FP7 Project ASAP
Adaptable Scalable Analytics Platform

ASAP D2.1
Preliminary Definition of ASAP Programming Model

WP 2 – A unified analytics programming model

Nature: Report

Dissemination: Public

Version History

Version	Date	Author	Comments
0.1	21 Feb 2015	K. Murphy, J. Sun, H. Vandierendonck	Initial Version

Acknowledgement This project has received funding from the European Union's 7th Framework Programme for research, technological development and demonstration under grant agreement number 619706.

Executive Summary

This document describes the current status of the definition of the ASAP unified programming model. The programming model is used to implement individual steps in a workflow. The workflow coordination language is distinct and is defined in *Deliverable D5.1 "Workflow Management Model"*.

The unified programming model is defined at two levels. It defines high-level operators which provide a high level of abstraction and simplify programming. These include operations such as map, filter and group-by. These operators are internally implemented in a task-oriented parallel programming language that uses advanced concepts in dataflow-based dependence resolution and region-based memory management to achieve high degrees of efficiency.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Programming models describe how programs can be constructed and how the resulting constructions are executed. Parallel programming models in particular extend sequential or single-threaded programming models with syntax and execution methods for executing parallel programs collaboratively with a set of procesors.

Programming models for big data analytics need to satisfy a wide range of properties, ranging from richness of expression of analytical algorithms, ease of expression of such algorithms, high performance on small-scale and large-scale distributed systems, fault-tolerance, interfacing with data management systems, etc. Work package 2 of the ASAP project focusses on the aspects of high performance and expressiveness.

In order to achieve high performance and expressiveness, ASAP uses two distinct programming languages. The rationale hereto is that one programming language caters for the abilities and expertise of domain experts whose expertise is, in this instance, in data analytics. The other programming language is used internally to develop the ASAP system and is geared towards experts in parallel and distributed computing. An automated conversion of programs expressed in the high-level language to the low-level language provides for an overall easy-to-use and high-performance programming environment.

ASAP aims for expressiveness and productivity in the high-level language by defining operators that manipulate data sets. The operators are akin to SQL queries in the level of abstraction aimed for. However, the ASAP operators are embedded in a general-purpose programming language in order to allow arbitrary manipulation of data. The operators can, in principle, be defined in any desired programming language.

ASAP aims for high-performance using a specially designed low-level programming language. The low-level language is structured around a task-based programming language. Tasks are units of work and are scheduled to execute on nodes and processors at runtime by the scheduler. Tasks are furthermore annotated with the precise data sets that they will access. As such, the scheduler knows precisely where the required data is located and can optimize the scheduling of tasks to minimize data movement, or decide on the minimal amount of data movement to perform.

One of the key goals of ASAP is to tackle practical deployment issues surrounding the diver-

sity of programming environments and the heterogeneity of data stores. The ASAP programming model contributes to this goal by defining a high-performant internal language that supports arbitrary workloads, of which graph analytics and map/reduce are only two specific subsets.

This document is structured as follows. In Chapter 2 we review existing programming environments in the big data landscape. This review mandates our decision to define an internal (low-level) programming language for experts in parallel and distributed computing and a set of high-level operators for use by data analysts. Chapter 3 provides details and design rationale for the internal programming language, while Chapter 4 provides information on the operators. Next, Chapter 5 gives some thought on the translation of operators to the internal programming language. Chapter 7 draws conclusion to this document.

Chapter 2

Review of Existing Programming Models

Parallel programming models for big data to date fall within 2 broad categories: distributed data flow models like MapReduce have proved very popular and progressive for Big Data Analytics applications but are considered to be too low level for complex analysis tasks; and graph analytics models which provide flexibility for specifying complex parallel computations by representing structured and unstructured data as graphs. A summary of the leading models within each is given below.

2.1 Distributed Data-flow Models

MapReduce [9] is a functional data flow model for transforming large amounts of data read from distributed file systems, for example Hadoop Distributed File System (HDFS), on large clusters in parallel. The input data is partitioned and transformed in parallel by optimised map and reduce functions on worker nodes. It is a popular framework due to its programmability, scalability and fault-tolerance. The libraries have been written in many programming languages; application programmers need only implement key functions, most notably map and reduce, in their language of choice. Computation is pushed into processing elements that are close to local disks and tasks are scheduled on machines that contain replicas of the most relevant data partitions.

One of the main criticisms of MapReduce is its lack of support for iterative processing, which is a key requirement of ASAP's use case applications. MapReduce tasks are restricted to a two phase disk-based pipeline operation where the output from map is an intermediate data file which is consumed by reduce. It is therefore of limited use for algorithms which require asynchronous iterative processing on stateful entities such as machine learning algorithms. Additionally, the input is largely restricted to structured input flat files which are insufficient for expressing many parallel algorithms naturally expressed as graph models with sparse computational dependencies.

Spark [41] overcomes inefficiencies in Hadoop by creating Resilient Distributed Dataset (RDD) in-memory structures that can be queried and processed iteratively. RDDs allow data held in external storage systems to be loaded into memory as a read-only collection of objects partitioned

across a set of machines in a cluster. Access to data represented by RDDs are therefore much faster than access to data on disk in traditional MapReduce. RDDs contain lineage information used for rebuilding an object to its current state. This lineage information typically starts from data in reliable storage. So data can be queried and manipulated repeatedly because it is held in the clusters memory. Spark exposes RDDs as abstractions in Scala, Java or Python. Transformations are applied on existing RDDs with operations map, filter, reduce and join. (See section Resilient Distributed Datasets 4.1.1 for more details on RDD operations). ASAP leverage Spark to overcome the limitation of iterative processing in MapReduce.

HAMA [31] extends MapReduce to create a Bulk Synchronous Parallel version for large-scale distributed data processing which, like Spark, is more effecient for iterative data analysis. It layers the HBase interface on top of HDFS for a flexible data management interface and has been shown to have better scalability than MPI implementations of MapReduce.

2.2 Graph Analytics Models

Graph Parallel models of late have implemented a variety of novel features to overcome inefficiencies in traditional distributed dataflow models such as MapReduce [9]. The models target either a distributed or shared memory environment. Some designs focus on finding efficiences for particular domain specific characteristics (sparse/dense graphs, etc.) whilst others take the approach of defining abstractions to unify models and features for multiple application domains. Some of the most notable models and their characteristic features are summarised.

2.2.1 Programming Abstraction

Computation in graph analytics models are expressed as vertex programs which run in parallel on nodes in a graph. Typically users of parallel models are expected to define code for vertex programs in a high level language such as C++ or Java. The program performs transformations on data gathered from neighbouring nodes and edges and propagates results in the form of messages along edges to neighbouring nodes as input for further computation. The mode of computation may be synchronous or asynchronous. In synchronous mode computation progresses as a sequence of super-steps [14]. Within each step the vertex program is executed on active vertices. Changes to the node/edge data are committed at the end of the step and will take effect within the next step [14]. In asynchronous mode computation on active vertices typically occurs iteratively, so the same vertex is visited many times. Update values are committed immediately and take take effect in subsequent computation in neighbouring vertices. In either mode, repeated execution of vertex programs refines intermediate values and is expected to yield accurate final results when values converge sufficiently. In some models a single vertex-program can span multiple machines, for example in Powergraph [14] this makes it possible to achieve an even assignment of edges to machines.

```
1  interface  GASVertexProgram(u) {
2        // Run on gather_nbrs(u)
3        gather(Du, D(u,v),  Dv)  Accum
4        sum(Accum left, Accum right)  Accum
5        apply(Du,Accum) Du new
6
7        // Run on scatter_nbrs(u)
8        scatter(Dnew u ,D(u,v),Dv)  (Dnew (u,v), Accum)
9  }
```

Figure 2.1: Powergraph GASVertexProgram interface

As an example of a programming abstraction, the PowerGraph model [14] defines an interface for GASVertexProgram (Figure 2.1) and the user of Powergraph must provide definitions of the functions *gather*, *apply* and *scatter* as well as providing definitions for graph data types.

2.2.2 Distributed Models

Two models which aim to save on message passing overheads are Active Pebbles [38] and Giraph [24]. Active Pebbles minimises the negative impact of locking contention in message sending by introducing coalescing and intelligent routing of fine grained light weight messages. It is particularly suited to algorithms that create many fine-grained asynchronous tasks such as Breadth First Search. Efficiency gains from routing can be observed most in models with higher node numbers where more savings can be made from avoiding contention.

Giraph [24] is a Bulk Synchronous Parallel (BSP) model which cuts down on message overheads by use of message combiner abstraction and aggregators at different levels of graph computation. Giraph fine tunes the partitioning and placement of vertices and their outgoing edges by creating hooks for user defined custom assignment functions. Giraph supports a mutable graph topology where added or deleted vertices take effect in subsequent super-steps.

BSP models like the above mentioned are inherently limited in parallelism for iterative computational algorithms such as machine learning. This is due to the necessity for processes to wait, at barrier synchronisation, for all other processes to reach the same barrier.

Graph models such as GraphLab and Powergraph [20] [14] have been developed to exploit parallelism for iterative algorithms where computational and data dependencies can be simultaneously represented at finer levels of granularity in graph components. Computation is stalled only between minimally dependent graph components when asynchronous communications occurs between them. Iterative communication and computation occurs at multiple levels of the graph with no artificial synchronisation barriers.

PowerGraph [14] extends elements of GraphLab [20] for efficient processing of high degree power-law graphs by factoring computation over edges. It factors vertex computation over edges

and avoids the characteristic message passing congestion of highly skewed graphs. In contrast to GraphLab, PowerGraph supports both BSP and Asynchronous modes of computation. Parallelism is exploited by streaming vertex cuts where partitioning can be done at the same time as computation, and execution cycles are saved by caching results from computations so subsequent cycle computations can be skipped when there has been no change to neighbor values. Communication overheads are reduced by storing edge data once. Vertex data must be communicated to all machines its computation is factored over but they define heuristics for partitioning edges to cut down on vertex data replication.

2.2.3 Shared Memory Models

X-Stream [30] shared memory model provides BSP computation that works with in-memory and out-of-core graphs on a single shared-memory machine. Sequential bandwidth is larger than random access bandwidth so performance gains in X-Stream are made by reading edges sequentially from file, avoiding random access to edges. It is therefore particularly beneficial where graphs are edge heavy with data compared to vertex data. The streaming partitions feature means edges appear in the same partition as their source vertex and it implements work stealing for high-degree edge partitions. X-Stream implements an efficient 2-tier index/chunk array for fast access into streamed buffers (used for edges, disk updates and shuffles).

Ligra [32] is a graph traversal framework built on top of Cilk++ for shared memory machines. It provides abstractions for edge and vertex computations (edgemap and vertexmap). The API aims to make writing graph traversal algorithms (such as BFS) simple. Of note the representation of active vertices switches dynamically to more efficient structures sensitive to graph and active vertex set density. Under future work they plan to make changes to the input graph possible.

Galois [28] is a shared memory system which presents a novel design for a unified model by defining abstractions for algorithms. It conducts an analysis of algorithms (tao) to extract properties important for parallelism by expressing actions on data structures. Properties of algorithms are categorised by 1) detail of the graph topology, 2) how active nodes become active and 3) operator actions on active nodes. The extent to which dynamics come into play in the building of dependence graphs is determined by the type of algorithm (eg. data/topology driven, ordered/unordered, structured/unstructured). Galois defines ordered and unordered sets which may be added to at execution of iterations. Unlike Ligra, it doesn't abstract away the internals of iteration loops from the user; the user must identify active elements directly for each iteration [28]. The decision to use data structures like dense arrays or adjacency lists for graphs is also left to the user, but there is no indication that a dynamic switch between these structures is possible in reaction to graph density changes.

Polymer [42] targets efficiency gains on shared-memory multi-core systems. Sequential inter-node memory accesses have much higher bandwidth than both intra- and inter-node random ones. As such Polymer, like X-Stream [30], minimises both random and remote memory access by optimising graph data layout and access strategies at allocation. This maximises data locality and

parallelism. For Graph topology data each thread allocates its own data structures. For application data Polymer constructs a sequentially accessed virtual contiguous memory from discrete physical addresses. For runtime states, frequently reconstructing virtual address spaces cant be justified and so Polymer uses a fast lock-free structure lookup table for all partitions. It achieves further efficiency savings by dynamically changing the leaf data structure from a bitmap to a queue as algorithms converge. It reduces remote accesses to data on neighboring vertices owned by other partitions by replicating immutable topology data with agents.

2.3 Direction for ASAP Programming Model

Application development simplified by high-level programming model (map/reduce or vertex programs). Conceptually simply, but to get performance one must be very careful on how to orchestrate the repeated application of these operations on the data, and how to manage data.

The ASAP programming model retains the proven approach. We will define high-level operators that have common occurence in data analytics. Data analysts can use these operators as building blocks to construct complex analyses.

In order to get efficient computation, the ASAP project defines an internal parallel programming model that uses low-level concepts to describe cumbersome notions in parallel computing: the presence of parallel operations, data management and locality, scheduling, etc. The high-level operators will be implemented in the internal programming language and will be provided to the data analysts as a library of internally parallel components.

The remainder of this report describes the internal programming model and a preliminary selection of operators that we wish to support. Moreover, we will show example implementations of high-level operators in the internal programming language.

Chapter 3

Internal Programming Model

The internal programming model is designed to implement the operators. As such, it is targeted at experts in parallel and distributed programming and builds on advanced concepts in parallel and distributed programming.

The internal programming language builds on a task-oriented abstraction. Programs specify tasks, indicate dependences or task ordering constraints and provide hints towards memory management. Upon execution of these programs, a runtime system makes on-the-fly decision towards mapping tasks to compute nodes, which trades-off load balancing and optimization of data locality.

3.1 Programming Model Definition

The internal programming model builds on the Cilk language [5], an established and high-performance language for parallel computing, and on the Swan language, a recent experimental extension for data-flow style execution [36].

In Cilk and Swan, tasks are defined as functions (procedures, methods) and are organized in a hierarchical structure, i.e., the language builds heavily on recursion. Parallelism is indicated by a *spawn* statement. The spawn keyword, followed by a function call, indicates that the called function may be executed in parallel with the *continuation* of the parent. As such, every spawn statement increases the degree of parallelism. Parallelism continues until the calling function executes a *sync* statement. A function executing a sync statement blocks until all spawned children have completed.

It is up to the Cilk runtime system to decide which tasks are executed in parallel. The scheduler makes these choices using *work stealing*, i.e., parallelism is exploited only when processors are idle.

Swan extends Cilk by adding the option for data-flow-driven execution [36]. Data-flow dependences between tasks are indicated through program variables: one task may write to a program variable, while another task may read this variable. This pattern established a data-flow relationship between these tasks. The data-flow relationship enriches the set of parallel programs that can

be expressed in Cilk, e.g., it allows the expression of pipeline-parallel programs. More importantly, the data-flow information expresses what data is accessed by a task. The ASAP project will exploit this aspect of Swan to enable efficient execution of data analytics operators.

Swan assumes a Cilk-like syntax for the *spawn* and *sync* statements, extended with task dependency types on function arguments. If a task does not have any arguments labeled with a dependence type, then this task will spawn unconditionally, as a Cilk spawn. If a task lists at least one argument with a dependency type, then this task will check whether the dependences are satisifed, and may not be executed immediately. Similarily, the *sync* statement may also be unconditional (wait for all children) or conditional (wait for specific children, or specific variables). The latter type of sync statement will take a set of objects as arguments, and suspend the task until all children operating on any of the arguments have completed.

Swan enriches the set of parallel programs that can be specified. While Cilk is traditionally limited to divide-and-conquer parallelism, Swan can also support pipeline parallelism by virtue of the task dependences. These dependences can enforce at runtime a scheduling order of tasks that respects pipeline parallelism constraints [35, 37].

3.1.1 Memory Footprints and Side-Effect Annotations

We define the *memory footprint* of a task is the set of memory locations, or more abstractly, the data, that it accesses. The memory footprint is crucial to schedule the task for data analytics, as it allows us to schedule the task on the node where the data is located.

The footprints of tasks are described by annotating their arguments with side-effects. It is assumed that tasks do not access global variables. The side-effects indicate whether an argument is *read*, *written* or both by the task. The side-effects induce data-flow dependences between tasks: a task reading a variable must wait until a prior task that writes to the variable has completed. Moreover, the language induces also false dependences: a task writing to a variable must wait until a prior task that reads the variable has completed. If not, the variable may be overwritten by the former task, which may result in unintended behavior.

The runtime scheduler checks the side-effect annotations of a task when it executes a spawn statement. If all arguments are ready, then the task may be spawned immediately. Otherwise, the scheduler will defer the execution of the task. Hereto, it registers the dependence of the task on the prior executing task(s), such that it can be woken up when the prior task(s) have completed. In the mean time, the scheduler continues executing the task calling the spawn statement.

Note that if there are no task arguments that describe side-effects, Swan behaves identical to Cilk.

Following side-effect annotations are utilized in the ASAP internal programming model:

- indep – The task will access argument in read-only method, other tasks need to write to this data.

- outdep –The task will over-write each single element of the argument. It may also read the

element which has been writing.

- inoutdep –The task will read and write individual elements of the arguments, or this argument as a whole.

Swan use the versioned objects to allow efficient and automatic dependency resolution, and renaming of objects. It allows the runtime system to track producer-consumer relations and to create multiple versions of these objects during parallelism execution. It is useful to increase parallelism. Versioned objects are a type of hyperobject [12] as they can automatically provide distinct views to threads that reference the same variable [37].

Versioned objects encapsulate the meta-data that is used to track task argument dependencies. A versioned object combines two pieces of information:

- The object metadata that tracks the status of the object, such as task reading, task writing (input,output,inoutput)
- and a pointer to the dynamically allocated memory which holds an instance of the object.

Versioned objects allow versioning of arbitrary data structures. *Versioning* (renaming) occurs only when the spawning tasks with an output dependency. And it happens only if prior readers or writers are pending. Output dependency (*outdep*) may call the function of renaming the object, for instance, when this object is in use by pending tasks at the time of a spwan task. In this procedure, the child receives the parent's view (existing version) and the parent receives a *new empty view*. For the versioned objects with *indep* and *inoutdep*, the parent and child procedures will share the same view of the object.

3.1.2 Regions

While Swan is a proven programming language for shared-memory systems [35, 36, 37], an efficient implementation for distributed memory systems (i.e., clusters) requires an additional concept to manage data transfers and optimize data locality: regions.

Regions are most often used in region-based memory allocation. A region is a large, consecutive chunk of memory from which small pieces of memory may be allocated [13]. The efficiency of regions is in the de-allocation, as the whole region may be deallocated at once. Some programs that allocated large numbers of small objects with the same lifetime can benefit significantly from this.

In the context of ASAP, however, we will use regions to describe large data sets that can be managed in a distributed memory system [22, 4]. Myrmics [22, 27] is a task-based runtime system for heterogeneous message-passing that organizes data in regions. It supports nested parallelism and pointer-based, irregular data structures by replicating the virtual address space across all nodes in the cluster. This way, regions containing pointers can be freely moved between nodes while retaining the validity of the pointers.

```
1 let x = ref 1 in
2 let y = ref 2 in
3 task(x){
4           x := x + 3;
5           task(x){ x := 42 }
6           wait(x);
7           x := !x   4
8         }
9 task(y){ y := !y + 2 }
10 task(x){ x := 0 }
```

Figure 3.1: Example program

ASAP organizes regions in tree-based structures: the top-level region is the root of the tree and may contain itself other regions. Every region can have at most one parent, except the root of a region tree has none. Programmers using the internal ASAP programming language explicitly decide how deep the region tree is and in which part of the tree data is allocated. The runtime system distributes the regions across the nodes of the system. Typically, one would match the structure of the region tree with the organization of the underlying storage systems. E.g., each leaf region in the region tree would correspond with a single chunk of data stored in HDFS.

Regions are first-class objects in the ASAP programming language. As such, they can be passed as arguments to tasks and the side-effects of tasks on the regions can be annotated with the side-effect annotations. These annotations help the scheduler to understand what data is accessed by a task (e.g., which chunk of HDFS data) and to schedule the task accordingly.

3.2 Operational Semantics

This section presents λ^{TASK}, a simple task-parallel calculus with runtime dependency resolution. We define the semantics of sequential and parallel executions of λ^{TASK} programs, and show that our dependency-aware scheduler is correct, i.e., it produces parallel executions that are equivalent to a sequential execution of the program.

3.2.1 Example

Consider the simple program in Figure 3.1. This program allocates two memory locations, referenced by x and y and initializes them to 1 and 2 respectively (lines 1–2). It then creates a task that operates on location x (lines 3-8), a task that operates on location y (line 9) and a task that operates on x (line 10). The first task recursively spawns another task that also operates on x (line 5).

Locations		ℓ	$\in$	$\mathcal{L}$
Values	$v \in \mathcal{V}$	$::=$	$n \mid \texttt{()} \mid \ell \mid \lambda\, x\,.\, e$
Expressions	e	$::=$	$x \mid e\ e \mid \texttt{ref}\ e \mid e := e \mid\, !\, e$
		$\mid$	$\texttt{task}(e)\,\{e\} \mid \texttt{waiton}\ e$
Types	τ	$::=$	$\mathit{int} \mid \mathit{unit} \mid \tau \to \tau \mid \tau\ \mathit{ref}$
Contexts	E	$::=$	$[\cdot] \mid E\ e \mid v\ E \mid \texttt{ref}\ E \mid E := e \mid v := E$
		$\mid$	$!\, E \mid \texttt{task}(E)\,\{e\} \mid \texttt{waiton}\ E$
Task queues	$q \in \mathcal{Q}$	$::=$	$\emptyset \mid e, q$
Dependency maps	D	$::$	$\mathcal{L} \to \mathcal{Q}$
Stores	S	$::$	$\mathcal{L} \to \mathcal{V}$
Execution state	Σ	$::=$	$\langle S, D, e\rangle \mid (\Sigma \| \Sigma)$

Figure 3.2: A simple task-based parallel language

3.2.2 Language

To simplify the presentation of our algorithms for task scheduling and runtime dependency resolution, we first formalize a core calculus that abstracts over most complex features of real programming languages.

Syntax Figure 3.2 presents λ^{TASK}, a simple task-parallel programming language. λ^{TASK} is a simply-typed lambda calculus extended with dynamic memory allocation and updatable references, task creation and synchronization. Values include integer constants n, the unit value `()`, functions $\lambda\, x\,.\, e$ and pointers ℓ. Program expressions include variables x, function application $e_1\ e_2$, memory operations and task operations. Specifically, expression $\texttt{ref}\ e$ allocates some memory, initializes it with the result of evaluating e, and returns a pointer ℓ to that memory; expression $e_1 := e_2$ evaluates e_1 to a pointer and updates the pointed memory using the value of e_2; and expression $!\, e$ evaluates e to a pointer and returns the value in that memory location. Expression $\texttt{task}(e_1)\,\{e_2\}$ evaluates e_1 to a pointer and then evaluates e_2, possibly in parallel. The task body e_1 must always return `()` and can only access the given pointer; if e_1 is evaluated in a parallel task, the expression immediately returns `()`. Finally, expression $\texttt{waiton}\ e$ evaluates e to a pointer and blocks the execution until no child task has that memory.

3.2.3 Semantics

We define the operational semantics for both sequential and parallel execution of λ^{TASK} programs.

Sequential execution Figure 3.3 presents small-step operational semantics for the sequential evaluation of λ^{TASK} programs. Small-step rules have the form $\langle S, e\rangle \to_s \langle S', e'\rangle$ where S is the

$$\boxed{\langle S, e\rangle \rightarrow_s \langle S', e'\rangle}$$

$$\text{[E-CTX-S]}\frac{\langle S, e\rangle \rightarrow_s \langle S', e'\rangle}{\langle S, E[e]\rangle \rightarrow_s \langle S', E[e']\rangle}$$

$$\text{[E-APP]}\frac{}{\langle S, (\lambda\, x\, .\, e)\ v\rangle \rightarrow_s \langle S, e[x \mapsto v]\rangle}$$

$$\text{[E-REF]}\frac{\ell - \mathit{fresh}}{\langle S, \texttt{ref}\ v\rangle \rightarrow_s \langle S \cup (\ell, v)\,, \ell\rangle}$$

$$\text{[E-DEREF]}\frac{S(\ell) = v}{\langle S, !\,\ell\rangle \rightarrow_s \langle S, v\rangle} \qquad \text{[E-ASSIGN]}\frac{\ell \in \mathit{dom}\,(S)}{\langle S, \ell := v\rangle \rightarrow_s \langle S[\ell \mapsto v], v\rangle}$$

$$\text{[E-FORK-S]}\frac{\ell \in \mathit{dom}\,(S)}{\langle S, \texttt{task}(\ell)\,\{e_1\}\rangle \rightarrow_s \langle S, e_1\rangle}$$

$$\text{[E-WAITON]}\frac{S(\ell) = v}{\langle S, \texttt{waiton}\ \ell\rangle \rightarrow_s \langle S, v\rangle}$$

Figure 3.3: Sequential semantics

starting store, mapping pointers ℓ to values v; e is the original program text; S' is the store after taking one step in e; and e' is the final program text.

Rule [E-CTX-S] defines evaluation inside a context E, defined in figure 3.2. Note that we only evaluate inside the first subexpression of a $\texttt{task}(e_1)\,\{e_2\}$ expression, as shown by the definition of evaluation contexts in Figure 3.2. Rule [E-APP] is standard function application by capture-avoiding substitution of the actual argument for the formal in the function body. Rule [E-REF] evaluates memory allocation expressions $\texttt{ref}\ v$ by allocating fresh memory in the store, indexed by the fresh pointer ℓ, and assigns it to the value v. The result of the evaluation is the pointer ℓ. Rule [E-DEREF] evaluates pointer dereference expressions $!\,\ell$ to the value v assigned to ℓ in the store. Rule [E-ASSIGN] updates the value stored in a memory location ℓ, when the pointer is valid (is defined in the store), and evaluates to the stored value. Rule [E-FORK-S] defines the sequential execution of a task $\texttt{task}(\ell)\,\{e_1\}$. It checks that the requested memory ℓ exists in the store, and inlines the task body e_1 in the current execution. Finally, in rule [E-WAITON] expression $\texttt{waiton}\ \ell$ steps to the value v where the ℓ points to in the store.

Parallel execution Figure 3.4 presents small-step operational semantics for the parallel execution of λ^{TASK} programs. Small-step judgements have the form $\Sigma \rightarrow_p \Sigma$, where Σ is either the execution state of a task, or (recursively) two parallel execution states, as defined in Figure 3.2.

$\boxed{\Sigma \to_p \Sigma}$

$$\text{[E-SEQ]}\frac{\langle S \setminus dom\,(D)\,, e\rangle \to_s \langle S', e'\rangle}{\langle S, D, e\rangle \to_p \langle S' \cup (S \cap dom\,(D))\,, D, e'\rangle}$$

$$\text{[E-CTX-FORK]}\frac{\langle S, D, e\rangle \to_p \langle S', D', e'\rangle \,\|\Sigma}{\langle S, D, E[e]\rangle \to_p \langle S', D', E[e']\rangle \,\|\Sigma} \qquad \text{[E-CTX-1]}\frac{\Sigma_1 \to_p \Sigma_2}{\Sigma_1\|\Sigma \to_p \Sigma_2\|\Sigma}$$

$$\text{[E-CTX-2]}\frac{\Sigma_1 \to_p \Sigma_2}{\Sigma\|\Sigma_1 \to_p \Sigma\|\Sigma_2} \qquad \text{[E-CTX-3]}\frac{\Sigma_1 \to_p \Sigma_1' \qquad \Sigma_2 \to_p \Sigma_2'}{\Sigma_1\|\Sigma_2 \to_p \Sigma_1'\|\Sigma_2'}$$

$$\text{[E-TASK]}\frac{\ell \in dom\,(S) \cup dom\,(D) \qquad D' = D[\ell \mapsto (D(\ell), e)]}{\langle S, D, \texttt{task}(\ell)\,\{e\}\rangle \to_p \langle S, D', \texttt{()}\,\rangle}$$

$$\text{[E-START]}\frac{\begin{array}{cc}\ell \in dom\,(S) & D(\ell) = e_t, q\\ S_1 = S \setminus \{\ell\} \qquad S_2 = (\ell, S(\ell)) & D' = D[\ell \mapsto q]\end{array}}{\langle S, D, e\rangle \to_p \langle S_1, D', e\rangle \,\|\, \langle S_2, \emptyset, e_t\rangle}$$

$$\text{[E-JOIN-1]}\frac{\Sigma \sqcup \langle S, D, \texttt{()}\,\rangle = \Sigma'}{\Sigma\|\, \langle S, D, \texttt{()}\,\rangle \to_p \Sigma'}$$

$$\text{[E-JOIN-2]}\frac{\Sigma_1 \sqcup \langle S, D, \texttt{()}\,\rangle = \Sigma_1'}{\Sigma_1\|\, (\langle S, D, \texttt{()}\,\rangle \,\|\Sigma_2) \to_p \Sigma_1'\|\Sigma_2}$$

$$\text{[JOIN-1]}\frac{dom\,(S_1) \cap dom\,(S_2) = \emptyset \qquad dom\,(D_1) \cap dom\,(D_2) = \emptyset}{\langle S_1, D_1, e\rangle \sqcup \langle S_2, D_2, \texttt{()}\,\rangle = \langle S_1 \cup S_2, D_1 \cup D_2, e\rangle}$$

$$\text{[JOIN-2]}\frac{\Sigma_1 \sqcup \langle S, D, \texttt{()}\,\rangle = \Sigma_1'}{(\Sigma_1\|\Sigma_2) \sqcup \langle S, D, \texttt{()}\,\rangle = \Sigma_1'\|\Sigma_2}$$

Figure 3.4: Parallel semantics

The parallel execution state of each task is a triple $\langle S, D, e\rangle$ of a store S, a *dependency map* D that maps memory pointers to a queue of tasks and the task body e.

Rule [E-SEQ] allows the parallel execution to revert into serial execution for any expression in the program. When a task $\langle S, D, e\rangle$ sequentially takes a step in e, any outstanding dependencies D remain unchanged in the new task state $\langle S', D, e'\rangle$.

Rule [E-CTX-FORK] states that when a subexpression e of a task $\langle S, D, E[e]\rangle$ creates a new task Σ, then Σ executes in parallel with its parent task $E[e]$.

Rules [E-CTX-1], [E-CTX-2] and [E-CTX-3] define that the execution of two parallel tasks $\Sigma_1 \| \Sigma_2$ takes a step, when the parent task Σ_1, or the child task Σ_2, or both, recursively take a step.

Rule [E-TASK] defines the execution of a $\texttt{task}(\ell)\,\{e\}$ expression by creating a new task for the evaluation of expression e. The memory location needed should be owned by the creating task, either immediately $\ell \in dom\,(S)$, or by one of its children $\ell \in dom\,(D)$. Then, the task expression e is added to the queue $D(\ell)$ of tasks waiting for ℓ, and the whole expression evaluates to `()`.

Rule [E-START] defines the beginning of a task e_t that operates on a memory location ℓ. A new task can be started at any time that its dependencies are satisfied, hence the starting state $\langle S, D, e\rangle$ of the conclusion does not constrain the original task e. The first premise requires the location ℓ to exist in the current store. The second premise requires the new task to be at the top of the queue for that location in D. The third and fourth premises split the store S into two parts, store S_1 containing everything in S except the memory ℓ required by the new task, and store S_2 containing only ℓ. The last premise defines the new dependency queue for ℓ, which does not contain the new task e_t. Finally, the resulting state in the conclusion is the parallel execution of two tasks, the original starting task with the new store and dependencies $\langle S_1, D', e\rangle$, in parallel with the new task $\langle S_2, \emptyset, e_t\rangle$. The newly created task e_t does not yet have subtasks and its dependency map is empty.

Rule [E-JOIN-1] defines that when a task $\langle S, D, \texttt{()}\rangle$ ends and it does not have child tasks that are still running, its state is merged with its parent task. In the last two rules of Figure 3.4, [JOIN-1] and [JOIN-2] we recursively define a join operator $\sqcup$ that, given an execution state Σ and a finished task $\langle S, D, \texttt{()}\rangle$, produces an execution state Σ' where the finished task is "collected" and merged with the "oldest" task running in Σ'. Rule [JOIN-1] is the base-case where Σ is a single task $\langle S_1, D_1, e\rangle$, where the join is valid if both the stores and the outstanding dependency maps refer to disjoint memory locations. Rule [JOIN-2] is the recursive case, where Σ is the parallel execution state of two other states Σ_1 and Σ_2, in which case the join always happens with the leftmost (and longest-lived) execution state Σ_1.

Rule [E-JOIN-2] defines the end of a task $\langle S, D, \texttt{()}\rangle$ that is executing in parallel with a parent execution state Σ_1 and children tasks Σ_2. In this case, we join the finished task with its parent execution state Σ_1. Note that the top-level task of the program cannot be joined with any other task. There is no fundamental reason for that restriction, but adding support for that would add unnecessary complexity by requiring another semantic rule for that case.

```
1 template<typename Functor>
2 vertexSubset VertexMap(vertexSubset U, Functor F) {
3       vertexSubset Out;
4       swan_for(vertexSubset::const_iterator  i=U.begin();i<U.end();i++)
5           if  (F(i)==true)
6               Out.insert(i);
7     return Out;
8 }
```

Figure 3.5: The VertexMap operator expressed in Swan.

3.3 Examples

Ligra and Polymer which are based on Cilk language for a single machine. Swan is also basis of Cilk. So, we can use the task-based language model to rewrite these graph analytics programs. For instance, rewite the Ligra's VertexMap algorithm and EdgeMap algorithm for graph analytics in swan, which applies an operation to every vertex in a set, may be represented in Swan as in Figure 3.5. Here, U is the subset of vertices of a graph where the functor F is applied. Functor F can run in parallel, in Swan, which can use spawn statements to parallelism. In shared-memory systems, it is assumed that F can access its argument as well as any location in shared memory such as the edge list and any vertex targeted by an out-edge of F's argument. However, it is also assumed that executing F on multiple distinct vertices is free of data races. VertexMap returns a vertexSubset representing the subset $Out = \{u \in U \mid F(u) = true\}$,where U is the subset of vertices passed to vertexMap.

The Ligra edge map function example coded in Swan is provided in Figure 3.6.

For a given graph $G = (V, E)$, U is a vertex subset representing a set of vertices $U \subseteq V$. The EdgeMap operator appplies the function F to all edges with source vertex in U and target vertex satisfying C. F is applied to each element in the active edge set. EdgeMap returns the vertex subset containing active vertices: $Out = \{v \mid (u, v) \in E_a \wedge F(u, v) = true\}$

In this framework, F and C can execute in parallel, in Swan, which can use spawn statements to indicate parallelism. The edge map function will call one of sparse-edge-map and dense-edge-map based on $|U|$ and the number of outgoing edges of U. The threshold in here determines when edgeMap switches between edgemapSparse and edgemapDense based on the threshold value. If the vertex subset size plus its number of outgoing edges is less than threshold value, edgeMap calls edgemapSparse, and otherwise calls edgesmapDense. Edge map sparse loops will through all vertices present in U in parallel and applies F to all of $u \in U$ neighbors neighbor in G in parallel. It returns a vertex subset will represented sparsely. In edge map dense, it throughs all the vertices in V in parallel.

```
1 template <typename Functor1, typename Functor2, typename vertex>
2 void edgeMapDense(graph<vertex> G, vertexSubset U, Functor1 F, Functor2 C) {
3     int numVertices = G.v;
4     vertex *G = G.U;
5     vertexSubset Out;
6
7     swan_for(vertexSubset::const_iterator i=U.begin(); i<numVertices; i++){
8         Out[i] = 0;
9         if(C(i)==true) {
10            for(int j=0; j<G[i].getInDegree; j++){
11                int ngh = G[i].getInNeighbor(j);
12                if(F(ngh)==true){
13                    Out.insert(i);
14                }
15            }
16        }
17        if(C(i)==false) break;
18    }
19    return Out;
20 }
```

Figure 3.6: The EdgeMap operator expressed in Swan.

Chapter 4

Operators

A significant inefficiency in MapReduce [9] results from the costly operation of writing to and subsequently reading from data sets held on disks at each transition to the next pipeline. Another drawback of the distributed data flow model is its inability to represent and compute graph algorithms. It is proposed we leverage GraphX [15] to overcome these shortcomings for ASAP. GraphX will allow graphs to be cast to flatter data-flow structures whilst preserving graph views so we can also take advantage of graph optimisations. Additionally GraphX builds on Spark's RDDs to remove the need for intermediate disk storage of datasets between pipeline stages.

4.1 Data Distribution

4.1.1 Resilient Distributed Datasets

A RDD [41] is a collection of objects partitioned over distributed nodes in a cluster for computation in parallel. Each Spark operation makes a transformation on an RDD and in doing so generates a new immutable object. The lineage of the RDD is retained for fault tolerance.

In Spark, the programmer can optionally specify that an RDD should persist over multiple iterations. This would allow significant performance gains in iterative algorithms. Users can choose to persist RDDs in memory or on disk. They may opt for serialisation to save space and/or choose to replicate RDDs to other nodes in the cluster.

GraphX leverages RDDs making it possible to persist data in memory between dataset transformations and make great performance savings by removing the need for expensive disk I/O activity.

In GraphX data is loaded as a graph from graph input formats (eg. edge list, ...). But its use of separate collections for vertices and edges means a flatter data-flow representation of a graph can be created and computed in a distributed data-flow engine style. Equally, through the use of graph abstractions and APIs, logic graph representations can be built in the form of triplets and allow for the expression and optimisations of iterative graph algorithms [15].

By leveraging RDDs, GraphX can keep data in memory greatly reducing access times for

iterative algorithms. Additionally RDDs allow GraphX to reduce memory overhead by index reuse across views and iterations. It achieves low cost fault tolerance through lineage data built into RDDs [41].

4.2 Definition

The parallel model adapted for ASAP is GraphX and Spark. Notably GraphX will leverage the fault tolerance, persitent properties available with Spark's RDD to allow efficient in-memory iterative processing on graph applications.

4.2.1 Spark Operators

Operations on RDDs fall into the category of transformations or actions. Transformations return a new RDD formed by execution of user defined code. In contrast actions return a *value* (rather than an RDD) which has been generated from user defined code [2].

RDDs are only computed when Spark needs to return a result to its driver program. RDDs are only computed when Spark needs to return a result to its driver program. Therefore if you transform an RDD, it may be computed each time unless you persist it in memory [2].

Transformational Operators

Some of Spark's main transformation operations include [2]:

map(func)	Return a new distributed dataset formed by passing each element of the source through a function func.
filter(func)	Return a new dataset formed by selecting those elements of the source on which func returns true.
intersection(otherDataset)	Return a new RDD that contains the intersection of elements in the source dataset and the argument.
reduceByKey(func)	Values for each key are aggregated using the given reduce function
aggregateByKey(func)	Values for each key are aggregated using the given combine functions
join(otherDataset)	Joins datasets under a common key
coalesce(numPartitions)	Decrease the number of partitions in the RDD.
repartition(numPartitions)	Reshuffle the data in the RDD randomly to create either more or fewer partitions and balance it across them.

Table 4.1: Spark transformation operators

Action Operators

Some of the main actions are:

reduce(func)	Aggregate the elements of the dataset using the provided function
collect()	Return all the elements of the dataset as an array at the driver program.
saveAsTextFilePath()	Write the elements of the dataset as a text file in the local filesystem, HDFS or any other Hadoop-supported file system.
foreach(func)	Run a function func on each element of the dataset.

Table 4.2: Spark action operators

4.2.2 GraphX Operators

GraphX operations take user defined functions and produce new graphs with transformed properties and structure [1]

The three main transformational operators in GraphX, which maintain structure and indices for efficiency of re-use are [1]:

mapVertices	Yield a new graph with the vertex property changed by the user defined map function.
mapEdges	Yield a new graph with the edge property changed by the user defined map function.
mapTriplets	Yield a new graph with the edge property changed by the user defined map function using source and destination vertex properties.

Table 4.3: GraphX transoformation operators

Additionally, there are a set of **structural operators** which generate new version or subsets of the graph based on for example reversing the edge directions, generating subgraphs of a graph or merging parallel edges. **Join operators** allow us to join data in a graph with other RDDs. For example, the *joinVertices* operator joins the vertices with the input RDD and returns a new graph with the vertex properties obtained by applying the user defined map function to the result of the joined vertices. The **aggregator operator** *mapReduceTriplet* is key to many algorithms for aggregating information about the neighborhood of vertices. It applies a user defined map function to each triplet and applies the results to target and/or source vertices. There are operators to compute the degree of each vertex, and an operator to cache a whole graph which is useful when a graph is repeatedly computed [1].

Chapter 5

Translation of High-Level Primitives to Low-Level Language

Region-based memory allocation organizes the allocation of data in regions, which are large, consecutive chunks of memory. Regions are also efficient to transfer pointer-based data structures in large chunks between nodes in a cluster [23]. We assume regions may be organized in a tree structure using specific allocation routines: *swan_region_new(), swan_region_malloc() and swan_region_free()*. For instance, the vertices in graph analytics are typically represented by a large array. We can represent the array as a tree of regions where the leaves of the trees are chunks of memory that hold large slices of the array.

Given this data organization, we aim to design programming abstractions that allow the automatic translation of the code given previously for VertexMap to code along the lines depicted in Figure 5.1. Code like this above would be generated from the short version given in Section 3.3, page 23, and will not written directly by the programmer. The point behind using regions is that the scheduler can decide to send spawned tasks to specific nodes in the cluster depending on what nodes are storing the regions used in the arguments. The construction of the *Out RegionArray* may furthermore be implemented using the concept of reducer hyperobjects [12]. The code above still needs to resolve issues with the accesses made by the functor F. To this end, we can borrow ideas from other graph frameworks, such as the graph-parallel abstractions of PowerGraph[14]. It can split vertices crossing node boundaries, and gather split values together finally. It also can borrow the ideas of Polymer [42] for NUMA-aware graph partitioning.

```
1  template<typename Functor>
2  RegionArray<bool> swan_for_region(
3          RegionArray<int> vertices, RegionArray<bool> U,
4          size_t Low_index, size_t Up_index, indep<Functor> F) {
5      RegionArray<bool> Out;
6      RegionArray<bool>::region_type * lo_region, * up_region;
7      lo_region = vertices.get_region(Low_index);
8      up_region = vertices.get_region(Up_index);
9      if ( lo_region == up_region ){// target vertices in a same region
10         RegionArray<bool>::region_type * region = lo_region;
11         swan_for(I=region>begin(Low_index),E=region>end(Up_index);I!=E;++I){
12             if ( F(*I) == true ) Out.insert( *I );
13         }
14     } else { // Traverse region tree recursively.
15                // This assumes a binary region tree
16         RegionArray<bool>::region_type * region, * left, * right;
17         region = lo_region>common_ancestor(up_region);
18         left = region> get_left_sub_region ();
19         right = region> get_right_sub_region ();
20         RegionArray<bool> OutLeft, OutRight;
21         OutLeft = swan_spawn swan_for_region(
22             vertices.getArrayForRegion(left),
23             U.getArrayForRegion(left),
24             std::max(left>getStartIndex(),Low_index),
25             std::min(left>getEndIndex(),Up_index),
26             F);
27         OutRight = swan_spawn swan_for_region(
28             vertices.getArrayForRegion(right),
29             U.getArrayForRegion(right),
30             std::max(right>getStartIndex(),Low_index),
31             std::min(right>getEndIndex(),Up_index),
32             F);
33         swan_sync;
34         Out = OutLeft + OutRight;
35     }
36     return Out;
37 }
```

Figure 5.1: Low-level code for the VertexMap operator for graph analytics.

Chapter 6

Case Study on The ASAP Internal Language

6.1 Introduction

We have performed a case study on the implementation of map-reduce programs on the ASAP internal programming language. This study uses a surrogate programming language, Cilk, which does not contain all of the features of the ASAP internal language. We will compare the expressiveness and performance of Cilk when applied to map-reduce programs against a specialized map-reduce programming system, in particular Phoenix++. This comparison provides evidence that (i) the ASAP internal programming language provides a high degree of expressiveness and ease of programming and (ii) the ASAP internal language promises to out-perform specialized runtime systems for map-reduce.

This study is limited in its application to a shared-memory (single node) setting. Later in this project we will extend it to the distributed memory (cluster) setting.

6.1.1 Map-Reduce

The map-reduce programming model is centered around the representation of data by key-value pairs. For instance, the links between internet sites may be represented by key-value pairs where the key is a source URL and the value is a list of target URLs. The data representation exposes high degrees of parallelism, as individual key-value pairs may be operated on independently.

Computations on key-value pairs consist, in essence, of a map function and a reduce function. The map function transforms a single input data item (typically a key-value pair) to a list of key-value pairs (which is possibly empty). The reduce function combines all values occurring for each key. Many computations fit this model [8], or can be adjusted to fit this model.

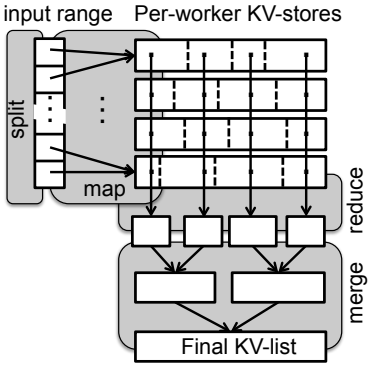

Figure 6.1: Schematic overview of Phoenix++ runtime system

6.1.2 Phoenix++ Implementation

The Phoenix++ shared-memory map-reduce programming model consists of multiple steps: split, map-and-combine, reduce, sort and merge (Figure 6.1). The split step splits the input data in chunks such that each map task can operate on a single chunk. The input data may be a list of key-value pairs read from disk, but it may also be other data such as a set of HTML documents. The map-and-combine step further breaks the chunk of data apart and transforms it to a list of key-value pairs. The map function may apply a combine function, which performs an initial reduction step of the data. It has been observed that making an initial reduction is extremely important to performance as it reduces the intermediate data set size [34].

It is key to performance to store the intermediate key-value list in an appropriate format. An naive implementation would hold these simply as lists. However, it is much more efficient to tune these to the application [34]. For instance, in the word count application the key is a string and the value is a count. As such, one should use a hash-map indexed by the key. In the histogram application, a fixed-size histogram is computed. As such, the key is an integer lying in a fixed range. In this case, the intermediate key-value list should be stored as an array of integers. For this reason, we say the map-and-combine step produces key-value data structures, rather than lists.

The output of the map-and-combine step is a set of key-value data structures, one for each worker thread. Let KV-list $j = 0, \ldots, N-1$ represent the key-value data structure for the j-th worker thread. These N key-value data structures are subsequently split in M chunks such that each chunk with index $i = 0, \ldots, M-1$ in the intermediate key-value list j holds the same range of keys. All chunks i are then handed to worker thread N, which reduces those chunks by key. This way, the reduce step produces M key-value *lists*, each with distinct keys.

Finally, the resulting key-value lists are sorted by key (an optional step) and they are subsequently merged into a single key-value list.

Phoenix++ allows the programmer to specify a map function, the intermediate key-value data structure, a combine function for that data structure, the reduce function, a sort comparison function and a flag whether sorting is required.

```
1  int fib (int n) {
2      if (n < 2) {
3          return n;
4      } else {
5          int x = cilk_spawn fib(n 1) ;
6          int y = cilk_spawn fib(n 2) ;
7          cilk_sync;
8          return x+y;
9      }
10 }
```

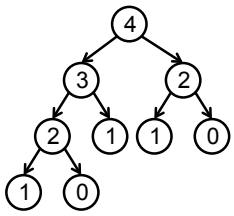

Figure 6.2: Fibonacci example (left) and the spawn tree for `fib(4)` (right).

6.2 Map-Reduce Using Cilk

The map-reduce format is a specific parallel skeleton. As such, any general-purpose programming language can implement it. In this work we focus on the Cilk language [11], in particular because of its support for generalized reductions [12]. We use Intel's Cilkplus version of the language, but Cilk Arts' Cilk++ may be used as well except for the array notation, an Intel-specific addition to the language that facilitates vectorization.

6.2.1 Cilk

Cilk is a task-oriented parallel programming model. The key way to create parallelism in Cilk is by a spawn statement, which has a syntax similar to a function call, except that it is preceded by the `cilk_spawn` keyword. The spawn statement indicates that the spawned function may execute in parallel with the continuation of the containing function. The scope of parallelism extends until the containing function executes a `cilk_sync` statement. Once the `cilk_sync` statement has completed, one can be sure that all side effects of the functions that were spawned by the containing function have taken effect.

Cilk is a faithful extension of the C/C++ languages. When the `cilk_spawn` and `cilk_sync` statements are removed from the program, then a correct sequential C/C++ program is obtained. This program is called the *serial elision* of the Cilk program. In the absence of data races, every Cilk program is equivalent to the *serial elision* of that program.

The simplicity of the Cilk language is often illustrated by a parallel program that calculates the nth Fibonacci number (Figure 6.2). A recursive specification defines the nth Fibonacci number as the sum of the $n-1$th and $n-2$th Fibonacci numbers. Recalculating these numbers in parallel generates a fairly scalable parallel program (although this is an inefficient way to calculate Fibonacci numbers). The return values `x` and `y` of the recursive spawns are available for further computation after the `cilk_sync` statement.

The spawn tree for this program (Figure 6.2, right) shows the recursive function calls. Each node corresponds to a call to `fib` and the number in the node corresponds to the argument to `fib`. The spawn tree should, ideally, be a balanced tree. This structure works beneficially with the Cilk scheduler. Programs with unbalanced spawn trees may incur excessive scheduling overhead.

Cilk programs are executed by a randomized work-stealing scheduler [11]. The scheduler consists of a number of worker threads, usually one per CPU core. Each worker has a spawn *deque* (double-ended queue), on which it pushes and pops work items. The execution of a Cilk program starts on a single worker with sequential execution. When a spawn statement is encountered, the worker continues with the execution of the spawned function. The continuation of the containing function is pushed on the worker's spawn deque. When the spawned function has completed, the continuation is popped from the spawn deque, in the same order as a sequential program is executed.

Idle workers attempt to steal continuations from other workers' spawn deques. Workers are selected at random. If the selected worker has continuations on its spawn deque (apart from the top-most continuation that is being executed), then the bottom-most continuation is stolen and transferred to the thief's spawn deque. It is subsequently executed.

A Cilk program expresses parallelism by means of its spawn tree. The key property of the Cilk work stealing scheduler is that it selects parallelism from the top of the spawn tree down. As such, stolen work items are initially extremely coarse-grain. In a balanced spawn tree, the first work item stolen represents half of the work in the parallel section of the program. It has been shown that the number of work stealing activities is proportional to the span of the program [3]. When the spawn tree is balanced and the majority of work is distributed evenly across the leaves of the tree, then the span is proportional to the 2-logarithm of the degree of parallelism.

Programs with unbalanced spawn trees are, however, executed less efficiently. The same rule applies, but now the span of the program is, in the worst case, proportional to the total amount of work in the program. Cilk generally performs better on programs with (nearly) balanced spawn trees. In some cases, when the leaf tasks perform little work, the overhead of executing the recursively spawning procedures that generate a balanced spawn tree is excessive. In such cases, an unbalanced search tree may perform better.

6.2.2 Parallel For

Parallel for loops are a common idiom. Such loops can be cast to the spawn/sync structure by creating a function that recursively divides the iteration range of the for loop. The recursion stops when the range is "*too short*", in which case the range is stepped through sequentially and the loop body is executed for those iterations. Cilk provides the `cilk_for` syntax to perform this translation behind the scenes. A `cilk_for` loop is like a C/C++ for loop with restrictions applied to the format of the initializer, conditional expression and increment expression [16].

A `cilk_for` construct may also use C++ iterators as control variable, e.g., STL iterators. In this case, it is required that the iterator is a `random_access` iterator [33].

A `cilk_for` loop is terminated by an implicit `cilk_sync`. However, this implicit `cilk_sync` synchronizes only the iterations of the `cilk_for` loop, but does not synchronize the loop with prior spawn statements.

6.2.3 Generalized Reductions

The Cilk language provides definitions for generalized reductions that are associative but not necessarily commutative [12]. As the reduction operation need not be commutative, many operations such as list prepend/append and hash-map insert can now be expressed as reduction operations. In these cases it is guaranteed that the reduction variable contains the same value as computed by the serial elision of the Cilk program.

A Cilk generalized reduction is defined by three components: a data type, an associative operation and an identity value for that operation. These components are defined in a monoid data structure that serves as the basis for a *reducer* class definition.

Figure 6.3 shows the definition of a Cilk reducer for a hash-map data type. It is assumed that the template parameter `map_type` defines a hash-map type that is compatible to the C++ standard's `std::map`. The definition consists of a Monoid class (Line 3), which defines the base type (through the `monoid_base` template parameter), the identity value (through an initialization function, Line 10) and the reduction function (Line 4). It is assumed that hash-maps are reduced by taking the join of all keys and that the values for common keys are further reduced using an `operator +=`. This behavior is specified in the `reduce` function.

Note that the reduction operation should ideally execute in constant-time, otherwise the execution time of the program will depend on the number of reduction operations performed [18]. The number of reduction operations is, in any case, proportional to the number of steal operations [12].

The monoid class definition plays a crucial rule in the operation of reducers. Reducers dynamically create copies of the reduction variable, and reduce those copies, as needed. These copies are called *views*. There can be at most one view of the reduction variable per Cilk worker at any time. The view is maintained when a worker spawns a task. When an idle worker steals a continuation from another worker's deque, a new view is created for the thief and initialized with the identity element. Views are reduced when a worker completes a spawned task leaving its spawn deque empty, or when a worker executes a `cilk_sync` statement. Further details are provided by Frigo *et al* [12].

Throughout the execution, the relative order of tasks is maintained to reflect the order of tasks as they are executed in the serial elision. The view used during a certain interval of the execution is reduced only with views corresponding to the intervals that directly precede or follow the first interval. This ensures the reducer calculates the same value during any parallel execution as it does during the serial elision.

The running example (Figure 6.3) defines a `map_reducer` class. The member value `imp_` (Line 14) is declared as an instance of the `reducer` class, specialized by the `Monoid` definition. The object `imp_` manages the creation, lookup and destruction of views. The `map_reducer` class

```
1  template<class map_type>
2  class map_reducer {
3      struct Monoid : cilk::monoid_base<map_type> {
4          static void reduce(map_type * left, map_type * right) {
5              for(typename map_type::const_iterator
6                      I=right>cbegin(), E=right>cend(); I != E; ++I)
7                  (* left )[ I> first ] += I>second;
8              right>clear();
9          }
10         static void identity (map_type * p) const {
11             new (p) map_type();
12         }
13     };
14     cilk::reducer<Monoid> imp_;
15
16 public:
17     map_reducer() : imp_() { }
18     typename map_type::value_type & operator[](
19             const typename map_type::key_type & key) {
20         return imp_.view()[key];
21     }
22     typename map_type::const_iterator cbegin() {
23         return imp_.view().cbegin();
24     }
25     typename map_type::const_iterator cend() {
26         return imp_.view().cend();
27     }
28     void swap(map_type & other) {
29         return imp_.view().swap(other);
30     }
31     map_type & get_value() {
32         return imp_.view();
33     }
34 };
```

Figure 6.3: Example of a hash-map reducer for counting occurrences of words.

further provides access to the underlying view through the `operator []` in order to add items to the hash-map.

Figure 6.4 shows how the `map_reducer` may be used in parallel code. The `cilk_for` construct creates parallelism. Each concurrently executing loop iteration references the same instance of the `map_reducer` class, but the `cilk::reducer` object `imp_` serves up different views in

```
1 map_reducer<std::map<std::string,size_t>> map;
2 cilk_for (std::vector<std::string>::const_iterator
3           I=vec.cbegin(); I != vec.cend(); ++I) {
4   map[*I]++;
5 }
```

Figure 6.4: Example use of the map reducer: a word count is calculated in parallel for all words in a vector by accumulating the count in a hash-map.

concurrently executing iterations. All views are reduced prior to completion of the `cilk_for` loop.

6.2.4 Array Notation

Intel Cilkplus supports an array notation that facilitates auto-vectorization [16]. The array notation allows for 3 fields in an array section expression: `a[i:l:s]`, where `i` is the start index of the array section, `l` is the length and `s` is the stride. Each element of the array notation is optional, but at least one colon must be present. Default values are 0 for `i`, the length of the array for `l`, provided it is known at compile-time, and 1 for `s`. E.g., `a[:]` indicates the full array if its size is statically known, while `a[:10:2]` indicates the elements at indices 0, 2, 4, 6, 8.

Expressions may be built up using array notations, e.g., the statement `c[:] = a[:]+2*b[:];` is equivalent to

```
1 for(int i=0; i < n; ++i)
2   c[i] = a[i] + 2*b[i];
```

assuming each array was declared with length `n`.

One can also map functions over all elements of an array section. E.g., `a[:] = pow(b[:])` applies the function `pow` to each element of array `b` and stores the result in the corresponding element of array `a`. Reductions are specified using built-in functions that may be applied to arbitrary array sections. E.g., `__sec_reduce_add(a[::2])` returns the sum of the array elements at even positions of `a`.

The key advantage of the array notation is that it enables the compiler to auto-vectorize the code. Vectorization can be important towards performance as map-reduce programs often exhibit a data streaming pattern.

6.2.5 Map-Reduce Template

Map-reduce applications can be encoded quite easily in Cilk, given the presence of `cilk_for` to apply the map task in parallel, and the functionality of reducers to correctly implement the reduction phase. Please note again that writing Cilk does not require programmers to worry extensively

```
1 MR * mr = ...;
2 MR::reducer_t output;
3 cilk_for (auto I=mr>begin(), E=mr>end(); I != E; ++I) {
4     mr>map(*I, output);
5 }
6 // Optional
7 cilk_pub::sort( output.get_value().begin(),
8                 output.get_value().end(), MR::cmp_t() );
```

Figure 6.5: Map-reduce template with balanced spawn tree

about scheduling issues or the application of reduction operations. This is taking care of by the runtime.

Figure 6.5 shows template code for a map-reduce program in Cilk. Application-specific types and functions have been extracted in an application-specific namespace "`MR`". It is assumed the programmer has defined the `MR::reducer_t` data type that describes intermediate key-value data structures and the way by which it can be reduced, conforming Section 6.2.3. Moreover, it is assumed the programmer has defined the functions `MR::begin()` and `MR::end()` that define the range of data elements to iterate over. These functions return C++ `random_iterators`. When dereferenced, these iterators return a data element of type `MR::partition_t` which is passed to the application specific function `MR::map()` together with a reference to a reducer object.

Finally, an optional sorting step may be applied to the reducer data structure using the application-specific comparison function of type `MR::cmp_t`. Sorting is performed in parallel, using a parallel merge-sort/quicksort algorithm [26]. Depending on the source data structure, sorting may require to first copy the data to a sortable data structure, e.g., change from a hash map to an array.

One may observe that there is little boiler-plate code in the code template (Figure 6.5).

The balanced code template (Figure 6.5) is applicable when the input data can be iterated over using an STL iterator. This is not always the case. E.g., text files need to be parsed, which often requires sequential access. Figure 6.6 shows an unbalanced template for map-reduce. In this case, the user defines a function `MR::partition()` that isolates a single work item and returns false when no more work items can be found. Each work item is subsequently passed on to the `MR::map()` function.

Note that the unbalanced template generates an unbalanced spawn tree. As such it potentially scales less well than the balanced template.

Figure 6.7 summarizes the data types and methods that the programmer should provide for either code template.

The Cilk code templates allow a natural specification of map-reduce problems as illustrated by the implementation of **wc** (Figure 6.8). The program consists of a loop that first partitions the input data on word boundaries. Once a partition is identified, a task is spawned off to compute the

```
1  MR * mr = ...;
2  MR::partition_t chunk;
3  MR::reducer_t output;
4  while(mr->partition(chunk)) {
5      cilk_spawn [&](MR::partition_t work) {
6          mr->map(work, output);
7      }(chunk);
8  }
9  cilk_sync;
10 // Optional
11 cilk_pub::sort( output.get_value().begin(),
12                  output.get_value().end(), MR::cmp_t() );
```

Figure 6.6: Map-reduce template with unbalanced spawn tree

```
1  class MapReduceApplication {
2      typedef ... partition_t ;
3      typedef ... reducer_t;
4      void map( const partition_t &, const reducer_t & );
5      // Unbalanced template
6      bool partition ( partition_t & );
7      // Balanced template
8      typedef ... iterator ; // random access
9      iterator begin();
10     iterator end();
11     // Optional comparison for sorting
12     typedef ... cmp_t;
13 };
```

Figure 6.7: Overview of template members for the `MR` class describing application-dependent types and methods.

word counts for a partition. These tasks may execute in parallel with each other as well as with the partitioning code. The hash-map reducer (Figure 6.3) is used to reduce the hash maps constructed by individual tasks.

Note that Phoenix [34] does not allow overlap between partitioning of the input data and applying the map task.

```
1  typedef unordered_map<const char *, size_t> umap;
2  void wc( char * data, size_t size, umap & out_dict,
3           size_t chunk_size=1UL<<20 ) {
4      map_reducer<umap> dict;
5      size_t pos = 0;
6      while( pos < size ) {
7          // Partition input at word boundary
8          size_t end = std::min(pos + chunk_size, size);
9          while(end < size && !non_word_char(*end))
10             end++;
11         data[end] = '\0';
12
13         // Process partitions in parallel
14         cilk_spawn [&](char * chunk, size_t len) {
15             size_t i=0;
16             while(i < len) {
17                 size_t start = i;
18                 // Search the end of the word
19                 while(i < len && !non_word_char(chunk[i]))
20                     i++;
21                 if ( i > start ) {
22                     chunk[i] = '\0';
23                     dict[&chunk[start]]++;
24                 }
25             }
26         }( &data[pos], end-pos );
27         pos = end;
28     }
29     cilk_sync;
30     // O(1) operation to hide reducer object from caller
31     dict.swap(out_dict);
32 }
```

Figure 6.8: Cilk implementation of **wc**.

6.3 Map-Reduce Benchmarks

We have implemented all 7 Phoenix++ benchmarks in Cilk. Table 6.1 describes their properties when programmed in the Phoenix++ programming model. The first column shows the key multiplicity as *m:e*, where m indicates how many map tasks can generate a unique key and e indicates how many keys can be emitted by a map task. These properties were listed in the Phoenix++ publication [34], however, we found the report deviates from the distributed code. For matmul

Table 6.1: Phoenix++ implementation details: the map task multiplicity, combiner data type, whether key-value pairs are sorted, merge task and reduction operation.

	map	**combiner**	**sort**	**reduction**
histogram	*:768	array	N	array add
lreg	*:5	array	N	array add
wc	*:*	hash	Y	hash map join
kmeans	*:K	array	N	array add
matmul	*:0	n/a	N	n/a
pca	1:1	array	N	array add
strmatch	*:0	n/a	N	n/a

Table 6.2: Cilk implementation details: the reduction type and whether the implementation has a balanced spawn tree (bal), uses multi-level parallelism (nest), vectorization (vec) and sorting (sort). The matmul code deviates from the map-reduce template.

	reduction	**bal**	**nest**	**vec**	**sort**
histogram	fixed-size array add	B	N	(Y)	N
lreg	5-scalar struct add	B	N	Y	N
wc	hash table union	U	N	N	Y
kmeans	cluster center add	B	N	(Y)	N
matmul	n/a	B	N	N	N
pca	scalar integer add	B	Y	Y	N
strmatch	none	U	N	N	N

and strmatch, the multiplicity **:0* indicates that the map tasks emit zero key-value pairs. Instead, shared memory operations are used to produce output results. This goes against the spirit of the map-reduce model.

The second column of Table 6.1 shows the intermediate key-value data structure used. In most cases, a generic key-value list is optimized to an array indexed by an integer key. In the case of word count, intermediate key-value pairs are stored in a hash map indexed by a character string key.

Details of the Cilk implementations of the benchmarks are provided in Table 6.2. The reduction data structures are similar to those used in the Phoenix++ versions of the code. In some cases, they are specialized further to the benchmarks and are semantically richer, e.g., a struct of scalars vs. an array for *lreg*. The table further indicates whether the balanced (B) or unbalanced (U) template is used, and also if the benchmark uses nested parallelism, vectorization or sorting. The label (Y) indicates that vectorization is possible, but did not return performance improvements.

6.4 Evaluation

We evaluate the programming systems on a dual-socket 2 GHz Intel Xeon E5-2650 v2, totaling 16 threads. The operating system is CentOS 6.5 with the Intel C compiler version 14.0.1. We compare against Phoenix++ version 1.0 using the large, medium and small input data sets and command line flags provided with it.

6.4.1 Performance Evaluation

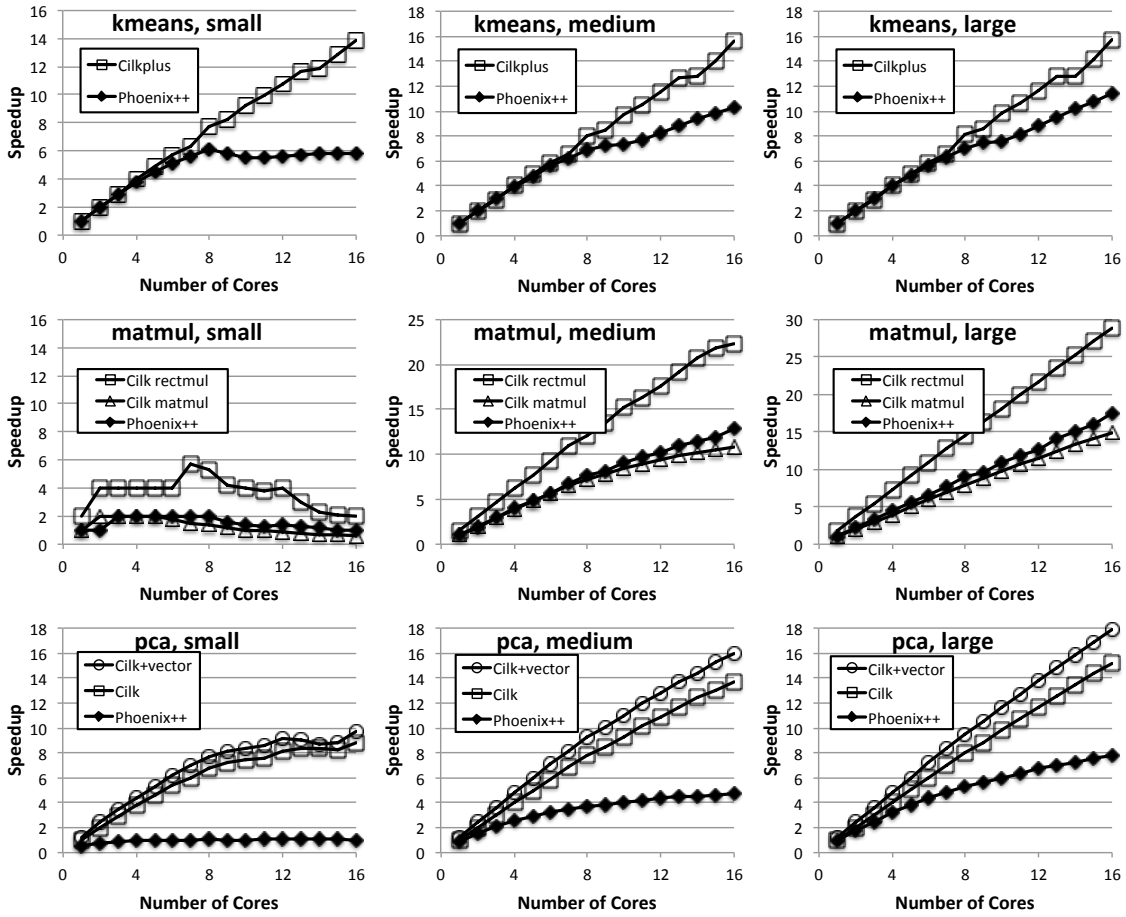

Figure 6.9: Results (a): Applications dominated by map time (compute-bound)

Figures 6.9–6.11 present the speedup using Cilk and Phoenix++ over the sequential version of the benchmarks. These codes have not been vectorized. Figure 6.9 shows the benchmarks

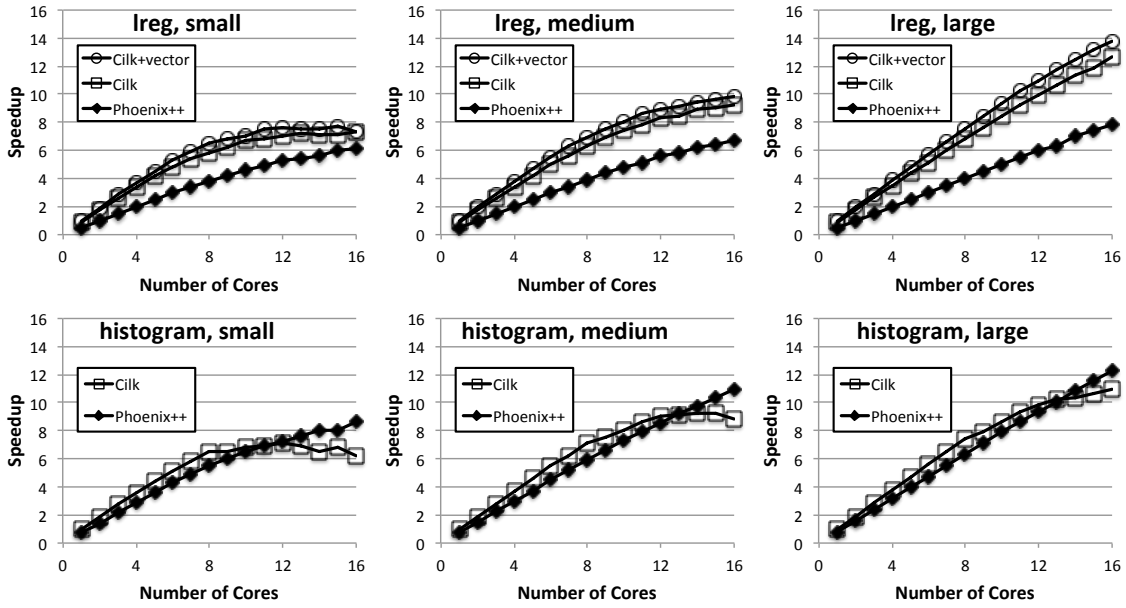

Figure 6.10: Results (b): Memory-bound applications

dominated by computation in the map phase: **matmul**, **pca** and **kmeans**. Kmeans is an iterative map-reduce application. Phoenix++ requires repeated serialization and de-serialization of the centers to key-value lists. This is redundant and reduces scalability.

For **matmul** we use two matrix multiply implementations distributed with MIT Cilk [11]. The **matmul** version uses recursive decomposition of the problem where on each level of recursion the problem is split along its largest dimension. The **rectmul** version splits the target matrix along both dimensions on each level of recursion and has a much higher degree of parallelism. Moreover, its leaf task, a 16x16 block multiply, is highly optimized. In Figure 6.9, performance is normalized to the sequential version of **matmul**. It is clear that specifically optimized codes outperform a generic map-reduce framework like Phoenix++. This exposes the pitfall that applying the map-reduce concept to every problem is not sensible. Moreover, note that the map-reduce runtime is used inappropriately for matrix multiply as it accesses shared memory from within the map task and does not emit key-value pairs.

The memory-bound benchmarks **histogram** and **lreg** show good scalability on both programming systems (Figure 6.10). Both benchmarks perform very few operations per input byte, namely 4 integer operations for **histogram** and 7 for **lreg**. The scalability of the Cilk version saturates for the highest core counts, which suggests that the Cilk scheduler carries a higher burden than the Phoenix++ scheduler.

Finally, the benchmarks **wc** and **strmatch** use the unbalanced Cilk template (Figure 6.11). The performance of **strmatch** is identical to that of the Phoenix++ version while **wc** outperforms

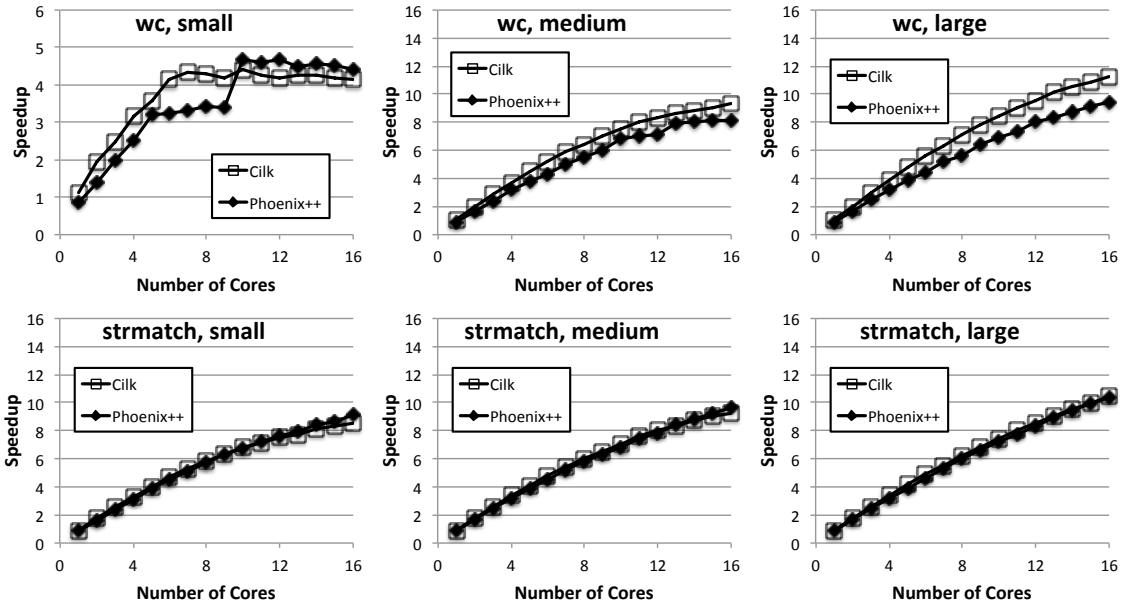

Figure 6.11: Results (c): Applications with unbalanced spawn trees.

Phoenix++ due to an improved hash table implementation, which is described below.

Overall, we observe that the Cilk codes are outperformed by the Phoenix++ codes only on **histogram** and marginally on **wc** when executing the small input data set.

6.4.2 Performance Considerations

Internal Data Structures

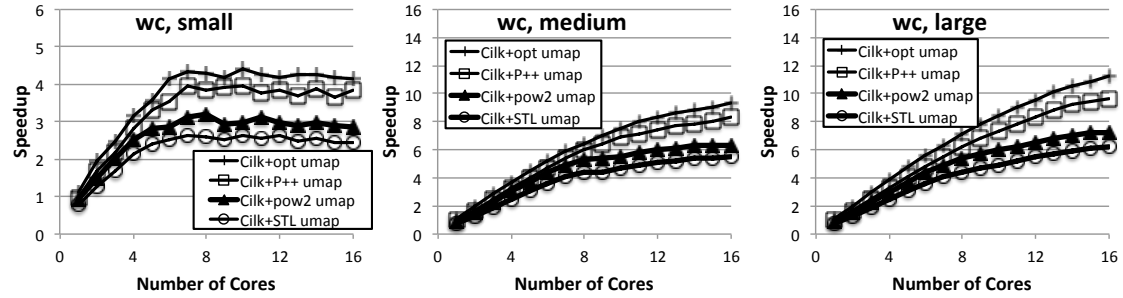

Figure 6.12: Performance of **wc** with various hash table implementations.

Our initial Cilk implementation of **wc** performed poorly. The reason hereto was that we used

the default C++ STL `unordered_map`. This hash map data structure performs badly for this application (Figure 6.12, line "STL umap") because it is optimized to balance performance against space. In a typical map-reduce application, however, insert operations dominate the execution time, so there is no scope to trade-off performance.

We optimized the STL unordered map by defining a resizing policy that restricts the hash table size to a power of 2 and a hash function that selects the lowest bits of the integer key to select a bucket in the hash table. This version performs better (Figure 6.12, line "pow2 umap"), but is still not as performant as the Phoenix++ hash table as the STL code uses dynamically allocated linked lists to store elements that map to the same bucket.

We further optimized the Phoenix++ hash table to store the full-length hash values in the hash table along with the key. This is possible as the hash function computes a hash that is independent of the hash table size. The hash table selects the appropriate bits from this to index its table. Storing the full-length hash saves time recalculating hash values when the hash table is resized. Moreover, we initialize the hash table to an application-specific size. This is possible because we know both the chunk size and the average number of words per character of text. We assume there is one unique word for every 16 bytes in a chunk. The optimized hash table out-performs the Phoenix++ hash table by 7.9% to 17.4% on 16 threads (Figure 6.12, line "opt umap").

We conclude that map-reduce applications are extremely sensitive to the performance of the data structures due to the generally low amount of computation per data structure access. This observation also motivated the specialization of data structures in Phoenix++ [34]. However, the only data structure that posed problems is the hash map in **wc**. All other applications use fixed-length arrays or structures, which introduce no performance overhead.

Vectorization

Cilk's array notation allows the expression of vectorizable code. This improves the speedup of **pca** on 16 threads from 15.2 to 17.9 over unvectorized sequential code (Figure 6.9). Note that applying the array notation to **pca** is trivial.

We furthermore applied the array notation to **lreg**. The regression computation requires integer multiplications, for which our x86-64 target has no efficient vector operations. The vector multiplications offered reduce the number of elements in a vector while increasing the precision of each element in order to handle possible overflows correctly. In this code, however, this is redundant as the initial coordinates are 8 bits wide and the target of the multiplication is 64 bits. As such, the main loop is not vectorized. However, the array notation results in more efficient assembly code. Also, the reduction operation is vectorized efficiently, resulting in an overall performance improvement by 8.99% over non-vectorized Cilk code on the large data set.

We also applied vectorization to **histogram** and **kmeans**. In these benchmarks, the vectorized code is restricted to the reduction operation, which contributes to a minor fraction of total execution time. We have not shown these results.

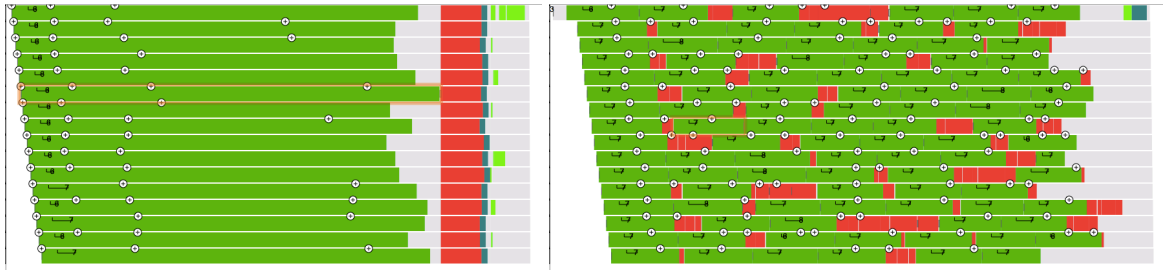

Figure 6.13: Traces for the **wc** benchmark executing on Phoenix++ (left) and Cilk (right). Legend: Map tasks (dark green), reduction (red), merge (light green) and sort (greyish blue). Cross marks show hash table resizes while black lines in map tasks indicate a rapid succession of resize operations. The Cilk version executes more slowly in this measurement because substantially more trace events are recorded.

Nested Parallelism

Finally, a language like Cilk allows nested parallelism, while Phoenix++ does not. In the case of **pca**, this means that Cilk allows to express a higher degree of parallelism than Phoenix++. We verified that removing nested parallelism from the Cilk implementation reduced the speedup on 16 threads from 15.2 to 13.9.

6.4.3 Analysis

Figure 6.13 illustrates the difference in execution pattern between Phoenix++ and Cilk. On the left hand side, an execution trace is shown for Phoenix++ showing the activity on each of the 16 threads. The trace shows sharply delineated phases where all threads first co-operate on the map task, then the reduce task, subsequently sort and finally merge. There is strong thread imbalance between stages.

We recorded a trace for the Cilk version when using the Phoenix++ hash table for fair comparison (Figure 6.13, right). Here, the map and reduce tasks are interleaved as a result of work stealing. **wc** follows the unbalanced pattern, implying that one hash table reduction is performed per map task. The Cilk version spends 60% more time in reduction compared to the Phoenix++ version.

In the case of Cilk, the merge step consists solely of serialization, i.e., transforming the hash table to a key-value list. The merge step is executed by a single thread. Sorting is executed in parallel but the contribution of individual threads is not recorded in the trace.

The trace reveals that hash tables are resized much more frequently in the Cilk version, about 5.5x times more often. This occurs because the size of a hash table is reset for every new map task and must gradually grow, while Phoenix++ has one hash table per thread and does not resets its size during the map phase.

6.4.4 Discussion

Our performance evaluation demonstrates that the specialization applied to the Phoenix++ runtime does not offer increased performance. Implementations of the same algorithms in a generic parallel programming language deliver significantly higher performance in all but one algorithm. The only major performance issue we encountered was the sensitivity of the **wc** benchmark on its unordered hash map data structure.

One may draw different conclusions from this work, either with practical consequences or conceptual consequences. A practical consequence is that the implementation of Phoenix++ needs to be revisited. Alternatively, one may decide to develop a specialized map-reduce runtime system building on the Cilk language, which should theoretically deliver the same results as we have found.

We believe, however, that the concept of a specialized map-reduce programming system *for shared memory systems* is conceptually broken. While map-reduce systems offer various advantages for distributed memory systems, such as support for fault-tolerance, data movement and straggler detection [8], they can provide only two benefits for shared memory systems: performance and ease of programming. On the down-side, the map-reduce model puts restrictions on the expressiveness of programs. The class of programs that can be represented in the map-reduce model is limited theoretically [10], but also from a practical point of view, e.g., see [19, 17]. To deliver performance, a shared memory map-reduce system like Phoenix [34, 6, 25, 21] offers two things: (i) a manner to express the map-reduce parallel pattern and (ii) data structures that are highly tuned to the application properties. Of these, (i) is offered by your favourite parallel programming language and (ii) can be delivered by means of a library.

As for the ease of programming, the Cilk code pattern is not complicated to apply. Moreover, a subtly undefined aspect of map-reduce is the commutativity of reductions [39]. This aspect of the programming model is most often not even documented, for instance in the Phoenix systems [29, 40, 34]. However, executing non-commutative reduction operations on a runtime system that assumes commutativity can lead to real program bugs [7] even in extensively tested programs [39]. We believe strongly that the non-commutative nature of Cilk reductions provides a better programming abstraction than the commutative abstractions applied in the majority of map-reduce runtimes.

Moreover, it has been documented repeatedly that not all algorithms fit the map-reduce model. Forcing an algorithm in the map-reduce straight-jacket is demanding for the programmer and will most likely lead to reduced performance. Matrix multiply and K-means are just two examples of algorithms that should not be programmed as map-reduce problems.

6.5 Conclusion

We have evaluated the suitability of the internal programming language for implementing map-reduce style applications. We performed this study by using a subset of the internal language,

named Cilk, and found that it provides better performance than specialized map-reduce systems on shared-memory machines. We will extend this result to distributed memory machines using the region concept, as discussed in Chapter 3.

We implemented a number of map-reduce benchmarks in the Cilk parallel programming language. Comparison against Phoenix++, a state-of-the-art shared-memory map-reduce runtime, shows that the Cilk versions of the code increase performance by 40–95%, while performance is reduced by 10% for one application, which has the lowest number of operations per byte transferred.

The key conclusions from this study are that, in order to achieve best performance, one should (i) use high-performance data structures with minimal overhead, (ii) avoid representing the data as key-value pairs unless required, especially in iterative map-reduce problems, and (iii) use existing optimized implementations of kernels when available. Note that these considerations are not hidden from the programmer by specialized map-reduce systems such as Phoenix or even Hadoop. As such, they must be known to data analysts. The ASAP project will remove this requirement on the data analyst by providing suitable operator abstractions, while such implementation details will be hidden in the implementation of the operators, i.e., at the level of the internal programming language.

Chapter 7

Conclusion

Analytics platforms provide high degrees of programmability by providing operators that describe common data manipulation operations at a high level of abstractions. These are easy to use for data analysts. The operators hide the details of parallel and distributed computing, data management, fault tolerance, etc. Internally, however, these operators are implemented in a parallel or distributed programming framework, but this is hidden from the data analyst.

The ASAP programming model follows a similar structure. We define high-level operators that describe commonly occurring patterns of operation. Moreover, we extend the way that these high-level operators may be combined. In particular, we allow nesting of operations on data sets in order to better manage important aspects of the computation, such as degree of parallelism, granularity of parallelism, locality optimization, etc.

Moreover, we define an internal programming language to implement the operators. The internal language is task-oriented, where tasks are operations on specific data sets. These data sets are describe explicitly by means of regions. Regions were originally introduced as a memory management technique, but in this case regions can also be used to manage large data sets in a distributed computing environment.

This document provides our current snapshot of the definition of the internal programming language and the high-level operators. We expect that some changes and extensions will be made during the course of the ASAP project.

Acronyms

HDFS Hadoop Distributed File System. 8, 9, 21

RDD Remote Distributed Dataset. 19–21

Bibliography

[1] Graphx 1.1.1 programming guide.

[2] Spark 1.1.1 programming guide.

[3] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '98, pages 119–129, New York, NY, USA, 1998. ACM.

[4] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: expressing locality and independence with logical regions. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pages 1–11. IEEE, 2012.

[5] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, pages 207–216, New York, NY, USA, 1995. ACM.

[6] R. Chen and H. Chen. Tiled-mapreduce: Efficient and flexible mapreduce processing on multicore with tiling. *ACM Trans. Archit. Code Optim.*, 10(1):3:1–3:30, April 2013.

[7] Christoph Csallner, Leonidas Fegaras, and Chengkai Li. New ideas track: Testing mapreduce-style programs. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 504–507, New York, NY, USA, 2011. ACM.

[8] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[9] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun.ACM*, 51(1):107–113, jan 2008.

[10] Benjamin Fish, Jeremy Kun, Ádám Dániel Lelkes, Lev Reyzin, and György Turán. On the computational complexity of mapreduce. *CoRR*, abs/1410.0245, 2014.

[11] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *PLDI '98: Proceedings of the 1998 ACM SIGPLAN conference on Programming language design and implementation*, pages 212–223, 1998.

[12] Matteo Frigo, Pablo Halpern, Charles E Leiserson, and Stephen Lewin-Berlin. Reducers and other cilk++ hyperobjects. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pages 79–90. ACM, 2009.

[13] David Gay and Alex Aiken. Memory management with explicit regions. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pages 313–323, New York, NY, USA, 1998. ACM.

[14] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 17–30, Berkeley, CA, USA, 2012. USENIX Association.

[15] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. Graphx: Graph processing in a distributed dataflow framework. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 599–613, Broomfield, CO, 2014. USENIX Association. ID: 9.

[16] Intel. *Intel Cilk Plus Language Extension Specification*, version 1.2. 324396-003us edition, September 2013.

[17] Silvio Lattanzi, Benjamin Moseley, Siddharth Suri, and Sergei Vassilvitskii. Filtering: A method for solving graph problems in mapreduce. In *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 85–94, New York, NY, USA, 2011. ACM.

[18] Charles E. Leiserson and Tao B. Schardl. A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers). In *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, pages 303–314, New York, NY, USA, 2010. ACM.

[19] Jimmy Lin and Michael Schatz. Design patterns for efficient graph algorithms in mapreduce. In *Proceedings of the Eighth Workshop on Mining and Learning with Graphs*, MLG '10, pages 78–85, New York, NY, USA, 2010. ACM.

[20] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proc.VLDB Endow.*, 5(8):716–727, apr 2012.

[21] Mian Lu, Lei Zhang, Huynh Phung Huynh, Zhongliang Ong, Yun Liang, Bingsheng He, R.S.M. Goh, and R. Huynh. Optimizing the mapreduce framework on intel xeon phi coprocessor. In *Big Data, 2013 IEEE International Conference on*, pages 125–130, Oct 2013.

[22] Spyros Lyberis, Polyvios Pratikakis, Dimitrios S. Nikolopoulos, Martin Schulz, Todd Gamblin, and Bronis R. de Supinski. The myrmics memory allocator: Hierarchical,message-passing allocation for global address spaces. In *Proceedings of the 2012 International Symposium on Memory Management*, ISMM '12, pages 15–24, New York, NY, USA, 2012. ACM.

[23] Spyros Lyberis, Polyvios Pratikakis, Dimitrios S Nikolopoulos, Martin Schulz, Todd Gamblin, and Bronis R de Supinski. The myrmics memory allocator: hierarchical, message-passing allocation for global address spaces. *ACM SIGPLAN Notices*, 47(11):15–24, 2013.

[24] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[25] Y. Mao, R. Morris, and F. Kaashoek. Optimizing MapReduce for multicore architectures. Technical Report MIT-CSAIL-TR-2010-020, MIT Computer Science and Artificial Intelligence Laboratory, 2010.

[26] M. Mccool, J. Reinders, and A. Robison. *Structured Parallel Programming: Patterns for Efficient Computation*. Morgan-Kaufman, 2012. ISBN: 0-12-415993-1.

[27] Vassilis Papaefstathiou, Manolis G.H. Katevenis, Dimitrios S. Nikolopoulos, and Dionisios Pnevmatikatos. Prefetching and cache management using task lifetimes. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 325–334, New York, NY, USA, 2013. ACM.

[28] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. The tao of parallelism in algorithms. *SIGPLAN Not.*, 46(6):12–25, jun 2011. ID: 6.

[29] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA '07, pages 13–24, Washington, DC, USA, 2007. IEEE Computer Society.

[30] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 472–488, New York, NY, USA, 2013. ACM.

[31] Sangwon Seo, Edward J. Yoon, Jaehong Kim, Seongwook Jin, Jin-Soo Kim, and Seungryoul Maeng. Hama: An efficient matrix computation with the mapreduce framework. In *Proceedings of the 2010 IEEE Second International Conference on Cloud Computing Technology and Science*, CLOUDCOM '10, pages 721–726, Washington, DC, USA, 2010. IEEE Computer Society.

[32] Julian Shun and Guy E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 135–146, New York, NY, USA, 2013. ACM.

[33] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 4th edition, 2013. ISBN 978-0321563842.

[34] J. Talbot, R. M. Yoo, and C. Kozyrakis. Phoenix++: Modular mapreduce for shared-memory systems. In *Proceedings of the Second International Workshop on MapReduce and Its Applications*, MapReduce '11, pages 9–16, New York, NY, USA, 2011. ACM.

[35] H. Vandierendonck, P. Pratikakis, and D. S. Nikolopoulos. Parallel programming of general-purpose programs using task-based programming models. In *Proceedings of the 3rd USENIX Workshop on Hot Topics in Parallelism (HotPar)*, page 6, May 2011.

[36] Hans Vandierendonck, George Tzenakis, and Dimitrios S Nikolopoulos. A unified scheduler for recursive and task dataflow parallelism. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 1–11. IEEE, 2011.

[37] Hans Vandierendonck, George Tzenakis, and Dimitrios S Nikolopoulos. Analysis of dependence tracking algorithms for task dataflow execution. *ACM Transactions on Architecture and Code Optimization (TACO)*, 10(4):61, 2013.

[38] Jeremiah James Willcock, Torsten Hoefler, Nicholas Gerard Edmonds, and Andrew Lumsdaine. Active pebbles: Parallel programming for data-driven applications. In *Proceedings of the International Conference on Supercomputing*, ICS '11, pages 235–244, New York, NY, USA, 2011. ACM.

[39] Tian Xiao, Jiaxing Zhang, Hucheng Zhou, Zhenyu Guo, Sean McDirmid, Wei Lin, Wenguang Chen, and Lidong Zhou. Nondeterminism in mapreduce considered harmful? an empirical study on non-commutative aggregators in mapreduce programs. In *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, pages 44–53, New York, NY, USA, 2014. ACM.

[40] R. M. Yoo, A. Romano, and C. Kozyrakis. Phoenix rebirth: Scalable mapreduce on a large-scale shared-memory system. In *Proceedings of the 2009 IEEE International Symposium*

on Workload Characterization (IISWC), IISWC '09, pages 198–207, Washington, DC, USA, 2009. IEEE Computer Society.

[41] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'13, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[42] Kaiyuan Zhang, Rong Chen, and Haibo Chen. Numa-aware graph-structured analytics. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 183–193, New York, NY, USA, 2015. ACM.

FP7 Project ASAP
Adaptable Scalable Analytics Platform

End of ASAP D2.1
Preliminary Definition of ASAP Programming Model

WP 2 – A Unified Analytics Programming Model

Nature: Report

Dissemination: Public