Chapter 2

PROGRAMMING MODELS AND RUNTIMES

Georges Da Costa[1] and Alexey L. Lastovetsky[2] and Jorge G. Barbosa[3] and Juan C. Díaz-Martín[4] and Juan L. García-Zapata[5] and Matthias Janetschek[6] and Emmanuel Jeannot[7] and João Leitão[8] and Ravi Reddy Manumachu[9] and Radu Prodan[10] and Juan A. Rico-Gallego[11] and Peter Van Roy[12] and Ali Shoker[13] and Albert van der Linde[14]

Keywords: Performance Modelling, Energy Modelling, Heterogeneous Platforms, Optimization Techniques, Cloud Computing; Edge Computing; Process Placement; Graph Partition; Sustainability; Energy awareness; Availability; Scalability.

Several millions of execution flows will be executed in Ultrascale Computing Systems (UCS), and the task for the programmer to understand their coherency and for the runtime to coordinate them is unfathomable. Moreover, in link with USC large scale and their impact on reliability the current static point of view is not more sufficient. A runtime cannot consider to restart an application because of the failure of a single node as statically several nodes will fail every days. Classical management of these failures by the programmers using checkpoint-restart are also too limited due to the overhead at such scale.

[1]University of Toulouse, France
[2]University College Dublin, Ireland
[3]Faculdade de Engenharia da Universidade do Porto, Portugal
[4]University of Extremadura, School of Technology, Spain
[5]University of Extremadura, School of Technology, Spain
[6]Institute of Computer Science, University of Innsbruck, Austria
[7]INRIA Bourdeaux Sud-Ouest, LaBRI, Université de Bourdeaux, France
[8]Universidade Nova de Lisboa, Portugal
[9]University College Dublin, Ireland
[10]Institute of Information Technology, University of Klagenfurt, Austria
[11]University of Extremadura, School of Technology, Spain
[12]Université Catholique de Louvain, Belgium
[13]HASLab, INESC TEC & University of Minho, Portugal
[14]Universidade Nova de Lisboa, Portugal

Emerging programming models that facilitate the task of scaling and extracting performance on continuous evolving platforms, while providing resilience and fault tolerant mechanisms to tackle the increasing probability of failures throughout the whole software stack, are needed to achieve scale handling (optimal usage of resources, faults), improve programmability, adaptation to rapidly changing underlying computing architecture, data-centric programming models, resilience, energy efficiency.

One key element on the ultrascale front is the necessity of new sustainable programming and execution models in the context of rapid underlying computing architecture changing. There is a need to explore synergies among emerging programming models and runtimes from HPC, distributed systems and big data management communities. To improve the programmability of future systems, the main changing factor will be the substantially higher levels of concurrency, asynchrony, failures and heterogeneous architectures.

UCS need new sustainable programming and execution models, suitable in the context of rapidly changing underlying computing architecture, as described in [1]. Advances are to be expected at three levels: Innovative programming models with higher level abstraction of the hardware; breakthrough for more efficient runtimes at large scale; cooperation between the programming models and runtime levels.

Furthermore all the programming ecosystem must evolve. A large number of scientific applications are built on the message passing paradigm which needs a global point of view during the programming phase and usually require global synchronization during execution. But even at lower granularity, classical libraries must evolve. As an example, a large number of scientists use the linear algebra BLAS libraries for their optimized behavior on current supercomputers. Improving the performance of this library on UCS would prove largely beneficial.

This chapter explores programming models and runtimes required to facilitate the task of scaling and extracting performance on continuously evolving platforms, while providing resilience and fault-tolerant mechanisms to tackle the increasing probability of failures throughout the whole software stack. However, currently no programming solution exists that satisfies all these requirements. Therefore, new programming models and languages are required towards this direction. The whole point of view on application will have to change. As we will show, the current wall between runtime and application models leads to most of these problem. Programmers will need new tools but also new way to assess their programs. Also, data will be a key concept around which failure-tolerant high number of micro-threads will be generated using high-level information by adaptive runtime. One complex element comes from the difficulty to test these approaches as UCS systems are not yet available. Most of the following explorations are extrapolated to UCS scale but only actually proven an currently existing infrastructure.

The complexity of UCS computing architecture integrating in a hierarchical heterogeneous way multicore CPUs and various accelerators makes many traditional approaches to the development of performance and energy efficient applications ineffective. New sustainable approaches based on accurate and sustainable application-level performance and energy models have a great potential to improve

the performance and energy efficiency of applications and create a solid basis for the emerging USC programming tools and runtimes. Section 2.1 of this chapter covers this topic by describing accurate models of the hardware and software usable during the design phase, but also means or reasoning on these models. With these tools, it becomes possible to adapt and tune finely applications during the design phase to run efficiently on large scale heterogeneous platforms.

Optimizing UCS usage is difficult due to the large number of possible use-cases. In particular ones such as Scientific workflow, it becomes possible to use a dedicated abstraction. As scientific workflow scheduling for UCS is a major challenge, the impact of proposing a particular abstraction along-with dedicated runtime harnessing the particularities of this abstraction leads to a high improvement of the efficiency of using a UCS. The approaches to solve this challenge are covered in section 2.2. In this section, both the *Abstract part* (linked with the design and programming of the workflow) and the *Concrete part* (linked with its actual scheduling and execution) are described. This specific high-level abstraction shows that link between programming models and runtime helps to simplify the task of programmers to harness the power of the underlying large scale heterogeneous systems.

With the emergence of UCS, a new computing revolution is coming: Edge computing. Instead of harnessing computing power directly from large scale datacenters, new proposal comes from the possibility to interconnect and coordinate large number of distributed computing nodes. Due to the explosion of IoT applications the aggregated Edge computing power is increasing extremely fast. These two systems (Edge and UCS) share the difficulty to manage large number of distributed execution flows in a dynamic and heterogeneous environment. These similarities is explored in Section 2.3 where key elements of programming models and runtime for large scale Edge computing are explored.

Due to the scale of UCS, even classical management operation of the platform becomes complex. As an example, section 2.5 shows how a simple operation such as graph partitioning becomes complex at large scale. This operation is central in the management of a platform as it is needed to minimize communication between nodes when used for placing the tasks. In this section several challenges are addressed such as the scale but also the heterogeneity of tasks, computing nodes and networking infrastructure.

This chapter concludes with a description of the main global challenges linked to programming models, runtimes and the link between these two as described in NESUS roadmap[2].

2.1 Using Performance and Energy Models for Ultrascale Computing Applications

Ultrascale systems, including high performance computing, distributed computing and big data management platforms, will demand a huge investment of heterogeneous

computing and communication equipment. Ensuring the availability of current and future social, enterprise and scientific applications with efficient and reliable execution on these platforms remains nowadays an outstanding challenge. Indeed, reducing their power footprint while still achieving high performance has been identified as a leading constraint in this goal. Model-driven design and development of optimal software solutions should play a critical role in that respect.

Energy consumption is one of the main limiting factors for designing and deploying ultrascale systems.

Using monitoring and measurement data, Performance and Energy models contribute to quantify and gain insights into the performance and power consumption effects of system components and their interactions, including both hardware and the full software stack. Analysis of the information provided by the models is then used for tunning applications and predicting its behavior under different conditions, mainly at scale.

This chapter describes methods, facilities and tools for building performance and energy models, with the goal of aiding in the design, development and tunning of data-parallel and task-parallel applications running on complex heterogeneous parallel platforms.

2.1.1 Terminology

In this section, we describe the various terms related to power and energy predictive models used in this work.

There are two types of power consumptions in a component: dynamic power and static power . Dynamic power consumption is caused by the switching activity in the component's circuits. Static power is the power consumed when the component is not active or doing work. Static power is also known as idle power or base power. From an application point of view, we define dynamic and static power consumption as the power consumption of the whole system with and without the given application execution respectively. From the component point of view, we define dynamic and static power consumption of the component as the power consumption of the component with and without the given application utilizing the component during its execution respectively.

There are two types of energy consumptions, static energy and dynamic energy. We define the static energy consumption as the energy consumption of the platform without the given application execution. Dynamic energy consumption is calculated by subtracting this static energy consumption from the total energy consumption of the platform during the given application execution. That is, if P_S is the static power consumption of the platform, E_T is the total energy consumption of the platform during the execution of an application, which takes T_E seconds, then the dynamic energy E_D can be calculated as,

$$E_D = E_T - (P_S \times T_E) \tag{2.1}$$

2.1.2 Performance Models of Computation

In this section, we survey prominent models used for prediction of the cost of computations in the execution of ultrascale computing applications.

The seminal models are the parallel random access machine (PRAM) [3], the bulk-synchronous parallel model (BSP) [4], and the LogP model [5]. All these models assume a parallel computer to be a homogeneous multiprocessor.

The PRAM is the most simplistic parallel computational model . It consists of p sequential processors sharing a global memory. It assumes that synchronization and communication is essentially cost free. However, these overheads can significantly affect algorithm performance. Many modifications to the PRAM have been proposed that attempt to bring it closer to practical parallel computers.

The BSP model is a bridging model that consists of p parallel/memory modules, a communication network, and a mechanism for efficient barrier synchronization of all the processors. A computation consists of a sequence of supersteps. During a superstep, each processor performs synchronously some combination of local computation, message transmissions, and message arrivals.

Finally, LogP (covered later in much detail) abstracts the performance of a system with four parameters, L, o, g, and P, which stand for network delay, overhead or cycles that a CPU devotes to sending the message, gap per message or minimum time interval between two consecutive injections to the network, and, finally, number of processes. It has been successfully used for developing fast and portable parallel algorithms for (homogeneous) supercomputers and has become a foundation for numerous subsequent models.

A dominant class models parallel computation by Directed Acyclic Graph (DAG) where the nodes represent local computation and the edges signify the data dependencies. This model forms the fundamental building block of runtime schedulers in KAAPI [6], StarPU [7], and DAGuE [8].

Graphical models are commonly used to structure mesh-based scientific computations. The objective of a graph partitioning problem is then to divide the vertices of the graph into approximately equal-weight partitions (balance computations) and minimize the number of cut edges between partitions (minimize total runtime communication) [9], [10], [11], [12].

We will now review performance models of computation for heterogeneous platforms where they are even more paramount.

Performance Models of Computation for Heterogeneous HPC Platforms

Realistic and accurate performance models of computation are the fundamental building blocks of data partitioning algorithms. Over the years, load balancing algorithms developed for performance optimization on parallel platforms have attempted to take into consideration the real-life behavior of applications executing on these platforms. This can be discerned from the evolution of performance models for computation used in these algorithms.

The simplest models used positive constant numbers and different notions such as normalized processor speed, normalized cycle time, task computation time, average execution time, etc to characterize the speed of an application [13], [14], [15]. A

common crucial feature of these efforts is that the performance of a processor is assumed to have no dependence on the size of the workload.

The most advanced load balancing algorithms use functional performance models (FPMs) , which are application-specific and represent the speed of a processor by continuous function of problem size but satisfying some assumptions on its shape [16],[17],[18],[19]. These FPMs capture accurately the real-life behaviour of applications executing on nodes consisting of uniprocessors (single-core CPUs).

Modern multicore platforms have complex nodal architectures with highly hierarchical arrangement and tight integration of processors where resource contention and Non-uniform Memory Access (NUMA) are inherent complexities. On these platforms, load balancing algorithms based on the traditional and state-of-the-art performance models (FPMs) will return sub-optimal solutions due to the complex nature of the performance models. Therefore, there is a need for novel performance models of computation that take into account these inherent complexities.

Lastovetsky et al. [20], [21] present an advanced performance model of computation (FPMs) that contains severe variations reflecting the resource contention and NUMA inherent in the modern multicore platforms. These models (or performance profiles) have complex shapes (non-linear, non-convex), which do not satisfy the assumptions on shape that allow load balancing algorithms based on smooth FPMs to return optimal workload distribution. The authors then propose data partitioning algorithms that use these advanced FPMs as building blocks to minimize the computation time of the parallel application.

2.1.3 Performance Models of Communications

This section fairly describes the issue of optimizing communication using analytical representations of the transmissions departing from a given workload balance of the computation between the processes of an application. We also introduce foundational analytical communication performance models and we apply one of the models to an example of a real-world kernel.

Ultra-scale Computer Systems are composed of heterogeneous multi-core processors and accelerators, connected by a hierarchy of communication channels. Such heterogeneity is partially due to the necessity of increasing the system performance keeping the energy cost at a reasonable level. Scientific applications executing on UCS platforms are composed of *kernels*, computationally intensive tasks conceived for being executed by a set of heterogeneous processors. Usually, every processor runs the same code on a different *data region* of a global data space. UCS applications face the challenge of obtaining as much performance as possible from the specific platform.

During execution of a kernel, each of the processes needs data from other processes to compute its own values. Therefore, the necessity of communication appears periodically during its execution. The challenge is not only to balance the overall computational load of the kernel among the available computing resources, but also to optimize the completion time of its communications.

Current approach is based on design and implement evaluation tests, execute them in the target platform, hence consuming computational resources along a signifi-

cant amount of time, and extrapolate estimations obtained to the whole application. A model-based methodology replaces the previous test-based approach by a fully analytical modeling of the behavior of the application. Optimization of computation and communication in data parallel applications are usually addressed separately. First, computational load is distributed between processors according to their capabilities, following different approaches (see section 2.1.2). Then, communication optimization is addressed by building communication performance models and applying them for searching a distribution of the data space to the processes that reduces the communication cost.

A communication performance model provides with an analytic framework that represents communications as a parameterized formal expression. The evaluation of this expression determines the cost of the communication in terms of time, as function of system parameters. Many models have been proposed, covering different aspects of the communication. They can be generally classified in two types: *hardware models* and *software models*. Following, we introduce some of the representative models of each type.

Hardware models use hardware related parameters to build the analytical expression representing the cost of communications. LogP [5] is a foundational model representing the cost of a communication by four parameters: L is the *network delay*, and represent the latency of the network, o is the *overhead* or cycles that a CPU devotes to send the message, g is the *gap per message* and represents the minimum time interval between two consecutive injections to the network, and, finally, P is the number of processes. LogP model was improved by LogGP [22], which includes a new parameter G (gap per message) allowing to represent the influence of the network bandwidth in the transmission of large messages. In LogGP, the cost of a point-to-point transmission of a message of size m is represented as: $T_{p2p}(m) = 2\,o + L + (m-1)\,G$. More advanced models have been proposed, as PLogP [23], that considers parameters gap per message and overhead linear functions of the message size, achieving higher accuracy. Derived models have been proposed to represent communication costs in heterogeneous platforms, by extending previous purely homogeneous models with additional parameters representing specific features of the platform, as HLogGP [24] and LMO [25].

Software models address the modeling of the *middleware* costs of a communication. They abstract from hardware and use middleware-related parameters to build analytical expressions representing the costs associated to data movement. log_nP [26] considers a point-to-point transmission as a sequence of *transfers* (copies) through intermediate buffers between the endpoints of a homogeneous platform. The aggregation of the costs of the individual transfers yields the cost of the transmission in an expression as: $T(m) = \sum_{i=0}^{n-1} (o_i + l_i)$, where o (*overhead*) is the per transfer time dedicated by the CPU to a contiguous message, and the *latency* l is the additional cost if the message is non-contiguous in memory. τ-*Lop* model [27, 28] addresses the challenge of accurately modeling MPI communications on heterogeneous ultra scale platforms. It relies on the concept of *Concurrent Transfers* of data, and uses this concept as a building block to represent the communications on hierarchical communication channels, capturing the impact of contention and process mapping. The cost of

a point-to-point message transmission is modeled using two parameters: the *overhead* $o^c(m)$ represents the time needed to start the injection of data in the communication channel c from the invocation of the operation, and the *transfer time* $L^c(m,\tau)$ is the time invested in each one of the data movements composing the transmission, and depends on the message size and the number of concurrent transfers progressing through the channel c. The parameter τ allows to the model to represent the cost derived from contention, and hence the channel bandwidth sharing, appearing naturally in collective and kernel communications. The τ-*Lop* expression describing the cost of a message transmission in n equal transfers is $T^c(m) = o^c(m) + n \times L^c(m,1)$. To represent the cost in complex heterogeneous platforms, τ-*Lop* adopts a compositional approach for representing the concurrency of full point-to-point transmissions, by using the concurrency operator $||$. As an example, the cost of the pair of concurrent transmissions is represented as $T^c(m) \;||\; T^c(m) = 2 \;||\; T^c(m) = o^c(m) + n \times L^c(m,2)$. Note how the amount of concurrent transmissions represented using the concurrency operator is propagated to the τ parameter of the transfers.

Using analytical models to optimize performance of complex heterogeneous kernels requires a high level of accuracy in the predictions and enough representation capabilities for the high amount of convoluted communications of the processes. Accuracy has to do with the representation of the cost, but also with the parameter measurement in the specific platform. A methodology for measuring the parameter values that captures the parameter meaning is essential for achieving accurate predictions of the communication cost.

Following, we develop an example of a simple communication optimization for a real data parallel kernel. The kernel (named *Wave2D*) uses the technique of finite differences to numerically solve the following wave equation in a $N \times N$ data space:

$$\frac{\partial^2 u}{\partial t^2} = c^2 \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) \tag{2.2}$$

Along time t, $u(x,y,t+1)$ is generated from its previous instances $u(x,y,t)$ and $u(x,y,t-1)$. The left side of Fig. 5 shows this matrix at a given step of the algorithm.

The communication optimization procedure departs from a previously established process distribution to the resources of the platform, involving multicore CPUs and accelerators. The first step is to balance the computational load between the processors. In a heterogeneous platform the processors have different computing capabilities, therefore, this step involves the characterization of the speeds of the processes by a vector $s = \{s_0, \ldots, s_{P-1}\}$, and the assignation to p_i of an amount of data proportional to its speed s_i. Usually, such speed characterization is done through benchmarking, that outputs a speed number per process, or a function describing the speed as a function of the task size (see section 2.1.2).

Regions of data distributed to the processes must tile the entire data space. Partitioning and distributing the data space in P regions of sizes proportional to s is subject to multiple variations, called *data mappings* . Alternatives data mappings can be evaluated to choose that which minimizes the communication cost. Note that, for the set of possible data mappings, every process performs the same amount of computational work on a different set of data points, and hence, the workload

balance does not change, but does the communication cost. An example of a data mapping is shown at the right part of Figure 2.1. It represents the kernel running in an experimental platform composed of two nodes identified by a background color. The $P = 8$ processes communicate through shared memory or network depending on their location. Inside each node, each process may run on a set of assigned resources of different type. *FuPerMod* tool [29] was used to provide a load-balanced partition following a column-based approach [30]. Partitioning algorithms do not take into account the communication cost of the kernel, but only the relative speed of the processes. In this example, we use τ-*Lop* analytical framework to find a more efficient data mapping in terms of its communication costs.

In homogeneous systems, models of point-to-point and collective operations basically contains expressions in the forms $n \,||\, T^c(m)$ representing the cost of n concurrent transmissions of a message of size m through a communication channel c, and $T^c(m_1) \,||\, T^c(m_2)$, representing the cost of a sequence of two transmissions of different message sizes through the same communication channel. Communication models in heterogeneous systems become more complex. τ-*Lop* provides with extensions to evaluate these types of complex expressions [28] which shuffle concurrent and sequential transmissions of different message lengths progressing through the same or different communication channel, e.g. $T^{c_1}(m_1) \,||\, T^{c_2}(m_2)$. Anyway, expressions of actual kernels rapidly become complex enough to require an automatic evaluation. The τ-*Lop* toolbox[15] is a package that provides with a C++ function interface to describe and automatically evaluate the communication cost expressions of a data parallel kernel. Their inputs are the τ-*Lop* parameters built for the platform and a description of the data mapping and the kernel communications, both point-to-point and collectives. The toolbox provides with facilities to provide such description and to efficiently evaluate its communication cost. It allows to evaluate efficiently a set of partitions, leading to an optimal election.

[15] http://hpc.unex.es/taulop

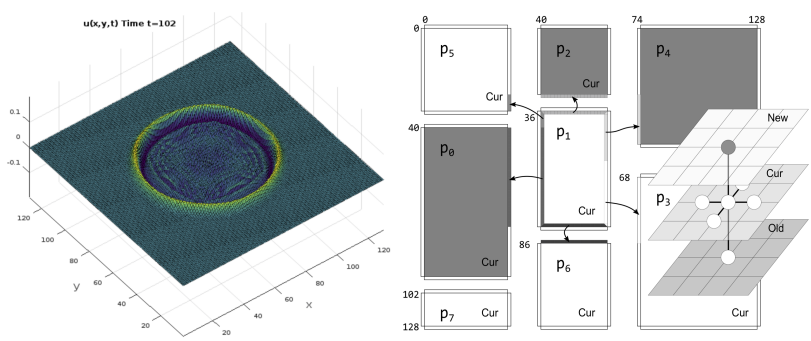


Figure 2.1: Left: visualization of discrete solution $u(x,y,t)$ of a wave equation in a $N \times N$ data mesh with $N = 128$, at time $t = 102$, for particular initial and boundary conditions. Right: an example data space partition and distribution to $P = 8$ processes with different computational capabilities running in two nodes (background color).

Algorithm 1 Code for evaluating the communication cost of the Wave2D kernel in a heterogeneous platform.

```
int P = 8;
int nodes = {0, 1, 0, 1, 0, 1, 1, 1}; // Node mapping
Process *p[P];
int *η[P];
for rank in {0, P-1}:
p[rank] = new Process (rank, nodes[rank]);
for rank in {0, P-1}:
η[rank] = new Neighbors (p);
TauLopConcurrent *conc = new TauLopConcurrent ();
for rank in {0, P-1}:
TauLopSequence *seq = new TauLopSequence ();
for dst in {η[rank]}:
m = getMsgSize (p, dst) * sizeof(double);
seq→add (new Transmission (p[rank], p[dst], m));
conc→add(seq);
TauLopCost *tc = new TauLopCost ();
conc→evaluate (tc);
double t = tc→getTime ();
```

As shown in the right part of Figure 2.1, at each time $t+1$, every data point in matrix *New* is calculated as a combination of the neighbor points in matrix *Cur*, which requires a previous communication stage of the needed data from neighbor processes at step t. Such communications are represented in the figure for process p_1. As the computation is (unevenly) load balanced, all processes come into the communication phase at the same time. Hence, all processes interchange their boundaries simultaneously. From this assumption, we can derive a communication cost expression of the kernel:

$$\Theta = t \times \left[\mathop{||}_{p=0}^{P-1} \Theta_p \right], \; with \; \Theta_p = \sum_{i \in \eta_p} T^{c(i)}(m(i)). \tag{2.3}$$

All of the processes communicate concurrently, so the total cost Θ is calculated using the concurrency operator $||$ for every process communication over t steps. A process p transmits its boundary data to its neighbor processes (the set η_p) using the channel $c(i)$ for transmitting the message of size $m(i)$ to the neighbor i. The transmissions of a process to its neighbors are accomplished sequentially, hence the sum.

Extending previous cost expression to every individual cost transmission is indeed complex enough to require evaluation using an automatic tool. Code 1 uses τ-*Lop* toolbox to describe and evaluate the previous cost model representing the communications and data mapping of the kernel. Array `node` represents the mapping of processes to nodes, numbered 0 and 1. Following, an array of processes is created, with the rank number and mapping node of each `Process`. Then, `Neighbors()` function is used to create the neighbor set of each process (η_p). Neighbor sets are

specific for a given arrangement of the rectangles in the data space (data mapping) and determine the destination and amount of data transmitted through different communication channels, and, as a consequence, the final cost. As an example, Figure 2.2 shows two partitions with different communication costs. At the left figure the number of data points in process p_5 boundaries for transmitting through the network to processes in $\eta_5 = \{0,1,2\}$ is 76, while at the right figure, with $\eta'_5 = \{0,3,7\}$, the number is 40, reducing the transmissions through the slower network communication channel, and hence, the cost. The rest of the code composes and evaluates the cost for the specific partition. The cost expression is composed using the `TauLopConcurrent` and `TauLopSequence` objects. All Transmissions added to a `TauLopSequence` object will be evaluated under the assumption that they progress sequentially. Then, `TauLopSequence` objects added to a `TauLopConcurrent` object will be evaluated under the assumption that they progress concurrently, applying the transfer time parameter values for specific m and τ. The communication channel used for each transmission is internally figured out from the node location obtained from the processes. Finally, `TauLopCost` object evaluates the cost expression and returns a time in seconds.

By executing the algorithm using all possible data mappings, the optimum arrangement is obtained. Actually, this procedure is unfeasible when the number of processes grows, because the number of combinations grows exponentially. In practice, only a reduced set of possible data mappings is evaluated. A straightforward heuristic-based optimization decision for Wave2D, proposed by Malik et al [31], is based on the rearrangement of the regions assigned to processes running on the same node to be as close as possible, which decreases the network communication, more expensive in terms of time.

2.1.4 Power and Energy Models of Computation

In this section, we will survey research works that have proposed power and energy predictive models for optimization of applications for energy on ultrascale systems.

There are several ways to classify power and energy predictive models for ultrascale computing systems and applications.

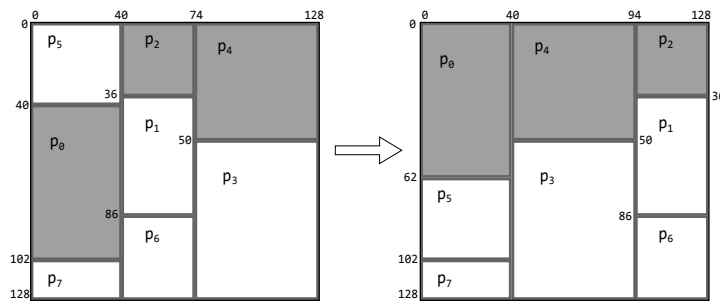


Figure 2.2: Rearranging the data regions assigned to processes in the 2D mesh data space in such a way that network transmissions have been minimized.

First classification is based on three dominant approaches used for modeling power and energy consumption.

- ***System-level***: The approach is to use system-level physical measurements using power meters.
- ***On-chip sensors***: On-chip sensors supplied by the vendors and their APIs are used for obtaining the power and energy consumptions.
- ***Performance Monitoring Events (PMCs)***: This dominant approach uses the PMCs provided by a vendor as parameters for the models.

Second classification is based on the characteristic *Level of abstraction*, which specifies how the model captures the inherent hierarchical and heterogeneous nature of modern processor architectures.

- ***Linear Independence*** - All the components of a node are modeled independently. The model for a node is a linear combination of the models of its components.
- ***Linear Dependence*** - The components of a node are modeled taking into account the dependencies (shared structures) between them and expressing these dependencies linearly. For example:
  - The models for CPUs are constructed taking into account the shared resources (Last level cache) between them.
  - For an application employing both CPUs and accelerators, the models for CPUs and the accelerators are constructed taking into account the shared resources (last level cache) between the CPUs and the communication link (PCIe) connecting the CPUs and the accelerators.
- ***Non-linear Independence*** - All the components of a node are modeled independently. However, the model for a node is a non-linear combination of the models of its components.
- ***Non-linear Dependence*** - This is the most complex model. The components and dependencies between them (shared resources, communication links) are modeled non-linearly by taking into account their inherent hierarchical and heterogeneous nature.

From our survey, almost all the models fall into the category of *linear independence*.

However, we divide our survey into categories using the following more readable classification: a). Models for CPUs, b). Models for GPUs, c). Models for Xeon Phis and FPGAs, d). Application-specific models, and e). Critiques of PMC-based models.

Owing to length constraints, we will look at only the most prominent works in each category.

Power and Energy Models for CPUs. The first notable model in this category is [32], which is based on events such as integer operations, floating-point operations, memory requests due to cache misses, etc. that the authors believed to strongly correlate with power consumption. Icsi et al. [33] propose a methodology to determine unit-level power estimates based on hardware performance counters. They select 22 strictly collocated physical units based on an annotated P4 die photo. The total power consumption is then estimated as the sum of the power consumptions of the

22 physical units plus the base power. The power estimate for each unit is a linear function of the access rate of it, with the exception of few issue logic units where an extra parameter is introduced to model the non-linear behavior.

Several models employed as predictor variables utilization metrics of the key components such as CPU, memory, disk, and network. The most comprehensive model in this group is proposed in [34] that used as parameters, the utilization metrics of CPU, disk, and network components and hardware performance counters for memory. Here, the general model can be described as follows:

$$P = C_{base} + C_1 \times U_{CPU} + C_2 \times U_{Mem} + C_3 \times U_{Disk} + C_4 \times U_{Net} \tag{2.4}$$

where C_{base} is the base power consumption of a node and U_{CPU}, U_{Mem}, U_{Disk}, and U_{Net} are the CPU, memory, disk, and network utilizations respectively.

Basmadjian et al. [35] construct a power model of a server as a summation of power models of its components, the processor (CPU), memory (RAM), fans, and disk (HDD). Bircher et al. [36] propose a non-linear model to predict power using PMCs. They use PMCs that trickle down from the processor to other subsystems such as CPU, disk, GPU, etc and PMCs that flow inward into the processor such as Direct Memory Access (DMA) and I/O interrupts.

Power and Energy Models for GPUs. GPUs are now an integral part of high performance computing systems due to their enormous computational powers and energy efficiency (performance/watt). In a node, the GPU is used as a coprocessor and is connected to a CPU through a PCI-Express (PCIe) bus. Work is offloaded from a CPU to the GPU.

The first comprehensive model developed for GPUs was by [37]. The GPU power consumption in their prediction model is modelled similar to the PMC-based unit power prediction approach of [33]. In their model, the power consumption is calculated as sum of power consumptions of all the components composing the Streaming Multiprocessor (SM) and GDDR memory.

Majority of other models employ machine learning methods. [38] propose power and energy prediction models that employ a configurable, back-propagation, artificial neural network (BP-ANN). The parameters of the BP-ANN model are ten carefully selected PMCs of a GPU. The values of these PMCs are obtained using the CUDA Profiling Tools Interface (CUPTI) [39] during the application execution. [40] use the technique of program slicing to model GPU power consumption. The source code of an application is decomposed into slices and these slices are used as basic units to train a power model based on fuzzy wavelet artificial neural networks (FWNN). So, unlike earlier research efforts which use PMCs, slicing features are extracted from the programs and used in their model.

Power and Energy Models for Xeon Phis and FPGAs. In this category, we cover the other accelerators that are used in high performance computing systems.

There is an abysmal shortage of power and energy prediction models for Xeon Phis. We found just one for Xeon Phis even though this accelerator enjoys a noticeable space in the Top500 [41] supercomputers. [42] construct an instruction-level energy model of a Xeon Phi processor and report an accuracy between 1% and 5% for real world applications.

To the best of our knowledge, there are no linear regression models using PMCs because PMCs are not yet offered by FPGAs. [43] construct a linear energy prediction model based on instruction level energy profiling. [44] propose a linear component-based model to predict energy consumption of a reconfigurable Multiprocessors-on-a-Programmable-Chip (MPoPCs) implemented on Xilinx FPGAs. [45] propose a linear instruction-level model to predict dynamic energy consumption for soft processors in FPGA. The model considers both inter-instruction effects and the operand values of the instructions.

Application-specific Models. Here, we present studies for saving power and energy in HPC applications. Previous sections dwelt on power and energy models for dominant components in a node that predicted power and energy consumptions for all kinds of applications executing on these components. Our focus in this category is application-specific.

Lively et al. [46] propose application-centric predictive models for power consumption. For each kernel in an application, multivariate linear regression models for system power, CPU power, and memory power are constructed using PAPI performance events [47] as predictors.

[48] compare the power consumptions of two high performance dense linear algebra libraries i.e., LAPACK and PLASMA. Their results show that PLASMA outperforms LAPACK both in performance as well as energy efficiency.

[49], [50] propose system-wide power prediction models for HPC servers based on performance counters. They cluster real-life HPC applications into groups and create specialized power models for them. They then use decision trees to select an appropriate model for the current system load.

Lastovetsky et al. [51] present an application-level energy model where the dynamic energy consumption of a processor is represented by a function of problem size. Unlike PMC-based models that contain hardware-related PMCs and do not consider problem size as a parameter, this model takes into account highly non-linear and non-convex nature of the relationship between energy consumption and problem size for solving optimization problems of data-parallel applications on homogeneous multicore clusters for energy.

Critiques of PMC-based models. In this category, we review attempts that have critically examined and highlighted the poor prediction accuracy of PMCs for energy predictive modeling.

Economou et al. [34] highlight the fundamental limitation, which is the inability to obtain all the PMCs simultaneously or in one application run. They also mention the lack of PMCs to model energy consumption of disk I/O and network I/O. McCullough et al. [52] evaluate the competence of predictive power models for modern node architectures and show that linear regression models show prediction errors as high as 150%. They suggest that direct physical measurement of power consumption should be the preferred approach to tackle the inherent complexities posed by modern node architectures. Hackenberg et al. [53] present a study of various power measurement strategies, which includes *Intel RAPL* [54]. They report that the accuracy of *RAPL* depends on the type of workload and is quite poor for workloads that use the hyper-threading feature. They also report that the accuracy is poor for applications with

small execution times and becomes better only for applications with longer execution times since the predictions are energy averages.

O'Brien et al. [55] survey predictive power and energy models focusing on the highly heterogeneous and hierarchical node architecture in modern HPC computing platforms. Using a case study of PMCs, they highlight the poor prediction accuracy and ineffectiveness of models to accurately predict the dynamic power consumption of modern nodes due to the inherent complexities (contention for shared resources such as Last Level Cache (LLC), NUMA, and dynamic power management). Arsalan et al. [56] propose a novel selection criterion for PMCs called *additivity*, which can be used to determine the subset of PMCs that can potentially be considered for reliable energy predictive modelling. They study the *additivity* of PMCs offered by two popular tools, *Likwid* [57] and *PAPI* [47], using a detailed statistical experimental methodology on a modern Intel Haswell multicore server CPU. They show that many PMCs in *Likwid* and *PAPI* are *non-additive* and that some of these PMCs are key predictor variables in energy predictive models thereby bringing into question the reliability and reported prediction accuracy of these models.

Prominent Surveys on Power and Energy Predictive Models.

In this category, we present recent surveys summarizing the power and energy efficiency techniques employed in high performance computing systems and applications.

Mobius et al. [58] present a survey of power consumption models for single-core and multicore processors, virtual machines, and servers. They conclude that regression-based approaches dominate and that one prominent shortcoming of the these models is that they use static instead of variable workloads for training the models.

Inacio et al. [59] present a literature survey of works using workload characterization for performance and energy efficiency improvement in HPC, cloud, and big data environments. They report a remarkable increase in research papers proposing energy modelling and energy efficiency techniques from 2009 to 2013 thereby suggesting an increasing importance of energy saving techniques in the HPC, cloud, and big data environments.

Tan et al. [60] survey the research on saving power and energy for HPC linear algebra applications. They separate the surveyed efforts into two categories: 1) Power management in HPC systems and 2) Power and energy efficient HPC applications (Cholesky, LU, QR). They construct a linear model of a HPC system as a summation of power consumptions of all the nodes in the system. The power consumption of a node is modelled as the sum of all the major components (CPU, GPU, RAM) of a node.

Dayarathna et al. [61] present an in-depth and voluminous survey on data center power modelling.

O'Brien et al. [55] survey the state-of-the-art energy predictive models in HPC and present a case study demonstrating the ineffectiveness of the dominant PMC-based modeling approach for accurate energy predictions.

2.1.5 Holistic Approaches to Optimization for Performance and Energy

In this section, we will review research that has proposed solutions for optimization of scientific applications on ultrascale platforms for both performance and energy. We believe that realistic and accurate performance and energy models of computations and communications are fundamental to the effectiveness of these solution approaches.

The methods solving the bi-objective optimization problem for performance and energy (*BOPPE*) can be broadly classified as follows:

- *System-level*: Methods that aim to optimize several objectives of the system or the environment (for example: clouds, data centers, etc) where the applications are executed. The leading objectives are performance, energy consumption, cost, and reliability. A core characteristic of the methods is the use of application-agnostic models for predicting the performance of applications and energy consumption of resources in the system.
- *Application-level*: Methods focusing mainly on optimization of applications for performance and energy. These methods use application-level models for predicting the performance and energy consumption of applications. This category can be further sub-classified into methods that target intra-node optimization and methods that target both intra-node and inter-node optimization.

System-level: Mezmaz et al. [62] propose a parallel bi-objective genetic algorithm to maximize the performance and minimize the energy consumption in cloud computing infrastructures. Fard et al. [63] present a four-objective case study comprising performance, economic cost, energy consumption, and reliability for optimization of scientific workflows in heterogeneous computing environments. Beloglazov et al. [64] propose heuristics that consider twin objectives of energy efficiency and Quality of Service (QoS) for provisioning data center resources. Kessaci et al. [65] present a multi-objective genetic algorithm that minimizes the energy consumption, CO2 emissions, and maximizes the generated profit of a cloud computing infrastructure. Durillo et al. [66] propose a multi-objective workflow scheduling algorithm that maximizes performance and minimizes energy consumption of applications executing in heterogeneous high-performance parallel and distributed computing systems.

Application-level: Freeh et al. [67] propose an intra-node optimization approach that analyzes the performance-energy trade-offs of serial and parallel applications on a cluster of DVFS-capable AMD nodes. In their study, they consider three intra-node parameters to characterize the performance and energy of serial and parallel applications. Ishfaq et al. [68] formulate a bi-objective optimization problem for power-aware scheduling of tasks onto heterogeneous and homogeneous multicore processor architectures. Their solution method targets intra-node optimization. They consider intra-node parameters such as DVFS, computational cycles, and core architecture type. Balaprakash et al. [69] is an intra-node optimization approach that explores trade-offs among power, energy, and performance using various application-level tuning parameters such as number of threads and hardware parameters such as DVFS.

Drozdowski et al. [70] propose a concept called an iso-energy map, which represents points of equal energy consumption in a multi-dimensional space of system and application parameters. They study three analytical models, two intra-node and one inter-node. For the inter-node model, they consider eight parameters. From all the possible combinations of these parameters, they study twenty-eight combinations and their corresponding iso-energy maps. However, one of the key assumptions in their model is that the energy consumption is constant and independent of problem size. Marszakowski et al. [71] analyze the impact of memory hierarchies on performance-energy trade-off in parallel computations. They study the effects of twelve intra-node and inter-node parameters on performance and energy. In their problem formulations, they represent performance and energy by two linear functions of problem size, one for in-core computations and the other for out-of-core computations.

Reddy et. al. [72] study the bi-objective optimization problem for performance and energy (*BOPPE*) for data-parallel applications on homogeneous clusters of modern multicore CPUs, which is based on only one but heretofore unstudied decision variable, the problem size. They present an efficient and exact global optimization algorithm that solved the *BOPPE*. It takes as inputs, functions of performance and dynamic energy consumption against problem size, and outputs the globally Pareto-optimal set of solutions. These solutions are the workload distributions, which achieve inter-node optimization of data-parallel applications for performance and energy.

2.2 Impact of Workflow Enactment Modes on Scheduling and Workflow Performance

In the past decade, computer architectures have experienced an important paradigm shift. From a single processor containing a few homogeneous cores, computers have evolved to complex dynamic systems containing tens or hundreds of heterogeneous computing resources, the so-called *manycore* computers. Despite these trends, the majority of popular parallel programming languages, development tools and compilers remain to be based on the old symmetric multi-processing paradigm. Past efforts to make parallel computers more accessible for programmers resulted in a multitude of different and often incompatible programming libraries and language extensions, including successful standards like OpenMP, OpenCL and MPI.

On distributed computing infrastructures (DCIs), *scientific workflows* emerged in industry, business and science as an easy way to develop large-scale applications as a composition of smaller loosely-coupled components [73]. Existing DCI workflow engines are currently mature and come with rich ecosystems which support the user in all aspects of a workflow lifecycle from creating to execution, monitoring and results retrieval, interfaced towards the domain scientists and ease of use rather than the computer science underneath [74, 75, 76, 77, 78, 79]. Because of the similarity in terms of scale and heterogeneity, workflow systems represent today a promising alternative for development and execution of scientific applications on shared memory heterogeneous manycore architectures. However, existing workflow engines targeted at DCIs are prone to high overheads and latencies [80]. While such overheads are

acceptable on DCIs, tightly-coupled manycore computers are much more sensitive to latencies and other form of overheads.

To overcome these problems, Janetschek *et al.* [80] presented a *Manycore Workflow Runtime Engine (MWRE)* that efficiently exploits the low latency characteristics of heterogeneous manycore computers and which performs significantly better than traditional workflow engines on manycore computers.

There are two different strategies for enacting a workflow determining how and when the workflow engine evaluates a workflow execution plan: *early* and *late evaluation mode*. In theory, early enactment mode produces a better workflow schedule, while also having more enactment overhead. Late enactment mode theoretically produces a worse workflow schedule, while having less enactment overhead. The practical implications of early and late enactment modes on scheduling performance are still unclear, therefore, in this work we simulated the execution of a large number and variety of random MWRE workflows with both early and late evaluation mode to gain more insights on how much early evaluation mode improves scheduling performance and to be able to deduct some guidelines on when to use early evaluation mode and when to use late evaluation mode.

Next, the following topics are addressed. Section 2.2.1 introduces the scientific workflow model, followed by an introduction to workflow enactment in Section 2.2.2 and to workflow scheduling in Section 2.2.3. Section 2.2.4 explains the MWRE workflow engine for manycores. Section 2.2.5 discusses the theoretical implications of an incomplete workflow execution plan on scheduling, followed by an explanation of the methodology used to conduct the experiments presented in Section 2.2.6. Section 2.2.7 discusses experimental results, and Section 2.2.8 presents conclusions.

2.2.1 Scientific Workflow Model

A workflow consists of two parts: an *abstract* part and a *concrete* part. A short overview of these two parts is presented next.

Workflow Abstract Part

The abstract part (see Figure 2.3) of a scientific workflow comprises a hardware and middleware agnostic (and therefore portable) description of its structure, the activities involved (identified by a unique name and a type), and the data and control-flow dependencies between the activities. The individual activities are treated as black-boxes where only the input and output signatures are known.

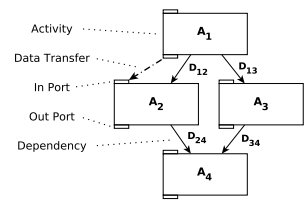


Figure 2.3: Abstract part of a scientific workflow.

There are usually two different types of workflow activities:

1. *Atomic activities* are basic indivisible units of computation;

2. *Composite activities* combine several fine granular activities, including atomic and other composite activities, to form coarse grained activities and impose a control flow on the contained inner activities.

Typical composite activities are sequential and parallel loops, conditional activities and sub-workflows.

Workflow Concrete Part

The concrete part of a workflow contains the hardware and middleware-dependent implementations of the atomic activities and their accompanying meta-information. This part is often highly specific to each individual workflow system and the underlying computing infrastructure. It usually contains information about the available activity implementations, locations where they are installed, how they can be executed, and any other further information intended to help the workflow engine in selecting the most appropriate activity implementation.

2.2.2 Workflow Enactment

A workflow engine executes a workflow instance (operation usually called *workflow enactment*) by traversing the DAG representing the workflow structure, determining the state of the individual activities, transferring data from finished activities to their successors in the dependency graph, unrolling composite activities and replacing them with the resulting subgraph, and delegating the actual execution of atomic activities to the scheduling and execution subsystems. We call the resulting DAG, where composite activities have been replaced with their contained subgraphs and enriched with additional state information, a *workflow execution plan (WEP)* .

We distinguish between two types of workflow enactment modes [80]:

1. *Early enactment mode*, where the engine reevaluates the WEP as soon as there are activity state changes, and completes it as early as possible. This mode usually comes with a much higher overhead, but results in a more complete WEP comprising more information, which allows the scheduler to better plan the workflow execution on the underlying resources;
2. *Late enactment mode* (also called lazy evaluation mode), where the engine only partially reevaluates and completes the WEP when it is absolutely necessary for further workflow enactment. This mode has less overhead, but also results in a less complete WEP with less information available for the scheduler to plan the workflow execution.

2.2.3 Workflow Scheduling

Workflow scheduling describes the process of mapping atomic activities to available computing resources where they are executed. The resulting mapping of activities to computing resources is called *workflow schedule*. The scheduler optimizes the

workflow schedule by maximizing or minimizing a given utility function, typically the overall execution time. Some scheduler implementations take more than one objective into account, some of which being in conflict with each other and requiring *multi-objective optimization* [81], or by considering one variable as a constraint [82] while optimizing the other.

Generating a full-ahead schedule is an NP-hard problem [83] and therefore, most existing full-ahead scheduling methods are approximate heuristic algorithms [84]. Existing scheduling heuristics can be broadly divided into the two following categories [85]:

1. *Just-in-time scheduling algorithms*: only consider the next activities to be scheduled when deciding on a mapping and ignore the rest of the WEP. They are usually linear in complexity with the number of activities (i.e. $O(N)$) and have a low overhead, but as a consequence produce poorer schedules;
2. *Full-ahead scheduling algorithms*: use the entire WEP when deciding on a mapping. They usually present a higher overhead, but consider more workflow information and therefore, produce in general better results.

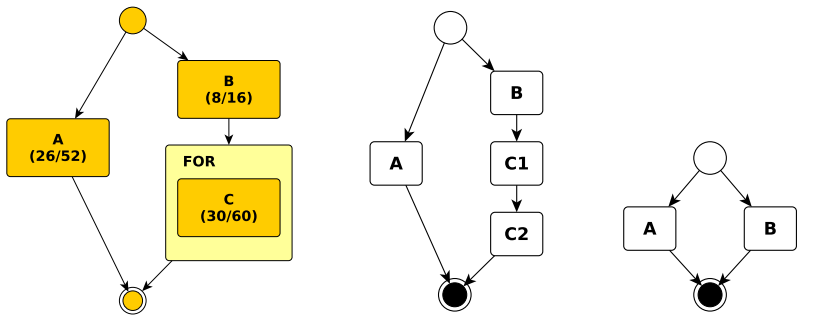
2.2.4 Manycore Workflow Runtime Engine

We designed and developed a workflow engine called *Manycore Workflow Runtime Engine* (MWRE) [80], specifically tuned for shared-memory heterogeneous manycore parallel computers. Our motivation is to exploit the workflow paradigm, highly successful for programming DCIs (like Clouds), for programming heterogeneous manycore architectures, while supporting and integrating existing established parallel programming paradigms, such as OpenMP. Traditional workflow applications in DCIs usually have a rather simple structure, feature a coarse-grained parallelism with relatively few long-running parallel tasks, and exhibit large task submission and data transfer overheads. In contrast, shared memory manycore applications usually have a much more complex structure, feature a more fine-grained parallelism with a lot of short running parallel tasks, and hardly have any task submission and data transfer overheads.

The defining feature of our engine is compiling workflows into semantically-equivalent C++ programs using a source-to-source compiler (and not interpreting workflows like most traditional engines for DCIs). The workflow engine is linked to the C++ program in the form of a shared library that uses a novel callback-driven enactment mechanism, where the engine is only responsible for maintaining and traversing the WEP. Dependency resolution and data transfers are implemented in callback functions, specifically tailored to the concrete workflow and are part of the workflow specification. This keeps the engine clean and minimizes the enactment overhead.

2.2.5 Impact of Incomplete WEP on Full-Ahead Scheduling

When using a full-ahead scheduling algorithm, the workflow enactment mode can theoretically have a huge influence on the scheduling performance. Full-ahead



(a) Example workflow.

(b) Early enactment mode.

(c) Late enactment mode.

Figure 2.4: WEPs in early and late enactment mode at workflow execution start for an example workflow, where the numbers in brackets represent the execution times on resources $R1$ and $R2$.

scheduling considers the entire WEP when calculating a schedule, therefore an incomplete WEP may lead to a comparatively worse workflow schedule.

For example, let us assume the workflow in Figure 2.4a executed on a heterogeneous system consisting of two different computing resources: resource $R1$ and resource $R2$. Resource $R1$ has a fast CPU, and resource $R2$ has a twice as slow CPU. The example workflow consists of two parallel atomic activities A and B, and a sequential `for` loop with a data dependency on activity B containing a single atomic activity C. The number of iterations of the `for` loop is known from the beginning and assumed here as two. The number in brackets represents the activity execution times on resources $R1$ and $R2$, respectively.

Most full-ahead scheduling algorithms try to prioritize the atomic activities lying on the workflow's *critical path*, defined as the longest path from the start to the end of the workflow, and the *length* of the critical path is defined as the sum of the activity execution times on the critical path. The activities on the critical path have the most influence on performance, and any delay on the critical path delays the entire workflow.

The critical path of our example workflow consists of activities B, $C1$ and $C2$ (where $C1$ refers to the instance of C in the first loop iteration, and $C2$ to the instance of C in the second loop iteration), and the minimum length of the critical path is 68. Therefore, an optimal workflow schedule maps activity A to resource $R2$, and activities B, $C1$ and $C2$ to resource $R1$ to achieve a workflow makespan of 68.

When using early enactment mode the `for` loop is immediately evaluated and the resulting WEP (see Figure 2.4b) contains all the necessary information to find the correct critical path. Therefore a full-ahead scheduling algorithm can calculate an optimal workflow schedule as depicted above.

In late enactment mode, the evaluation of the `for` loop is deferred until activity B has finished its execution. Therefore, the resulting WEP (see Figure 2.4c) initially

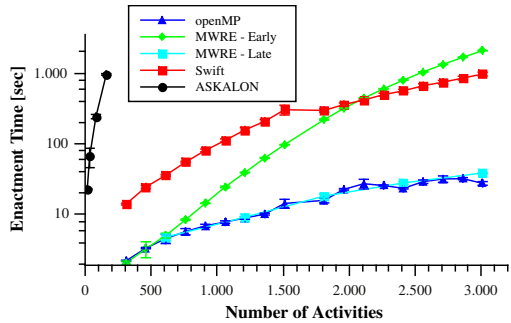


Figure 2.5: Enactment times of the Montage workflow [80].

misses the activities $C1$ and $C2$, and a full-ahead scheduler would base it's calculation of a workflow schedule on incomplete information. It may be deducted from the WEP that the critical path only consists of activity A and map it onto the fastest resource $R1$, while activity B is mapped onto the slower resource $R2$. The `for` loop will be evaluated only after activity B has finished and the WEP will look like Figure 2.4b. At this time, the critical path activity B has already been executed by the slower resource, and resource $R1$ is still occupied executing activity A. Therefore, the scheduler can only map activity $C1$ onto the slower resource, $C2$ is the only critical path activity mapped to the fastest resource. The workflow makespan in this scenario is 106, which is about 56% larger than the optimal makespan.

Based on this observations, one may conclude that early enactment mode should always be preferred to late enactment mode. However, our experience with MWRE has shown that depending on the particular workflow to be executed, early enactment mode can exhibit drastic performance losses and a limited scalability compared to late enactment mode. For example, Figure 2.5 (taken from [80]) shows the enactment overhead of the Montage workflow executed with MWRE, referring to the time spend in the engine not including the execution times of the atomic activities. In this experiment we executed the Montage workflow several times with a different number of atomic activities. The enactment time in late enactment mode stays close to the enactment time of an equivalent OpenMP program for the whole experiment. In contrast, the enactment time of early enactment mode is also close to the enactment time of the OpenMP version in the beginning, but significantly increases beyond 600 activities.

2.2.6 Methodology

To evaluate the impact of early and late evaluation mode on scheduling performance, we simulated the execution of a large number and variety of workflows on manycore architectures. Due to the lack of a sufficient number of complex real-world workflows, we used an algorithm to generate a large number of random workflows with varying parameters.

Algorithm 2 Random hierarchical workflow generation.

```
1:  procedure genRandomWorkflow(v, α, o, w, β, l)
2:      W ← origGenRandomWorkflow(v, α, o, w, β)
3:      if l > 1 then
4:          s ← selectRandomActivity(W)
5:          t ← selectRandomCompositeType(if, parallel for)
6:          SW[0] ← genRandomWorkflow(v, α, o, w, β, l − 1)
7:          if t = if then
8:              SW[1] ← genRandomWorkflow(v, α, o, w, β, l − 1)
9:          end if
10:         convertAtomicToComposite(s, t, SW)
11:         p ← selectRandomPredecessor(s)
12:         h ← createHelperNode(t)
13:         insertNode(h, p, s)
14:     end if
15:     return W
16: end procedure
```

Random Workflow Generation

For generating random workflows, we used an existing algorithm [84] that creates workflows consisting of solely atomic activities, extended to cover composite ones, as shown in Algorithm 2. The algorithm considers the following parameters as input to influence the shape and structure of the generated workflows:

- *Average number of activities v* in the workflow;
- *Workflow shape* α by randomly generating the workflow height from a uniform distribution with a mean value of $\frac{\sqrt{v}}{\alpha}$ and the width of each level from a uniform distribution with a mean value of $\sqrt{v} \cdot \alpha$;
- *Output degree o* of an activity, which is the maximum number of successors a workflow activity is allowed to have;
- *Average execution time w* of an atomic activity;
- *Computational heterogeneity* β by randomly selecting the execution time of an activity on a specific resource from the interval $\left(w \cdot \left(1 - \frac{\beta}{2}\right), w \cdot \left(1 + \frac{\beta}{2}\right)\right)$;
- *Maximum nesting level l* of the composite activity.

At first, a workflow is generated using the original algorithm (line 2). As long as the maximum nesting level has not been reached, a random activity is selected (line 4), a random composite activity type is chosen (line 5), one or two sub-workflows representing the body of the composite activity are created by recursively calling the algorithm (lines 6 –9), and finally the selected activity is converted into the corresponding composite activity (line 10). Next we select a random predecessor (line 11) of the composite activity, which supplies it with specific input data, such as conditional argument for `if` activities and loop counter boundaries for *parallel for* activities. To ease implementation of the algorithm composite activity specific

Parameter	*Symbol*	*Value set*
Average number of activities	v	10
Workflow shape	α	$\{0.1, 0.5, 1.0, 1.5, 2.0\}$
Activity output degree	o	$\{1, 3, 20\}$
Activity average execution time	w	3 seconds
Computational heterogeneity	β	3.0
Maximum nesting level	l	$\{1, 2\}$

Table 2.1 Random workflow generation parameters.

Configuration	*Description*
Configuration 1	4 different single-core CPUs
Configuration 2	8 different 10-core CPUs
Configuration 3	1 4-core CPU and 2 different GPUs

Table 2.2 Simulated hardware configurations.

data is supplied by a helper activity inserted between the selected predecessor and the composite activity (lines 12 and 13). The helper activity randomly chooses for `if` activities whether the supplied condition is `true` or `false`, and the loop iteration count between 2 and 10 for `parallel for` activities.

Experimental Setup

We conducted our experiments by generating five different workflows for each parameter combination (see Table 2.1), and then simulated the execution of each workflow five times for both evaluation modes on three heterogeneous hardware configurations (see Table 2.2) using seven different schedulers.

The workflow generation parameters were chosen to best represent the characteristics of manycore workflow applications, characterized by a relatively high number of short running activities. The generated workflows consist of $20 - 110$ unique activities, each having a different randomly chosen execution time of 0.1 to 6 seconds for each resource type. The workflows have highly different shapes, ranging from nearly sequential to workflows with a high degree of parallelism, and from workflows with very few dependencies between activities to nearly fully connected ones. Larger workflows were not created, as MWRE early evaluation mode leads from our experience to a significant increase in enactment overhead (e.g. see Figure 2.5) beyond a few hundred workflow activities. For the experiments, we aimed to have early and late evaluation modes with roughly the same enactment overhead to not bias the results.

To get meaningful results independent from a specific scheduler, the following schedulers implemented in MWRE were used:

- *Minimum Completion Time* (MCT) [86] is a just-in-time algorithm that assigns ready-to-execute tasks in no particular order to the resource with the minimum completion time.
- *Heterogeneous Earliest Finish Time* (HEFT) [84] is a list based heuristic consisting of two phases. In the ranking phase all tasks are assigned a rank representing the longest path from the task to the exit node. In the processor selection phase the tasks are assigned to a free processor with the earliest finish time in the order of their ranks.
- *Predict Earliest Finish Time* (PEFT) [87] is also a list based heuristic similar to HEFT, which uses the average path from the task to the exit node for assigning a rank.
- The *Lookahead* [88] algorithm is another variant of HEFT also taking the children of a task into account in the processor selection phase.
- The *Min-Min* [86] is a batch mode heuristic consisting of two phases. In the first phase the minimum expected completion time is calculated for each task, and in the second phase the tasks are assigned to processors according to their minimum expected completion time in the order of the overall minimum expected completion time.
- The *Max-Min* [86] scheduling algorithm is very similar to Min-Min except that the second phase takes the maximum expected completion time into account.
- The *Sufferage* [86] scheduling algorithm assigns tasks to processors according to how much the task would "suffer" in terms of expected completion time if it is not assigned to that processor.

For each workflow the average makespan, hardware configuration, scheduler and evaluation mode combination are registered. The results are grouped according to the scheduler, hardware configuration, workflow shape, activity output degree and composite activity nesting level, and the relative time difference $\Delta T_{rel} = \frac{T_{late} - T_{early}}{T_{late}}$ of the makespan of early evaluation mode T_{early} compared to the makespan of late evaluation mode T_{late} is calculated. If the relative time difference is less than $\pm 2.5\%$, it is assumed there is no significant difference. It is determined the relative number of experiments showing no significant performance improvement, the relative number of experiments showing a significant performance improvement with early evaluation mode, and the relative number of experiments showing a significant performance degradation.

Simulations are run on an Intel Core i7-2600K running at 3.40GHz with 16GB RAM.

2.2.7 Experimental Results

The results of all experiments are shown in Figure 2.6. For 85.7% of the experiments, the workflow makespan in early evaluation mode is nearly the same as the makespan in late evaluation mode. For 11.6% of all experiments the early evaluation mode is faster, while for 2.7% it is slower than late evaluation mode. In the best case early evaluation mode is 43% faster, and in the worst case early evaluation mode is 39.6% slower. The experiments for which early evaluation mode is faster show an average

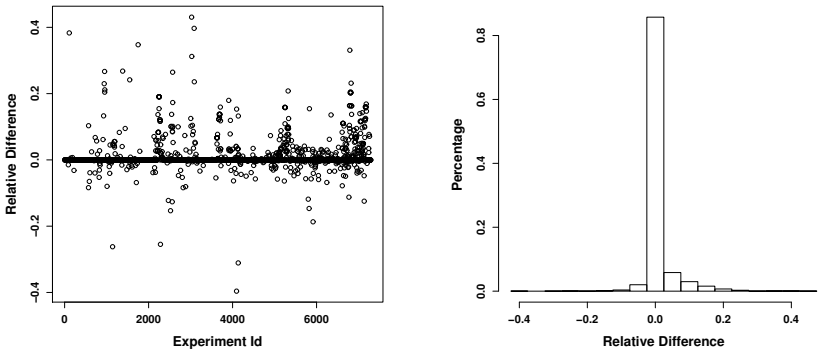


Figure 2.6: Result overview of all experiments.

Scheduler	*No change*	*Early better*	*Late better*	*Average improvement*	*Average degradation*
MCT	74.5%	15.9%	9.5%	8.2%	-6.7%
HEFT	89.3%	10.1%	0.5%	10.9%	-5.6%
PEFT	84.5%	12.2%	3.3%	10.2%	-5.8%
Lookahead	84.9%	12.6%	2.5%	8.8%	-5.1%
Min-Min	89.6%	9.3%	1.1%	10.7%	-5.6%
Max-Min	87.9%	10.7%	1.4%	8.6%	-18.2%
Sufferage	89.3%	10.2%	5.5%	9.2%	-11.4%

Table 2.3 Results by scheduler type.

performance improvement of 9.4%, and the experiments for which is slower show an average performance degradation of -7.3%.

These results indicate that for the majority of workflows, using early or late evaluation mode has practically no significant impact on scheduling performance. Only for a minority of 10%, the executed workflows in early enactment mode caused a performance improvement of 10%. It is also observed that 3% of the workflows executed in early enactment mode led to worse performance. The reason for this result is that the schedulers are suboptimal heuristics and that more but still incomplete information can still cause the scheduler to misjudge the critical path (see Section 2.2.5), while with less information the scheduler may correctly guess the critical path.

Table 2.3 and Figure 2.7 show the experimental results by scheduler type. MCT schedules activities to the fastest available machine as they are passed to the scheduler and it does not take the rest of the WEP into account. Therefore, it is the least stable, and its results roughly form a Gaussian distribution. However, MCT still shows a slight bias towards early evaluation mode, 6% more workflows showing better performance. The full-ahead scheduler shows rather stable performance with 80%-90% of the workflows having no significant performance difference between early

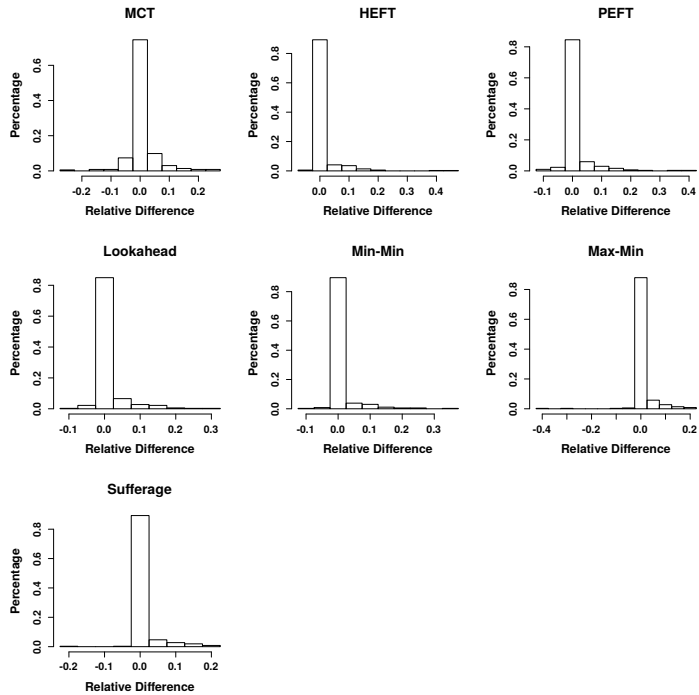


Figure 2.7: Histograms of relative performance by scheduler type.

Hardware config	*No change*	*Early better*	*Late better*	*Average improvement*	*Average degradation*
Config 1	84.2%	12.1%	3.7%	9.3%	-7.8%
Config 2	84.3%	12.1%	3.6%	9.5%	-7.5%
Config 3	84.2%	12.6%	3.2%	9.2%	-7%

Table 2.4 Results by hardware configuration.

and late evaluation mode. For the workflows where there is a significant performance difference, it is early evaluation mode showing a better performance in the majority of cases. The only exception is Sufferage, where only twice as many workflows show better performance with early evaluation mode.

Table 2.4 and Figure 2.8 show the experimental results by the hardware configuration. There is no significant difference in the results for the different hardware configurations. For all hardware configurations, 84% of all experiments show no significant difference between early and late evaluation mode, 12% show 9% better performance with early evaluation mode, and 4% show 7% of worse performance.

Table 2.5 and Figure 2.9 show the experimental results by workflow shape α. Also here, there is hardly any difference between different workflow shapes. For all workflow shapes, 84% of the experiments show no significant difference between early

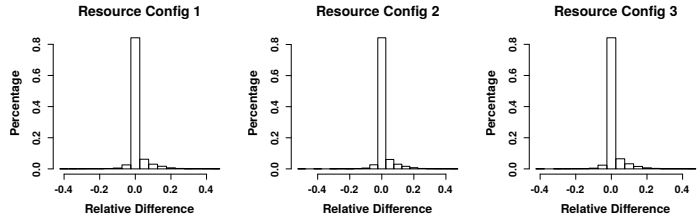


Figure 2.8: Histograms of relative performance by hardware configuration.

Workflow shape	*No change*	*Early better*	*Late better*	*Average improvement*	*Average degradation*
0.1	85.8%	11.4%	2.8%	9.3%	-7.1%
0.5	86.2%	10.8%	3%	9.3%	-7.4%
1.0	83.7%	13.2%	3.1%	13.2%	-3.1%
1.5	83.9%	12.5%	3.6%	9.2%	-7.4%
2.0	84.4%	12.1%	3.5%	9.2%	-7.9%

Table 2.5 Results by workflow shape α.

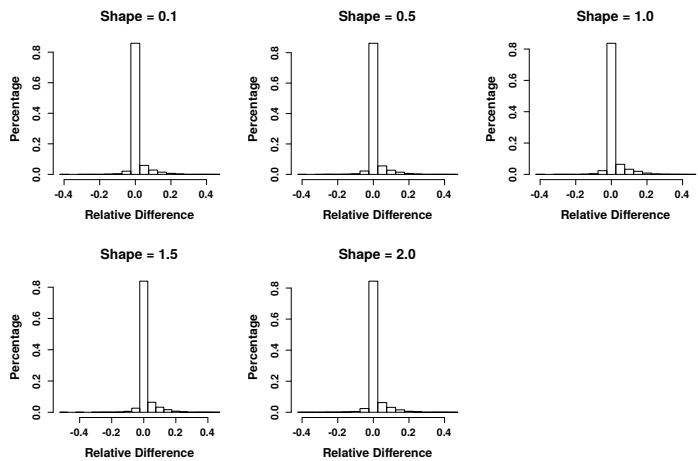


Figure 2.9: Histograms of relative performance by workflow shape α.

and late evaluation mode, 12% show 9% better performance with early evaluation mode, and 3% show 7% of worse performance. The only difference is $\alpha = 1.0$, which shows an average performance improvement of 13.2% instead of 9%, and an average performance degradation of -3.1% instead of -7%. For $\alpha = 1.0$, the workflow height and width is the same, which means that all activities are equally distributed. This gives the scheduler the most opportunities for improving the mapping.

Table 2.6 and Figure 2.10 show the experimental results concerning the output degree of workflow activities. Also here there is hardly any difference between

Outdegree	*No change*	*Early better*	*Late better*	*Average improvement*	*Average degradation*
1	84.4%	11.8%	3.8%	9.1%	-6.9%
3	83.8%	12.8%	3.4%	9.5%	-7.4%
20	84.6%	12.1%	3.3%	9.3%	-8.2%

Table 2.6 Results by outdegree.

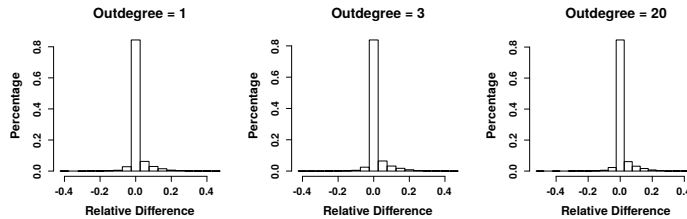


Figure 2.10: Histograms of relative performance by outdegree.

different output degrees. For 84% of the experiments there is no significant difference between early and late enactment mode, for 12% of the experiments early enactment mode causes 9% of better performance, and for 3% of the experiments the early enactment mode causes 7% of worse performance.

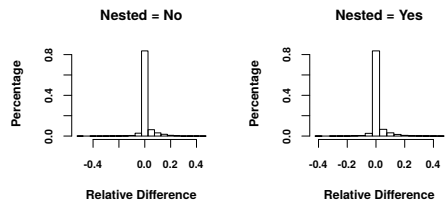


Figure 2.11: Histograms of relative performance by composite nesting level.

Table 2.6 and Figure 2.10 show the experimental results considering whether there is nested composite activities in the workflow. Also here, there is hardly any difference, 84% showing the same performance, 12% showing better performance with early enactment mode with a performance improvement of 9% and 3% show 7% of worse performance.

2.2.8 Conclusion

The impact of early and late enactment modes on workflow execution performance were evaluated. Early evaluation mode provides more information to the scheduler, which can calculate a potentially better schedule, improving the workflow performance. On the other hand, early evaluation mode causes a significant increase in workflow enactment overhead degrading workflow execution performance and limit-

Nested	*No change*	*Early better*	*Late better*	*Average improvement*	*Average degradation*
No	83.7%	12.4%	3.9%	9.3%	-7.9%
Yes	83.6%	12.7%	3.7%	9.3%	-7%

Table 2.7 Results by composite nesting level.

ing scalability. In order to find guidelines to when early evaluation mode significantly improves workflow performance, results were broken down according to several parameters defining workflow shape and structure.

The first relevant result is that for 85.7% of the experiments we could not find a significant difference in workflow performance between early and late execution mode. We conclude that it is safe to use late evaluation mode for most workflows to get better scalability and less enactment overhead without the fear of loosing performance because of a suboptimal workflow schedule. Only for 11.6% of the experiments we observed a significantly better performance with early enactment mode with an average improvement of 11.6% and a maximum improvement of 43%. For 2.7% of the experiments, we observed a significantly worse performance with early evaluation mode with an average performance degradation of 7.3% and a maximum performance degradation of 39.6%.

The second relevant result is that for 14.3% of the experiments, while there is a significant difference in performance between early and late enactment mode, no decisive guidelines were identified when a workflow performs better. The best enactment mode is highly individual for each workflow and no correspondence can be made to a specific parameter defining workflow shape or structure. The only way to determine whether early or late enactment will cause a better performance is to execute the workflow using both modes and compare the results.

Based on these results, the late enactment mode was selected as the default mode in MWRE. According to the experiments, the potential performance improvement of early enactment mode due to a better scheduling is too insignificant compared to the downsides of a higher enactment overhead and worse scalability.

MWRE is a workflow engine for shared-memory heterogeneous manycore computers, and thus, MWRE workflows have different characteristics than DCI (Cloud) workflows. More precisely, they feature a more complex workflow structure with a higher number of shorter running activities. The experimental results reflect this and, therefore, only have limited validity for common DCI environments. Because DCI workflows have a simpler structure with a lower number of longer running activities, the early evaluation mode here has less impact on scheduling performance.

2.3 Towards General Purpose Computations at the Edge

Originally designed to exploit the power of multi-core processors through virtualization, Cloud Computing [89] has changed over the past decade to support ultrascale computations. The new paradigm, often called *aggregation*, collects a large number

of resources in a pool to form a single service with huge storage and computation capacities. Unfortunately, with the huge amounts of data generated via modern applications, the cloud center has become a bottleneck and a single point of failure. This advocated an extended paradigm, called *Edge Computing*, that brings part of the data storage and computation closer to the user. The benefits are plenty: reduced delays, high availability, low bandwidth usage, improved data privacy, etc. In this section, we introduce recent advances in edge computing that makes the coordination of edge networks synchronization-free and convergent. We address the main challenges facing applications on the data management and communication aspects. The section also provides convenient runtime environments for different categories of edge computing scenarios[16].

2.3.1 Motivation

Edge Computing offers the opportunity to build new and existing ultrascale applications that take advantage of a large and heterogeneous assortment of edge devices and environments. Fully realizing the opportunities that are created by edge computing, requires dealing with a set of key challenges related with the high number of different components that compose such systems and the interactions among them. In this work, was address the main challenges on the communication and data management levels allowing for robust communication and available data access.

On the communication frontend, the fact that applications are composed of components running in heterogeneous environments requires robust and efficient solutions for tracking these components. This implies the development of highly robust and adaptive membership services and mechanisms that allow efficient communication among these components. Among the promising class of gossip-based communication protocols are those "hybrid" ones [90, 91], in which payloads are propagated though an elected logical *spanning tree*, supported by lightweight meta-data across the graph for recovery (reconstructing another logical tree) under failures.

The consequences of such hostile environments are also present on the data management level. Since application components run on different administrative domains scattered across heterogeneous environments, communication links between these components can be disrupted by external factors (i.e, network partitions) frequently. This implies that the progress of computations executed across different application components cannot depend on continuous communication with other components, or in other words, cannot depend on synchronous interactions. This advocates the use of synchronization-free (i.e., sync-free) programming abstractions backed by sync-free data propagation and replication techniques. An interesting approach is to make use of Conflict-free Replicated Data Types (CRDTs) [92, 93, 94] that are proven abstractions designed to achieve convergence under such conditions (this is explained later in more details).

[16]Credits go to all team members contributed to the success of this work within the EU FP7 Syncfree project and EU H2020 LightKone project. The research leading to these results has received funding from the European Union's Horizon 2020 - The EU Framework Programme for Research and Innovation 2014-2020, under grant agreement No. 732505, LightKone project.

Finally, heterogeneity is the norm in ultrascale edge applications, and it exists at various layers: execution environments, communication media, data sources, operating systems, programming languages, etc. Addressing this heterogeneity can be achieved by leveraging on different run-time supports and frameworks that provide a more unified vision of resources to application developers. These different run-time and frameworks will have to inter-operate through the use of standard protocols and common data representation models.

In the following we refine the challenges associated with tapping on edge computing to design ultrascale applications, and discuss enabling technology that paves the way to tackle these challenges, and finally discuss a set of run-time and framework support that can simplify the design of such applications.

2.3.2 Edge Computing Opportunities

Edge Environments. To the contrary of cloud computing where the data and computation is centralized at the cloud data centers, the edge computing paradigm encompasses a large number of highly distinct execution environments that are defined by the network topology, connectivity, locality, and the storage and computation capacities of the devices used. In particular, we identify we identify the following interesting edge environments:

- Fog Computing: a variant of cloud computing where the cloud is divided into smaller cloud infrastructures located in the user vicinity. In such environments, each fog cloud often serves as an individual cloud, although the data can eventually be incorporated with other fog [95, 96].
- Mobile Cloudlets: small cloud datacenters that are located at the edge and are tailored to support mobile applications with powerful computations and low response times, e.g., in ISP gateways or 5G towers [97, 98, 99].
- Hardware-based Clouds: self-contained devices, such as routers, gateways, or set-up boxes, that are enriched with additional computational and storage capabilities like [100, 101].
- Peer-to-Peer (P2P) Clouds: these environments try to leverage existing devices, e.g., user mobiles, laptops, and computers in volunteer networks, aiming to cooperate towards achieving a common goal [102, 103, 90].
- Things and Sensor Network Clouds: resource constrained devices, e.g., Internet of Things devices, sensors, and actuators, capable of performing some computations on data without accessing or delegating to the (possibly unreachable) cloud center [104].

All of these different scenarios are characterized by having highly heterogeneous devices in terms of processing power and memory, but also regarding their connectivity to the backbone of the Internet or even their up-times (being continually running or being operating for only small periods of time). These different devices naturally, run different operating systems, from general purpose Linux based operating systems in the case of servers in cloud and private infrastructures, to proprietary operating systems in the case of set-up boxes, mobile operating systems, general purpose multi-user operating system or even single process operating systems in the case of small sensors

and actuators. Gathering the capacity of devices with very different properties is highly challenging, and devising solutions that can exploit devices located in different edge devices brings additional challenges. Next we will discuss some of the key high level challenges in tapping the potential of the edge.

Challenges at the Edge Despite the diversity of edge computing environments, components, and properties, the major challenges are common to most of the scenarios. In particular, we recognize the following four challenges:

Scalability. One of the reasons to move the data and computation off the cloud data center to the edge is to reduce the I/O overload on the cloud and avoid bottlenecks related with the limited network capability connecting clients to the cloud infrastructures. Nevertheless, this raises another challenges on handling the data and computation in a distributed way especially in ultra-scale systems composed of, potentially, many data centers and thousands of edge devices. This scale requires special techniques across the data, computation, and communication planes. As captured by the CAP theorem [105], and because scaling out will increase the potential for network portions, link failures, and arbitrary communication delays, ensuring availability—as an essential requirement for most applications including novel edge applications—requires relaxing the consistency model employed in the design and implementation of these solutions. Consequently, the computation should also be decentralized and coordinated to achieve the common goals of the entire system. Finally, the communication middlewares should also scale to afford a high number of nodes, e.g., through asynchronous, P2P, or gossip protocols.

Interoperability. Considering the edge categories discussed above, one can notice the notable diversity level of the devices and platforms used within the same or across edge clouds. This brings interoperability challenges if all components shall communicate with each others, thus requiring well studied interfaces and possibly introducing a common layer that all components can understand without compromising the characteristics deemed essential.

Resilience. While cloud datacenters use high quality equipment for the network and devices, edge computing often use commodity equipment that are far from perfect regarding failures. The problem is extrapolated with edge network problems that are likely to be loosely connected, mobile, and hostile. This threatens the quality of the service and makes the data and communication components even more complex. That said, one must consider the performance as well as the cost trade-offs (being a major factor due to the constrained resources).

Security and Privacy. Given the heterogeneity of the edge applications, security and privacy measures must be analyzed and tackled individually. However, in general, it is desired to find a common security layer or security measures that govern a wide range of applications. Security and privacy on the edge need to be addressed on the infrastructure and data levels. The former can be deployed at the communication or network layer, ranging from establishing secure connections to enforcing secure group dynamics, and cover several dimensions including data integrity, data privacy, or resilience to DoS attacks. On the other hand, edge applications often deal with

sensitive data which likely requires lightweight encryption and data sanitization techniques to control the disclosure of such data. These may also include secret-sharing, anonymization, noise addition or partitioning, etc., depending on the specific security and functional requirements of the implementations.

Use Cases. As discussed in the edge environments, edge computing supports a plenty of applications and use-cases. In this section, we focus on three categories in which most of the use-cases lie:

- Time series applications. This category spans a multitude of applications with the popularity of IoT. The scenario is often a type of time series where data is generated by the IoT devices, e.g., sensors, and pushed to the edge devices to get stored, aggregated, and partially computed. The aggregated data is then pushed to the center of the cloud for further handling. The data-flow can sometimes be in the opposite sense if actuator devices exist; in this case, the processed data in the cloud is pushed back to the actuators to do some action. Consequently, this scenario represents a hybrid model of light and heavy devices, different types of networks (e.g., Zig-bee, WIFI, WAN, etc), as well as data-flow direction.
- Mobile edge applications. This category covers all the applications in which devices are mobile and public. This makes the model very hostile as link failure and delays are expected, and the availability of nodes cannot be guaranteed (e.g., a mobile device can be switched off). The communication in such use cases does not follow a particular data-flow pattern, but it is often P2P or gossip-based due to the dominant dynamic graph-like network of nodes. In such applications, devices have moderate storage and computation resources that makes the interaction symmetric. Obviously, the main challenges in such use-cases are reseliance and availability. In some cases, access points, towers, or routers with more capacities can assist in storage, computation, and communication, which can be used as third party authority when needed.
- Highly available databases. This category is a natural evolution of scalable databases in cloud and cluster systems. The intuition is to replicate the database geographically, brining replicas or cache servers closer to the user. In this scenario, devices are at least commodity computers or servers with non-scarce capacities, and then network is often the Internet. In addition to availability, the challenge in such use-cases is to tolerate network partitions and optimize data locality (especially when partial replication is used). These scenarios are close to Fog Computing and Cloudlets with the difference that all node must work as a single (often loosely) coordinated system.

2.3.3 Enabling Technologies for the Edge

Synchronization-Free Computing. Edge devices and edge networks are both unreliable. This follows both from their design, e.g., they are low-power systems that are often offline, and from the nature of the edge itself, e.g., it is directly involved

with real world activities, such as in Internet of Things. Despite this unreliability, we would like to perform computations directly on the edge.

To perform computations directly on the edge, we need distributed data structures and operations that tolerate the unreliability of the edge. Synchronization-free computing fits the bill because of its very weak synchronization requirement. A prominent example is Conflict-free Replicated DataType (CRDT), which is a replicated data type that is designed to support temporary divergence at each replica, while guaranteeing that when all updates are delivered to all replicas of a given instance, they will converge to the same state. (More details about CRDTs can be found in Chapter 4 or by referring to [92, 93, 94].) CRDTs naturally tolerate node problems, namely nodes going offline and online and node crashes, and network problems, namely partitions, message loss, message reordering, and message duplication. Node crashes are tolerated as long as the desired state exists on at least one correct node. The following results on CRDT computations are summarized from [106].

CRDT Definition. For the purposes of this section, we define a *CRDT instance* to be a replicated object that satisfies the following conditions:

- Basic structure: It consists of n replicas where each replica has an initial state, a current state, and two methods, query and update, that each executes at a single replica.
- Eventual delivery: An update delivered at some correct replica is eventually delivered at all correct replicas.
- Termination: All method executions terminate.
- Strong Eventual Consistency (SEC): All correct replicas that have delivered the same updates have equal state.

This definition is slightly more general than the one given in the original report on CRDTs [92]. In that report, an additional condition is added: that each replica will always eventually send its state to each other replica, where it is merged using a join operation. This condition is too strong for CRDT composition, since it no longer holds for a system containing more than one CRDT instance. We explain the conditions needed for CRDT composition in the next section.

CRDT Composition. The properties of CRDTs make them desirable for computation in distributed systems. It is possible to extend these properties to full programs where the nodes are CRDTs and the edges are monotonic functions. To achieve this, it is sufficient to add the following two conditions on the merge schedule, i.e., the sequence of allowed replica-to-replica communications:

- Weak synchronization: For any execution of a CRDT instance, it is always true that eventually every replica will successfully send a message to each other replica.
- Determinism: Given two executions of a CRDT instance with the same set of updates but a different merge schedule, then replicas that have delivered the same updates in the two executions have equal state.

The first condition allows each CRDT instance to send the merge messages it requires to satisfy the CRDT conditions. The second condition ensures that the execution of each CRDT instance is deterministic, which makes it a form of functional

programming. We remark that SEC by itself is not enough for this, since the states of replicas *in different executions* that have delivered the same updates can be different, even though SEC guarantees that they are equal in the same execution. In practice, enforcing determinism is not difficult but it depends on the type of the CRDT instance. Article [106] explains how to do it for a set that has add and remove operations (the so-called Observed-Remove Set).

We define a *CRDT composition* to be a directed acyclic graph where each node is a CRDT instance, and each node with at least one incoming edge is associated to a function of all incoming edges arranged in a particular order. Given the first of the two conditions introduced above, we can show that the execution of a CRDT composition satisfies the same properties as a single CRDT instance. If the second condition is added, then the CRDT composition behaves like a functional program.

Hybrid Gossip Communication. Gossip is a well known and effective approach for implementing robust and efficient communication strategies on highly dynamic and large-scale system [103, 91]. In its most simple form, in a gossip protocol, each node periodically interacts with a randomly selected node. In this interaction both exchange information about their local state (and potentially merge it). Since all nodes do this in parallel and in an independent fashion, after approximately one round-trip time, all nodes will have performed, at least, one merge step, and on average two merge steps (one initiated by the node itself and another initiated by some peer). We usually call this period of interactions a *cycle*. After a small number of cycles, the network converges to a globally consistent vision of the system state. This simple approach cab be used, for instance to compute aggregate functions, such as inferring the network size or load. Interestingly, this can also be used for other, and more complex, purposes such as managing the membership of large-scale system, which implies building and maintaining an overlay (i.e, logical) network topology, in a way that is both robust and scalable, but also to support robust data dissemination in such systems.

Gossip-based approaches have been shown to be highly resilient to network faults, due to the inherent redundancy that its core to the design of gossip protocols. Unfortunately, this redundancy also leads to efficiency penalties. Hybrid gossip addresses this aspect of gossip protocols. In a nutshell, the key idea of hybrid gossip is to leverage on the feedback produced by previous gossip interactions among nodes, such that an effective and non-redundant structure of communication can naturally emerge. The topology of this *emergent structure* depends on the computation being performed by nodes, and it enables nodes significantly improve the communication and coordination cost by restricting the exchange of information among node to the logical links that belong to this structure, lowering the among of redundant communication.

Key to maintaining the fault-tolerance of gossip protocols in hybrid gossip is the use of the remaining communication paths among nodes (those that are not selected to be part of the emergent structure) to convey minimal control information. This control information enables the system to detect (and recover) from failures that might

affect the emergent structure. Moreover, in highly dynamic scenarios, the additional communication paths allow nodes to fall back to a pure gossip strategy, for instance, when there are a significant number of concurrent nodes crashes or network failures.

Interesting, hybrid gossip solutions naturally allow different components of the system to operate using either the emergent structure or a pure gossip approach simultaneously. Hence, components of the system that are in stable conditions (i.e, low membership dynamics and low failures) will operate resorting to the emergent structure, while components of the system that are subjected to high churn or network/node failure will fallback to use pure gossip while still being able to inter-operate with the components using the emergent structure.

Therefore, hybrid gossip approaches enable applications to, effectively and transparently, benefit from the resilience of a pure gossip approach entwined with the efficiency of a gossip approach that leverages an emergent communication topology. The hybrid gossip approach has been introduced in [90, 107]. The Plumtree protocol in particular, shows how to build an efficient and robust spanning tree connecting large number of nodes to support reliable application-level broadcast. This solution is currently used in industry, for example, the Basho Riak database uses it to manage the underlying structure of its ring topology which is used to map data object keys into nodes (through consistent-hashing).

2.3.4 Runtime for Edge Scenarios

Above we have discussed enabling technologies that can be leveraged to build new and exciting edge applications in the ultrascale domain. Tapping into these enabling technologies can however, be a complex task for developers. Therefore, it becomes relevant to provide frameworks, tools, and other artifacts that exploit these technologies in a coherent way, providing high level abstractions to programmers that aim at developing their ultrascale edge applications. We now discuss some existing runtime support tools and frameworks that have been recently proposed to this end.

Antidote. Antidote is a geo-replicated key-value store, designed for providing strong guarantees to applications while exhibiting high availability, thus providing a good compromise in the consistency versus availability trade-off in the design of cloud databases. These proprties make Antidote a strong candidate as an edge database especially when edge nodes have non-scarce resources (e.g., commodity servers).

In particular, some cloud databases adopt a strong consistency model by enforcing a serialization in the execution of operation, leading to high latency and unavailability under failures and network partitions. Other databases adopt a weak consistency model where any replica can execute any operation, with updates being propagated asynchronously to other replicas. This approach leads to low latency and high availability even under network partition, but replicas can diverge. On the other hand, Antidote allows any operation to execute in any replica, but provides additional guarantees to the application as we explain next.

First, Antidote relies on CRDTs for guaranteeing that concurrent updates are merged in a deterministic way. Antidote provides a library of CRDTs with different concurrency semantics, including registers, counters, sets and maps. The applications

programmer must select the most appropriate CRDT, considering its functionality and concurrency semantics (e.g., add-wins, remove-wins).

Second, Antidote enforces causal consistency, guaranteeing that whenever an update *u* may depend on update *v*, if a client observes update *u* he also observes update *v*. Applications can leverage this property to guarantee their correctness when the correctness depends on the order of updates, e.g., an update executed after changing the access control policies should not be visible in a replica with the old access control policies.

Third, Antidote provides a highly available form of transactions, where reads observe a causally-consistent snapshot of the database and writes are made visible atomically. Unlike standard transactions, write-write conflicts are solved by merging the concurrent update. Applications can leverage these highly-available transactions to guarantee that a set of updates is made visible atomically.

Fourth, Antidote provides support for efficiently enforcing numeric invariants, such as guaranteeing that the value of a counter remains larger than 0. To this end, it includes an implementation of a Bounded Counter CRDT [108], a shared integer that must remain within some bounds. The implementation uses escrow techniques [109] for allowing an operation to execute in a replica without coordination in most cases.

Finally, associated with Antidote, we have developed a set of tools to verify whether an application can execute correctly under weak consistency, and when this is not the case, what coordination is necessary. These tools are backed by a principled approach to reason about the consistency of distributed systems [110].

Antidote is designed to be deployed in a set of geo-distributed data centers. Within each cluster, data is sharded among the servers. Data is geo-replicated across data centers. The execution of transactions in Antidote, and the replication of updates across data centers, is controlled by Cure [111], a highly scalable protocol that enforces transactional causal+ consistency (combining CRDTs for eventual consistency, causal consistency and highly available transactions).

Legion. Legion [112] is a new framework for developing collaborative web applications that transparently leverage on the principles of edge computing by enabling direct browser-to-browser communication. Legion was implemented in *javascript* and it uses the *Web Real-Time Communications* (https://webrtc.org) to establish direct communication channels among web application users. At its core, Legion enables applications to transparently replicate, in the form of CRDTs, relevant application state in clients. Clients can then modify the application state locally, and through the use of hybrid gossip mechanisms, synchronize directly among them, without the need to go through the web application server. The server however is still used both to ensure the durability of the application state, but also to assist in the operation of Legion, namely to simplify the task of creating the initial webRTC connections among clients when they enter the application.

A simplified architecture of Legion is illustrated in Figure 2.12. Legion can be used by a web application simply by importing a javascript script. This script provides the application access to the *Legion API*. The API exposes to the application the ability to manipulate data objects that can be used to model the application state. These data objects include records, counters, lists, and maps. All of these objects

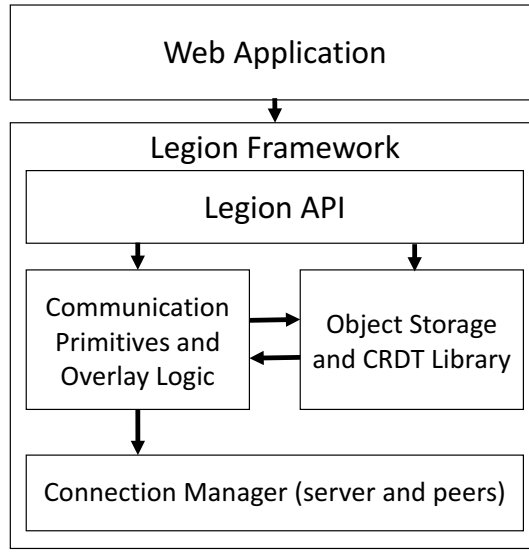


Figure 2.12: The Legion architecture (adapted from [112])

are internally represented by Legion through CRDTs which simplifies the the direct synchronization among clients of shared application state. This is provided by an extensible CRDT Library that is part of the *Object Store* component of Legion . The synchronization of objects among clients (and that of a subset of clients with the server to ensure durability) is transparently managed by the Object Store.

To guide the synchronization process, Legion leverages on an unstructured overlay network, whose construction is guided by the principles of hybrid gossip, and takes into consideration the relative distance of each client among them. This allows clients to mostly interact and synchronize with clients that are in their vicinity. While the typical use case in Legion is to have clients interacting through the manipulation of shared data objects, web applications also have access to communication primitives that enable them to disseminate messages among the currently active clients of the application in a decentralized fashion. This is achieved by a gossip-based broadcast protocol that operates on top of the legion overlay network.

Finally, Legion also takes into account security, by ensuring that before clients can start to replicate and manipulate application data objects they authenticate on a server. Moreover, Legion exposes an adapter API, that allows developers to integrate their Legion-backed applications with existing backends. The framework provides adapters to the Google Real Time API[17]. These adapters allow the developers to leverage this backed to do any combination of the following: authentication and access control, data storage for durability, and support to the WebRTC signaling protocol required to create webRTC connections among browsers. More details on the design and operation of Legion can be found in [112]. Legion is open source and available, along side some demo applications through https://legion.di.fct.unl.pt.

[17]https://developers.google.com/google-apps/realtime/application

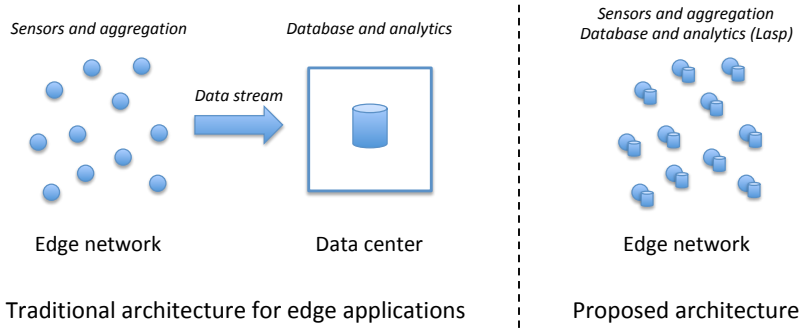


Figure 2.13: Proposed architecture for edge applications using Lasp

Lasp. The Lasp language and programming system [113] was designed for application development on unreliable distributed systems, and in particular for edge computing. Lasp allows developers to write applications by composing CRDTs, as explained above [106]. In addition to composition, Lasp also provides a monotonic conditional operation that allows executing application logic based on monotonic conditions on CRDTs. The Lasp implementation combines a programming layer based on synchronization-free computing with a communication layer based on hybrid gossip. This makes the implementation highly resilient and well-adapted to edge networks.

Many of today's edge applications use the cloud as a database to store data coming from the edge. By using Lasp as their database, such applications can be translated to fully run on the edge (see Figure 2.13). This cannot be done with traditional cloud databases since they are not designed to run on unreliable edge networks. In the proposed architecture, the edge network runs everything: the sensors and aggregation software on individual edge nodes, and the database (Lasp) on all edge nodes. Analytics computations can be run either as an internal Lasp computation or external to Lasp on individual nodes, using Lasp just as a database.

Example Lasp program. A typical application for Lasp is the scenario of advertisements counter that counts the total number of times each advertisement is displayed on all client mobile phones, up to a preset threshold for each. Figure 2.14 defines graphically part of the Lasp program for this application. The actual code is a straightforward translation of this graph. The application has the following properties:

- Replicated data: Data is fully replicated to every client in the system. This replicated data is under high contention by each client.
- High scalability: Clients are individual mobile phone instances of the application, thus the application should scale to millions of clients.
- High availability: Clients need to continue operation when disconnected as mobile phones frequently have periods of signal loss (offline operation).

This application can be implemented completely on the edge, as explained previously, or partly on the cloud. For this application we have demonstrated the scalability of

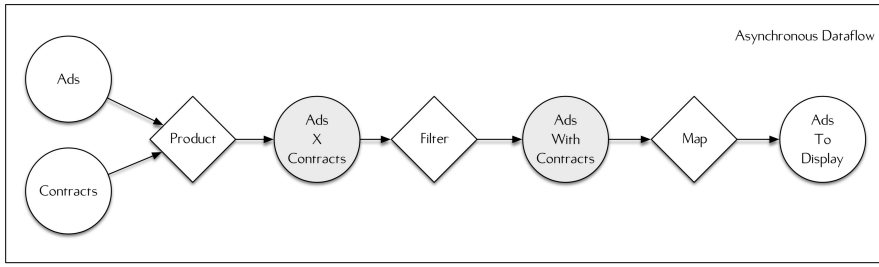


Figure 2.14: A Lasp computation to derive the set of displayable advertisements in the advertisement counter scenario. On the left, Ads and Contracts give information for the advertisements, including how many times they have been displayed, and their contracts, including the threshold for each advertisement. On the right are the advertisements that can be displayed. All data structures are sets, similar to database relations, and the computation is similar to an incremental SQL query.

the Lasp prototype implementation up to 1024 nodes by using the *Amazon* cloud computing environment to simulate the edge network [114].

2.3.5 Future Directions

Building additional tools and support for a new generation of ultrascale edge applications is quite relevant and challenging. The varied nature of edge computing environments, which can combine small private clouds and data centers, specialized routing equipment and 5G towers, users desktops, laptops and even cellphones, to small things sensors and actuators, makes it a daunting task to build a single runtime support that can efficiently operate on all such devices and deal with their heterogeneity.

While we presented a set of tools and frameworks that can ease the development of ultrascale edge computing applications and services, these do not cover all possible execution scenarios. That path to build such support requires not only the development of specialized runtimes for different edge settings, but also devising standard protocols and data representation models that allow the natural integration of different runtimes in a cohesive and effective edge architecture.

Current solutions for data replication and management are also unsuitable for the ultrascale that one is expected to find in emerging edge computing applications. The use of CRDTs to address the requirements of data management in this setting presents a viable approach. However, further efforts have to be dedicated in designing new and

efficient synchronization mechanisms that can naturally adapt to the heterogeneity of the execution environment.

2.4 Spectral Graph Partitioning for Process Placement on Heterogeneous Platforms

It is customary in the literature to model a distributed application as a graph, whose vertices are processes, or computing tasks, and an edge between tasks denotes a communication between them. The edges are weighted with a positive value to mark the magnitude of this communication. Frequently used magnitudes to measure the communications are the data volume, in total number of bytes, or the number of messages interchanged [115].

In this setting, spectral techniques divide the set of vertices in two parts, equal in number of vertices, in such a way that the total communications from one part to the other is lesser than between the two parts of any other partition. The practical interest for this is to assign each part to a computation node, so the slow communication link between two nodes are used less than the quick intra-node links. It is imposed that the computational nodes are similar in performance, and also similar the computational requirements of the vertices, in order for the assignation be balanced. The theoretical resource that allows to compute this in an effective way is the Fiedler eigenvector of the Laplacian matrix of the graph [116]. The study of the eigenvalues and eigenvectors of a matrix is called spectral resolution [117], hence the name of the method.

In this subsection we describe the spectral method as it is customarily used. We also propose to extend the previous scheme in two directions. First, we consider that each vertex has assigned a volume or weight, positive but possibly different depending on the vertex. To divide the set of vertices into two parts, so that the part have the same volume (possibly with different number of vertices), we consider the Fiedler eigenvalue of a generalized Laplacian (that we will define) which has similar properties to the standard Laplacian. The practical interest of this extension comes from the fact than the computational requirements of each vertex (process) can be different, and we are interested in a partition in vertex subsets with equal computational load (not necessarily equal number of vertices).

A second extension is to consider the division in two parts, where the fraction of total volume assigned to each part is not the same, but can be predefined to p and $1-p$ to each part, for a fraction p of the total of vertices. The Fiedler eigenvector of the generalized Laplacian can be used to this end. This is of interest for the case where the two subsets of vertices/processes will be assigned to computational units that are not equal in speed, being instead proportional to p and $1-p$. Hence, the partition of tasks is conformal with the speed of the intended processors. We also discuss the problem of partitioning in more than two parts. We find difficult to put it in this scheme.

For the structure of the subsection, in the following subsubsection we describe notation about graphs and Linear Algebra. Then we introduce the Spectral Partitioning technique using the Laplacian . The material is standard but our presentation emphasizes the operator view (that is, avoid references to coordinates as much as

possible). The subsubsection 2.4.3 is our work about weighted graph partitioning using a potential over the vertices. We use a finite element model as example. After a numerical comparison of performance against other partition methods, using the software Scotch, we draw some conclusions.

2.4.1 Graphs and Matrices. Examples

A graph $G = (V,E)$ consists of a set V of *vertices*, and a set E of *edges*, being each edge a set $\{u,v\}$ of two vertices $u,v \in V$. Each edge $\{u,v\}$, also noted $u \sim v$, is said that joins u and v. Note that this structure does not models loops or directed arrows.

A *weight on edges* is a map

$$w : E \to \mathbb{R}.$$

The weight of the edge $u \sim v$ is denoted $w(u,v)$. If a weight on the edges is not specified, implicitly the constant unit weight must be considered (that is, $w(u,v) = 1$ for each $(u \sim v) \in E$).

The *degree* of a vertex is the number of vertices adjacent to it.

A *potential on vertices* is a map

$$p : V \to \mathbb{R}.$$

The set of all potentials (that is, of all functions $V \to \mathbb{R}$) is denoted $\mathbb{R}^V$.

We choose an ordering of the set of vertices, $V = \{v_1, \ldots, v_n\}$. The *adjacency matrix* of G (for this conventional ordering) is the $n \times n$ symmetric matrix A:

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & & a_{2n} \\ \vdots & & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \text{ with } a_{ij} = \begin{cases} 1 & \text{if } v_i \sim v_j \\ 0 & \text{if not} \end{cases} \quad \text{for } i,j \in \{1,\ldots,n\}.$$

For a edge weight w, its *weighted adjacency matrix* is A_w with entries a_{ij} where

$$a_{ij} = \begin{cases} w(v_i,v_j) & \text{if } v_i \sim v_j \\ 0 & \text{if not} \end{cases}$$

The adjacency matrix is the matrix of the constant unit weight $w(u,v) = 1$ if $u \sim v$. We will consider mainly *positive edge weights* (that is, weigths w such that $w(u,v) > 0$ for each $(u \sim v) \in E$), with the notable exception of the Laplacian.

We represent a vertex potential $p : V \to \mathbb{R}$ as the vector $p = (p(v_1), p(v_2), \ldots, p(v_n))$. A weighted adjacency matrix A_w operates in $\mathbb{R}^V$, the set of vertex potentials, as a matrix multiplication.

$$\begin{aligned} A_w : \mathbb{R}^V &\longrightarrow \mathbb{R}^V \\ p &\longmapsto A_w p \end{aligned}$$

That is, the vector $A_w p$ has as j-entry the value $\sum_{j=0}^{n} a_{ij} p(v_j)$.

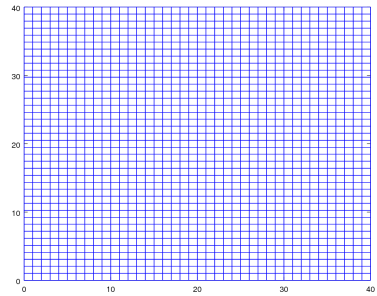


Figure 2.15: Square mesh of size 40×41

To give an intuitive interpretation of this setting, we consider a easily visualizable graph: the vertices are a square lattice of dots, and four edges join each one with those placed up, down, left and right (three edges for lateral vertices and two for the corners, figure 2.15). This type of graph is used in finite elements computations. It is symmetric.

As example of weight in this graph, let's take that each edge has weight one, so $A_w = A$ is the adjacency matrix. As an example of potential p_0, we consider that $p_0(v_0) = 1$ in one vertex v_0, and $p_0(v) = 0$ in the other ones, $v \neq v_0$. The application of A to that potential, Ap_0, transfers the value 1 to the vertices adjacent to v_0. That is, $p_1 = Ap_0$ takes the value 1 in vertexes adjacent to v_0, and 0 in others. A second application $p_2 = A^2 p_0$ widens the circle of influence: $p_2(v)$ is the number of paths of two edges from v_0 to v. The iterated application $p_k = A^k p_0$ produces a sequence $p_0, p_1, p_2, \ldots$, in a transfer process. We can assign to the sum of potential $\sum_{i=0}^{n} p_k(v_i)$ the meaning of the total amount of material that comprises p_k. In the process induced by A the total amount of material is not constant, but is multiplied by four in the majority of the vertices, the inner vertices. Therefore is not exactly a diffusion process. Taking another weight on edges, being $w(u,v)$ the inverse of the number of arrows that come out of u, it can be seen that $\sum_{i=0}^{n} p_k(v_i)$ (with $p_k = A_w^k p_0$) is constant, and the process is properly a diffusion process.

It is pertinent to mention iterations in the above example because the eigenvalues p are those potentials that verify $Ap = \lambda p$ (equivalently $A^n p = \lambda^n p$). And they are precisely the potentials invariant (except for a factor λ^n) under iterations of A.

2.4.2 Laplacian and Partitions

A *partition of a set* V is an array (V_1, V_2) of two subsets of V such that

$$V_1 \cup V_2 = V \text{ and } V_1 \cap V_2 = \emptyset.$$

A *partition of a graph* $G = (V, E)$ is a partition of the underlying set of vertices. An edge $u \sim v$ in G is *cut by a partition* (V_1, V_2) if $u \in V_1$ and $v \in V_2$ or vice versa ($u \in V_2$

and $v \in V_1$). If the graph is weighted, the *total weight of the cut*, or total cut, is $\text{cut}(V_1, V_2) = \sum_{\substack{u \in V_1 \\ v \in V_2}} w(u,v)$.

If there are several partitions in a graph, usually is preferable that which minimum number of cuts (or total cut, if weighted). We are interested in partitions with minimal cut, but with balanced number of vertices, that is $\mid V_1 \mid = \mid V_2 \mid$ (if $\mid V \mid$ is even, $\mid V_1 \mid = \mid V_2 \mid \pm 1$ if it is odd). We express the combinatorial problem of finding these partitions using Linear Algebra, in particular the spectrum (that is, eigenvalues and eigenvectors[18]) of the Laplacian matrix, later defined.

Let us suppose given an order $V = \{v_1, v_2, \ldots, v_n\}$ in the set of vertices. A vector $x = (x_1, \ldots, x_n)$ corresponding to a potential of $\mathbb{R}^V$ has an entry x_i for each v_i. The *characteristic vector* c_S of a set $S \in V$ is $c_S = (c_1, \ldots, c_n)$ with:

$$c_i = \begin{cases} 1 \text{ if } v_i \in S \\ 0 \text{ if } v_i \notin S \end{cases}$$

Sometimes is preferable to use other values than 0 or 1 in the vector expression of a combinatorial object like a subset or partition [116]. For two real values b_1, b_2, the (b_1, b_2)-*indicator vector* of a partition (V_1, V_2) is the vector $(x_1, \ldots, x_n)$ with

$$x_i = \begin{cases} b_1 \text{ if } v_i \in V_1 \\ b_2 \text{ if } v_i \in V_2 \end{cases}$$

For example, the (0,1)-indicator is the characteristic of the second set of the partition. We use mainly (1,-1)-indicators.

The following proposition summarizes some graph and combinatorial properties expressed in Linear Algebra language. We denote with a dot $\cdot$ the inner product in $\mathbb{R}^V$, and with $\mathbf{1} = (1, \ldots, 1)$ the vector all whose entries are 1. The *degree* of a vertex $u \in V$ is the cardinal of the set $\{v \in V \text{ such that } u \sim v\}$, that is, the number of vertices adjacent to u. The *degree vector* is $(d_1, d_2, \ldots, d_n)$ where d_i is the degree of v_i.

Proposition 1. *Let $G = (V, E)$ a graph of adjacency matrix A. For two sets $S, T \subset V$ of characteristics c_S, c_T:*

1. *$\mathbf{1} \cdot c_S$ is the cardinal of S, that is, $\mathbf{1} \cdot c_S = \mid S \mid$. Also $c_S \cdot c_S = \mid S \mid$.*
2. *$c_S \cdot c_T = \mid S \cap T \mid$.*
3. *The vector $A\mathbf{1}$ has, in the i-th entry, the degree of v_i, that is, $A\mathbf{1}$ is the degree vector*

$$A\mathbf{1} = (d_1, d_2, \ldots, d_n).$$

Also $\mathbf{1} \cdot A\mathbf{1} = \sum_i d_i$.

4. *Ac_S has, in entry i-th, the number of edges to v_i from a vertex in S. That is, calling $S \sim v = \{s \in V \text{ such that } s \sim v \text{ and } s \in S\}$,*

$$Ac_S = (x_1, x_2, \ldots, x_n) \text{ with } x_i = \mid S \sim v_i \mid$$

[18] We recall, that, given a matrix A, λ is an eigenvalue of A if there exist an vector v such that $Av = \lambda v$. In this case, v is the associated eigenvector of λ.

If A_w is a weighted adjacency matrix, the i-th entry of $A_w c_S$ is the sum of the weight of the edges of the form $s \sim v_i$ with $s \in S$. That is,

$$A_w c_S = (x_1, x_2, \ldots, x_n) \text{ with } x_i = \sum_{u \in S \sim v_i} w(u)$$

Proof. It is easy to do the computations for these claims from the definitions. For example, for *c*), we have that the *i*-th entry of $A\mathbf{1}$ is $\sum_{j=0} n a_{ij} \cdot 1$. As a_{ij} is 1 if $v_i \sim v_j$ (and 0 in other case), then $\sum_{j=0}^{n} a_{ij} = \sum_{j|v_i \sim v_j} 1$, that is precisely the number of vertices adjacent to v_i. □ □

If we call D_g the matrix with the degree vector in the diagonal and zero off-diagonal:

$$D_g = \begin{pmatrix} d_1 & 0 & \cdots & 0 \\ 0 & d_2 & & 0 \\ \vdots & & \ddots & \vdots \\ 0 & 0 & \cdots & d_n \end{pmatrix}$$

from a similar easy computation we have $\mathbf{1} \cdot D_g \mathbf{1} = \sum_i d_i$. For any partition, if x is it (1,-1)-indicator, we also have $x \cdot D_g x = \sum_i d_i$, because the minus signs compensate in the entries where them appear.

In this context, it is traditional to define the Laplacian matrix L as

$$L = D_g - A.$$

See for example [118] or [119]. The rationale behind this definition is the following relationship between the cut of a partition and the transform by L of its characteristic vectors.

Theorem 1. *For a partition (V_1, V_2), of (1,-1)-indicator x, we have;*

$$cut(V_1, V_2) = \frac{x \cdot Lx}{4}$$

Proof. For a partition (V_1, V_2), being c_1 and c_2 characteristic vectors of its sets, the sum of the weight of the edges $u \sim v$ with $u \in V_1$ and $v \in V_2$ is $c_1 \cdot A c_2$. Therefore, $\text{cut}(V_1, V_2) = c_1 \cdot A c_2$. By the symmetry of A, it is also equal to $c_2 \cdot A c_1$.

If x is the (1,-1)-indicator of (V_1, V_2), then $x = c_1 - c_2$, and:

$$\begin{aligned} x \cdot Ax = (c_1 - c_2) \cdot A(c_1 - c_2) = \\ = c_1 \cdot Ac_1 + c_2 \cdot Ac_2 - (c_1 \cdot Ac_2 + c_2 \cdot Ac_1) = \\ = c_1 \cdot Ac_1 + c_2 \cdot Ac_2 - 2\text{cut}(V_1, V_2) \end{aligned}$$

Besides, as $c_1 + c_2 = \mathbf{1}$, that is $c_1 = \mathbf{1} - c_2$, then $c_1 \cdot Ac_1 = c_1 \cdot A(\mathbf{1} - c_2) = c_1 \cdot A\mathbf{1} - c_1 \cdot Ac_2$. Likewise $c_2 \cdot Ac_2 = c_2 \cdot A\mathbf{1} - c_2 \cdot Ac_1$, hence

$$\begin{aligned} c_1 \cdot Ac_1 + c_2 \cdot Ac_2 = c_1 \cdot A\mathbf{1} - c_1 \cdot Ac_2 + c_2 \cdot A\mathbf{1} - c_2 \cdot Ac_1 = \\ = (c_1 + c_2) \cdot A\mathbf{1} - (c_1 \cdot Ac_2 + c_1 \cdot Ac_2) = \mathbf{1} \cdot A\mathbf{1} - 2\text{cut}(V_1, V_2) \end{aligned}$$

Hence

$$x \cdot Ax = c_1 \cdot Ac_1 + c_2 \cdot Ac_2 - 2\text{cut}(V_1, V_2) =$$
$$= \mathbf{1} \cdot A\mathbf{1} - 2\text{cut}(V_1, V_2) - 2\text{cut}(V_1, V_2) = \sum_i d_i - 4\text{cut}(V_1, V_2)$$

That is, $4\text{cut}(V_1, V_2) = \sum_i d_i - x \cdot Ax$. As $\sum_i d_i = x \cdot D_g x$, we can express $\sum_i d_i - x \cdot Ax = x \cdot D_g x - x \cdot Ax = x \cdot Lx$. Therefore

$$\text{cut}(V_1, V_2) = \frac{x \cdot Lx}{4}$$

□ □

We have deduced this well known identity in matrix form, instead of summatory form as usual, to avoid the index chasing. This way also makes explicit the role of the values b_1, b_2 using in indicators (as it is done in [116]). For example if x is a $(\frac{1}{2}, -\frac{1}{2})$-indicator of (V_1, V_2), then $\text{cut}(V_1, V_2) = x \cdot Lx$. In general if x is a (b_1, b_2)-indicator the cost of its cut is $\frac{x \cdot Lx}{(b_2 - b_1)^2}$. This deduction also shows the role of the diagonal degree matrix.

In addition to the expression of cost as a bilineal form with matrix L, we express the requirement that the partition (V_1, V_2) be balanced as $\mathbf{1} \cdot x = 0$. Hence, the problem of finding the balanced partition of minimal cost is the following problem of combinatorial optimization:

$$\begin{array}{ll} \underset{x}{\text{Minimize}} & x \cdot Lx \\ \text{subject to} & x_i = \pm 1,\ i = 1, \ldots, n \\ & \mathbf{1} \cdot x = 0. \end{array}$$

To solve this combinatorial problem it is customary to relax the restrain $x_i = \pm 1$. The relaxed problem has several features that ease its numerical resolution: L is symmetric, hence its eigenvalues are real and there are a orthonormal basis of eigenvectors [120]. Besides, $\mathbf{1}$ is a eigenvector of eigenvalue 0, because $D_g\mathbf{1} - A\mathbf{1} = 0$. Also, the eigenvalues are non-negative [121] $0 = \mu_0 \leq \mu_1 \leq \cdots \leq \mu_{n-1}$ (numbering then without multiplicity $0 = \lambda_0 < \lambda_1 < \cdots < \lambda_k$). These features of L are generally deduced from its expression as summatory of squares, that we have avoided. Here we derive them from standard facts of numerical matrix analysis.

The main result in numerical eigenvalue computation is the Min-max Theorem [117]. In our case, this implies $\lambda_1 = \min_{x \neq 0} \frac{x \cdot Lx}{x \cdot x}$, the minimum is reached in a vector x_1 of norm 1, that is eigenvalue for λ_1. As the eigenvectors of different eigenvalues are orthogonal, $\mathbf{1} \cdot x_1 = 0$. That is x_1 is a solution of the relaxed problem.

The first non-null eigenvalue λ_1 is the Fiedler value and its eigenvector x_1 is the Fiedler vector, by [122]. It solves the relaxed problem, numerically with computational complexity of $O(n^3)$. Rounding x_i gives a $(1, -1)$-indicator of a partition. The solution of the relaxed problem is an approximation of the combinatorial problem. This problem is *NP*-hard [115], hence the interest of a relaxed approximation. A bound of the error of this approximation, involving λ_1, is given by the bound of Mohar

[123]. Being Δ_g the maximum vertex degree of G, $\Phi(G)$ the cost of the minimal cut, and $Sp(G)$ the cut obtained by Fiedler eigenvector:

$$\Phi(G) \leq Sp(G) \leq \sqrt{\lambda_1(2\Delta_g - \lambda_1)}.$$

These properties, included the bound of Mohar, can be translated for Laplacians with vertex potential, a generalization of the Laplacian that we define in the next subsubsection, and that allows us to extend the spectral partition to unequal vertex load.

2.4.3 Laplacian with Potential of Vertex Weights

A *potential* is a function $p : V \longrightarrow \mathbb{R}$, and its diagonal form is the matrix $D_p = (d_{ij})$ with $d_{ii} = p(x_i)$, $d_{ij} = 0$ if $i \neq j$. The *Laplacian with potential p* (or p-Laplacian) is:

$$L_p = L + D_p$$

That is $L_p = D_g - A + D_p$. Some properties of the p-Laplacian are similar to those of the Laplacian.

If the potential p is non-negative, L_p has a real eigenvalue that is positive and of maximum absolute value between the eigenvalues (known as *Perron eigenvalue*). There is an eigenvector of the Perron eigenvalue that is positive (the *Perron* eigenvector ρ).

The max-min theorem for the operator L_p gives us that $\lambda_1 = \min_{\substack{x \neq 0 \\ x \cdot \rho = 0}} \frac{x \cdot L_p x}{x \cdot x}$, and the minimum is reached in its eigenvectors. Conventionally, the eigenvector of λ_1 of norm 1 and with greater number of nonnegative values is the *Fiedler* vector ϕ.

The spectral decomposition of L_p assure that $\phi \cdot \rho = 0$. This can be viewed, like in the previous Laplacian, that the positive and negative values of the Fiedler vector is an indicator of two sets of vertices that cut V in two parts of equal sum of Perron values.

To build potential p in such a way that the Perron vector ρ have predetermined values ρ_i, we have developed the following result. $A = (a_{ij})$ is the adjacency matrix and $(A\rho)_i$ the i-th component of the vector $A\rho$.

Theorem 2. *The potential*

$$p(x_i) = 1 + a_{ii} - \frac{(A\rho)_i}{\rho_i}$$

has ρ as Perron vector.

By the above discussion, the p-Laplacian of this potential has a Fiedler vector orthogonal to the Perron vector (that is, it produces a partition in parts of equal total load at the vertices), and that in addition, by the extremal Max-min property, minimizes the cost of communications in the relaxed problem.

Also, by taking this Fiedler vector as an approximation to the combinatorial solution, that is, the unrelaxed problem, the error can be bounded with an expression

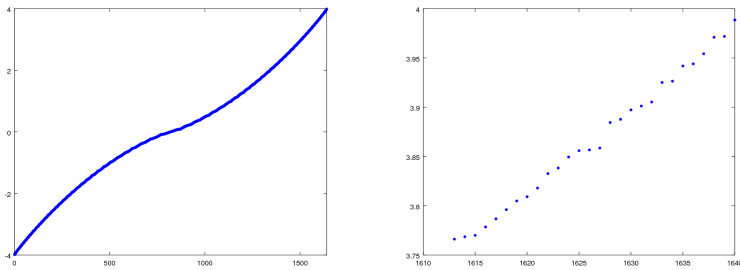


Figure 2.16: Eigenvalues of the mesh. The area of main eigenvalues is zoomed.

similar to that of Mohar. Being, as above, $\Phi(G)$ the cost of the minimal cut, and $Sp_p(G)$ the cut obtained by Fiedler eigenvector of L_p:

$$\Phi(G) \leq Sp_p(G) \leq \sqrt{2\lambda_1 \max_i \frac{d_i - a_{ii}}{\rho_i}}$$

With these results, we can mimic the traditional partition techniques, but incorporating the load at the vertices. In addition, the division into two parts can be done by assigning unequal proportions of the load (for example 30%-70%).

The unequal load has been addressed in the literature either by modeling as a generalized eigenvalue problem [124], or by using several eigenvectors [125]. Both approaches have their own drawbacks [115]. For our purposes, the main disadvantage is that the vertex load is not embodied in the Laplacian. As we want to consider mappings from application graph to machine graph, the loads should be included in the model.

2.4.4 Mesh graph

In this subsection first we give an example of spectral decomposition of the mesh graph of figure 2.15 . We will see that the eigenvector of the dominant eigenvalue partitions the square. In this example of Cartesian graph, the adjacency matrix has side $40 \cdot 41 = 1640$. The 1640 eigenvalues, in increasing order, are plotted in figure 2.16 and, as the matrix is symmetric, the Jordan form is diagonal. [126].

Each vector is a value in every vertex, so we can plot it as a z value of height above the xy plane were the square lattice is displayed. With this convention, the first and second eigenvectors (with respect the ordering eigenvalues) can be seen in figure 2.17.

Note that these are the eigenvectors of the adjacency matrix, not the Laplacian. However, the first eigenvector, bell-shaped and positive, is symmetrically posed in the square. We consider that each vertex has a load proportional to the corresponding entry of this first eigenvector. The second eigenvector, orthogonal to it, has positive and negative entries defining a partition of the mesh, whose two parts are equal in total load (measured by the first eigenvector).

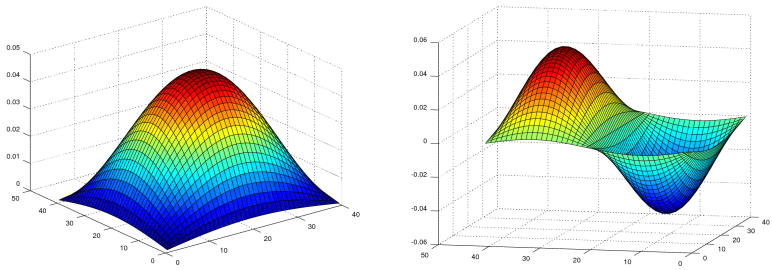


Figure 2.17: The first two eigenvectors of the mesh.

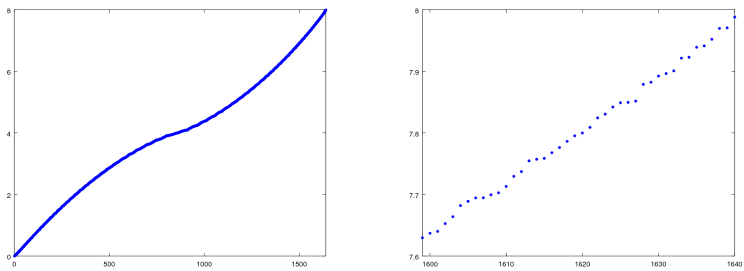


Figure 2.18: Laplacian eigenvalues of square lattice. The area of main eigenvalues is zoomed.

In the case of the Laplacian, first and second eigenvectors (Perron and Fiedler) are in figure 2.18. And these are also, as in the adjacency matrix, one positive and the other partitioning the vertices in two sets of vertices. The sets have equal load, measured by the first eigenvector, that being constant gives us equal number of vertexes in each part.

The adjacency matrix and the Laplacian matrix have been taken as examples. They differ only in the diagonal, so the adjacency matrix is a particular type of *p*-

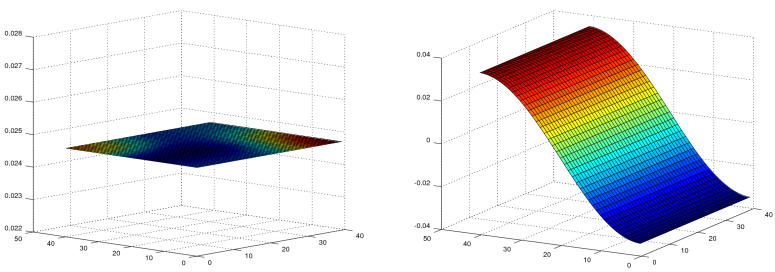


Figure 2.19: Eigenvectors of square lattice.

Laplacian: one that has as potential the degree at each vertex. This example has been considered because it is easy to represent the eigenvectors, and to see that the first eigenvector (Perron in the case of the adjacency matrix) corresponds to a load at each vertex (uniform in the case of Laplacian).

2.4.5 Numerical Experiment

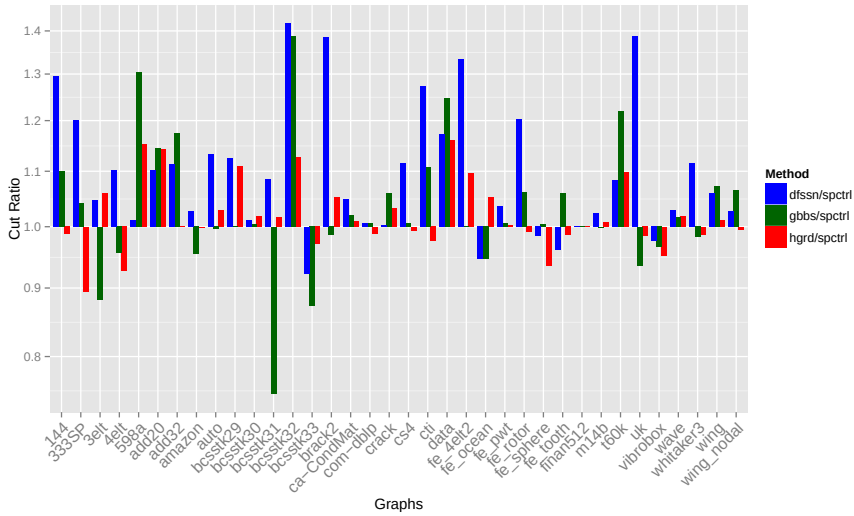


Figure 2.20: Ratio of cut improvement of the spectral method against others.

In this subsection we describe a comparison, using the partitioning software Scotch [127], of the spectral method described above against other method of graph partition. We have integrated the spectral bipartition method (with vertex loads) in Scotch. We resorted to the LAPACK library [128] for the eigenvector computation due to their availability, but it is preferable to use libraries specialized in sparse matrices, such as [129]. The Fiedler eigenvector is used as an initial method in a multilevel approach (see [115] for technical background). In Fig 2.20, we do a comparison with some of the initial methods present in Scotch (Diffusion, Gibbs-Pole-Stockmeyer, H-greedy) over the graphs of the Walshaw collection [130], and also some bigger graphs from the dataset of [131]. The cut produced with the spectral method, for bipartitions, is about 10% better than the other methods. There are some cases where the spectral method behaves equal or worse. In the figure, we plot the value "cost of the cut of other method / cost of the spectral cut ", hence a value of 1 means equal cost, greater than 1 means that spectral has lower cost.

The tests are meant to compare initial methods, leaving the contribution of coarsening-uncoarsening as equal as possible between methods. To be precise, the tests include, for each graph of Walshaw benchmark (excluding fe_body and MemPlus, which are not connected) five different coarsening processes: up to 64, or 128, or 256, or 512, or 1024 vertices. Then each of these initial graph is bipartitioned by

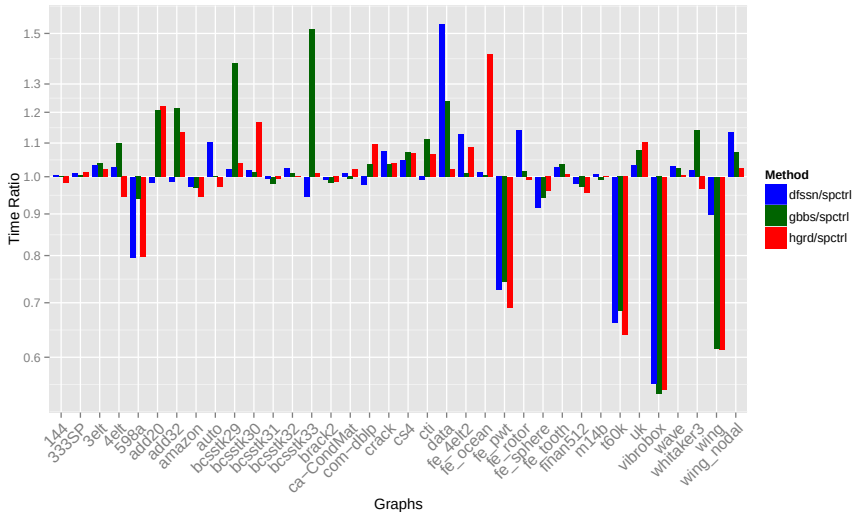


Figure 2.21: Ratio of time improvement of the spectral method against others

diffusion, Gibbs, hgreedy and spectral methods, and then are uncoarsened with the Fiduccia-Mattheyses algorithm [132]. The final measure is the cost of the cut obtained. Unit loads and weights of the input graph are considered. In the plot we use the mean of the five measures for each pair graph-method.

In Figure 2.21 we also compare the timing of running each method compared to the spectral method. We plot the ratio of the time of three other ones compared to the spectral method (hence the higher, the faster the spectral method). We see balanced results here where in some cases the spectral method is much faster but for some graphs (e.g. vibrobox) the ratio is lower than 0.6. However, the geometric mean of the ratio is 0.986, which means that, on average, the spectral method is comparable in terms of timings to the other methods. This is a good result as computing the eigenvectors can be very long. Actually, the coarsening phase that happens only in the other methods takes also a lot of time that has a strong impact on the timings.

2.4.6 Conclusions

The bipartition through the Fiedler eigenvector can be done by incorporating the loads of the vertices in the model of the graph, without the need to introduce these loads a posteriori in the resolution process. The classical techniques of analysis of the error of the approximation can be generalized to this new approach. Bipartition can be done in unbalanced parts, with a predetermined ratio, minimizing communications. However, extending this scheme to partitions in three or more parts does not seem straightforward. The numerical results, in comparison with the more usual methods, are favorable to the spectral method, especially in graphs of a certain type, such as those from social networks.

2.5 Summary

This chapter has shown some works related to UCS and the variety of systems that cloud be integrated in those complex environments. However, there are still several research topics and challenges that must be faced to cope with such complexity. Below, some of them are shown

Cloud/Fog/Dew Big Data Computing: In the future the highest opportunities lie in the availability of massive scale cloud infrastructure which will be omnipresent. To effectively use these available resources, massively federated and scalable software with orchestration through network awareness will be necessary. As an extension of links between UCS and Clouds, data access models for data mining in Exascale systems will be a key research topic. The integration will be between Cloud systems but also Fog and future type of infrastructure, leading to need on machine-to-machine computing and Cloud computing integration. Heterogeneity of such system will continue to increase, leading to the need to be able to integrate warehouse-scale computing using purpose-designed chips. Integrating the lowest, Dew-level devices will present additional challanges due to the extreme quantities of Physical Edge Devices, their severely low processing power and communication means, and the huge amounts of data generated.

HPC: One of the key point will be the availability of programming abstractions for the different fields of Exascale such as data analysis, machine learning, scientific computing, Big Data management, smart cities, that will be based on asynchronous algorithms for overlapping communication and computation. To reach this overlap, parallel applications (such as the MPI-based one) will need to be optimized using platform topology and performance information. One crucial research topic will be programmability of UCS as applications will run millions of parallel execution flows. New workflow programming for very large plate forms will be needed. But interoperability and sustainability will only be reached when code will be prevented to be platform specific and still efficient on different platforms. From a broader point of view, the scale of UCS will lead to Supercomputing on demand leading to a better use of the vast amount of available resources. The efficiency will be linked to researches on performance evaluation, modeling and optimization of data parallel applications on heterogeneous HPC platforms. Management of such large distributed systems will be based on future researches on complex systems modelling, self-organizing systems and cellular automata.

Application-driven topics: With the aim of harnessing the power of UCS, scientific community will be able to improve dramatically the quality of models. One key example will be the research focus on meteorology beyond wind simulation (Interfacing between different software packages and data formats, necessary for integration of simulations for complex tasks). New tools will be needed to use UCS for scientists from diverse fields, but tools only available to computer scientists will be needed such as the Hardware/Software Co-design models to guide together the development of hardware and software infrastructure.

Tool-driven: Several tools will be needed to use efficiently UCS. Some tools can be provided by software, but also abstract models and new programming paradigms helping programmers to better use the available resources are helpful. Due to the scale of the systems, one key element will be resource-efficient models for automatic recovery from minute-to-minute failures. As security is often forgotten by programmers, software-defined security models will be needed on large scale distributed infrastructure to simplify its usage. One way to increase security and privacy will be to create new secure Privacy-Preserving data management algorithms such as machine learning. To address code sustainability and adaptation evolution on code production is needed such as source-to-source translators and MDE (Model Driven Engineering) in order to adapt to the underlying hardware.

In order to support some of those challenges, several breaktroughs are expected in order to reach proper support for programmers and users in the Ultrascale context as described in the NESUS research roadmap[2]:

Improve the programmability of complex systems Due to the size of these systems, it is no more possible for the programmer to have a precised and detailed global view of the state of its application. Thus he needs to have support from programming frameworks to simplify this view;

Break the wall between runtime and programming frameworks Exascale sytems are so complex that runtime need high level information from the programmers and the programmer need some information on the runtime to understand how to harness its power;

Enabling behavioral sensitive runtime. Runtime cannot run application as black boxes anymore as large scale systems are composed of a large number of interconnected elements. Network profile must be known to reduce impact on neighboor applications for example.

References

[1] Da Costa G, Fahringer T, Gallego JAR, et al. Exascale machines require new programming paradigms and runtimes. Supercomputing frontiers and innovations. 2015;2(2):6–27.

[2] Sousa L, Kropf P, Kuonene P, et al. A Roadmap for Research in Sustainable Ultrascale Systems. University Carlos III of Madrid; 2017. 0. Available from: https://www.irit.fr/~Georges.Da-Costa/NESUS-research_roadmap.pdf.

[3] Fortune S, Wyllie J. Parallelism in Random Access Machines. In: Proceedings of the Tenth Annual ACM Symposium on Theory of Computing. STOC '78. New York, NY, USA: ACM; 1978. p. 114–118. Available from: http://doi.acm.org/10.1145/800133.804339.

[4] Valiant LG. A Bridging Model for Parallel Computation. Commun ACM. 1990 Aug;33(8):103–111. Available from: http://doi.acm.org/10.1145/79173.79181.

[5] Culler D, Karp R, Patterson D, et al. LogP: towards a realistic model of parallel computation. In: Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming. PPOPP '93. New York, NY, USA: ACM; 1993. p. 1–12.

[6] Gautier T, Besseron X, Pigeon L. KAAPI: A Thread Scheduling Runtime System for Data Flow Computations on Cluster of Multi-processors. In: Proceedings of the 2007 International Workshop on Parallel Symbolic Computation. PASCO '07. ACM; 2007. p. 15–23.

[7] Augonnet C, Thibault S, Namyst R, et al. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. Concurr Comput : Pract Exper. 2011 Feb;23(2):187–198.

[8] Bosilca G, Bouteiller A, Danalis A, et al. DAGuE: A Generic Distributed DAG Engine for High Performance Computing. In: 2011 IEEE IPDPSW; 2011. p. 1151–1158.

[9] Bui TN, Jones C. A heuristic for reducing fill-in in sparse matrix factorization. In: Proceedings of the 6th SIAM Conference on Parallel Processing for Scientific Computing. SIAM; 1993. .

[10] Hendrickson B, Leland R. A Multilevel Algorithm for Partitioning Graphs. In: Proceedings of the 1995 ACM/IEEE Conference on Supercomputing. Supercomputing '95. ACM; 1995. Available from: http://doi.acm.org/10.1145/224170.224228.

[11] Catalyurek U, Aykanat C. Decomposing Irregularly Sparse Matrices for Parallel Matrix-Vector Multiplication. In: Proceedings of the Third International Workshop on Parallel Algorithms for Irregularly Structured Problems. IRREGULAR '96. Springer-Verlag; 1996. p. 75–86.

[12] Hendrickson B, Kolda TG. Partitioning Rectangular and Structurally Unsymmetric Sparse Matrices for Parallel Processing. SIAM Journal on Scientific Computing. 2000;21(6):2048–2072.

[13] Cierniak M, Zaki MJ, Li W. Compile-time scheduling algorithms for a heterogeneous network of workstations. The Computer Journal. 1997;40(6):356–372.

[14] Beaumont O, Boudet V, Rastello F, et al. Matrix multiplication on heterogeneous platforms. Parallel and Distributed Systems, IEEE Transactions on. 2001;12(10):1033–1051.

[15] Kalinov A, Lastovetsky A. Heterogeneous Distribution of Computations Solving Linear Algebra Problems on Networks of Heterogeneous Computers. Journal of Parallel and Distributed Computing. 2001;61:520–535.

[16] Lastovetsky AL, Reddy R. Data partitioning with a realistic performance model of networks of heterogeneous computers. In: Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International. IEEE; 2004. p. 104.

[17] Lastovetsky A, Twamley J. Towards a realistic performance model for networks of heterogeneous computers. In: High Performance Computational Science and Engineering. Springer; 2005. p. 39–57.

[18] Lastovetsky A, Reddy R. Data partitioning with a functional performance model of heterogeneous processors. International Journal of High Performance Computing Applications. 2007;21(1):76–90.

[19] Lastovetsky A, Reddy R. Data Distribution for Dense Factorization on Computers with Memory Heterogeneity. Parallel Computing. 2007 Dec;33(12).

[20] Lastovetsky A, Szustak L, Wyrzykowski R. Model-based optimization of EULAG kernel on Intel Xeon Phi through load imbalancing. IEEE Transactions on Parallel and Distributed Systems. 2017;28(3):787–797.

[21] Lastovetsky A, Reddy R. New Model-Based Methods and Algorithms for Performance and Energy Optimization of Data Parallel Applications on Homogeneous Multicore Clusters. IEEE Transactions on Parallel and Distributed Systems. 2017;28(4):1119–1133.

[22] Alexandrov A, Ionescu MF, Schauser KE, et al. LogGP: incorporating long messages into the LogP model - one step closer towards a realistic model for parallel computation. In: Proc. of the seventh annual ACM symposium on Parallel algorithms and architectures. SPAA '95. NY, USA; 1995. p. 95–105.

[23] Kielmann T, Bal HE, Verstoep K. Fast Measurement of LogP Parameters for Message Passing Platforms. In: Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing. IPDPS '00. London, UK, UK: Springer-Verlag; 2000. p. 1176–1183.

[24] Bosque JL, Perez LP. HLogGP: a new parallel computational model for heterogeneous clusters. In: Cluster Computing and the Grid, 2004. CCGrid 2004. IEEE International Symposium on; 2004. p. 403–410.

[25] Lastovetsky A, Mkwawa IH, O'Flynn M. An accurate communication model of a heterogeneous cluster based on a switch-enabled Ethernet network. In: Parallel and Distributed Systems, 2006. ICPADS 2006. 12th International Conference on. vol. 2; 2006. p. 6 pp.–.

[26] Cameron KW, Ge R, Sun XH. $\log_m P$ and $\log_3 P$: Accurate Analytical Models of Point-to-Point Communication in Distributed Systems. Computers IEEE Transactions on. 2007;56(3):314–327.

[27] Rico-Gallego JA, Díaz-Martín JC, Lastovetsky AL. Extending τ–Lop to model concurrent MPI communications in multicore clusters. Future Generation Computer Systems. 2016;61:66 – 82.

[28] Rico-Gallego JA, Lastovetsky AL, Díaz-Martín JC. Model-Based Estimation of the Communication Cost of Hybrid Data-Parallel Applications on Heterogeneous Clusters. IEEE Transactions on Parallel and Distributed Systems. 2017 Nov;28(11):3215–3228.

[29] Clarke D, Zhong Z, Rychkov V, et al. FuPerMod: a software tool for the optimization of data-parallel applications on heterogeneous platforms. The Journal of Supercomputing. 2014;69:61– 69.

[30] Beaumont O, Boudet V, Rastello F, et al. Matrix Multiplication on Heterogeneous Platforms. IEEE Trans Parallel Distrib Syst. 2001 Oct;12(10):1033–1051. Available from: http://dx.doi.org/10.1109/71.963416.

[31] Malik T, Rychkov V, Lastovetsky A. Network-aware optimization of communications for parallel matrix multiplication on hierarchical HPC plat-

forms. Concurrency and Computation: Practice and Experience. 2016 03/2016;28:802–821.

[32] Bellosa F. The benefits of event: driven energy accounting in power-sensitive systems. In: Proceedings of the 9th workshop on ACM SIGOPS European workshop: beyond the PC: new challenges for the operating system. ACM; 2000. .

[33] Isci C, Martonosi M. Runtime power monitoring in high-end processors: Methodology and empirical data. In: 36th annual IEEE/ACM International Symposium on Microarchitecture. IEEE Computer Society; 2003. p. 93.

[34] Economou D, Rivoire S, Kozyrakis C, et al. Full-system power analysis and modeling for server environments. In: In Proceedings of Workshop on Modeling, Benchmarking, and Simulation; 2006. p. 70–77.

[35] Basmadjian R, Ali N, Niedermeier F, et al. A methodology to predict the power consumption of servers in data centres. In: 2nd International Conference on Energy-Efficient Computing and Networking. ACM; 2011. .

[36] Bircher WL, John LK. Complete System Power Estimation Using Processor Performance Events. IEEE Transactions on Computers. 2012 Apr;61(4):563–577.

[37] Hong H Sunpyand Kim. An Integrated GPU Power and Performance Model. SIGARCH Comput Archit News. 2010 Jun;38(3):280–289.

[38] Song S, Su C, Rountree B, et al. A Simplified and Accurate Model of Power-Performance Efficiency on Emergent GPU Architectures. In: 27th IEEE International Parallel & Distributed Processing Symposium (IPDPS). IEEE Computer Society; 2013. p. 673–686.

[39] CUPTI. CUDA Profiling Tools Interface; 2018. Available from: https://developer.nvidia.com/cuda-profiling-tools-interface.

[40] Wang H, Cao Y. Predicting power consumption of GPUs with fuzzy wavelet neural networks. Parallel Computing. 2015 May;44:18–36.

[41] Top500. Top 500. The List - November 2017; 2018. Available from: https://www.top500.org/lists/2017/11/.

[42] Shao YS, Brooks D. Energy Characterization and Instruction-level Energy Model of Intel's Xeon Phi Processor. In: Proceedings of the 2013 International Symposium on Low Power Electronics and Design. ISLPED '13. IEEE Press; 2013. .

[43] Ou J, Prasanna VK. Rapid energy estimation of computations on FPGA based soft processors. In: SOC Conference, 2004. Proceedings. IEEE International; 2004. .

[44] Wang X, Ziavras SG, Hu J. System-Level Energy Modeling for Heterogeneous Reconfigurable Chip Multiprocessors. In: 2006 International Conference on Computer Design; 2006. .

[45] Al-Khatib Z, Abdi S. Operand-Value-Based Modeling of Dynamic Energy Consumption of Soft Processors in FPGA. In: International Symposium on Applied Reconfigurable Computing. Springer; 2015. p. 65–76.

[46] Lively C, Wu X, Taylor V, et al. Power-aware predictive models of hybrid (MPI/OpenMP) scientific applications on multicore systems. Computer Science-Research and Development. 2012;27(4):245–253.

[47] PAPI. Performance Application Programming Interface 5.6.0; 2018. Available from: http://icl.cs.utk.edu/papi/.

[48] Bosilca G, Ltaief H, Dongarra J. Power profiling of Cholesky and QR factorizations on distributed memory systems. Computer Science-Research and Development. 2014;29(2):139–147.

[49] Witkowski M, Oleksiak A, Piontek T, et al. Practical Power Consumption Estimation for Real Life HPC Applications. Future Gener Comput Syst. 2013 Jan;29(1).

[50] Jarus M, Oleksiak A, Piontek T, et al. Runtime power usage estimation of HPC servers for various classes of real-life applications. Future Generation Computer Systems. 2014;36.

[51] Lastovetsky A, Manumachu RR. New Model-Based Methods and Algorithms for Performance and Energy Optimization of Data Parallel Applications on Homogeneous Multicore Clusters. IEEE Transactions on Parallel and Distributed Systems. 2017;28(4):1119–1133.

[52] McCullough JC, Agarwal Y, Chandrashekar J, et al. Evaluating the Effectiveness of Model-based Power Characterization. In: Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference. USENIXATC'11. USENIX Association; 2011. .

[53] Hackenberg D, Ilsche T, Schöne R, et al. Power measurement techniques on standard compute nodes: A quantitative comparison. In: Performance analysis of systems and software (ISPASS), 2013 IEEE international symposium on. IEEE; 2013. p. 194–204.

[54] Rotem E, Naveh A, Ananthakrishnan A, et al. Power-Management Architecture of the Intel Microarchitecture Code-Named Sandy Bridge. IEEE Micro. 2012 March;32(2):20–27.

[55] O'Brien K, Pietri I, Reddy R, et al. A Survey of Power and Energy Predictive Models in HPC Systems and Applications. ACM Computing Surveys. 2017;50(3). Available from: http://doi.org/10.1145/3078811.

[56] Shahid A, Fahad M, Reddy R, et al. Additivity: A Selection Criterion for Performance Events for Reliable Energy Predictive Modeling. Supercomputing Frontiers and Innovations. 2017;4(4).

[57] Treibig J, Hager G, Wellein G. Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In: Parallel Processing Workshops (ICPPW), 2010 39th International Conference on. IEEE; 2010. p. 207–216.

[58] Mobius C, Dargie W, Schill A. Power Consumption Estimation Models for Processors, Virtual Machines, and Servers. IEEE Transactions on Parallel and Distributed Systems. 2014;25(6).

[59] Inacio EC, Dantas MAR. A Survey into Performance and Energy Efficiency in HPC, Cloud and Big Data Environments. Int J Netw Virtual Organ. 2014 Mar;14(4).

[60] Tan L, Kothapalli S, Chen L, et al. A survey of power and energy efficient techniques for high performance numerical linear algebra operations. Parallel Computing. 2014 Dec;40.

[61] Dayarathna M, Wen Y, Fan R. Data Center Energy Consumption Modeling: A Survey. IEEE Communications Surveys & Tutorials. 2016;18(1):732–794.

[62] Mezmaz M, Melab N, Kessaci Y, et al. A parallel bi-objective hybrid metaheuristic for energy-aware scheduling for cloud computing systems. Journal of Parallel and Distributed Computing. 2011;71(11):1497 – 1508.

[63] Fard HM, Prodan R, Barrionuevo JJD, et al. A Multi-objective Approach for Workflow Scheduling in Heterogeneous Environments. In: Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (Ccgrid 2012). CCGRID '12. IEEE Computer Society; 2012. p. 300–309.

[64] Beloglazov A, Abawajy J, Buyya R. Energy-aware resource allocation heuristics for efficient management of data centers for Cloud computing. Future Generation Computer Systems. 2012;28(5):755 – 768. Special Section: Energy efficiency in large-scale distributed systems.

[65] Kessaci Y, Melab N, Talbi EG. A Pareto-based Metaheuristic for Scheduling HPC Applications on a Geographically Distributed Cloud Federation. Cluster Computing. 2013 Sep;16(3):451–468.

[66] Durillo JJ, Nae V, Prodan R. Multi-objective energy-efficient workflow scheduling using list-based heuristics. Future Generation Computer Systems. 2014;36:221 – 236.

[67] Freeh VW, Lowenthal DK, Pan F, et al. Analyzing the Energy-Time Trade-Off in High-Performance Computing Applications. IEEE Trans Parallel Distrib Syst. 2007 Jun;18(6).

[68] Ahmad I, Ranka S, Khan SU. Using game theory for scheduling tasks on multi-core processors for simultaneous optimization of performance and energy. In: Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on; 2008. p. 1–6.

[69] Balaprakash P, Tiwari A, Wild SM. In: Jarvis AS, Wright AS, Hammond DS, editors. Multi Objective Optimization of HPC Kernels for Performance, Power, and Energy. Springer International Publishing; 2014. p. 239–260.

[70] Drozdowski M, Marszalkowski JM, Marszalkowski J. Energy trade-offs analysis using equal-energy maps. Future Generation Computer Systems. 2014;36:311–321.

[71] Marszalkowski JM, Drozdowski M, Marszalkowski J. Time and Energy Performance of Parallel Systems with Hierarchical Memory. Journal of Grid Computing. 2016;14(1):153–170.

[72] Reddy R, Lastovetsky A. Bi-Objective Optimization of Data-Parallel Applications on Homogeneous Multicore Clusters for Performance and Energy. IEEE Transactions on Computers. 2018;64(2):160–177.

[73] Juve G, Chervenak A, Deelman E, et al. Characterizing and profiling scientific workflows. Future Generation Computer Systems. 2013;29(3):682–692.

[74] Fahringer T, Prodan R, Duan R, et al. ASKALON: A Development and Grid Computing Environment for Scientific Workflows. In: Taylor IJ, Deelman E, Gannon DB, et al., editors. Workflows for e-Science. Springer; 2007. p. 450–471.

[75] Altintas I, Berkley C, Jaeger E, et al. Kepler: an extensible system for design and execution of scientific workflows. In: Proceedings. 16th International Conference on Scientific and Statistical Database Management, 2004.; 2004. p. 423–424.

[76] Tristan Glatard DLXP Johan Montagnat. Flexible and Efficient Workflow Deployment of Data-Intensive Applications On Grids With MOTEUR. The International Journal of High Performance Computing Applications. 2008;22(3):347–360.

[77] Taylor I, Shields M, Wang I, et al. Triana Applications within Grid Computing and Peer to Peer Environments. Journal of Grid Computing. 2003 Jun;1(2):199–217.

[78] Kacsuk P. P–GRADE portal family for grid infrastructures. Concurrency and Computation: Practice and Experience. 2011;23(3):235–245.

[79] Deelman E, Vahi K, Juve G, et al. Pegasus, a workflow management system for science automation. Future Generation Computer Systems. 2015;46:17–35.

[80] Janetschek M, Prodan R, Benedict S. A Workflow Runtime Environment for Manycore Parallel Architectures. Future Generation Computer Systems. 2017;75:330–347.

[81] Durillo JJ, Prodan R, Barbosa JG. Pareto tradeoff scheduling of workflows on federated commercial clouds. Simulation Modelling Practice and Theory. 2015;58:95–111.

[82] Arabnejad H, Barbosa JG. Budget constrained scheduling strategies for online workflow applications. In: International Conference on Computational Science and Its Applications. Springer; 2014. p. 532–545.

[83] Ullman JD. NP-complete scheduling problems. Journal of Computer and System sciences. 1975;10(3):384–393.

[84] Topcuoglu H, Hariri S, Wu MY. Performance-effective and low-complexity task scheduling for heterogeneous computing. IEEE Transactions on Parallel and Distributed Systems. 2002 3;13(3):260–274.

[85] Wieczorek M, Hoheisel A, Prodan R. Towards a General Model of the Multi-Criteria Workflow Scheduling on the Grid. Future Generations Computer Systems. 2009;25(3):237–256.

[86] Maheswaran M, Ali S, Siegel HJ, et al. Dynamic mapping of a class of independent tasks onto heterogeneous computing systems. Journal of parallel and distributed computing. 1999;59(2):107–131.

[87] Arabnejad H, Barbosa JG. List scheduling algorithm for heterogeneous systems by an optimistic cost table. IEEE Transactions on Parallel and Distributed Systems. 2014;25(3):682–694.

[88] Bittencourt LF, Sakellariou R, Madeira ER. Dag scheduling using a lookahead variant of the heterogeneous earliest finish time algorithm. In: Parallel,

Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on. IEEE; 2010. p. 27–34.

[89] Armbrust M, Fox A, Griffith R, et al. A view of cloud computing. Communications of the ACM. 2010;53(4):50–58.

[90] Leitão J, Pereira J, Rodrigues L. Epidemic Broadcast Trees. In: Proceedings of SRDS.2007; 2007. p. 301–310.

[91] Leitão J, Pereira J, Rodrigues L. HyParView: A Membership Protocol for Reliable Gossip-Based Broadcast. In: Dependable Systems and Networks, 2007. DSN '07. 37th Annual IEEE/IFIP International Conference on; 2007. p. 419–429.

[92] Shapiro M, Preguiça N, Baquero C, et al. Conflict-free replicated data types. INRIA; 2011. RR-7687.

[93] Almeida PS, Shoker A, Baquero C. Efficient state-based crdts by delta-mutation. In: International Conference on Networked Systems. Springer; 2015. p. 62–76.

[94] Carlos Baquero PSA, Shoker A. Making Operation-Based CRDTs Operation-Based. In: Distributed Applications and Interoperable Systems - 14th IFIP WG 6.1 International Conference, DAIS 2014, Held as Part of the 9th International Federated Conference on Distributed Computing Techniques, DisCoTec 2014, Berlin, Germany, June 3-5, 2014, Proceedings; 2014. p. 126–140. Available from: http://dx.doi.org/10.1007/978-3-662-43352-2_11.

[95] Bonomi F, Milito R, Zhu J, et al. Fog Computing and Its Role in the Internet of Things. Proceedings of the first edition of the MCC workshop on Mobile cloud computing. 2012;p. 13–16. Available from: http://doi.acm.org/10.1145/2342509.2342513$\delimiter"026E30F$npapers2://publication/doi/10.1145/2342509.2342513.

[96] Yi S, Li C, Li Q. A Survey of Fog Computing: Concepts, Applications and Issues. In: Proceedings of the 2015 Workshop on Mobile Big Data. Mobidata '15. New York, NY, USA: ACM; 2015. p. 37–42. Available from: http://doi.acm.org/10.1145/2757384.2757397.

[97] Verbelen T, Simoens P, De Turck F, et al. Cloudlets: Bringing the cloud to the mobile user. In: Proceedings of the third ACM workshop on Mobile cloud computing and services. ACM; 2012. p. 29–36.

[98] Fernando N, Loke SW, Rahayu W. Mobile cloud computing: A survey. Future generation computer systems. 2013;29(1):84–106.

[99] Hu YC, Patel M, Sabella D, et al. Mobile edge computing—A key technology towards 5G. ETSI white paper. 2015;11(11):1–16.

[100] Cisco. Cisco IOx Data Sheet; 2016. Available from: http://www.cisco.com/c/en/us/products/collateral/cloud-systems-management/iox/datasheet-c78-736767.html.

[101] Dell. Dell Edge Gateway 5000; 2016. Available from: http://www.dell.com/us/business/p/dell-edge-gateway-5000/pd?oc=xctoi5000us.

[102] Milojicic DS, Kalogeraki V, Lukose R, et al. Peer-to-peer computing. 2002;.

[103] Jelasity M, Montresor A, Babaoglu O. Gossip-based aggregation in large dynamic networks. ACM Transactions on Computer Systems (TOCS). 2005;23(3):219–252.

[104] Akyildiz IF, Su W, Sankarasubramaniam Y, et al. Wireless sensor networks: a survey. Computer networks. 2002;38(4):393–422.

[105] Gilbert S, Lynch N. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services. SIGACT News. 2002 Jun;33(2):51–59.

[106] Meiklejohn C, Van Roy P. Lasp: A Language for Distributed, Coordination-Free Programming. In: Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming (PPDP 2015). ACM; 2015. p. 184–195.

[107] Carvalho N, Pereira J, Oliveira R, et al. Emergent Structure in Unstructured Epidemic Multicast. In: Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07). Edinburgh, Scotland, UK; 2007. p. 481 – 490.

[108] Balegas V, Serra D, Duarte S, et al. Extending Eventually Consistent Cloud Databases for Enforcing Numeric Invariants. In: Proceedings of SRDS 2015. Montréal, Canada: IEEE Computer Society; 2015. p. 31–36. Available from: http://lip6.fr/Marc.Shapiro/papers/numeric-invariants-SRDS-2015.pdf.

[109] Najafzadeh M, Shapiro M, Balegas V, et al. Improving the scalability of geo-replication with reservations. In: ACM SIGCOMM - Distributed Cloud Computing (DCC). Dresden, Germany; 2013. Available from: http://lip6.fr/Marc.Shapiro//papers/escrow-DCC-2013.pdf.

[110] Gotsman A, Yang H, Ferreira C, et al. 'Cause I'M Strong Enough: Reasoning About Consistency Choices in Distributed Systems. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. New York, NY, USA: ACM; 2016. p. 371–384. Available from: http://doi.acm.org/10.1145/2837614.2837625.

[111] Akkoorath DD, Tomsic A, Bravo M, et al. Cure: Strong Semantics Meets High Availability and Low Latency. INRIA; 2016. RR-8858.

[112] van der Linde A, Fouto P, Leitão J, et al. Legion: Enriching Internet Services with Peer-to-Peer Interactions. In: Proceedings of the 26th International Conference on World Wide Web. Republic and Canton of Geneva, Switzerland: International World Wide Web Conferences Steering Committee; 2017. p. 283–292. Available from: https://doi.org/10.1145/3038912.3052673.

[113] Lasp: The Missing part of Erlang distribution;. Accessed: 2018-04-27. http://www.lasp-lang.org.

[114] Meiklejohn C, Enes V, Yoo J, et al. Practical Evaluation of the Lasp Programming Model at Large Scale. In: Proceedings of the 19th International Symposium on Principles and Practice of Declarative Programming (PPDP 2017). ACM; 2017. p. 109–114.

[115] Bichot CE, Siarry P. Graph partitioning. John Wiley & Sons; 2013.

[116] Shewchuk JR. Allow Me to Introduce Spectral and Isoperimetric Graph Partitioning; 2016. Available from: http://www.cs.berkeley.edu/~jrs/papers/partnotes.pdf.

[117] Bellman R. Introduction to matrix analysis. vol. 960. SIAM;.

[118] Chung FR. Laplacians of graphs and Cheeger's inequalities. Combinatorics, Paul Erdos is Eighty. 1996;2(157-172):13–2.

[119] Spielman DA, Teng SH. Spectral partitioning works: Planar graphs and finite element meshes. Linear Algebra and its Applications. 2007;421(2):284–305.

[120] Gantmakher FR. The theory of matrices. vol. 131. American Mathematical Soc.; 1998.

[121] Berman A, Plemmons RJ. Nonnegative matrices. vol. 9. SIAM; 1979.

[122] Fiedler M. Algebraic connectivity of graphs. Czechoslovak mathematical journal. 1973;23(2):298–305.

[123] Mohar B. Isoperimetric numbers of graphs. Journal of Combinatorial Theory, Series B. 1989;47(3):274–291.

[124] Van Driessche R, Roose D. An improved spectral bisection algorithm and its application to dynamic load balancing. Parallel computing. 1995;21(1):29–48.

[125] Hendrickson B, Leland R. An improved spectral graph partitioning algorithm for mapping parallel computations. SIAM Journal on Scientific Computing. 1995;16(2):452–469.

[126] Lancaster P, Tismenetsky M. The theory of matrices: with applications. Elsevier; 1985.

[127] Chevalier C, Pellegrini F. PT-Scotch: A tool for efficient parallel graph ordering. Parallel computing. 2008;34(6):318–331.

[128] Anderson E, Bai Z, Bischof C, et al. LAPACK Users' Guide (Software, Environments and Tools) 3rd Edition;.

[129] Bergamaschi L, Bozzo E. Computing the smallest eigenpairs of the graph Laplacian. SeMA Journal. 2018;75(1):1–16.

[130] Soper AJ, Walshaw C, Cross M. A combined evolutionary search and multilevel optimisation approach to graph-partitioning. Journal of Global Optimization. 2004;29(2):225–241.

[131] Zheng A, Labrinidis A, Pisciuneri PH, et al. PARAGON: Parallel Architecture-Aware Graph Partition Refinement Algorithm. In: EDBT; 2016. p. 365–376.

[132] Fiduccia CM, Mattheyses RM. A linear-time heuristic for improving network partitions. In: Papers on Twenty-five years of electronic design automation. ACM; 1988. p. 241–247.