide only a coarse-grain yet relevant appreciation of the performance of these systems. Indeed, the prices used to compute the throughput-to-cost ratio is only accounting for the cost of acquisition, hence excluding operating expenditures. Similarly, the throughput-to-power metric is computed solely relies on the peak processor power, which only partially reflects the power requirements of the system.

Similar observations can be made with respect to the De Bruijn graph filtering algorithm, despite having less plotted points on the throughput figures shown in Fig. 4.4. In particular, the HC platform shows a quickly reached performance plateau that the LSCC system never achieves durably, hence advocating again for an increase in per-node memory. The HC platform is able to process a much wider range of problem sizes in comparison to the commodity cluster, as previously observed. In terms of pure performances, the LSCC system exhibits a peak system throughput of 5.74 MUPS, outperforming the peak throughput of 4.49 MUPS reached by the HC system. Hence, with respect to price or energy aspects, the LSCC system yet again favourably compares to the high-end compute cluster. Finally, it is to be noted that the performances exhibited on the DBG use-case are subject to change with respect to the algorithm parameters used (*e.g.* threshold or tolerance), which is in contrast with the trace analysis use-case; but also with dataset properties such as sequencing error-rates.

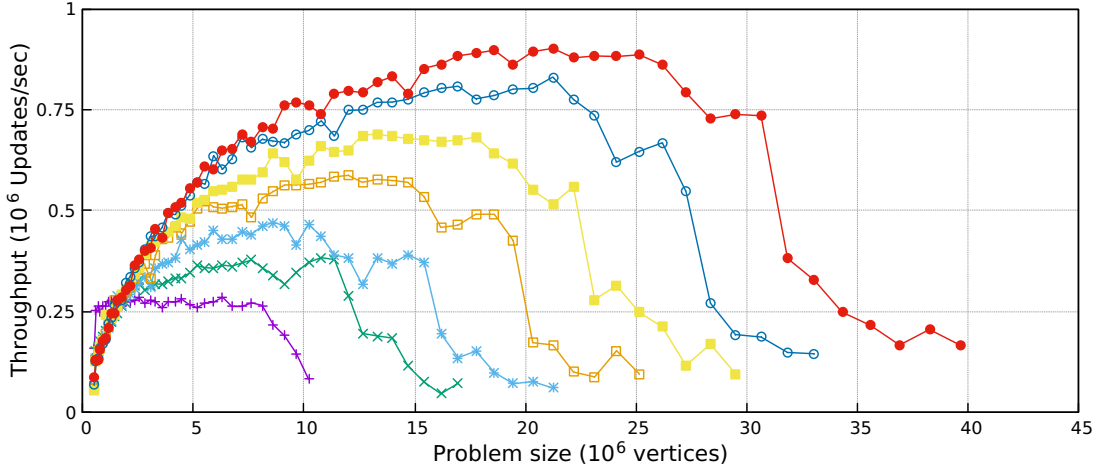
### 4.2.3 Whole-process update rates

In addition to the update rate, the *whole-process update rate* provides global insights on performance behaviors of our applications on the available clusters. As a larger execution time is considered, global throughput figures are expected to be lower than the previous raw performance figures. Figure 4.5 shows the whole-process update rates of the three architectures on the trace analysis use-case.

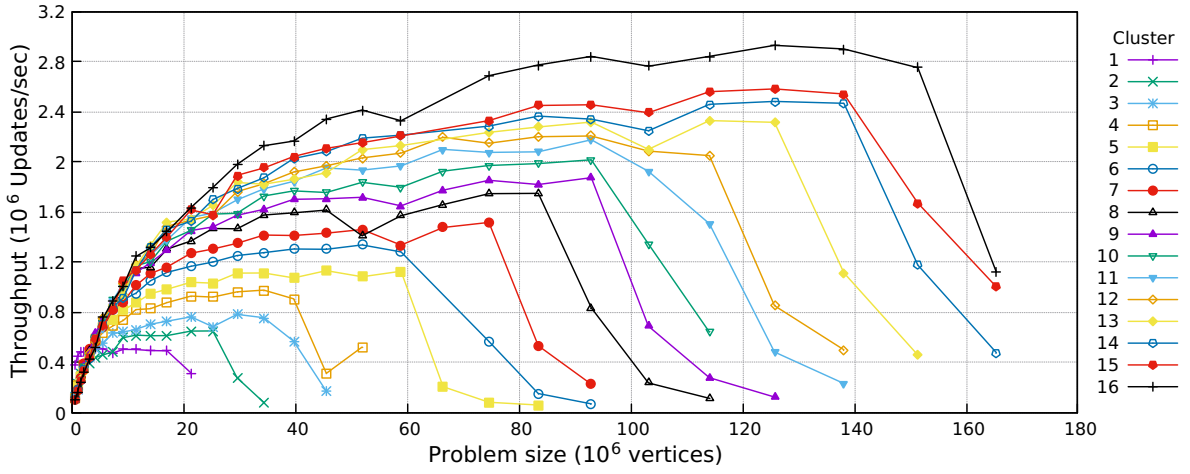
In details, the peak throughput of the LSCC system is divided by a factor of  $2.28\times$  when computed globally. This decrease is however of  $2.57\times$  and  $2.7\times$  for the HC and LECC respectively. This indicates that these platforms are globally behaving similarly with respect to non-processing tasks — *i.e.* the impact of such tasks is similar on both systems. When the share of time spent in the execution engine of the GraphLab program is decreasing, the global throughput is decreased similarly. This is particularly visible for the HC and LECC systems (Fig. 4.5(c) and 4.5(a)) which spend less time in pure processing, at peak throughput points, compared to the LSCC system. Indeed, when observing the profile shown in Fig. 4.1(c) at peak configuration points, the shares of processing time are 30%, 39% and 35%, for respectively the LECC, LSCC and HC systems.

Previously, we highlighted the fact that usually, the 2-node cluster configurations hardly outperform single-node baselines in terms of raw throughput performances. However, when considering things from a more global point of view, *i.e.* using the whole-process update rate, we observe that performances are actually improved, despite not being visible on the throughput-only figures. This is explained by the fact that the parsing and

(a) Whole-process update rate figures for the low-end commodity cluster



(b) Whole-process update rate figures for the larger-scale commodity cluster



(c) Whole-process update rate figures for the high-performance cluster

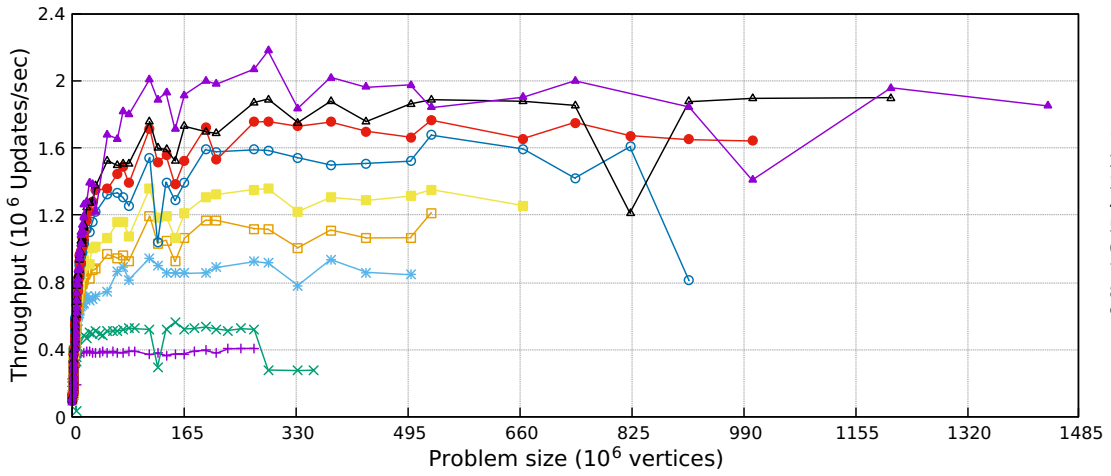


Figure 4.5: Global performance figures of available distributed memory systems for the program trace analysis use-case. The whole-process update rate is plotted for various cluster configurations with growing graph instances.



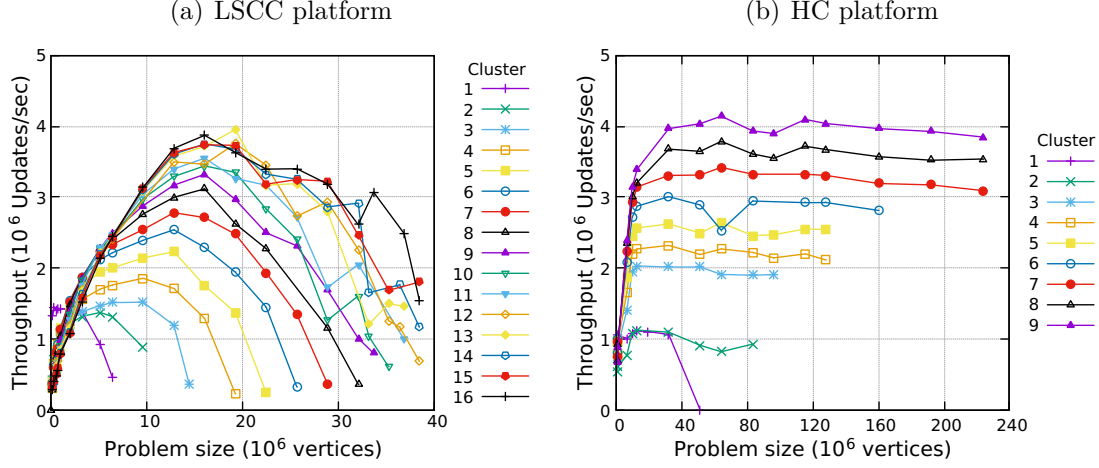


Figure 4.6: Global performance figures of the (a) LSCC and (b) HC systems for the De Bruijn graphs filtering algorithm.

graph commit steps are able to further benefit from the additional compute resources, hence compensating the distributed penalty observed during the processing.

As seen for the throughput analysis, the global performance charts of the DBG algorithm are similar to those of the trace analysis use-case. The most notable difference with the DBG use-case lies in the reduced performance decrease between throughput and global throughput exhibited by both systems, as shown in Fig. 4.6. This is mostly explained by the profiling figures seen in Fig. 4.2, which shows that the processing accounts for about 90% of the wall-clock time for both systems. Moreover, with this use-case, the peak global performance of both systems are rather close, with the HC slightly outperforming the LSCC platform, although only without accounting for power or cost metrics. However, we foresee that with an increased per-node memory, the LSCC would eventually exceed the HC performances. Further experimentations shall be conducted to validate this point.

#### 4.2.4 Asynchronous execution performances

During the study of the trace analysis algorithm in Chapter 3, the performance analysis of the algorithm on the LECC system shown that the asynchronous execution raised lower performances with respect to the synchronous execution<sup>4</sup>. We performed a similar analysis on the two other clusters in order to understand if larger processors with more cores would be able to benefit from the asynchronous execution. However, as previously explained in Chapter 3, the way the DBG filtering algorithm is implemented prevents the use of the asynchronous execution, hence it is not considered in the following.

Figure 4.7 shows that the asynchronous execution of the trace analysis algorithm still exhibits poorer performances than when using the synchronous engine, even on larger machines and configurations. In details, the LSCC system only reaches 39% of its peak synchronous throughput (comparable to the 38% shown by the LECC system in Chap-

<sup>4</sup>See Sec. 3.1.4



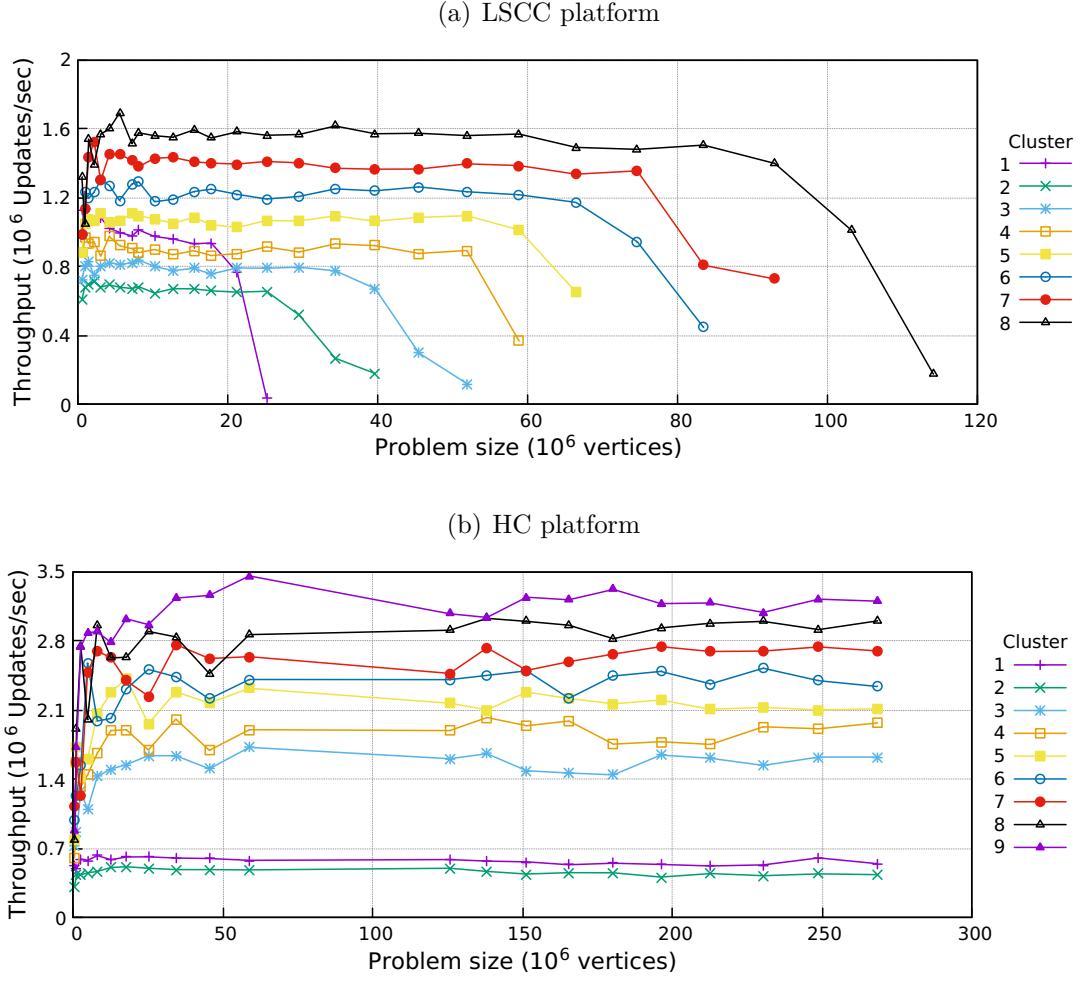


Figure 4.7: Throughput performances of the asynchronous engine on the trace analysis use-case for the (a) LSCC and (b) HC platforms.

ter 3). We observe also that the degradation when transitioning from the nominal to the overloaded operating range is slightly less abrupt with respect to the synchronous results. In comparison, the HC cluster is able to reach 58% of its peak synchronous performance. The fact that the HC system behaves slightly better than the LSCC platforms is explained by its larger number of per-node cores, more beneficially used in this context.

Finally, though observed performances are lower, further work should be conducted in fine tuning the scheduling options of GraphLab execution engines. Indeed, considering this algorithm, datasets exhibiting vertices with updates of variable length should benefit from asynchronous execution as every vertices can further execute iterations without waiting for global synchronization.

#### 4.2.5 Performance scalability

We define *vertical scalability* as the ability for a system to continuously improve its processing performances on a fixed-size graph, when adding compute resources. In contrast, we define *operating scalability* as the ability for a system to continuously improve perfor-

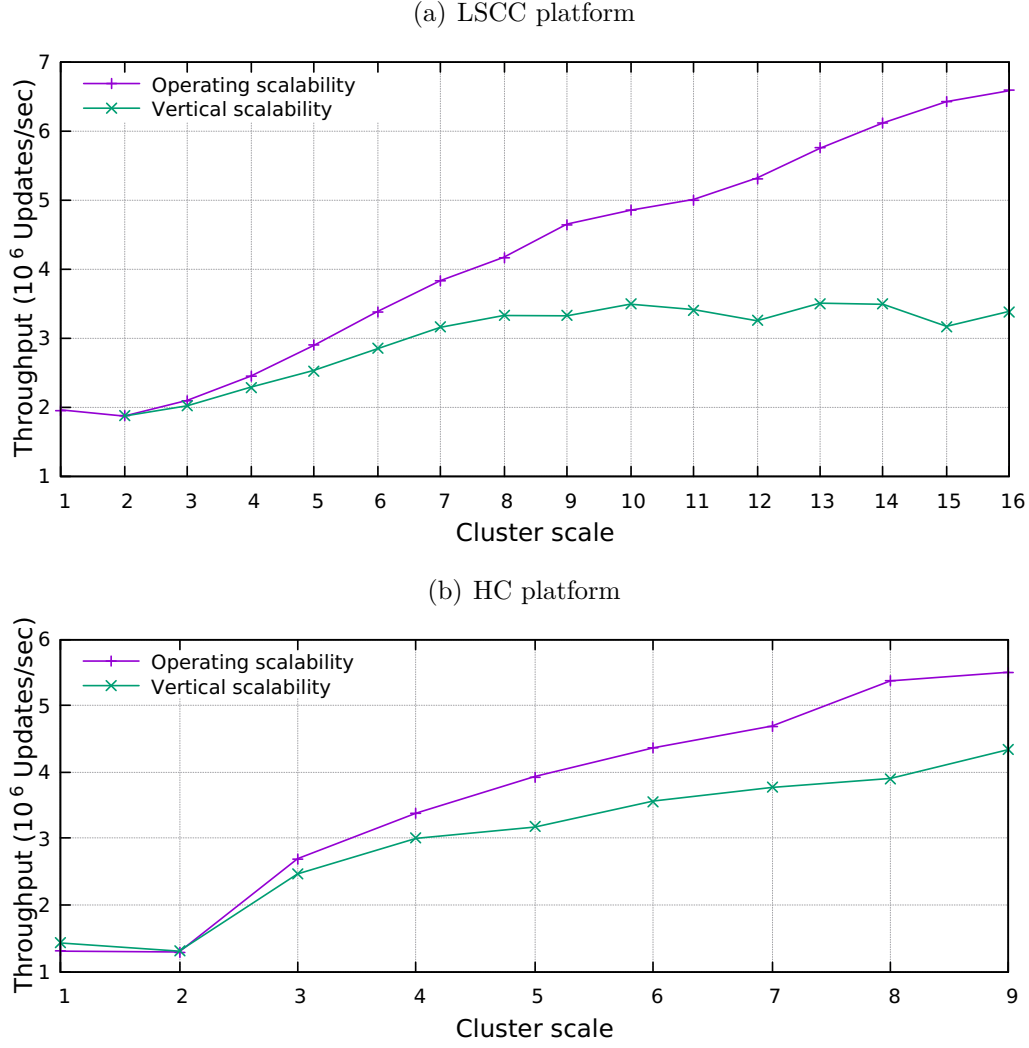


Figure 4.8: *Performance scalability of the trace analysis use-case for the (a) LSCC and (b) HC platforms. In purple are shown performances at a fixed operating point (peak configuration performance) and in green are shown throughput points for a fixed dataset of respectively 25 and 52 million vertices.*

manances with increasing cluster scale and problem size, hence at a fixed operating point (e.g. the previously defined *peak operating point*).

Figure 4.8 compares the more traditionally assessed *vertical scalability* to the *operating scalability* of the HC and LSCC systems, similarly to the study conducted in Chapter 3, for the LECC system<sup>5</sup>. In particular, these figures respectively show performances with respect to cluster scale, at a fixed problem size and at an efficient operating point. As seen with the LECC cluster, both systems exhibit a linear operating scalability compared with the bounded fixed-instance scalability on a fixed-size dataset.

In particular, when considering the comparison between fixed-instance and operating scalability curves in Fig. 4.7(a), it can be observed that, for the 25M-node graph, performances are linearly improved with increasing cluster scale. However, starting with the 8-node configuration, throughput saturates at about 3.3 MUPS, regardless of the cluster

<sup>5</sup>See e.g. Fig. 3.7

scale. In opposition, the throughput at a fixed operating point is improved linearly on the full cluster scale. Though not as clearly, the HC behaves similarly as the LSCC and LECC clusters and exhibits a limited fixed-instance scalability compared to the operating scalability.

Concerning the fixed-instance scalability plots shown, we decided to use problem sizes processable by the largest number of cluster configurations in both cases. However, the above statements remain correct with different fixed-instance scalability curves. Indeed, for a larger graph, the leftmost part of the curve cannot be plotted due to the graph being too large for the smallest cluster scales. Then, there should appear a somehow linear part of the curve, until the graph becomes too small for the cluster scale, resulting in stalling throughput performances.

More broadly speaking, we argue these curves are of particular relevance as they underline how scalability should be considered for HPDA applications. Indeed, in the context of graph-processing — or more generally, HPDA — the problem sizes are continuously scaling out, hence, operating scalability shall gain more interest than the more classic, vertical scalability.

#### 4.2.6 Synthesis

In this section, we compared the performances of two commodity clusters and a high-end compute servers, using two graph-processing algorithms implemented with GraphLab, which leverages the emerging vertex-centric programming model.

We observed that all systems exhibited a similar three-fold performance behavior comprising an underloaded, a nominal and an overloaded area. In terms of largest processed graph by the platforms, the HC platform is leading as expected, due to its larger amount of memory. However, when comparing peak performances (execution and global), the LSCC system outperforms it with respect to both algorithms. Moreover, this performance gap is further increased when weighing performances with other metrics such as cost of ownership or power, as the LSCC is composed of much cheaper/frugal nodes. This confirms yet again that commodity clusters constitute a hardware of choice for such memory-bound computations.

Both clusters have shown that the use of the asynchronous engine — without further tuning — decreased the performances with respect to the synchronous execution. However, in this context the HC has shown reduce decrease in comparison with the LSCC.

With respect to scalability, both platforms have exhibited a bounded scalability. However, when considering increasing problem and cluster scale, they show a linear operating scalability.

More interestingly, the study of the throughput allowed to highlight additional issues. Although convenient, the width of the nominal operating range of the HC system and its performance stall, indicate that the memory per processor ratio is too high. Hence, reducing the amount of per-node memory and adding supplementary nodes should be

beneficial to the system in terms of peak performances. However, it is to be noted that sharing the same total amount of memory on more compute nodes results in a slightly reduced maximum manageable graph size due to the higher vertex replication requirements. In contrast, the LSCC (and to some extent, the LECC) system exhibit a performance behavior where the nominal operating plateau is reduced, *i.e.* there is no clear stabilization of the performances in the nominal operating range, as seen for the HC platform. This indicates that the memory per processor ratio should be increased in order to further extend the nominal range but also to fully exploit processing performance to some extent.

More generally, we have seen that the study of system throughput figures provides guidance for node architecture upgrades. By way of example, a system too frequently underloaded may be in fact limited by the cost of parallelism, or by a not parallel enough problem. In contrast, often operating in the overloaded range may call for additional memory or compute nodes. Finally, the span of the nominal operating range provides useful hints on the memory-to-processor ratio. With this in mind, we investigate in the following section how the use of such performance analysis can further help in designing clusters for graph-processing applications.

## 4.3 Throughput-oriented dimensioning of a cluster

Having compared in details the LSCC and HC systems, we demonstrate in this section how such benchmarking analysis can be used not only to assess individual node efficiency, but also to adequately size a cluster for a given workload. To this extent, we focus on translating operating ranges into machine capacity in terms of graph vertices. We then study the impact of the graph's replication factor in order to evaluate practically the per-node capacity of a platform and its ability to scale. Finally, we investigate methods for the automatic cluster dimensioning before drawing a conclusion and present the perspectives of this work.

### 4.3.1 From operating ranges to machine capacity

Dividing throughput figures by the number of cluster nodes provides performances for a given per-node problem size. More interestingly, it shows the operating ranges observed with respect to the workload processed by each node, for different configurations. Hence, it gives the user an estimate of the maximum per-node graph size a configuration can sustain — or the unit capacity. Indeed, as the vertex type is user-defined and can thus vary widely depending on the application, it can be particularly difficult to assess how many vertices can fit on a cluster node prior such a study. However, knowing the unit capacity of a platform, the estimation of the number of required machines is at hand.

GraphLab defines *vertices* (or *true vertices*) as the amount of vertices in the processed graph, from the point of view of the user. In contrast, it defines *replicas* as the total

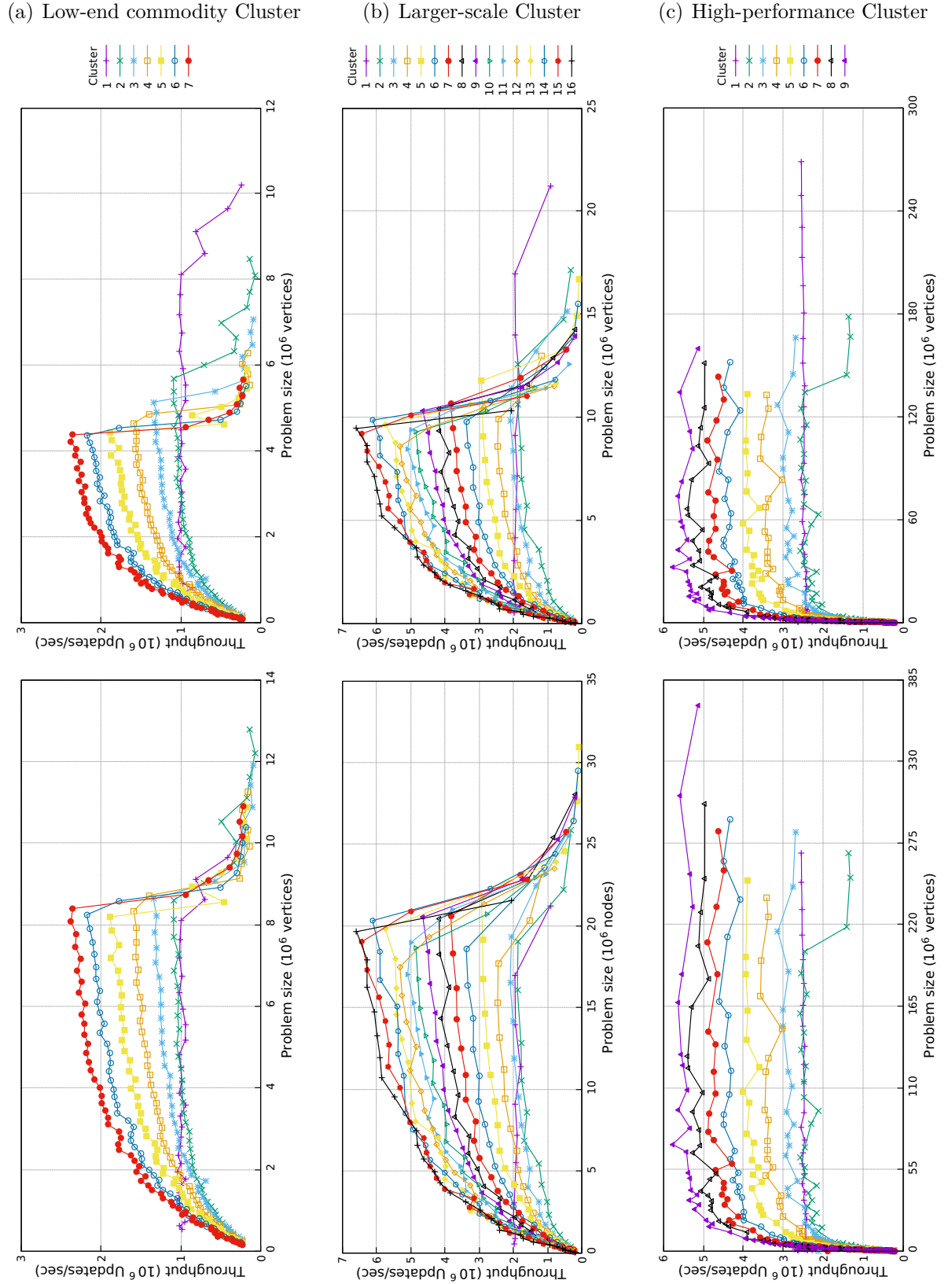


Figure 4.9: Update rate performance characteristics for the (a) LECC, the (b) LSCC and the (c) HC systems, on the program trace analysis use-case, with respect to the per-node number of graph vertices (up) and true vertices (down).

amount of instantiated vertices — in a distributed context, the effective vertex number in the graph structure. Hence, the number of replica is equal to the number of vertices multiplied by the replication factor as GraphLab uses a vertex-split policy. The replication factor is hence defined as:  $\frac{\text{replicas}}{\text{true vertices}}$ , which at best equals true vertices in a single-node configuration and varies with the ingress policy used. Consequently, we define two terms:

- The *true vertex unit capacity* of a system, which is the maximum number of graph vertices a single machine can hold in memory while keeping the system in the nominal operating range.
- The *replica unit capacity* of a system, which is the maximum number of replicas a single machine can hold in memory while keeping the system in the nominal operating range.

In Fig. 4.9 are shown *unit capacities* of the systems in terms of *true vertices* and *replicas*. These figures are obtained by plotting throughput divided by the number of cluster nodes per cluster configuration, with respect to true vertices and replicas. The replica unit capacity is constant regardless of the cluster scale — with scaling throughput. In contrast, the true vertex unit capacity is decreasing with larger cluster configuration as GraphLab needs replica vertices for data consistency requirements, hence artificially increasing the graph size — *i.e.* the amount of per-node true vertices decreases.

In fact, the true vertex unit capacity figure shows that, at a certain limit, the addition of machines may not allow the processing of larger graphs in the nominal operating range. Should this happen, an increase in per-node memory would be required in order to further extend the nominal operating range and improve performances. This decrease in true vertex capacity is directly related to the replication factor given by GraphLab, hence depends on the ingress method used. By way of example, we observed in Chapter 3 that the *oblivious* ingress method seems to converge around a replication factor of 2 for the trace analysis use-case. To investigate this issue, we performed an analysis of the replication factor later in this section.

It is important to note that, though performances may vary with respect to the dataset used, the unit capacity solely relies on the graph modelization we have implemented (*i.e.* the edge and vertex classes used), hence, is independent of the dataset properties. This is notably because no dynamically allocated members are held by the vertex classes. However, performances can vary if the graph properties have a negative impact on computations, such as observed for the **deriche** kernel on the trace analysis algorithm or with certain datasets on the DBG filtering algorithm, in Chapter 3. Indeed, these datasets hampered notably performances by reducing the degree of parallelism.

To summarize, we show in Tab. 4.1 unit replica capacities of our systems. Indeed, the HC provides the largest unit capacity, with an outstanding 340M replicas for the trace analysis use-case. In comparison, the LSCC only tolerate 22M replicas per node — a value about 16 times lower, which is comparable to the difference in available memory of the

<i>Systems</i>	LECC	LSCC	HC
<i>replica unit capacity (trace analysis)</i>	8.7M	22M	340M
<i>replica unit capacity (DBG filtering)</i>	N/A	7M	51M+

Table 4.1: *Replica unit capacities of the three platforms with respect to both algorithms.*

two systems. With respect to DBG use-case, the observed unit capacity of the LSCC is of about 7M vertices. However, it was more difficult to precisely assess the unit capacity of the HC, hence, though we observe a capacity of about 51M vertices, we expect it to be much larger and further experimentations shall be conducted to address this issue.

### 4.3.2 Replication factor

As the replication factor is the link between unit capacity in terms of true vertices and replicas, we focus now on the variations in replication factor with varying cluster scale and graph size. These variations are shown in Fig. 4.10, for the LSCC and HC systems on both the trace analysis and the DBG filtering algorithms.

Taken separately, it can be observed that for both algorithms, the replication factor for a given cluster scale seems to converge asymptotically — a consistent result with respect to the findings of Sec. 3.1.4. The replication factor is mostly influenced by the number of machines in the distributed cluster used. However, even with the LSCC system which shows a larger number of nodes, the convergence of the replication factor is visible. These figures shows as well that, though the DBG use-case has an almost fixed replication factor for a given cluster configuration, this factor evolves for the trace analysis use-case. Yet, even for the largest graphs processed by the 9-node HC configuration, the replication factor converges. This observed convergence is particularly important as it leads towards the possibility of rapidly approximating the replication factor expected for a larger cluster configurations. Hence, one can approximate the size of the processed graph in terms of replicas for a given configuration and compute the required number of compute nodes. Finally, at comparable node configurations (*e.g.* up to 9-node) and graph sizes (*e.g.* the range 0-120M nodes), coinciding replication factors are shown by both systems.

This study of the replication factor evolution shows that the ingress method used when launching a GraphLab instance is crucial. Indeed, all the experiments were performed using the *oblivious* ingress methods, as this method is the most flexible and yields the best partitioning results, as shown in Sec. 3.1.4. Moreover, this also shows that further optimizations in graph partitioning can lead to capacity improvements as well.

### 4.3.3 Throughput-based methods for cluster dimensioning

We argue that performing a throughput analysis — even at a moderate scale — is not only helpful in order to assess scalability and performance properties of a platform and/or an implementation; but also to perform shrewd resource allocation. First, a moderate-scale

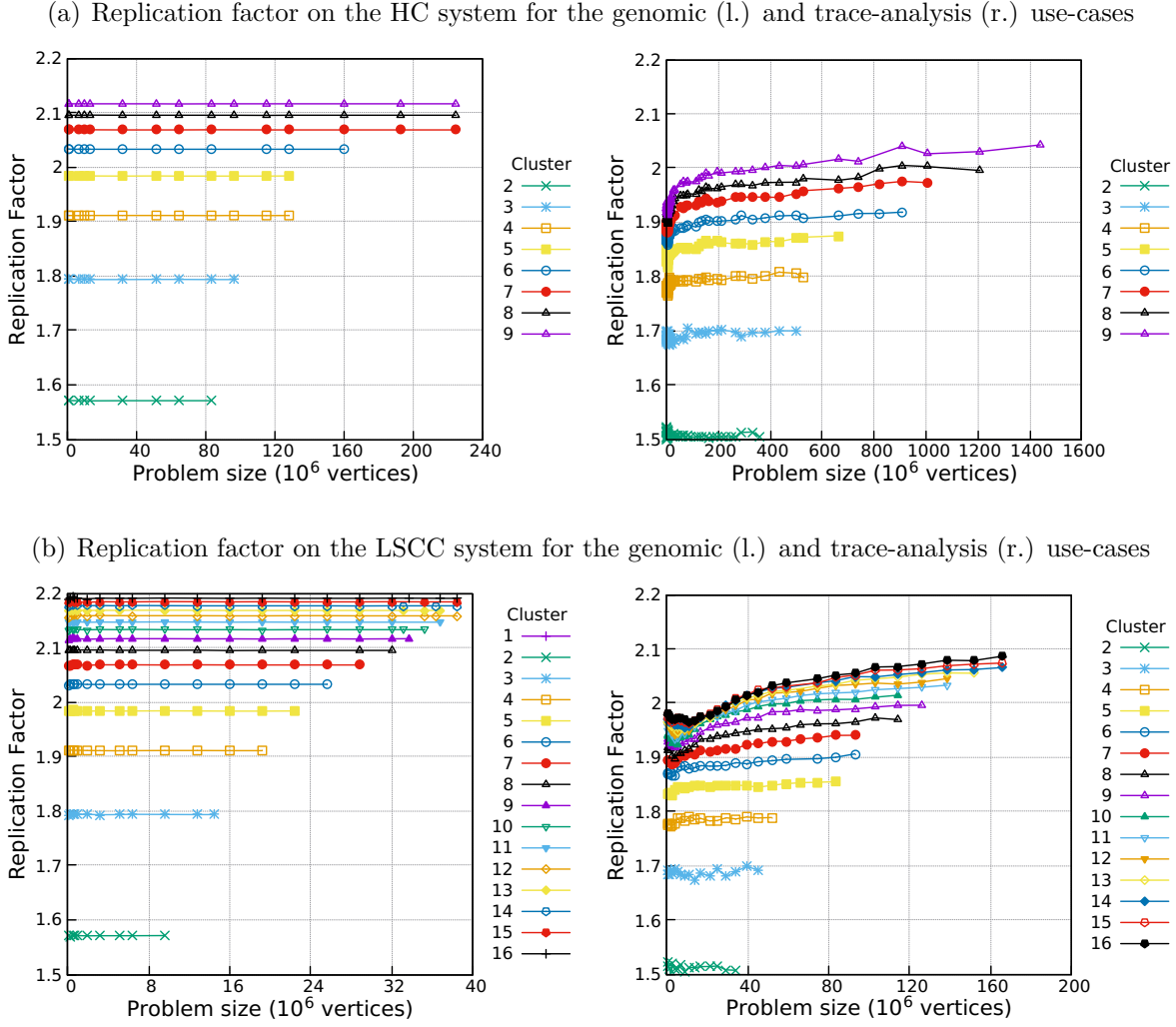


Figure 4.10: Evolution of the replication factor of the (a) HC and (b) LSCC systems with varying problem size and cluster scale. For each machines, results for the genome (respectively, the trace analysis) use-case are displayed on the leftmost (respectively, the rightmost) figure.

throughput analysis has to be performed in order to evaluate unit capacities of the system. Then, an evaluation of the replication factor must be performed. With this in mind, one can compute the approximate total number of replicas of a graph, once deployed. Finally, by dividing the replica number by the unit capacity and rounding the result to the upper unit, a cluster configuration (in node number) is obtained.

The obtained cluster configuration can be seen as a somehow lower bound in terms of cluster configuration, as it is the smallest number of machines able to process the given problem size in the nominal range. Yet, though this lower bound can be relatively easily computed, the upper bound is more difficult to reach. Indeed, the upper bound in terms of configuration can be seen as the maximum number of compute nodes that set the system in the nominal range — in such configuration, adding a compute node would place the cluster in the underloaded range. Thus, as throughput figures can vary with dataset properties, precisely assess such bound is a tough task, especially with a narrow nominal operating range such as the one exhibited by the LSCC system.



In the context of our experiments, we observed, as a rule of thumb, that doubling the number of machines given by the minimal configuration kept the system in the nominal range, allowing the achievement of higher global performances. However, these performances are obtained at the cost of a higher amount of resources leveraged, resulting in an increase in cost and power requirements.

To summarize, we foresee two bounds limiting the space in which dimensioning a cluster. Firstly, a performance-oriented policy, in which the largest number of machines will be used to process as fast as possible the graph, in the nominal operating range. In such operating point, the system exhibits its system peak throughput for the considered problem. Though this may result in the system operating near the underloaded regime, non-processing parts of the execution (*e.g.* parsing or commit) may benefit from additional compute resources yielding an improved global throughput as well. However, due to the changing performance behaviors with respect to the dataset, it may not be difficult to set-up such configuration without underloading the compute nodes.

Secondly, an efficiency-oriented policy, in which the smallest number of machines will be used to operate in the nominal range, near the configuration peak throughput point. In such case, as a fewer number of machines are used, it leaves room for concurrent executions of other instances of the algorithm on remaining compute nodes, if allowed by the network capabilities. Otherwise, they can be left idling in order to reduce the global power consumption. In contrast with the previous policy, this configuration is more easily computed, given that vertices have a fixed memory footprint.

## 4.4 Conclusion and perspectives

We compared in the course of this chapter the performances of two hardware trends — namely commodity clusters and high-end servers — for vertex-centric graph-processing using the two use-cases we developed. Figure 4.11 summarizes some of the findings of the first section of this Chapter. In particular, Fig. 4.11(a) shows the minimum wall-clock time obtained for each system, whatever the cluster scale, while Fig. 4.11(b) shows the maximum throughput obtained in similar conditions. Though both systems exhibited similar behaviors for all use-cases, the LSCC system consistently outperformed the HC platform, despite a smaller amount of per-node compute resources. This gap in performance is further enlarged when taking into account pricing or power consumption aspects. Yet, the HC system was able, thanks to its larger memory, to process larger scale problems.

In terms of architectures, the aspect of throughput curves of both systems calls for a decrease in per-node memory for the HC platform while the LSCC system would require the opposite. Hence, a perspective of interest is to investigate different RAM configurations on these systems in order to find an adequate memory-to-processor ratio.

In the second section, we proposed a practical method for cluster dimensioning by performing a throughput study. We saw that, accounting for the scalability properties of

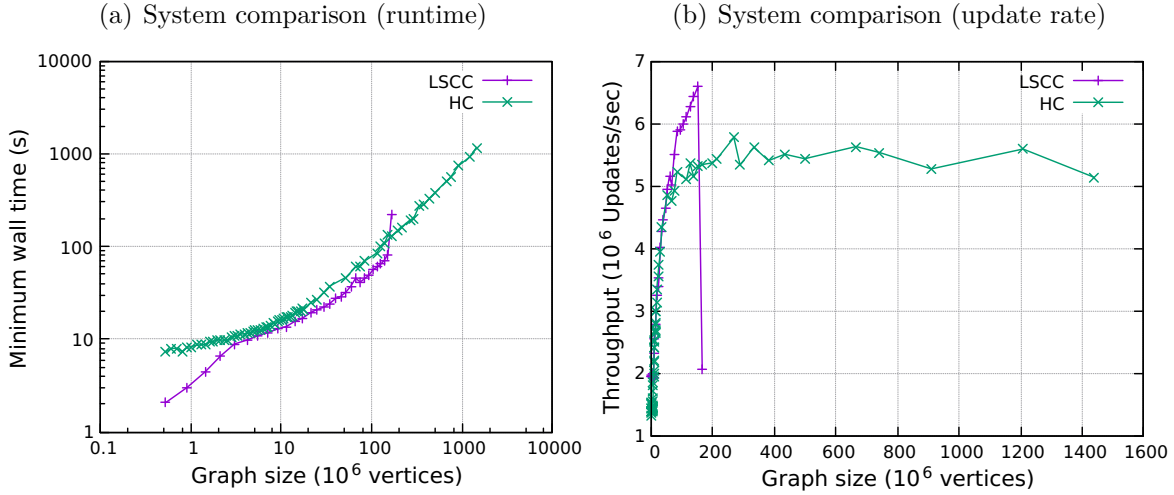


Figure 4.11: System for the program trace analysis of the LSCC and the HC systems in terms of (a) wall-clock time (lower is better, logarithmic scales) and (b) throughput (higher is better). For each metric, the best value achieved amongst every available cluster configuration is plotted.

GraphLab, the knowledge of the replication factor and the performance behaviors can lead to compute several valid cluster configurations for a given problem size. Such configuration prediction routines eventually lead to allocate resource appropriately, whether the aim is the most efficient or the fastest execution. We argue that such routines could be integrated into a runtime which would monitor executions and automatically allocate (after a learning process) resources.

Another perspective of interest could be to take benefits from the changing parallelism degree between the execution, parsing or commit steps. By way of example, the possibility to allocate the largest possible configurations at parsing, in order to fully exploit the inherent parallelism, and then reduce the number of used-machines to process the graph efficiently in the nominal operating range, shall be further investigated. Finally, as we glimpsed that the trace analysis algorithm was not able to leverage the large number of cores of the HC system, it might be interesting to adapt the number of GraphLab threads to reduce the overall consumption.

Since we compared performances of mainstream distributed architectures and investigated a cluster sizing mechanism, we propose in Chapter 5, architectural improvements for graph-processing servers. First, as we observed throughout this Chapter, a particularly interesting point of operation (in terms of throughput) is located towards the interface between the nominal and overloaded ranges. However, ensuring that the system will not transition to the overloaded range, resulting in drastically reduced performances, is a difficult task when the target system provides only a narrow nominal range. Hence, we describe the benefits a victim-swap mechanism to mitigate this problem and turn the abrupt throughput decrease into a more graceful degradation.

The performance experiments performed in this chapter have confirmed that graph-processing applications are memory-bound problems, if needed be. We have seen that

simpler architectures were able to outperform higher-end massively parallel servers. Consequently, we investigate the relevance of emerging ARM-based low-power computers for graph-processing in the next chapter. Indeed, though such platforms shall exhibit lower performances due to the more frugal processor, it shall be interesting to investigate its performances with respect to its promising power efficiency. To this extent, we ported GraphLab to such architectures and performed a throughput analysis of a moderate-scale ARM-based cluster.



# More efficient graph-oriented cluster design

## Contents

---

<b>5.1</b>	<b>Flash-based victim swap towards graceful performance degradation . . . . .</b>	<b>106</b>
5.1.1	Motivations . . . . .	106
5.1.2	Evaluation of flash-based swap memory . . . . .	107
5.1.3	Perspectives in graph-processing cluster design . . . . .	109
<b>5.2</b>	<b>Microserver-based architectures for efficient graph-processing</b>	<b>109</b>
5.2.1	Motivations . . . . .	110
5.2.2	Hardware architecture . . . . .	110
5.2.3	Using GraphLab on ARMv8 architectures . . . . .	110
5.2.4	Single-node operating performances . . . . .	112
5.2.5	Distributed operating performances . . . . .	113
5.2.6	Relevance of ARM-based platform for graph-processing . . . . .	115
<b>5.3</b>	<b>Conclusion and perspectives . . . . .</b>	<b>116</b>

---

In Chapter 4, we compared the graph-processing performances of two relevant hardware trends, namely commodity clusters and high-end compute servers leveraging the two use-cases introduced in Chapter 3. We observed so far that, though being of different architectures, the three systems exhibit a similar threefold performance behavior. Firstly, an underloaded range of operation can be witnessed, where the considered problem size is too small for the cluster configuration, hence the system is not able to fruitfully exploit the compute resources. Then, with increasing problem size, the system enters the nominal operating range, where adding compute nodes help in scaling out performances in a relevant and assessable way. Finally, once the processed graph exceeds the memory available, the swap operations decrease the throughput dramatically as the system operates in the overloaded area.

We have seen also that the evaluation of performance behaviors is not only helpful in understanding if the platform is efficient. Indeed, by estimating key characteristics from the performance charts, we have shown that it is possible to adequately size a cluster,

with respect to the design objective — maximum performance of minimum resource usage. Moreover, the study of these performances helped foreseeing hardware improvements to foster in order to increase performances of the benchmarked clusters.

The studies conducted in Chapters 3 and 4 have shown that, in general, the peak throughput point is located towards the border between the nominal and overloaded operating ranges. However, even having evaluated properly the unit capacity of a system, it can be difficult to target precisely this operating point as experimental variations may push the system in the overloaded area, outstandingly hampering performances. Moreover, it may be as well difficult to precisely assess the border between the operating and overloaded regime, hence increasing the risk of overwhelming the system inadvertently. To address this issue, we propose and evaluate in Section 5.1 a victim-swap mechanism aimed at turning the abrupt transition between the nominal and overloaded ranges in a more graceful degradation.

In Section 5.2, we then investigate further architectural aspects of graph-processing platforms and focus on emerging low-power systems. Indeed, the comparison of the two most relevant systems, namely LSCC and HC, has shown that, despite its more moderate price and compute resource amount, the LSCC system exhibited better performances than the higher-end HC platform. Hence, it becomes relatively natural to foster the use of lower consumption processor, with adequate memory amount and assess their relevance in such a memory-bound context. In order to investigate this issue, we benchmarked an emerging ARM-based, low-power processing module in order to evaluate its performance for graph-processing tasks. Though we expect a performance decrease due to inner microarchitectural details of the processor, we expect this decrease to be only moderate while being orders of magnitude more power efficient.

## 5.1 Flash-based victim swap towards graceful performance degradation

First, the motivations behind the exploration of the usage of a victim-swap mechanism are presented. Then, we evaluate the relevance of this approach in the context of a graph-processing system by investigating the performance gains brought by the victim-swap. Finally, some perspectives on its applicability are presented.

### 5.1.1 Motivations

As highlighted in previous chapters, the range of relevant performances is limited to a certain given problem size, determined by the amount of available memory and the data-structure implemented. This performance degradation, as observed, is mainly caused by the memory shortage, forcing the operating system to swap in and out large parts of the memory allocated to programs. As swap spaces are usually files or dedicated partitions

allocated on the hard disk drive, a severe bandwidth penalty is induced, hence slowing down the computation of the victim program and/or the whole system. Eventually, in extreme cases when programs saturate the memory and make use of the swap space for data being processed, the slowdown can render the machine unavailable, thus requiring a reboot.

Once a proper performance behavior analysis has been performed, hence the application has been characterized, the abrupt throughput decrease can be predicted and thus avoided by a careful cluster dimensioning — *e.g.* by allowing a sufficient number of machines. However, when operating a cluster around the maximum problem size — an operating point of interest to extract the maximum throughput of a system — it may happen that the system runs out of memory hence hampering drastically the observed throughput, possibly to the point where the cluster is rendered unusable.

To mitigate this issue, we argue that using dedicated Flash-based swap memory can help. Indeed, Flash-based memory is orders of magnitude faster than hard disk drives, hence, dedicating an amount of flash memory for swap operations could help recovering the system when the cluster is accidentally operating in the overloaded range. As of today, Solid State Drives (SSD) of small to moderate capacities (*e.g.* 8-16Go) can be purchased for less than 50 euros, making them an inexpensive addition helping mitigating swap-related system hanging.

### 5.1.2 Evaluation of flash-based swap memory

In order to measure to which extent this approach can be useful, we installed a SSD in a LECC node. The SSD was partitioned so that the swap space spanned the whole available disk space and other swap files were deactivated.

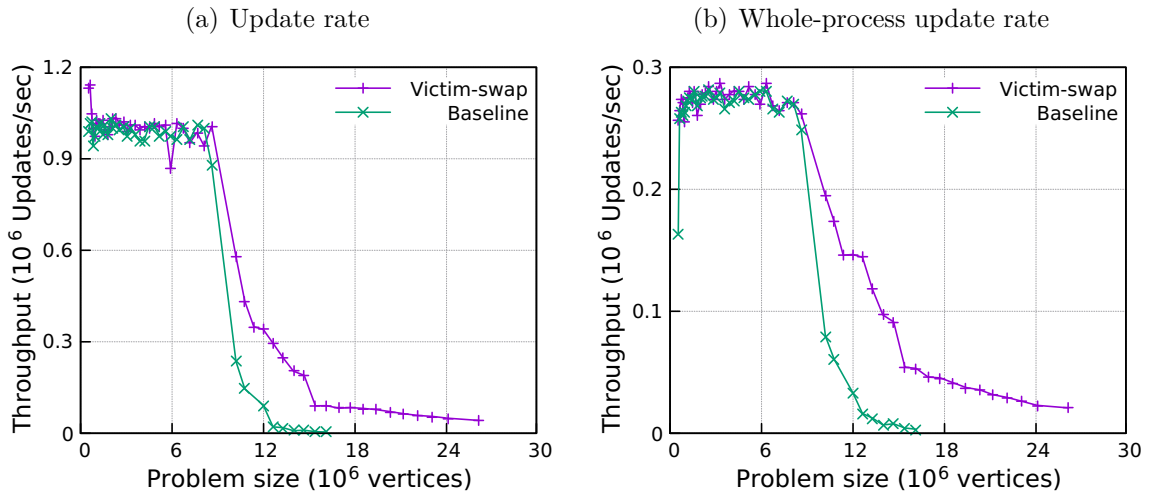


Figure 5.1: Update rates (a) and Whole-process update rates (b) comparisons of the victim-swap approach (purple curve) with the baseline (green curve), using the program trace analysis algorithm. Each performance curve is plotted for a single-node configuration.

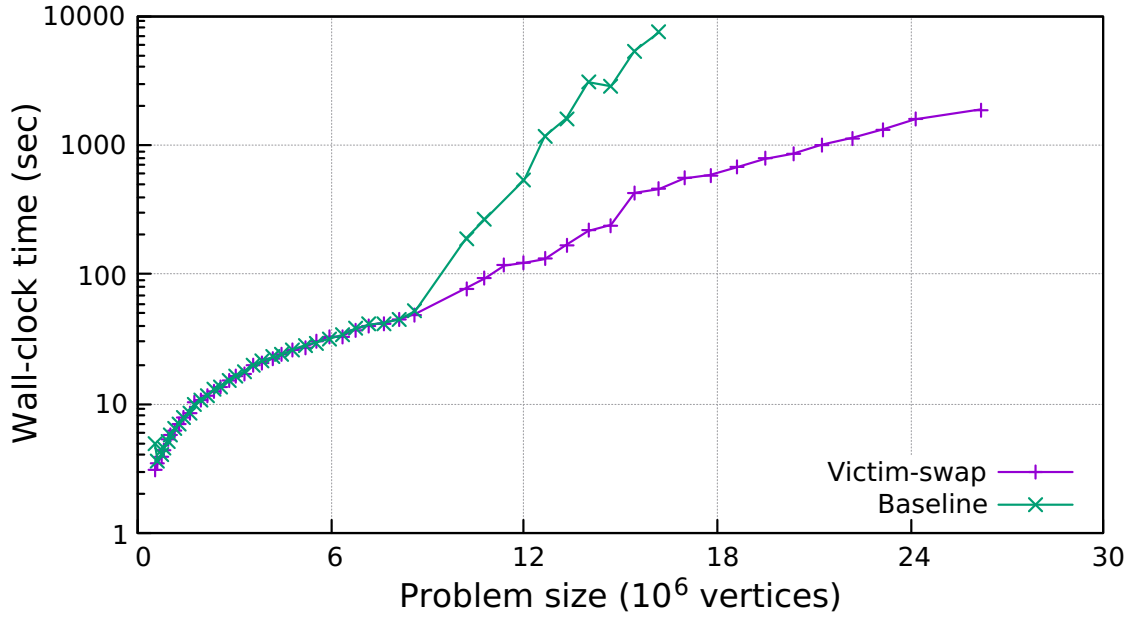


Figure 5.2: Wall-clock execution time comparison of the victim-swap approach (purple curve) with the baseline (green curve), using the program trace analysis algorithm (semilog axis).

Figure 5.1(a) shows the performance behaviors of two single-node configurations — the standard LECC node and a victim-swap-enabled node — on the trace analysis algorithm with the `matrix_multiply` kernel. The single-node operating performance behavior exhibits no underloaded range in both settings, as no communication penalty occurs. Both configurations exhibit a similar nominal operating range in terms of width and throughput. Finally, in the overloaded range, a more graceful performance degradation can be observed on the victim-swap configuration.

Figure 5.1(b) shows the global performances, in terms of whole-process update rates of the victim-swap enabled compute node and a standard LECC node. Analyzing global performances shows even more strikingly the benefits of using a flash-based victim swap, as global performances in the overloaded range are improved significantly. Such performance improvements are also shown by Fig. 5.2, in terms of wall-clock execution time.

In details, three zones are observed in the improved overloaded operating range. First, an important decrease in performance is observed between graph sizes of 11 and 14 million vertices. Although lowered, throughput remains around three times higher than the throughput observed using traditional disk-based swap space. Then, a second zone can be observed where the linear degradation is slightly less abrupt than previously, and where problems may still be processed reasonably. However, for graphs larger than 18M vertices, once the system gets further away in the overloaded range, the benefits of using a victim-swap are negligible and performances down to match standard HDD swap throughput performances. In this particular case, a victim-swap of at least the size of the available RAM, *i.e.* a partition of 4GB to 8GB, seems adequate.



### 5.1.3 Perspectives in graph-processing cluster design

We observed in previous chapters that, for a given problem size, a cluster configuration using the minimum number of machines is a particularly relevant setup. In such a case, the peak throughput point of a configuration is often met and the system operates in a setting where it can be close to the overloaded range. Due to unwanted experimental events or to the difficulty to precisely assess the limit between overloaded and nominal ranges, a system can easily get overwhelmed.

This section has shown that adding a dedicated flash-based swap partitions is helpful as a safety net mitigating the abrupt, swap-induced, performance degradation occurring in the overloaded range. When considering that such addition to a system is inexpensive and can be transparently installed into most mainstream machines, we argue that flash-based swap must be added to graph-processing servers. Indeed, the idea behind this extension would not be to operate the system in the overloaded range and benefits from the lesser decreased performances but to add flexibility to the machines. Moreover, this approach could be more easily generalized on microserver architectures where flash memory is mainstream to cope with somewhat limited RAM size.

## 5.2 Microserver-based architectures for efficient graph-processing

A traditional limitation in large server installations is the power consumption and heat generation, hence requiring complex cooling facilities and power sources. To address this issue, emerging so-called microservers can be leveraged. Microserver architectures are possibly-heterogeneous servers with low-power processors historically seen on embedded systems, such as ARM-based units. Usually, raw performances of such systems are lower than those of comparable high-end compute servers. However, such a performance decrease is often compensated by a drastically lowered energy consumption, cost of acquisition and operational expenditures.

As observed in Chapter 4, large high-end installations can be outperformed by commodity clusters with much simpler architectures. With this in mind, it could be of interest to measure performances of such platforms in order to evaluate their relevance in the context of graph-analytics. The remainder of this section addresses this question. First, a detailed introduction to the motivation of this study is provided, as with the presentation of the Nvidia's Jetson TX2. Then, we present the experimental context of this study and in particular the GraphLab modifications performed to extend its architectural support to the ARMv8 instruction set architecture (ISA). Finally, performance of the platform with GraphLab are discussed before perspectives of this work are detailed.

### 5.2.1 Motivations

Graph-processing is a domain known to be memory- rather than compute-bound. In Chapter 4, we confirmed this assumption as we observed that, performance-wise, simpler commodity clusters outperformed considerably more expensive higher-end compute servers. Building on this idea, we argue that emerging compute platform composed of a frugal processing element with an adequate memory amount can be of interest in the field of High-Performance Data-Analytics. In particular, such emerging platform are now composed of 64-bit processors with a fraction of the energy consumption seen on more traditional architectures, which can ultimately help in mitigating heating and consumption issues in data-centers. Moreover, lowering the energy needs for such compute clusters also has a positive impact on the operational expenditures, hence improving its performance to consumption and performance to cost figures.

Amongst the panel of available boards to conduct this experiment, the choice of using the TX2 board was motivated in particular by the following characteristics. First, the module total power consumption is under 7.5 Watts [107] which is an order of magnitude lower than the clusters previously used in this thesis work. Then, the available per-node memory amount (see Tab. 5.1) is one of the largest available for such platform and is comparable in that respect with the LSCC system. Moreover, the TX2 comes with an installed mainstream Linux distribution and a considerable support, thus facilitating its handling. Finally, the module retail price is relatively moderate, making the assembly of a compute cluster made out of linked TX2 modules a reachable goal, and rising its update rates per euro performances.

### 5.2.2 Hardware architecture

As a relevant example of an embedded computing node architecture, we chose to use a Nvidia Jetson TX2 module [107]. The module is build around a quad-core ARM A57 (64-bit), a dual-core Nvidia Denver2 and a NVIDIA Pascal GPU, with 8GB of memory (LPDDR4, 128-bit interface with a theoretical peak bandwidth of 59.7GB/s) and 8GB of swap space mounted on an SD card.

The main memory of the module is held by a 32GB eMMC 5.1 storage unit and numerous networking interfaces are provided, *e.g.* Gigabit Ethernet, Bluetooth4.1 or WLAN. The two TX2 boards are linked across a 1GB/s Ethernet network and each module is running Ubuntu 16.04.

### 5.2.3 Using GraphLab on ARMv8 architectures

We conducted a performance analysis of the board, using the two use-cases presented in Chapter 3, namely the trace analysis and the graph filtering algorithms. The performance comparison was firstly performed on a single-node configuration and was extended to a two-board configuration upon availability of the second TX2. As the TX2 comes

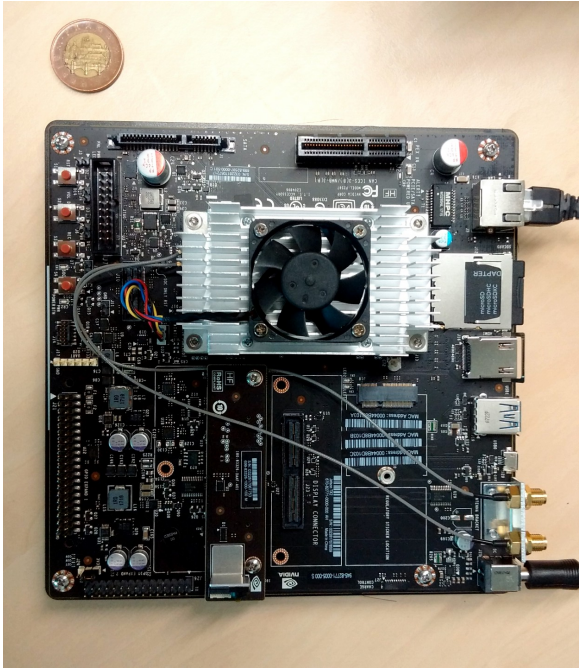
(a) Detailed description		(b) TX2 module	
			
	TX2		
Base configuration	Jetson TX2		
Per-node base price	EUR660		
Total Nodes	2		
Total Cores	8		
Total Threads	8		
Total Memory	16GB		
Node architecture	64-bit ARM A57		
Core number	4		
Cache	32kB D1, 2MB L2		
Max frequency	2.0GHz		
Board Power	7.5W		
Node memory	8GB		
Node swap	8GB		
Memory type	LPDDR4		
Memory speed	(128-bit interface) 1866MHz		
Network	1GBps/Ethernet		
OS	Ubuntu 16.04		
GraphLab version	v2.2 PowerGraph (for ARMv8)		
MPI layer	OpenRTE 1.6.5		

Table 5.1: (a) Detailed description of the Nvidia Jetson TX2 module used for the study [107]. (b) The core/threads numbers given correspond to the number of resources actually used in the experiment, hence excluding GPUs and accelerators. CZK50 coin for scale.

with an already configured Ubuntu Linux distribution, satisfying most of GraphLab’s dependencies was considerably facilitated.

However, a major hurdle for the execution of GraphLab implementations on the TX2 module was the lack of support of the ARMv8 ISA by the framework. Indeed, GraphLab is a large framework with a vast code-base designed to be compiled and executed on `x86_64` architectures, hence requiring a dedicated port for the 64-bit ARM A57 processor embedded in the TX2. The GraphLab port we performed in order to conduct our study constitutes itself a valuable contribution which will be helpful for the future evaluation of ARM-based platform for GraphLab applications. In further details, we upgraded some of the core dependencies of GraphLab and modified core assembler routines to make them ARMv8-compatible.

Architecture-wise, the Pascal GPU and the dual-core Denver units were not used as no support for such accelerator is provided by GraphLab at the time of writing. Input and output files were written to the embedded 32GB flash memory. Finally, due to a lack of mature support of the 64-bit ARM architecture, GraphLab was compiled with the standard `malloc` library instead of the more thread-friendly `libtcmalloc` recommended, contrary to other GraphLab implementations previously deployed.

### 5.2.4 Single-node operating performances

Single-node performances of the TX2 module are investigated in the following. In particular, Fig. 5.3 shows the performances of the module for the trace analysis use-case on three input kernel.

Firstly, we observe a comparable performance behavior with respect to the single node behavior of other systems on comparable conditions. As used in a shared-memory setting, no underloaded range can be seen resulting in only two operating ranges exhibited by the TX2 module, namely the nominal and overloaded ranges. The transition between these two regimes is delimited by the occurrence of swap operations, as indicated by the rise in pagefaults number in Fig. 5.3(c), 5.3(f) and 5.3(i).

In average, the performances exhibited by the system are lower than those of our previously-introduced Intel-based systems (Fig. 3.3 and 4.3), as expected for a much simpler and frugal processor architecture. However, the decrease in throughput is only moderate when compared to its closest competitors, the LSCC node, for a much lower energy consumption. Indeed, the whole TX2 board claims a power consumption of 7.5W (including memory systems, GPU and accelerators), whereas the sole TDP of the Intel i5 equipping the nodes of the LSCC systems is of 84W.

In further details, the TX2 single-node throughput on the nominal range is set at 1.3 MUPS for the trace analysis use-case, processing the `matmult` kernel<sup>1</sup>. Though it outperforms the moderate LECC system (1 MUPS) in a comparable setting, it only reaches 65% and 52% of the single node performance of the LSCC and HC platforms, respectively. However, when considering performances to consumption or performances to cost figures, the TX2 exhibits relevant performances. Indeed, it shows a remarkable 173.3 kUPS/W which is one order of magnitude higher than the LECC, LSCC and HC systems (respectively, 13.33, 23.81 and 13.88 kUPS/W). Accounting for its price, the TX2 single-node performance reaches 1.97 kUPS/€, a significantly higher value than the HC platform (0.41 kUPS/€). However, it is outperformed by both commodity clusters, which exhibit about 2.5kUPS/€ each.

It is to be noted that, in our experiments, we only performed computations using the ARM64 multicore, hence leaving the GPU at rest and thus decreasing the power consumption of the board. Unfortunately, due to the TX2 design limitations we were not able to precisely monitor the consumption of the system during the experiment, which is now a perspective of much interest to characterize such system under a graph-processing workload. Finally, though we do not expect GPUs to bring significant improvements in raw throughput due to the high level of indirection and the data-dependent nature of graph applications, they may bring performance gains for additional pre-/post-processing tasks. However, although maybe improving slightly performances, the programming challenge associated with the leveraging of such hardware accelerators remains significant.

---

<sup>1</sup>See Sec. 3.1.2

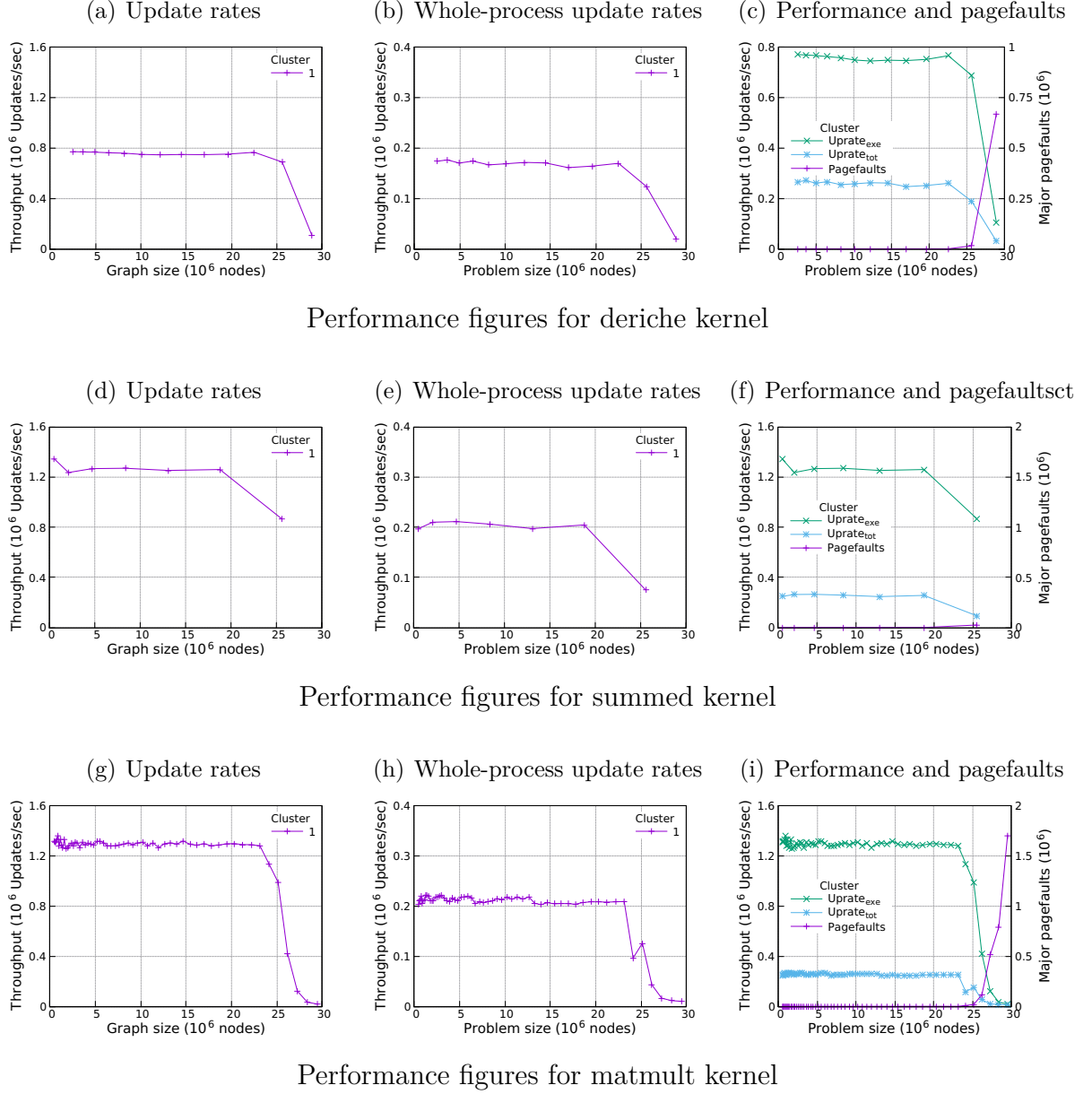


Figure 5.3: Operating performance of the TX2 board on the trace analysis use-case (*matrix\_multiply* kernel). Figure (a)(d)(g) shows the update rate of the system, or the number of updates performed per second during the processing part of the program. Figure (b)(e)(h) shows the whole-process update rate of the system, which is the number of updates performed per second during the program execution time. Figure (c)(f)(i) shows the amount of PageFaults impacting the system throughput.

### 5.2.5 Distributed operating performances

Having evaluated the performances on a single TX2 node, we now investigate the distributed performances of the TX2 platform. To this extent, we linked a second identical TX2 module using a Gigabit Ethernet switch. Both TX2 modules exhibited then 8GB of physical memory and an additional 8GB of swap space dispatched on an embedded SDCard due to the lack of space in the main flash memory of the modules.

Figure 5.4 shows the throughput of the system for both single- and dual-node configurations. As observed on other platform, a two-node configuration barely matches the

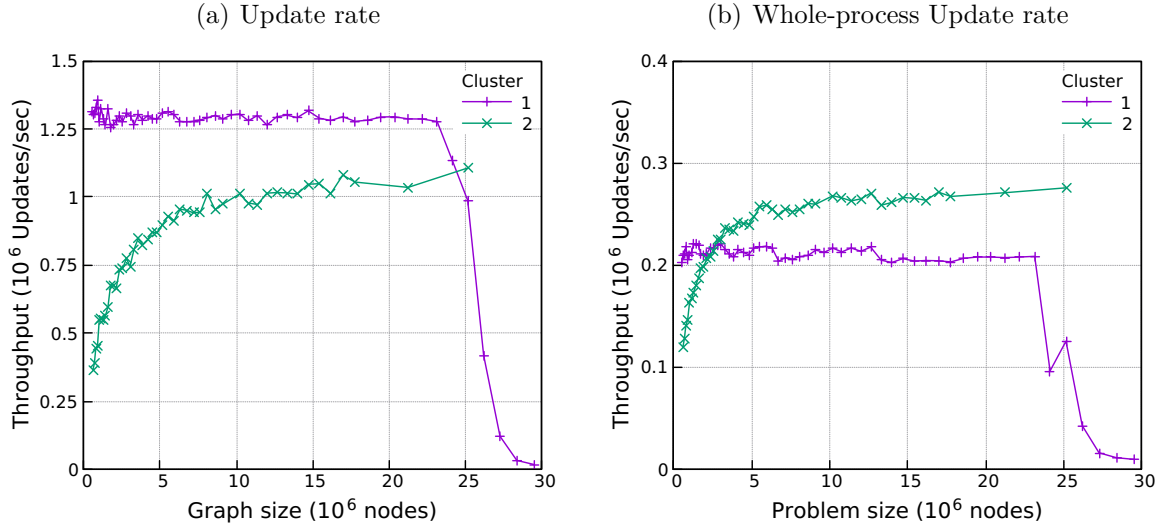


Figure 5.4: Distributed performances of the TX2-based platform on the trace analysis use-case (*matmult* kernel). Figure (a) shows the update rate while Fig. (b) shows the whole process throughput achieved by the system.

update rate of the single-node setting with respect to update rate. Moreover, with the two-node configuration, we were not able to process instances much larger than those of single-node capacity as we experienced network failures preventing the completion of larger runs. We found out that the problem might be related to the controller managing the Gigabit Ethernet module. However, though exhibiting lower raw throughput figures (as *e.g.* the LSCC system), the dual-node setting globally outperforms the single-node configuration with respect to the whole-process throughput figure.

In details, the peak dual-node throughput observed for the program trace analysis use-case is of 1.175 MUPS, which is above the LECC system dual-node performances. Yet, the TX2-based systems is outperformed by the LSCC and HC systems, which show a peak 2-node throughput of 1.9 and 2.5 MUPS, respectively. Similarly as observed for single-node performances, the TX2 dominates in terms of performance to consumption ratio, with 78.3 kUPS/W — a value to compare with those of the LECC, LSCC and HC platforms, respectively 7.33, 11.31 and 6.58 kUPS/W. With respect to price, the TX2-based system shows a notable 0.89 kUPS/€ outstandingly overcoming the HC platform (0.21 kUPS/€), yet remaining behind the commodity clusters (1.41 and 1.19 kUPS/€, respectively).

Similar observations can be made when investigating performances on the De Bruijn graph filtering algorithms. Performances visible in Fig. 5.5, shows that the dual-node configuration is outperformed by the single-node one, at the benefit of a (theoretically) larger capacity. Though we were not able to obtain results for graphs much larger than 6 million nodes due to the aforementioned networking problem, one can foresee that the dual-node setting should eventually outperforms the single-node setting on larger problems. Indeed, on the very last point we were able to gather, this tendency is initiated.

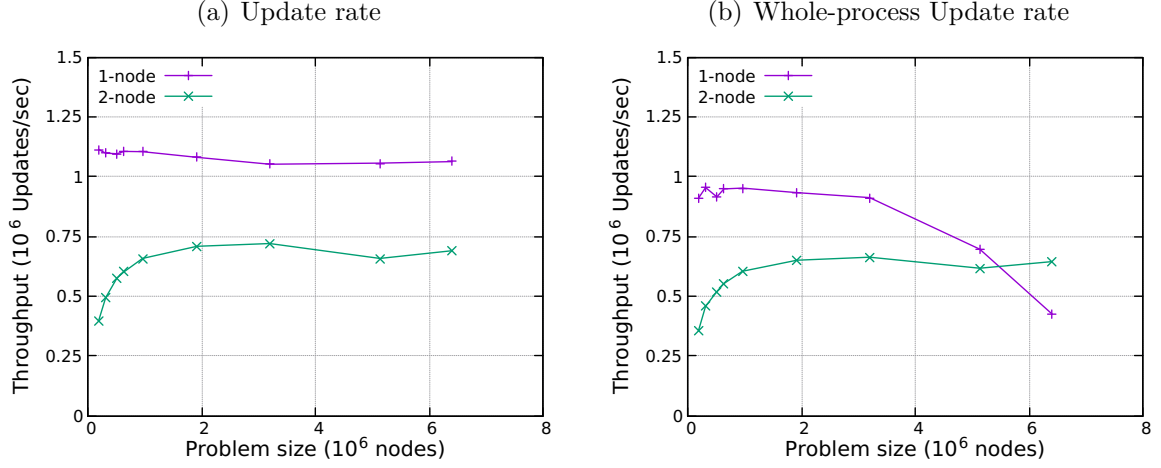


Figure 5.5: *Distributed performances of the TX2-based platform on the De Bruijn graph filtering use-case (Error rate: 0.75%, Tolerance: 2, Threshold: 0,  $k$ : 31, Operating coverage: 28, Sequencing depth: 40, Readlen: 100). Figure (a) shows the update rate while Fig. (b) shows the whole process throughput achieved by the system.*

Moreover, the decrease in throughput between the update and the whole-process update rates is really little on the dual-node setting, whereas it is more important with only one TX2 module.

Even though operating at a moderate scale, we foresee a unit capacity in terms of replica of about 22M and 7M vertices for the trace analysis and DBG filtering use-cases, respectively. These capacities are comparable to those of the LSCC platforms, which shares the same amount of physical memory. The exhibited 2-node replication factor converges toward 1.51 for the trace analysis algorithm and 1.57 for the DBG use-case. These values are comparable to the replication factors exhibited by the LSCC or HC systems in Fig. 4.10, hence we expect a similar behaviors on larger scales.

Using these values, we can estimate the maximum processable graph of the 2-node setting for the trace analysis use-case, even though some points are missing. By way of example, the 2-node configuration shall tolerate a 44M replica graph, or a 29M true vertex graph with respect to the 1.51 replication factor. Similarly, the system should be able to process graphs up to 11M true vertices with the DBG filtering algorithm. Finally, we expect the TX2 system to behave like the LSCC platform, due to their similar number of cores and memory amount, although exhibiting lower throughput and consumption at a comparable price.

### 5.2.6 Relevance of ARM-based platform for graph-processing

This section has presented early experimentations using emerging low-power nodes in the context of distributed graph-processing. A system composed of two TX2 nodes was setup and benchmarked using the two use-cases presented previously. Despite the networking problems, the presented system exhibits a similar performance behavior when compared

with the previously evaluated platforms. Having been in further details, we observed as well that the TX2 platform shows remarkable performances and often compares very favourably to other comparable systems, despite a much simpler processor. Indeed, when accounting for its price or lowered energy consumption, the TX2-based system outperforms clearly the HC system and, at least, matches performance figures of the commodity clusters.

For all these reasons, we argue that ARM-based microservers are a hardware path of interest toward more efficient graph-processing systems. To confirm these statement, a larger scale experiments should be conducted using more TX2 nodes to validate their behaviors on a more representative setting. Moreover, the GraphLab port for ARMv8 architectures we performed, will also be helpful for the performance evaluation of other similar hardware.

### 5.3 Conclusion and perspectives

In this chapter, we explored two hardware paths towards arguably more efficient graph-processing platforms.

The first explored proposition addresses the issue of unwanted operation of a system in the overloaded regime. The precise assessment of the unit capacity is a tough task and, as operating near the border between these two regimes is arguably an interesting operating point as shown by previous chapters, undesirable operations in the overloaded range can occur. In such cases, a dramatic performance decrease is observed, due to numerous swap-in/-out operations performed consequently to the memory exhaustion. The victim-swap mechanism proposed allows the system to compensate turn this abrupt decrease into a more graceful performance degradation by using flash-memory — a technology orders of magnitude faster than hard disk drive. Indeed, we observed on the performed experiments a significant improvement in global performances, hence upgrading a machine with a flash-based victim swap can be seen as an inexpensive safety net. However, the experiments have shown as well that this improvement is bounded: when leveraging this approach with large amount of swap, performances are lowered towards regular, hard-disk drive swap space. Finally, though we observed promising results on single-node cluster configurations, larger scale distributed experiments must be performed in order to assess potential gains brought by dedicated victim-swap partitions on distributed architectures.

The second part of this work explored a more profound hardware change. Previous chapters have shown that commodity clusters composed of simpler processors are able to compete and outperform much higher-end systems. With this in mind, we investigated the relevance of emerging computing platforms embedding low-power cores with relevant amount of memory. The experiments performed with a 2-node system showed promising results, even though further investigations on larger scales are required. Indeed, although with a tenfold decrease in power consumption, the TX2-based platform exhibited relevant



throughput performances. Thus, considering these statements and the relative lower price of such processors, clusters of ARM-based microservers seem a trend to investigate in the context of graph-processing applications. They shall further help in mitigating heat generation in data-centers as with reduce drastically the operating costs.



# Conclusions and perspectives

## Contents

---

<b>Synthesis . . . . .</b>	<b>119</b>
<b>Perspectives . . . . .</b>	<b>122</b>
Short-term . . . . .	122
Longer-term . . . . .	123

---

With the striking increase in the volume of processed datasets every year, larger and larger installations of computing resources are required by application from the rising High-Performance Data Analytics domain. However, graph-processing workloads are well-known for being strongly irregular, less structured and more data-dependent than traditional scientific computing applications. In such context, deploying a scalable and parallel implementation of an algorithm on distributed architectures is a challenging task. Though dedicated frameworks have emerged to facilitate the programming of graph algorithms, it is a tough task to understand and assess their scalability and performance behaviors — let alone appropriately size a cluster for a problem instance — using classic benchmarking approaches, an however utterly necessity.

In this thesis, we explored the parallel and scalable processing of large graphs on distributed architectures, with in mind to eventually propose cluster design guidelines towards more efficient executions. To this extent, we developed two graph-processing algorithms and set-up different hardware architectures. The performed experiments led us to propose practical methods for the appropriate sizing of clusters as with architectural directions for next-generation of graph-processing compute servers.

## Synthesis

As graph analytics algorithms are entering the field of High-Performance Computing (HPC) and the size of the processed datasets is increasing, the issue of understanding performance behaviors of such High-Performance Data-Analytics (HPDA) applications with respect to the underlying hardware or the scale of the problem is critical. To address this issue, we focused first on a threefold review of the HPC/HPDA landscape, in terms of hardware architecture, abstraction models and programming tools.

Firstly, having reviewed the landscape of high-performance distributed architectures enabled us to understand the hardware trends in the field. We particularly observed that although high-end compute servers are natural candidates for large-scale high-performance

computing, clusters of commodity workstations is a trend currently being exploited at comparably large-scales as a competitive alternative — this is notably due to its more affordable nature. In accordance with this trend, we decided to select three hardware architectures: two commodity clusters and a high-end installation.

Then, the state of the art of parallel programming models has been reviewed, in order to understand the ties between software and hardware. In particular — and especially in the context of parallel programming — it is extremely relevant to understand how threads of execution and communications are orchestrated. Moreover, it allows to better understand the freedom left to the programmer to exploit the hardware.

This review also helped in conceptualizing how software frameworks are built and facilitate the efficient implementation of algorithms through them. While the landscape of software tools for the implementation of parallel graph algorithms is wide, vertex-centric programming libraries have raised interests as they provide an acceptable compromise between increased productivity, performance and community acceptance. In particular, we identified and further investigated the GraphLab framework, a popular Pregel-like, vertex-centric C++ library for distributed architectures.

We consequently studied in details vertex-centric frameworks in terms of programming models and practical details. As such programs are composed of different parts exhibiting different degrees of parallelism, the use of traditional metrics may not be sufficient to grasp the behavior of the program with precision. This difficulty hence made more complex the task of identifying inadequacies between the hardware and the implementation.

In order to gain more real-life insights on the implementation of graph analytics workloads in such a context, we decided to focus on two real-life graph-related problems: a trace analysis algorithm and a De Bruijn graph filter. Moreover, the relatively distant application domains of these use-cases advocates graph analytics as a cross-domain discipline of much relevance nowadays.

A first experimental work enabled us to gain a practical experience with GraphLab by implementing a trace analysis algorithm using this framework and deployed on a distributed architecture. This novel algorithm was studied with respect to scalability and throughput for varying datasets and cluster scales, enabling us to highlight three operating ranges. Being able to identify operating ranges of interest and — more importantly — to set a system in such conditions for a given dataset is key for an efficient execution. This work was notably presented during the HPCS'2016 conference [100].

Then, a second use-case in the context of genomic data processing was developed. In this experimental work, a filtering algorithm for De Bruijn graphs constructed from Next Generation Sequencers (NGS) was proposed, after a thorough study of the whole-genome sequencing data processing field. The algorithm showed promising results with respect to both applicative results and performance aspects. In particular, we observed similar operating ranges with this different algorithm deployed on a different hardware platform, with respect to performance behavior of the algorithm, than findings of the first

---

algorithm. This work was presented at PDCO, an IPDPS'2017 workshop [104] and an extended version of this paper is under publication at the time of writing.

Having studied independently, software-related aspects of the operating performances of our use-cases on different hardware platforms, we then focused on understanding how the underlying architectures impacted the performance behaviors of our use-cases and in particular its scalability. To achieve this, we performed a benchmarking of our three available hardware platforms using the provided implementations. This study highlighted that despite its high-end architecture, the HC system could be outperformed in terms of throughput by a cluster of inexpensive commodity workstations. Moreover, this study of operating points of the clusters has also allowed us to propose cluster sizing methods based on throughput analysis. Indeed, analyzing operating ranges of a system leads to the evaluation of unit capacities that, with knowledge of the graph replication factor evolution, can be wielded to determine the number of cluster nodes required to process a given problem instance efficiently.

As highlighted throughout this thesis work, operating in the overloaded range hampers dramatically performances as memory is saturated. We proposed and evaluated a flash-based victim-swap approach to inexpensively mitigate the abrupt decrease in throughput when approaching the system memory capacity limits. Indeed, this operating point is often near the peak throughput point of a given configuration, that is, the highest operating performances for a given configuration. In other words, it is also the smallest cluster configuration operating in nominal conditions for the associated problem size, hence the most efficient nominal configuration. As it is thus a sound operating point, it is sensible to have a reasonable amount of flash memory holding a victim-swap partition as an inexpensive safety net in case of an unwanted transition toward the overloaded range. We observed that overloaded performances using the victim-swap approach significantly overtake those of a standard node and helped improve significantly global performances before the overloaded ranges.

Finally, we investigated hardware-related hints for the design of next generation compute cluster towards efficient graph-processing. In particular, embedded computing platforms built around low-power processors and a relevant amount of memory bring interesting performances in a context of applications bounded by the memory amount (and not the compute power amount). The NVIDIA Jetson TX2 module, with its frugal ARMv8 processor and its relevant amount of memory is a particularly representative example of such emerging platforms.

To investigate its relevance in this context, we ported the GraphLab library to the 64-bit ARM processor and set-up a cluster composed of two TX2 boards. Then, we conducted a performance behavior analysis in order to compare its performances to our previously evaluated systems. Eventually, we found out that the ARM-based module only exhibits a moderate decrease in throughput while being considerably more energy efficient at a moderate price, highlighting its relevance in such a context. Building on

unit capacity and performance projections, we foresee that a comparable scale, such as an ARM-powered cluster shall be able to match and even outperform commodity clusters in terms of performance to price or performance to power figures.

Today these results and activities are now part of and contribute to the CEA LIST roadmap and the laboratory activities. This thesis work has in particular opened several perspectives to be explored, as summarized in the following section.

## Perspectives

### Short-term

In the near future, it could be of interest to further develop strategies for the automatic resource allocation in distributed clusters. Indeed, as stated in Chapter 4, GraphLab programs are composed of different parts exhibiting varying degree of parallelism. Hence, it could be particularly interesting to augment GraphLab with capabilities to use an adequate amount of compute resources at each step of the program. By way of example, exploiting a large amount of cores during the embarrassingly parallel parsing step and a more moderate amount at compute time seems an interesting path to explore.

Concerning design aspects, it could be of interest to set-up experiments which could lead to the assessment of the memory to CPU ratio exhibiting the best throughput performances. We observed that the HC platform is likely to have a too large per-node memory amount whereas the LSCC cluster may benefit from larger amounts of per-node memory. Hence, having means to determine this ratio should be within reach by performing a throughput analysis and would eventually lead to tailoring the hardware to the expected performance behavior.

Moreover, at the scale of our experiments, even up to 16-node configurations, we did not observe striking network contentions. However, scaling out to a larger scale, such as fifty-some or hundreds of machines, may lead to potential network-related performance slowdowns. Hence, much larger scale experiments have to be performed in order to observe such potential network issues. Additionally, much larger scale experiments would be required to further validate the evolution of operating ranges. Furthermore, such experiments will allow the assessment of the nominal range behavior at larger scales.

We explored the relevance of power-efficient, ARM-based platform in the context of graph-processing using a system composed of NVIDIA TX2 nodes linked across an Ethernet network. The promising results obtained with a moderate number of nodes foreshadow interesting performances to energy figures. However, in order to further validate the relevance of such a hardware trend, larger-scale experiments are required and must be performed. Moreover, though as of today, measuring the consumption of the board is out of reach, it could be interesting to investigate means of characterizing on a finer-grain basis the energy required during runs.

---

## Longer-term

On a more longer-term basis, we foresee improvements on an architectural level, but also with respect to automatic job scheduling and resource allocations.

Designing from scratch microserver architectures based on low-power ARM cores with dedicated network interfaces and adequate amount of memory is to be explored. Indeed, operating costs of nowadays data-centers are mainly driven by cooling infrastructures and power requirements. Considering the fact that large-scale graph-mining applications are further expanding their already applicative field, the compute resource requirements will not stop to increase in the coming years. However, to make this a reality, compute clusters must become more energy efficiency. To this extent, designing dedicated graph-processing clusters leveraging low-power cores, with relevant memory amounts is a necessity.

With respect to software aspects, performance-wise job schedulers for graph-processing applications can help in adequately allocate resources. Indeed, by automatically *learning* performances of repeated runs of an application, such a scheduler could construct its representation of the application’s performance behavior. Hence, building on the given hardware allocation policy — maximum performance of minimum configuration — the scheduler could use the learned information to appropriately dimension resources upon availability. Going further in such a direction, and having evaluated potential network contentions, one can imagine a multi-instance, multi-application scheduler able to optimize a server farm’s usage. Coupled with the ability to make coarse-grain variations in allocated resources to match coarse parts of a graph-processing program, it can lead to flexible and optimized execution, towards a more energy-efficient processing of large instances.





# Personal publications

## International conference papers

- **J.Collet, T.Sassolas, Y.Lhuillier, R.Sirdey, J.Carlier:** *Exploration of de Bruijn graph filtering for de novo assembly using GraphLab*, in: 2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), Orlando/Lake Buena Vista (FL), US, 2017
- **J.Collet, T.Sassolas, Y.Lhuillier, R.Sirdey, J.Carlier:** *Leveraging distributed GraphLab for program trace analysis*, in: High Performance Computing & Simulation (HPCS), 2016 International Conference on, IEEE, Innsbruck, AT, 2016
- **J.Collet, T.Sassolas, Y.Lhuillier, R.Sirdey, J.Carlier:** *Parallel computing model for data-mining applications*. In Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems (ACACES), 2015 Hipeac's 11th International Summer school on, Fiuggi, IT, Jul. 2015

## Poster presentations

- **J.Collet, T.Sassolas, Y.Lhuillier, R.Sirdey, J.Carlier:** *Hardware support and dynamic compilation for online optimization of data transfers in distributed multi-processor architectures*. In Heudiasyc Lab PhD student forum, Université de Technologie de Compiègne, Compiègne, FR, Jun. 2016
- **J.Collet, T.Sassolas, Y.Lhuillier, R.Sirdey, J.Carlier:** *Parallel computing model for data-mining applications*. In Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems (ACACES), 2015 Hipeac's 11th International Summer school on, Fiuggi, IT, Jul. 2015

## Talks

- **J.Collet, T.Sassolas, Y.Lhuillier, R.Sirdey, J.Carlier:** *Distributed graph-processing using emerging vertex-centric frameworks*. Seminar ES201, ENSTA-ParisTech, Palaiseau, FR, Mar. 2017
- **J.Collet, T.Sassolas, Y.Lhuillier, R.Sirdey, J.Carlier:** *Hardware support and dynamic compilation for online optimization of data transfers in distributed multi-processor architectures*. In Heudiasyc Lab PhD student forum, Université de Technologie de Compiègne, Compiègne, FR, Jun. 2015

## Submitted publications

- **J.Collet, T.Sassolas, Y.Lhuillier, R.Sirdey, J.Carlier:** *Distributed de Bruijn graph filtering for de novo assembly using GraphLab*, in special issue of Journal of Parallel and Distributed Computing

# Bibliography

- [1] James E Thornton, “Design of a computer: the CDC 6600”, in: *Scott, Foresman & Co., Glenview, IL* (1970).
- [2] M Dungworth, “The Cray 1 computer system”, in: *Infotech State of the Art Report on Supercomputers 2* (1979), pp. 51–76.
- [3] Leslie G. Valiant, “A Bridging Model for Parallel Computation”, in: *Commun. ACM* 33.8 (Aug. 1990), pp. 103–111, ISSN: 0001-0782, DOI: 10.1145/79173.79181.
- [4] Jack J Dongarra, Hans W Meuer, and Erich Strohmaier, *Top500 supercomputer sites*, 1994.
- [5] Hiroaki Fujii, Yoshiko Yasuda, Hideya Akashi, Yasuhiro Inagami, Makoto Koga, Osamu Ishihara, Masamori Kashiyaama, Hideo Wada, and Tsutomu Sumimoto, “Architecture and performance of the Hitachi SR2201 massively parallel processor system”, in: *Parallel Processing Symposium, 1997. Proceedings., 11th International*, IEEE, 1997, pp. 233–241.
- [6] N. Hirose and M. Fukuda, “Numerical Wind Tunnel (NWT) and CFD research at National Aerospace Laboratory”, in: *High Performance Computing on the Information Superhighway, 1997. HPC Asia '97*, 1997, pp. 99–103, DOI: 10.1109/HPC.1997.592130.
- [7] Michael S Warren, John K Salmon, Donald J Becker, M Patrick Goda, Thomas Sterling, and W Winckelmans, “Pentium pro inside: I. a treecode at 430 gigaflops on asci red, ii. price/performance of USD50/mflop on loki and hyglac”, in: *Supercomputing, ACM/IEEE 1997 Conference*, IEEE, 1997, pp. 61–61.
- [8] Leonardo Dagum and Ramesh Menon, “OpenMP: an industry standard API for shared-memory programming”, in: *Computational Science & Engineering, IEEE* 5.1 (1998), pp. 46–55.
- [9] Eugene W Myers, Granger G Sutton, Art L Delcher, Ian M Dew, Dan P Fasulo, Michael J Flanigan, Saul A Kravitz, Clark M Mobarry, Knut HJ Reinert, Karin A Remington, et al., “A whole-genome assembly of Drosophila”, in: *Science* 287.5461 (2000), pp. 2196–2204.
- [10] Pavel A Pevzner, Haixu Tang, and Michael S Waterman, “An Eulerian path approach to DNA fragment assembly”, in: *Proceedings of the National Academy of Sciences* 98.17 (2001), pp. 9748–9753.

- 
- [11] Gagan Agrawal, Ruoming Jin, and Xiaogang Li, “Compiler and Middleware Support for Scalable Data Mining”, in: *Proceedings of the 14th International Conference on Languages and Compilers for Parallel Computing*, LCPC’01, Cumberland Falls, KY, USA: Springer-Verlag, 2003, pp. 33–51, ISBN: 3-540-04029-3, URL: <http://dl.acm.org/citation.cfm?id=1769331.1769334>.
  - [12] Luiz André Barroso, Jeffrey Dean, and Urs Holzle, “Web search for a planet: The Google cluster architecture”, in: *IEEE micro* 23.2 (2003), pp. 22–28.
  - [13] Xiaoqiu Huang, Jianmin Wang, Srinivas Aluru, Shiaw-Pyng Yang, and LaDeana Hillier, “PCAP: a whole-genome assembly program”, in: *Genome research* 13.9 (2003), pp. 2164–2170.
  - [14] Robert Numrich, “Co-Array Fortran Tutorial”, in: *SC’03*, 2003.
  - [15] Rolf Rabenseifner, “Hybrid Parallel Programming: Performance Problems and Chances”, in: *Proceedings of the 45th CUG Conference*, Columbus, Ohio, USA, 2003, URL: [www.cug.org](http://www.cug.org).
  - [16] Mark Chaisson, Pavel Pevzner, and Haixu Tang, “Fragment assembly with short reads”, in: *Bioinformatics* 20.13 (2004), pp. 2067–2074.
  - [17] Nikolaos Drosinos and Nectarios Koziris, “Performance Comparison of Pure MPI vs Hybrid MPI-OpenMP Parallelization Models on SMP Clusters”, in: *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS’04)*, 2004, URL: [www.cslab.ntua.gr/~nkoziris/papers/jpdc2003.pdf](http://www.cslab.ntua.gr/~nkoziris/papers/jpdc2003.pdf), [ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1239870](http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1239870), <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1592859>.
  - [18] David A Bader and Kamesh Madduri, “Design and implementation of the HPCS graph analysis benchmark on symmetric multiprocessors”, in: *HiPC 2005*, Springer, 2005, pp. 465–476.
  - [19] R. H. Castain, T. S. Woodall, D. J. Daniel, J. M. Squyres, B. Barrett, and G .E. Fagg, “The Open Run-Time Environment (OpenRTE): A Transparent Multi-Cluster Environment for High-Performance Computing”, in: *Proceedings, 12th European PVM/MPI Users’ Group Meeting*, Sorrento, Italy, 2005.
  - [20] Tarek El-Ghazawi, William Carlson, Thomas Sterling, and Katherine Yelick, *UPC: distributed shared memory programming*, vol. 40, John Wiley & Sons, 2005.
  - [21] Alan Gara, Matthias A Blumrich, Dong Chen, GL-T Chiu, Paul Coteus, Mark E Giampapa, Ruud A Haring, Philip Heidelberger, Dirk Hoenicke, Gerard V Kopsay, et al., “Overview of the Blue Gene/L system architecture”, in: *IBM Journal of Research and Development* 49.2.3 (2005), pp. 195–212.

- 
- [22] Ruoming Jin, Ge Yang, and Gagan Agrawal, “Shared memory parallelization of data mining algorithms: techniques, programming interface, and performance”, in: *Knowledge and Data Engineering, IEEE Transactions on* 17.1 (2005), pp. 71–89, ISSN: 1041-4347, DOI: 10.1109/TKDE.2005.18.
- [23] Ricco Rakotomalala, “TANAGRA : un logiciel gratuit pour l’enseignement et la recherche”, Fr, in: *Actes de EGC’2005, RNTI-E-3* Vol. 2 (2005), pp. 697–702, URL: <http://eric.univ-lyon2.fr/~ricco/tanagra/fr/tanagra.html>.
- [24] David R Bentley, “Whole-genome re-sequencing”, in: *Current opinion in genetics & development* 16.6 (2006), pp. 545–552.
- [25] Kaushik Datta, Dan Bonachea, and Katherine Yelick, “Titanium performance and potential: an NPB experimental study”, in: *Languages and Compilers for Parallel Computing*, Springer, 2006, pp. 200–214.
- [26] Ingo Mierswa, Michael Wurst, Ralf Klinkenberg, Martin Scholz, and Timm Euler, “Yale: Rapid prototyping for complex data mining tasks”, in: *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, ACM, 2006, pp. 935–940.
- [27] Michael R. Berthold, Nicolas Cebron, Fabian Dill, Thomas R. Gabriel, Tobias Köter, Thorsten Meinl, Peter Ohl, Christoph Sieb, Kilian Thiel, and Bernd Wiswedel, “KNIME: The Konstanz Information Miner”, in: *Studies in Classification, Data Analysis, and Knowledge Organization (GfKL 2007)*, Springer, 2007, ISBN: 978-3-540-78239-1.
- [28] Wu-chun Feng and Kirk W Cameron, “The green500 list: Encouraging sustainable supercomputing”, in: *Computer* 40.12 (2007), pp. 50–55.
- [29] Petr Konecny, “Introducing the Cray XMT”, in: *Proc. Cray User Group meeting (CUG 2007)*, 2007.
- [30] Elke Achtert, Hans-Peter Kriegel, and Arthur Zimek, “ELKI: A Software System for Evaluation of Subspace Clustering Algorithms”, English, in: *Scientific and Statistical Database Management*, ed. by Bertram Ludascher and Nikos Mamoulis, vol. 5069, Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2008, pp. 580–585, ISBN: 978-3-540-69476-2, DOI: 10.1007/978-3-540-69497-7\_41, URL: [http://dx.doi.org/10.1007/978-3-540-69497-7\\_41](http://dx.doi.org/10.1007/978-3-540-69497-7_41).
- [31] Kevin J Barker, Kei Davis, Adolffy Hoisie, Darren J Kerbyson, Mike Lang, Scott Pakin, and Jose C Sancho, “Entering the petaflop era: the architecture and performance of Roadrunner”, in: *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, IEEE Press, 2008, p. 1.

- 
- [32] Leonid Glimcher, Ruoming Jin, and Gagan Agrawal, “Middleware for data mining applications on clusters and grids”, in: *Journal of Parallel and Distributed Computing* 68.1 (2008), Parallel Techniques for Information Extraction, pp. 37–53, ISSN: 0743-7315, DOI: <http://dx.doi.org/10.1016/j.jpdc.2007.06.007>, URL: <http://www.sciencedirect.com/science/article/pii/S0743731507001128>.
- [33] Milind Kulkarni, “The Galois System: optimistic parallelization of irregular programs”, Adviser-Pingali, Keshav, PhD thesis, Ithaca, NY, USA: Cornell University, 2008, ISBN: 978-0-549-96812-2.
- [34] Adam M Phillippy, Michael C Schatz, and Mihai Pop, “Genome assembly forensics: finding the elusive mis-assembly”, in: *Genome biology* 9.3 (2008), p. 1.
- [35] Daniel R Zerbino and Ewan Birney, “Velvet: algorithms for de novo short read assembly using de Bruijn graphs”, in: *Genome research* 18.5 (2008), pp. 821–829.
- [36] Thomas Abeel, Yves Van de Peer, and Yvan Saeys, “Java-ML: A Machine Learning Library”, in: *J. Mach. Learn. Res.* 10 (June 2009), pp. 931–934, ISSN: 1532-4435, URL: <http://dl.acm.org/citation.cfm?id=1577069.1577103>.
- [37] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten, “The WEKA Data Mining Software: An Update”, in: *SIGKDD Explor. Newsl.* 11.1 (Nov. 2009), pp. 10–18, ISSN: 1931-0145, DOI: 10.1145/1656274.1656278, URL: <http://doi.acm.org/10.1145/1656274.1656278>.
- [38] Wei Jiang, V.T. Ravi, and G. Agrawal, “Comparing map-reduce and FREERIDE for data-intensive applications”, in: *Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on*, 2009, pp. 1–10, DOI: 10.1109/CLUSTER.2009.5289199.
- [39] Milind Kulkarni, Martin Burtscher, Rajeshkar Inkulu, Keshav Pingali, and Calin Căscaval, “How Much Parallelism is There in Irregular Applications?”, in: *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '09, Raleigh, NC, USA: ACM, 2009, pp. 3–14, ISBN: 978-1-60558-397-6, DOI: 10.1145/1504176.1504181, URL: <http://doi.acm.org/10.1145/1504176.1504181>.
- [40] Jared T Simpson, Kim Wong, Shaun D Jackman, Jacqueline E Schein, Steven JM Jones, and Inanç Birol, “ABYSS: a parallel assembler for short read sequence data”, in: *Genome research* 19.6 (2009), pp. 1117–1123.
- [41] Sébastien Boisvert, François Laviolette, and Jacques Corbeil, “Ray: simultaneous assembly of reads from a mix of high-throughput sequencing technologies”, in: *Journal of Computational Biology* 17.11 (2010), pp. 1519–1533.

- 
- [42] Ruiqiang Li, Hongmei Zhu, Jue Ruan, Wubin Qian, Xiaodong Fang, Zhongbin Shi, Yingrui Li, Shengting Li, Gao Shan, Karsten Kristiansen, et al., “De novo assembly of human genomes with massively parallel short read sequencing”, in: *Genome research* 20.2 (2010), pp. 265–272.
- [43] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein, “GraphLab: A New Parallel Framework for Machine Learning”, in: *UAI*, 2010.
- [44] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski, “Pregel: A System for Large-scale Graph Processing”, in: *SIGMOD*, Indianapolis, Indiana, USA, 2010, pp. 135–146, ISBN: 978-1-4503-0032-2, DOI: 10.1145/1807167.1807184.
- [45] Richard C Murphy, Kyle B Wheeler, Brian W Barrett, and James A Ang, “Introducing the graph 500”, in: *Cray User’s Group (CUG)* (2010).
- [46] Konrad Paszkiewicz and David J Studholme, “De novo assembly of short sequence reads”, in: *Briefings in bioinformatics* (2010), bbq020.
- [47] Leena Salmela, “Correction of sequencing errors in a mixed set of reads”, in: *Bioinformatics* 26.10 (2010), pp. 1284–1290.
- [48] Vijay Saraswat, George Almasi, Ganesh Bikshandi, Calin Cascaval, David Cunningham, David Grove, Sreedhar Kodali, Igor Peshansky, and Olivier Tardieu, *The Asynchronous Partitioned Global Address Space Model*, tech. rep., Toronto, Canada: IBM, 2010, URL: <http://www.cs.rochester.edu/u/cding/amp/papers/full/The%20Asynchronous%20Partitioned%20Global%20Address%20Space%20Model.pdf>.
- [49] Michael C Schatz, Ben Langmead, and Steven L Salzberg, “Cloud computing and the DNA data race”, in: *Nature biotechnology* 28.7 (2010), p. 691.
- [50] Xiaohong Zhao, Lance E Palmer, Randall Bolanos, Cristian Mircean, Dan Fasulo, and Gayle M Wittenberg, “EDAR: an efficient error detection and removal algorithm for next generation sequencing data”, in: *Journal of computational biology* 17.11 (2010), pp. 1549–1560.
- [51] Amol Ghoting, Prabhanjan Kambadur, Edwin Pednault, and Ramakrishnan Kannan, “NIMBLE: a toolkit for the implementation of parallel data mining and machine learning algorithms on mapreduce”, in: *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, ACM, 2011, pp. 334–342.
- [52] André E Minoche, Juliane C Dohm, and Heinz Himmelbauer, “Evaluation of genomic high-throughput sequencing data generated on Illumina HiSeq and genome analyzer systems”, in: *Genome biology* 12.11 (2011), R112.

- 
- [53] Wenyu Zhang, Jiajia Chen, Yang Yang, Yifei Tang, Jing Shang, and Bairong Shen, “A practical comparison of de novo genome assembly software tools for next-generation sequencing technologies”, in: *PloS one* 6.3 (2011), e17915.
- [54] Sébastien Boisvert, Frédéric Raymond, Élénie Godzaridis, François Laviolette, and Jacques Corbeil, “Ray Meta: scalable de novo metagenome assembly and profiling”, in: *Genome biology* 13.12 (2012), p. 1.
- [55] Miyuru Dayarathna, Charuwat Hounkaew, and Toyotaro Suzumura, “Introducing ScaleGraph: An X10 Library for Billion Scale Graph Analytics”, in: *Proceedings of the 2012 ACM SIGPLAN X10 Workshop*, X10 ’12, Beijing, China: ACM, 2012, 6:1–6:9, ISBN: 978-1-4503-1491-6, DOI: 10.1145/2246056.2246062, URL: <http://doi.acm.org/10.1145/2246056.2246062>.
- [56] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin, “PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs.”, in: *OSDI*, vol. 12, 1, 2012, p. 2.
- [57] Georg Hager, Gerhard Wellein, and SC12 Full-Day Tutorial, “The practitioner’s cookbook for good parallel performance on multi-and manycore systems”, in: (2012).
- [58] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun, “Green-Marl: a DSL for easy and efficient graph analysis”, in: *ACM SIGARCH Computer Architecture News*, vol. 40, 1, ACM, 2012, pp. 349–362.
- [59] Weichun Huang, Leping Li, Jason R Myers, and Gabor T Marth, “ART: a next-generation sequencing read simulator”, in: *Bioinformatics* 28.4 (2012), pp. 593–594.
- [60] Dounia Khaldi, Pierre Jouvelot, Corinne Ancourt, and François Irigoin, “Task parallelism and data distribution: An overview of explicit parallel programming languages”, in: *International Workshop on Languages and Compilers for Parallel Computing*, Springer, 2012, pp. 174–189.
- [61] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M Hellerstein, “Distributed GraphLab: a framework for machine learning and data mining in the cloud”, in: *Proceedings of the VLDB Endowment* 5.8 (2012), pp. 716–727.
- [62] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard, Version 3.0*, HLRS, 2012.
- [63] James Reinders, *An Overview of Programming for Intel Xeon processors and Xeon Phi coprocessors*, tech. rep., Intel Corporation, 2012.
- [64] Janez Demsar and Blaz Zupan, “Orange: Data Mining Fruitful and Fun - A Historical Perspective”, in: *Informatica* 37 (2013), pp. 55–60.



- 
- [65] Benoit Dupont de Dinechin, Pierre Guironnet de Massas, Guillaume Lager, Clement Leger, Benjamin Orgogozo, Jerome Reybert, and Thierry Strudel, “A Distributed Run-Time Environment for the Kalray MPPA-256 Integrated Manycore Processor”, in: *Procedia Computer Science* 18.0 (2013), 2013 International Conference on Computational Science, pp. 1654 –1663, ISSN: 1877-0509, DOI: <http://dx.doi.org/10.1016/j.procs.2013.05.333>, URL: <http://www.sciencedirect.com/science/article/pii/S1877050913004766>.
- [66] B. Elser and A. Montresor, “An evaluation study of BigData frameworks for graph processing”, in: *IEEE Int. Conf. on Big Data*, 2013, pp. 60–67, DOI: 10.1109/BigData.2013.6691555.
- [67] Torsten Hoefer, James Dinan, Darius Buntinas, Pavan Balaji, Brian Barrett, Ron Brightwell, William Gropp, Vivek Kale, and Rajeev Thakur, “MPI+MPI: A New Hybrid Approach to Parallel Programming with MPI Plus Shared Memory”, in: *Computing* 95.12 (2013), pp. 1121–1136.
- [68] Zuhair Khayyat, Karim Awara, Amani Alonazi, Hani Jamjoom, Dan Williams, and Panos Kalnis, “Mizan: a system for dynamic load balancing in large-scale graph processing”, in: *EuroSys*, ACM, 2013, pp. 169–182.
- [69] Sai Charan Koduru, Min Feng, and Rajiv Gupta, “Programming large dynamic data structures on a dsm cluster of multicores”, in: *7th International Conference on PGAS Programming Models*, 2013, p. 126.
- [70] Brent LeBack, Douglas Miles, and Michael Wolfe, “Tesla vs. Xeon Phi vs. Radeon: A Compiler Writer’s Perspective”, in: *in Proc. CUG’13*, 2013, URL: <https://www.pggroup.com/lit/articles/insider/v5n2a1.htm>.
- [71] Yucheng Low, “GraphLab: A Distributed Abstraction for Large Scale Machine Learning”, PhD thesis, University of California, Berkeley, 2013.
- [72] Niranjan Nagarajan and Mihai Pop, “Sequence assembly demystified”, in: *Nature Reviews Genetics* 14.3 (2013), pp. 157–167.
- [73] Semih Salihoglu and Jennifer Widom, “Gps: A graph processing system”, in: *SS-DBM*, ACM, 2013, p. 22.
- [74] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson, “From "think like a vertex" to "think like a graph"”, in: *Proc. of the VLDB Endowment* 7.3 (2013), pp. 193–204.
- [75] Ahmed Barnawi, Omar Batarfi, Radwa Elshawy, Ayman Fayoumi, Reza Nouri, Sherif Sakr, et al., “On Characterizing the Performance of Distributed Graph Computation Platforms”, in: *Performance Characterization and Benchmarking. Traditional to Big Data*, Springer, 2014, pp. 29–43.

- 
- [76] Barnaby Dalton, Gabriel Tanase, Michail Alvanos, George Almasi, and Ettore Tiotto, “Memory Management Techniques for Exploiting RDMA in PGAS Languages”, in: *The 27th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, 2014.
  - [77] Tobias Fleig, Oliver Mattes, and Wolfgang Karl, “Evaluation of Adaptive Memory Management Techniques on the Tilera TILE-Gx Platform”, in: *In Proc. Workshop ARCS’14*, 2014.
  - [78] Yong Guo, Marcin Biczak, A Varbanescu, Alexandru Iosup, Claudio Martella, and T Willke, “How well do graph-processing platforms perform? an empirical performance evaluation and analysis”, in: *IPDPS*, IEEE, 2014, pp. 395–404.
  - [79] Minyang Han, Khuzaima Daudjee, Khaled Ammar, M Ozsu, Xingfang Wang, and Tianqi Jin, “An experimental comparison of pregel-like graph processing systems”, in: *Proc. of the VLDB Endowment* 7.12 (2014), pp. 1047–1058.
  - [80] Mimi Lau, “China’s world-beating supercomputer fails to impress some potential clients” <http://www.scmp.com/news/china/article/1543226/chinas-world-beating-supercomputer-fails-impress-some-potential-clients>, English, South China Morning Post, 2014, URL: {<http://www.scmp.com/news/china/article/1543226/chinas-world-beating-supercomputer-fails-impress-some-potential-clients>}.
  - [81] Jure Leskovec and Andrej Krevl, *SNAP Datasets: Stanford Large Network Dataset Collection*, <http://snap.stanford.edu/data>, June 2014.
  - [82] Xiangke Liao, Liquan Xiao, Canqun Yang, and Yutong Lu, “MilkyWay-2 supercomputer: system and application”, in: *Frontiers of Computer Science* 8.3 (2014), pp. 345–356.
  - [83] Alessandro Lulli, “Distributed solutions for large scale graph processing”, in: (2014).
  - [84] Raghunath Nambiar, Meikel Poess, Akon Dey, Paul Cao, Tariq Magdon-Ismail, Andrew Bond, et al., “Introducing TPCx-HS The First Industry Standard for Benchmarking Big Data Systems”, in: *Performance Characterization and Benchmarking. Traditional to Big Data*, Springer, 2014, pp. 1–12.
  - [85] Olivier Serres, Abdullah Kayi, Ahmad Anbar, and Tarek El-Ghazawi, “Hardware Support for Address Mapping in PGAS Languages: A UPC Case Study”, in: *Proceedings of the 11th ACM Conference on Computing Frontiers*, CF ’14, Cagliari, Italy: ACM, 2014, 22:1–22:2, ISBN: 978-1-4503-2870-8, DOI: 10.1145/2597917.2597945, URL: <http://doi.acm.org/10.1145/2597917.2597945>.
  - [86] Daniel P Siewiorek and Philip John Koopman, *The architecture of supercomputers: Titan, a case study*, Academic Press, 2014.

- 
- [87] David Sims, Ian Sudbery, Nicholas E Illott, Andreas Heger, and Chris P Ponting, “Sequencing depth and coverage: key considerations in genomic analyses”, in: *Nature Reviews Genetics* 15.2 (2014), pp. 121–132.
- [88] Anish Varghese, Bob Edwards, Gaurav Mitra, and Alistair P. Rendell, “Programming the Adaptea Epiphany 64-Core Network-on-Chip Coprocessor”, in: *Proceedings of the 2014 IEEE International Parallel & Distributed Processing Symposium Workshops*, IPDPSW ’14, Washington, DC, USA: IEEE Computer Society, 2014, pp. 984–992, ISBN: 978-1-4799-4116-2, DOI: 10.1109/IPDPSW.2014.112, URL: <http://dx.doi.org/10.1109/IPDPSW.2014.112>.
- [89] Yue Zhao, Kenji Yoshigoe, Mengjun Xie, Suijian Zhou, Remzi Seker, and Jiang Bian, “Evaluation and Analysis of Distributed Graph-Parallel Processing Frameworks”, in: *JCSM* 3.3 (2014), pp. 289–316.
- [90] Anas Abu-Doleh and Ümit V Çatalyürek, “Spaler: Spark and GraphX based de novo genome assembler”, in: *Big Data (Big Data), 2015 IEEE International Conference on*, IEEE, 2015, pp. 1013–1018.
- [91] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen, “Powerlyra: Differentiated graph computation and partitioning on skewed graphs”, in: *ACM EuroSys*, 2015.
- [92] B. D. de Dinechin, “Kalray MPPA: Massively parallel processor array: Revisiting DSP acceleration with the Kalray MPPA Manycore processor”, in: *2015 IEEE Hot Chips 27 Symposium (HCS)*, 2015, pp. 1–27, DOI: 10.1109/HOTCHIPS.2015.7477332.
- [93] Yun Gao, Wei Zhou, Jizhong Han, Dan Meng, Zhang Zhang, and Zhiyong Xu, “An evaluation and analysis of graph processing frameworks on five key issues”, in: *Proc. of the 12th ACM Int. Conf. on Computing Frontiers*, ACM, 2015, p. 11.
- [94] Philip Leonard, “Exploring Graph Colouring Heuristics in GraphLab”, in: (2015).
- [95] Robert Ryan McCune, Tim Weninger, and Greg Madey, “Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing”, in: *ACM Computing Surveys* 48.2 (2015), p. 25.
- [96] Orange, <http://orange.biolab.si/>, *Data-mining - Fruitful and Fun*, 2015, URL: <https://orange.biolab.si/>.
- [97] OVH, *OVH HPC server offers* <https://www.ovh.com/fr/hpc/spot.xml>, French, OVH, 2015, URL: <https://www.ovh.com/fr/hpc/spot.xml>.
- [98] PRACE Research Infrastructure, <http://www.prace-ri.eu>, Eng. PRACE, 2015, URL: <http://www.prace-ri.eu>.
- [99] Torch7, <http://torch.ch/>, *Torch, A Scientific computing framework for LuaJIT*, 2015, URL: <http://torch.ch/>.

- 
- [101] Haohuan Fu, Junfeng Liao, Jinzhe Yang, Lanning Wang, Zhenya Song, Xiaomeng Huang, Chao Yang, Wei Xue, Fangfang Liu, Fangli Qiao, Wei Zhao, Xunqiang Yin, Chaofeng Hou, Chenglong Zhang, Wei Ge, Jian Zhang, Yangang Wang, Chunbo Zhou, and Guangwen Yang, “The Sunway TaihuLight supercomputer: system and applications”, in: *Science China Information Sciences* 59.7 (2016), p. 072001, ISSN: 1869-1919, DOI: 10.1007/s11432-016-5588-7, URL: <https://doi.org/10.1007/s11432-016-5588-7>.
- [102] Leena Salmela, Riku Walve, Eric Rivals, and Esko Ukkonen, “Accurate selfcorrection of errors in long reads using de Bruijn graphs”, in: *Bioinformatics* (2016), btw321.
- [103] Bull, *Supernode bullx S6000* <http://www.bull.com/fr/supernode-bullx-s6000>, French, Bull, 2017, URL: <http://www.bull.com/fr/supernode-bullx-s6000>.
- [105] DELL, *DELL PowerEdge RackServers*, 2017, URL: {<http://www.dell.com/en-us/work/shop/franchise/poweredge-rack-servers>}.
- [106] Kalray, *Kalray MPPA-256 Bostan*, French, Bull, 2017, URL: <http://www.kalrayinc.com/kalray/products>.
- [107] NVIDIA, *Jetson TX2 module datasheet 1.1*, tech. rep., 2017, URL: <https://developer.nvidia.com/embedded/dlc/jetson-tx2-module-data-sheet-1-1>.
- [108] Intel Corporation, *Core i5-4430 processor’s page on Intel website*, URL: [http://ark.intel.com/products/75036/Intel-Core-i5-4430-Processor-6M-Cache-up-to-3\\_20-GHz](http://ark.intel.com/products/75036/Intel-Core-i5-4430-Processor-6M-Cache-up-to-3_20-GHz).
- [109] Intel Corporation, *Core2 Extreme X6800 processor’s page on Intel website*, URL: [http://ark.intel.com/products/27258/Intel-Core2-Extreme-Processor-X6800-4M-Cache-2\\_93-GHz-1066-MHz-FSB](http://ark.intel.com/products/27258/Intel-Core2-Extreme-Processor-X6800-4M-Cache-2_93-GHz-1066-MHz-FSB).
- [110] Intel Corporation, *Xeon E5-2640 v3 processor’s page on Intel website*, URL: [https://ark.intel.com/products/83359/Intel-Xeon-Processor-E5-2640-v3-20M-Cache-2\\_60-GHz](https://ark.intel.com/products/83359/Intel-Xeon-Processor-E5-2640-v3-20M-Cache-2_60-GHz).
- [111] Intel Corporation, *Xeon-Phi Coprocessors’s page on Intel website*, URL: <http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html>.
- [112] Passmark Software, *CPUbenchmark website*, <https://www.cpubenchmark.net/>, URL: <https://www.cpubenchmark.net/>.
- [113] Richard Xia and Albert Kim, “MERmaid: A Parallel Genome Assembler for the Cloud”, in: ().