FP7 Project ASAP
Adaptable Scalable Analytics Platform

ASAP D2.3

Program analysis and transformation

WP 2 – A unified analytics programming model

Nature: Report

Dissemination: Public

Version History

Version	Date	Author	Comments
0.1	5 Feb 2017	H. Vandierendonck, K. Murphy, M. Arif, J. Sun, D.S. Nikolopoulos	Initial Version
0.2	28 Feb 2017	P. Pratikakis, H. Vandierendonck	Updated formatting
0.3	28 Feb 2017	H. Vandierendonck	Spelling and markup corrections

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 Swan programming language, designed for high-performance analytics, and demonstrates its application to a variety of analytics workloads, covering map-reduce workloads, graph analytics and text analytics.

Contents

1 Introduction

Data analytics workloads are consuming large amounts of computation time. As such, a detailed study of their computational patterns and properties is merited. The goal of Work Package 2 is to study these workloads in detail and to develop a programming language that enables programmers to express analytics workloads in such a way that they can be executed at high performance.

We developed the Swan language, an extension of the Intel Cilk parallel programming language, that facilitates high-performance execution of data analytics workloads. Swan adds three features to Cilk, namely: (i) data-flow extensions to express arbitrary parallel patterns and enable virtualization of memory; (ii) a scheduling hint for fine-grain parallel loops, and (iii) a scheduling hint to exploit locality-awareness in Non-Uniform Memory Access (NUMA) systems.

To a large extent, this work applies to shared memory systems, i.e., it is concerned with a single node in a data center. It should be noted, however, that servers with terabyte-scale main memory exist and offer a highly competitive alternative for workloads exhibiting frequent synchronization. The work is however not limited to shared memory systems as we apply some of our ideas also to Spark. Moreover, data-flow extensions may be used to orchestrate parallelism also in distributed memory systems without affecting the principles of the programming model [12, 72].

We demonstrate the usefulness of Swan for the data analytics problem by applying it to several analytics workloads. In first instance, we apply Swan to Map-Reduce workloads (Section 3). Map-reduce workloads are conceptually simple and are easily implemented using parallel loops with reducers. Contrary to popular frameworks such as Hadoop, our reducers have clearly defined semantics and do not require the *commutativity* property.

Next, we apply the Swan language to the problem of graph analytics (Section 4). Graph analytics differ from map-reduce problems in many respects. Most importantly, graph analytics problems involve irregular computations and have a high dynamic range of parallelism. In our study, we found that graph analytics are highly sensitive to the organization of the memory system in a server. We demonstrate how NUMA-aware scheduling has a significant impact on the performance of graph analytics.

Furthermore, we demonstrate how several of the map-reduce workloads as well as the graph analytics workloads benefit from the fine-grain scheduling hint (Section 5). The fine-grain scheduling hint expresses that certain parallel loops have very low operational intensity, i.e., they perform very few computations per byte transfered through the memory system. This property appears in several map-reduce workloads and, due to the high dynamic range of parallelism, also in a significant number of parallel loops in the graph analytics workloads.

Next, we turn our attention to text analytics problems. Text analytics problems again have the property of low operational intensity. In our case study of term frequency/inverse document frequency, however, this property implies that performance is highly sensitive to the organization of the data structures, the memory management of inter-

mediate data and efficiently managing parallelism. On the basis of this, we propose a number of operators and a library that are generally useful in text analytics. These are explained in Section 6. Finally, we demonstrate that our proposals for HPTA are also applicable to Spark (Section 7).

Overall, these case studies demonstrate that Swan is an appropriate programming language to express a variety of data analytics workloads. Rigorous performance evaluation, involving a comparison against state-of-the-art solutions for each of the problem domains, demonstrates that the goal of high-performance analytics is achievable with Swan.

2 The Swan Parallel Programming Language

Swan is an extension to the Cilk parallel programming language, which was originally designed at the Supercomputing Technologies group at MIT, and is currently supported by Intel under the name *Cilk Plus* [39]. Swan extends Cilk by adding dataflow dependences to express more complex parallel patterns than Cilk. One of these is pipeline parallelism. Moreover, Swan adds annotations to parallel for loops that help to increase performance.

2.1 The Cilk Parallel Programming Language

The following is a brief overview of the key aspects of Cilk. Full details are provided online [17].

2.1.1 Spawn and Sync

Parallelism is expressed by indicating that two (or more) pieces of code may execute in parallel. Typically, this implies that these pieces of code do not write to variables or memory locations that the other reads or writes to. These pieces of code may be any legal C/C++ code. In practice, Cilk requires that at least one piece of code is extracted in a function or isolated in a C++ lambda expression (an anonymous function).

Parallelism is introduced by adding the **cilk_spawn** keyword to the function call statement. We say that the function is *spawned* rather than *called*. The spawned function may execute in parallel with the remainder of the calling function. This is called the *continuation* of the calling function. The parallelism exists until a **cilk_sync** statement is encountered, or until the end of the calling function, whichever is encountered first.

The Cilk runtime is allowed to execute the spawned function in parallel with the continuation of the calling function, but is not obliged to do so. In fact, the Cilk runtime only executes in parallel as many spawns as is required to keep all CPU cores busy. Beyond this, it executes the spawned statements in a sequential manner, as this is much more efficient. Adding spawn statements to a program, thus, has little overhead in case they are not selected for parallel execution by the runtime.

```
void qsort(int *begin, int *end) {
    if (begin != end) {
        --end; // Exclude last element (pivot) from partition
        int * middle = std::partition (begin, end,
                                       std::bind2nd(std::less<int>(), *end));
        std::swap(*end, *middle);     // move pivot to middle
        cilk_spawn qsort(begin, middle);
        qsort(++middle, ++end); // Exclude pivot and restore end
        cilk_sync;
    }
}
```

Figure 1: Quicksort expressed in Cilk.

```
cilk_for (int i=0; i < n; ++i) {
    a[i] = ...;
}
```

Figure 2: Parallel loops in Cilk.

An example of Quicksort expressed in Cilk is shown in (Figure 1). Quicksort first partitions the range of values to sort using a pivot. It then recursively sorts the range of values less than the pivot and the range of values larger than the pivot. As these subranges are independent (non-overlapping), they can be sorted in parallel. This is indicated by labelling the first recursive call with **cilk_spawn**.

Note that one could also add the **cilk_spawn** keyword to the second recursive call. This is however redundant as it was already apparent that this call may execute in parallel with the first recursive call.

2.1.2 Parallel for loops

The spawn/sync mechanism is very versatile, (Figure 2). It can be used to express parallel for loops, a common idiom, as well. However, this is somewhat tedious. The Cilk compiler allows programmers to annotate parallel for loops using the **cilk_for** keyword as shown in Figure 2.

The compiler outlines the body of a **cilk_for** loop in a distinct function and generates code to call the loop body in parallel, using the **cilk_spawn** statement.

Every iteration of the loop should modify distinct memory locations. There are moreover restrictions on the structure of the loop iteration. In essence, the number of iterations of the loop must be known at execution time just before starting the loop. This implies that the loop should not have **break** statements and that the loop iteration variable (**i**) is modified only by the loop increment statement **++i**, the third part in the for loop syntax.

```
template<class map_type>
class map_reducer {
    struct Monoid : cilk::monoid_base<map_type> {
        static void reduce(map_type * left, map_type * right) {
            for(typename map_type::const_iterator
                    I=right->cbegin(), E=right->cend(); I != E; ++I)
                (* left )[I->first] += I->second;
            right->clear();
        }
        static void identity (map_type * p) const {
            new (p) map_type();
        }
    };
    cilk::reducer<Monoid> imp_;

public:
    map_reducer() : imp_() { }
    typename map_type::value_type & operator[](
            const typename map_type::key_type & key) {
        return imp_.view()[key];
    }
    typename map_type::const_iterator cbegin() {
        return imp_.view().cbegin();
    }
    typename map_type::const_iterator cend() {
        return imp_.view().cend();
    }
    void swap(map_type & other) {
        return imp_.view().swap(other);
    }
    map_type & get_value() {
        return imp_.view();
    }
};
```

Figure 3: Cilk reducer for hash-map.

2.1.3 Generalized Reductions

Cilk provides definitions for generalized reductions that are associative but not necessarily commutative. 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.

Cilk defines reductions with three components: a data type, an associative operation and an identity value for that operation. These components are defined in a monoid class that serves as the basis for a **reducer** class definition.

The definition of a Cilk reducer for a hash-map data type is shown in (Figure 3)

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

Figure 4: map_reducer code.

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, which defines the base type (through the **monoid_base** template parameter), the identity value (through an initialization function) and the reduction function. 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.

The runtime system dynamically creates copies of the reduction variable, and reduces those copies as needed. These copies are called *views*. A view is created for a worker thread when it first accesses the reduction variable. The view is initialized with the identity element. The worker retains the view when spawning a task. When an idle worker steals a continuation from another worker's deque, it does not receive a view for that reduction variable. The view is created only on the first access to the reduction variable. When a worker completes a spawned task leaving its spawn deque empty, or when a worker executes a **cilk_sync** statement, the view is reduced with that of a sibling task.

The example above defines a **map_reducer** class. The member value **imp_** 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 further provides access to the underlying view through the **operator []** in order to add items to the hash-map.

The **map_reducer** class may be used in parallel code as shown in (Figure 4).

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

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. The number of reduction operations is, in any case, proportional to the number of steal operations.

2.1.4 Array Notation

Cilk Plus supports an array notation that facilitates auto-vectorization, i.e., the use of SIMD vector instructions to accelerate processing. 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

```
for(int i=0; i < n; ++i)  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.

2.2 Swan's Dataflow dependences

Dataflow dependencies enable Swan to schedule dependent tasks as soon as prior tasks complete [97, 98]. We have demonstrated that Swan out-performs state-of-the-art systems for scheduling dependent tasks in high-performance computing [95]. Dependencies are tracked at the object level. An object must be declared as a **versioned** object in order to enable dependency tracking. Versioned objects support automatic tracking of dependencies as well as creating new versions of the object in order to increase task-level parallelism (a.k.a. renaming).

Dependency tracking is enabled on tasks that take particular types as arguments: the **indep**, **outdep** and **inoutdep** types. These types are little more than a wrapper around a versioned object that extends its type with the memory access mode of the task: input, ouput or input/output (in/out). The language allows only to pass versioned objects to such arguments.

When spawning a task, the scheduler analyzes the signature of the spawned procedure for arguments with a memory access mode. If none of the arguments describe a memory access mode, then the spawn statement is an *unconditional spawn* and it has the same semantics as a Cilk spawn. Otherwise, the spawn statement is a *conditional spawn*. The memory accesses of the task are tracked and, depending on runtime conditions, the task either executes immediately or it is queued up in a set of pending tasks.

The **sync** statement in our language has the same semantics as the Cilk sync statement: it postpones the execution of a procedure until all child tasks have finished execution.

```
typedef float (*block_t)[16]; // 16x16 tile
typedef swan::versioned<float[16][16]> vers_block_t;
typedef swan::indep<float[16][16]> in_block_t;
typedef swan::inoutdep<float[16][16]> inout_block_t;

void mul_add(in_block_t A, in_block_t B, inout_block_t C) {
    block_t a = (block_t)A; // Recover pointers
    block_t b = (block_t)B; // to the raw data
    block_t c = (block_t)C; // from the versioned objects
    // ... serial implementation on a 16x16 tile ...
}

void matmul(vers_block_t * A, vers_block_t * B,
            vers_block_t * C, unsigned n) {
    for( unsigned i=0; i < n; ++i ) {
        for( unsigned j=0; j < n; ++j ) {
            for( unsigned k=0; k < n; ++k ) {
                cilk_spawn mul_add( (in_block_t)A[i*n+j],
                                    (in_block_t)B[j*n+k],
                                    (inout_block_t)C[i*n+k] );
            }
        }
    }
    cilk_sync;
}
```

Figure 5: Matrix multiplication by blocks.

We consider only situations where dependencies are tracked between the children of a single parent procedure. Each dynamic procedure instance may have a task graph that restricts the execution order of its children. This restriction ensures that all parallel executions compute the same value as the sequential elision of the program. Note that the sequential elision of the program always respects the dependencies in the program: by deducing dependencies from input/output properties, there can never be backward dependencies in the sequential elision. Furthermore, by having multiple independent task graphs in a program, we can mitigate the performance impact of building the task graph in serial fashion.

Our model allows arbitrarily mixing fork/join style and task graph execution. The only problematic issue to allow this is that we must take care when nesting task graphs, in particular when passing versioned objects across multiple dependent spawns. To make this work correctly, we must use distinct metadata for every dependent spawn to track its dependencies separately.

Figure 5 shows an example of square matrix multiplication expressed in Swan using runtime tracking and enforcement of task dependencies. Here, the matrix multiplication is performed *by blocks*, i.e., matrices are partitioned in sub-blocks and parallelism between operations on sub-blocks is made explicit using data-flow annotations.

```
    int chunk = (len + num_numa_domains − 1) / num_numa_domains;
#pragma cilk numa(strict)
    cilk_for (int d=0; d < num_numa_domains; ++d)
        cilk_for (int i=d∗chunk; i < std::min((d+1)∗chunk,len); ++i)
            a[i] = ...;
```

Figure 6: Cilk numa annotation.

2.3 NUMA-Aware Loop Scheduling

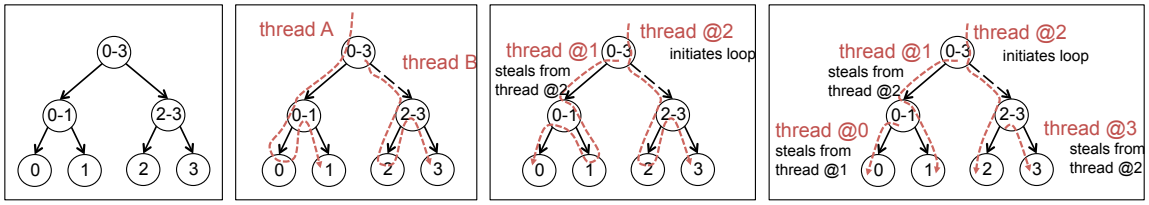
Cilk is a highly efficient work-stealing scheduler for parallel programs that achieves its performance through randomized, greedy scheduling. Such scheduling, however, is agnostic of the memory hierarchy and favors cache-oblivious programs [27, 106]. NUMA-awareness, however, relates to *scheduling* of memory accesses whereas cache-obliviousness relates to *locality* of access. These are distinct problem dimensions. For this reason, Swan extends the Cilk language and runtime system to support programmers to express NUMA-aware scheduling and work stealing. We have deliberately searched for a minimalistic modification to Cilk in order to retain its space- and time-efficiency [11].

We focus this extension exclusively on parallel loops. Parallel loops are, as we will demonstrate later, the most important idiom in data analytics. Swan adds an annotation to **cilk_for** loops which informs the runtime how to schedule tasks on CPU cores in a NUMA-aware manner.

The **numa** annotation facilitates performance tuning for systems with a Non-Uniform Memory Architecture (NUMA), e.g., multi-socket machines. This annotation indicates that the iterations of the loop should be scheduled on distinct NUMA domains (sockets). It is the programmer's responsibility to ensure that there are no more loop iterations than NUMA domains.

Figure 6 shows how the annotation is used. The outer loop (with loop iteration variable **d**) is annotated as a NUMA loop. Each iteration of this loop will be executed on a distinct NUMA domain. The iterations of the inner loop (using loop iteration variable **i**) are spread over the CPU cores of one the NUMA domain of the corresponding **d** value.

The assumption that the number of loop iterations does not exceed the number of NUMA domains is a pragmatic one. Longer iteration ranges are supported by distributing the loop range over a set of nested loops. The outer loop is NUMA-aware. The inner loop is a normal **cilk_for** loop that inherits the NUMA scheduling restriction from its calling context. Note that the NUMA-awareness property percolates to all code called recursively from the loop, until another loop with the NUMA pragma is encountered.



(a) Cilk spawn tree (b) Cilk work-first (depth-first) execution (c) NUMA-aware execution order for threads 1 and 2 (d) NUMA-aware execution order with four threads

Figure 7: NUMA-aware work-stealing. “Thread @n” represents any thread executing on socket “n”.

2.3.1 Implementation

Cilk implements parallel loops using a helper function that recursively splits the iteration range of the loop in half. Once the iteration range is shorter than a heuristically determined threshold the helper function executes the loop sequentially over this part of iteration range.

Figure 7 (a) shows the call tree of the helper function for a loop with 4 iterations. Each node represents an invocation of the helper function. Edges indicate a parent-child relationship between function calls. Nodes in distinct subtrees are independent and may execute concurrently. Cilk uses a work-first scheduler [11] which translates into a depth-first traversal of the tree (Figure 7 (b)). Work stealing is used to distribute work. Every idle thread attempts to steal work from a randomly selected victim thread. Threads steal the continuation of the oldest function on their victim’s call stack, i.e., the one nearest to the root of the call tree. E.g., if thread A starts execution of the range 0-3 in depth-first order it will first execute the sub-range 0-1. Meanwhile, thread B may steal the continuation of the oldest function and execute the sub-range 2-3.

We provide a NUMA-aware helper function that changes the execution order of loop iterations. The thread that executes an instance of the helper function checks its current NUMA domain and first executes the sub-range that matches its NUMA domain. E.g., if a thread on NUMA domain 2 initiates execution of the loop, it executes the range 2-3 before the range 0-1 (Figure 7 (c)). This strategy is applied recursively: a thread on NUMA domain 3 will first execute loop iteration 3. This way, work is distributed to the correct NUMA domains with a minimal number of work stealing events (Figure 7 (d)).

Work stealing is modified to respect the NUMA constraints. Every dynamic function call is marked by the helper function with the range of NUMA nodes where the function may execute. This range reflects the iteration sub-range of the loop. The range is copied over to recursively called functions. A worker that selects a victim thread inspects the NUMA range of the victim’s oldest function and aborts the work stealing attempt if the NUMA range does not contain its own NUMA node. By default, NUMA ranges are not set and work stealing proceeds as normal.

2.4 Fine-Grain Scheduling Hint

Swan furthermore accelerates execution of fine-grain parallel loops. Fine-grain parallel loops are loops that perform little work overall and have a parallel execution time that is typically less than 10 ms. For such loops it is often a difficult choice whether a programmer should mark them as parallel loops or not, as scheduling loop iterations over multiple CPU cores incurs an overhead, known as the scheduler burden. The scheduler burden can offset, or even annihilate, the performance gains from parallelization. As such, marking a fine-grain loop as parallel may reduce performance rather than improve it.

The fine-grain scheduling hint aims to address this problem: fine-grain loops are treated specially by a scheduler which has a lower burden, but is in turn less flexible in its schedule. A particular short-coming of the fine-grain scheduler is its inability to load balance parallel loops. However, we find that parallel loops in data analytics are often well-balanced. As such, the short-coming of the fine-grain scheduler is rarely a problem.

2.4.1 Language Extension

We introduce a new pragma "`#pragma cilk finegrain`" that can be added immediately before **cilk_for** loops:

```
    cilk::reducer<cilk::op_add<int> > r;
#pragma cilk finegrain
    cilk_for(int i = 0; i < n; i++)
        *r += a[i];
```

This tells the compiler that the annotated loop is a fine-grain parallel loop and should be scheduled using an optimized scheduler rather than using the dynamic scheduler. This optimized scheduler can handle these fine-grain loops as it has a lower burden by design. In this work, the burden is reduced by pre-calculating the distribution of loop iterations over threads and not allowing work stealing. Other scheduling techniques may be used as well without diminishing the contributions of this work. Moreover, we assume that no nested parallelism exists in the fine-grain loop. This is reasonable as otherwise the loop would likely not be fine-grain. We demonstrate that the fine-grain scheduler retains the functionality of Cilk hyperobjects, which is an important contributor to the ease-of-use of Cilk.

2.4.2 Cilk Application Binary Interface

The Cilk compiler replaces Cilk keywords with Application Binary Interface (ABI) calls [40]. Figure 8 shows a Cilk program that accumulates the values in an array using a reducer `r`. The operation `*r` is a short-hand to lookup the current thread's view for `r` and serves the relevant view for the the calling thread.

Figure 9 shows equivalent C++ code for Figure 8 rather than assembly code to ease the exposition. The `accumulate_data` structure is a capture for the free variables

```
int accumulate( int *a, int n ) {
    cilk::reducer<cilk::op_add<int> > r;
    cilk_for (int i = 0; i < n; i++)
        *r += a[i];
    return r.get_value();
}
```

Figure 8: A Cilk loop with a reducer.

```
struct accumulate_data {
  int *a;
  cilk::reducer<cilk::op_add<int> > *r;
};
void accumulate_helper( void *data, int start, int end ) {
    accumulate_data * d = (accumulate_data *)data;
    int * view = __cilkrts_hyper_lookup ( d->r );
    for(int i = start; i < end; i++)
        *view += (d->a)[i];
}
int accumulate( int *a, int n ) {
    cilk::reducer<cilk::op_add<int> > r;
    struct accumulate_data data = { a, &r };
    __cilkrts_cilk_for_32 (
        accumulate_helper, (void *)&data, n, 0 );
    return r.get_value();
}
```

Figure 9: Code tranformed by Cilk compiler.

in the loop body. The `accumulate_helper` function sequentially executes a sub-range of the loop as given by its arguments. An ABI call to `__cilkrts_hyper_lookup` is used to retrieve the current view for the reducer. Note that this call is made from within the `cilk::reducer` class and is not inserted by the compiler. The compiler however hoists this call out of the loop for performance reasons [40]. Finally, the loop is replaced by the `__cilkrts_cilk_for_32` ABI call that enables parallel execution of the loop. It recursively decomposes the loop iteration range and executes the helper function on short iteration ranges.

2.4.3 ABI Extension

Fine-grain parallel loops are handled in a similar way, albeit with two changes (Figure 10): (i) a new ABI function `__cilkrts_cilk_for_static_32` is called; (ii) some preparatory work is performed to optimize the lookup of reducers, and the `__cilkrts_hyper_lookup` calls is replaced by a version tuned to the parallel pattern. The motivation behind these changes is described in the next Section.

```
struct accumulate_data {
  int *a;
  __cilkrts_hyperobject_base **hyper_array;
};
void accumulate_helper( void *data, int start, int end ) {
    accumulate_data * d = (accumulate_data *)data;
    int * view = __cilkrts_hyper_array_lookup (d->hyper_array, 0);
    for(int i = start; i < end; i++)
        *view += (d->a)[i];
}
int accumulate( int *a, int n ) {
    cilk::reducer<cilk::op_add<int> > r;
    __cilkrts_hyperobject_base * obj[1] = { &r }; // r at index 0
    struct accumulate_data data = { a, obj };
    __cilkrts_cilk_for_static_32 (
        accumulate_helper, (void *)&data, n, 0, obj, 1);
    return r.get_value();
}
```

Figure 10: Equivalent code of the accumulation loop with fine-grain pragma.

2.4.4 Scheduling Fine-Grain Parallel Loops

We consider scheduling of parallel loops without cross-iteration dependences, except for the presence of reduction operations. Such loops are statically scheduled by following 4 steps. These steps are initiated by the thread that encounters the parallel loop, which we call the master thread [68]:

1. **Scheduling**: The master thread divides the loop iteration range in equal chunks, one for each of the threads that will contribute to the execution of the loop.

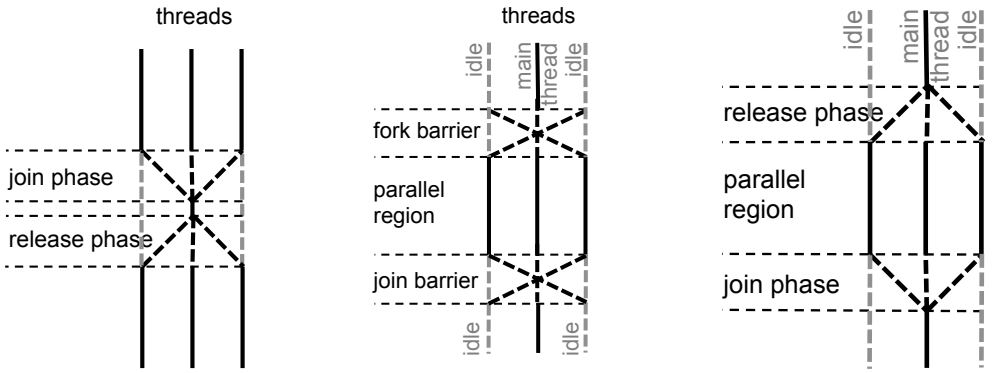
2. **Work distribution**: The master thread sends work descriptions to the workers. These include the task (typically identified by a function pointer) and the portion of the loop iteration range assigned to the worker.

3. **Parallel execution**: Worker threads initialize local copies of reduction variables and start executing their work as soon as they have obtained it. The master typically also executes part of the work.

4. **Synchronizing on completion**: The master thread typically needs to wait for the workers to complete. Partial results for reduction variables are reduced into the actual reduction variable.

This template assumes that the loop iteration range is known when the loop is encountered and that there are is no unexpected control flow leaving the loop.

Steps 2 and 3 involve synchronization and are typically implemented using barriers. Barriers involve two phases of synchronization: the *join phase* and the the *release phase* (Figure 11(a)). The join phase records the arrival of threads. The release phase



(a) Barrier structure (b) Parallel loop structure (c) Half-barrier schematic

Figure 11: Schematic structure of threads and synchronization in parallel loops and barriers

signals threads that they can enter the next phase of computation. Step 2 is known as a *fork barrier*. Step 4 is a *join barrier* (Figure 11(b)). As such, an implementation of a parallel loop scheduler has at least two barriers per parallel loop. Additional synchronization may be required. E.g., OpenMP supports dynamic teams, a grouping of worker threads that will be involved in executing the current parallel region [68]. Identifying what threads are available and which ones will be involved enforces additional synchronization.

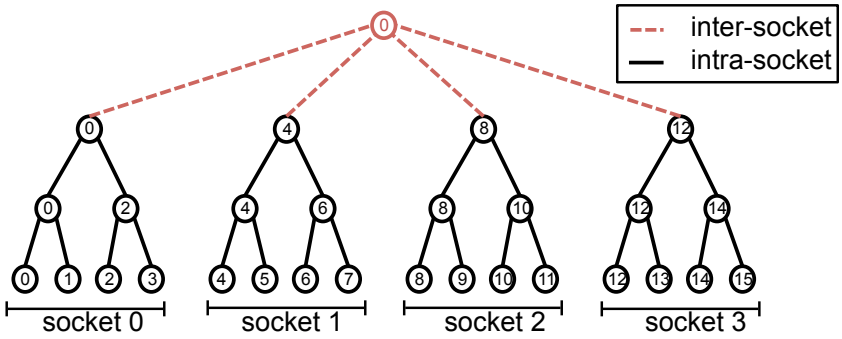
Half-Barrier Pattern

The barrier synchronization pattern involves redundant synchronization when synchronizing parallel loops. We build on the assumption that the worker threads are associated to a specific master thread. The master encounters a parallel code region and shares the work with the workers. Under this assumption, which is true in many runtimes, the following observation can be made:

Observation #1: The worker threads are available at the start of a parallel region. The workers are available to take on work because they are associated to the master. If the master is not executing inside a parallel region, then the workers must be idle. As such, it is not necessary for the master to ensure that the workers have arrived at the fork barrier. Workers need to wait, however, to receive their assigned work.

Observation #2: The worker threads are independent of one another. As we focus on loops where tasks are independent and synchronization-free units of work, there is no dependence between workers executing distinct tasks. Combining Observations 1 and 2, we conclude that it is unnecessary to execute the join part of the fork barrier.

Observation #3: When worker threads leave the parallel region, they are independent of the activities of the master thread during the parallel region. There is no data or control dependency that needs to be enforced between the master's computation during the parallel region and the workers past the parallel region. As such,



(a) Tree-structured barrier synchronisation

Figure 12: NUMA-aware tree barrier. Each leaf node is associated to a thread, numbered 0–15 in this example. Internal nodes of the tree represent point-to-point synchronization actions between the threads listed in the node's children.

the release phase of the join barrier is a redundant synchronization step.

Figure 11(c) illustrates the observations schematically. Compared to Figure 11(b), redundant synchronization is removed as the master thread signals workers that new work has arrived, without waiting for acknowledgement of the workers. Similarly, once workers have confirmed the completion of their work, there is strictly no need for the master to acknowledge receipt of the message to the workers. This way, the work distribution and synchronization overhead is reduced to one instead of two barriers.

The half-barrier pattern is also applicable to nested parallelism provided that the worker threads in the inner region are uniquely assigned to a worker in the outer region that acts as their master.

NUMA-Optimized Half-Barrier

The performance of barriers is dependent on the read and write patterns to shared memory locations used by the barrier. Scalable algorithms ensure that each shared variable is read by at most one thread and written by at most one thread [58]. The most efficient barrier styles are the dissemination barrier [13, 36] and the tree barrier [36, 58, 63].

In the dissemination barrier, each of P threads sends a signal to another thread during $\lceil \log_2 P \rceil$ rounds of communication [13], totalling $\mathcal{O}(P \log P)$ messages. The same synchronization pattern is used during join and release phases.

Tree barriers are the most efficient on architectures with broadcast capability in the cache coherence protocol, e.g., those using snooping protocols [63, 21]. As our experimental platform uses a snooping bus, we focus on tree barriers.

Tree barriers organize mini-tournaments between pairs of threads. The nodes of the tree implement centralized barriers for 2 threads. Threads perform the join phase of the centralized barriers in each node on the path from their leaf node to the root during the join phase. They traverse the tree from the root down to their leaf during the release phase [36, 58]. The tree barrier has a critical message path of $\lceil \log_2 P \rceil$,

which is similar to the dissemination barrier. However, during each round at most two messages are sent per thread. This makes it more bandwidth-efficient, which is important for bus-based architectures.

Tree barriers can be tuned to the underlying architecture by tuning the branching factor of the tree, by using different branching factors (known as fan-in and fan-out) in the join and release phases, and by using different branching factors at different levels of the tree [63].

We have optimized the half-barrier for best performance on our experimental machine, which is a 4-socket multi-core. We experimented with combinations of the centralized and tree barriers and matched their composition to the machine organization. The best performing configurations are:

- **NUMA-aware centralized barrier**: a distinct centralized barrier is used per socket. One thread on each socket is designated to synchronize with the other sockets using a centralized barrier.
- **NUMA-aware tree barrier**: a tree barrier is used with distinct branching factors at each level (Figure 12(a)). At the top-most level, the branching factor equals the number of sockets. Each sub-tree below this level is bound to a specific socket and has a branching factor of 2. We found no benefit from using a different branching factor in the join and reduce phases.

The NUMA-aware tree barrier is illustrated in Figure 12(a). Note that by using pairwise synchronization in the tree barrier it is possible to implement the barrier without costly atomic operations [63].

Integration With Dynamic Scheduler

We extend the Cilk work stealing algorithm such that applications can utilise both static and dynamic scheduling. Normally, an idle Cilk worker thread executes the random work stealing algorithm, whereby a worker randomly selects a victim worker and attempts to steal a stack frame from its double-ended work queue. The work steal attempt is successful when the victim's queue is not empty. Execution of the stolen stack frame is resumed and, when the worker returns to being idle, the process is repeated.

We extend the work stealing algorithm by alternating a cycle of the random work stealing algorithm with listening on the worker's flag variable in the tree barrier. Eventually, one of these actions will succeed and the worker continues either the static or dynamic scheduling protocol. This incurs a minor overhead on either scheduler type as additional instructions are executed only when the worker is idle.

The static scheduler may, if necessary, revert to sequential execution of parallel loops, or to execute them on the default scheduler. This is necessary, e.g., when the fine-grain scheduler is called concurrently from multiple threads. In such cases, one needs to trade-off executing the loop sequentially, or postponing the execution until the fine-grain threads are available. We opted to revert to sequential execution.

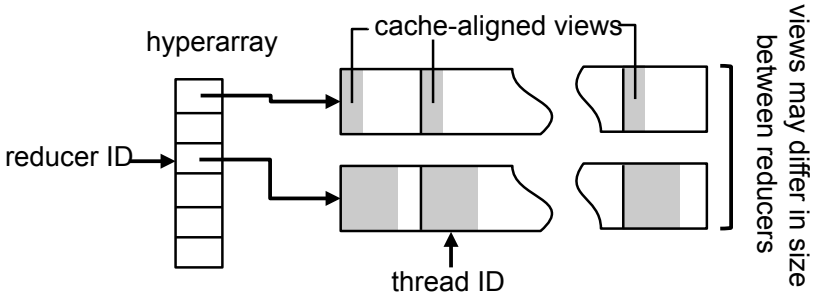


Figure 13: The hyperarray lays out views in consecutive, cache-aligned memory locations. It is indexed using the reducer's numeric ID within the loop and the thread ID.

2.4.5 Reductions

The half-barrier pattern can be leveraged to structure reductions, allowing to parallelize associative operations such as addition, multiplication and list concatenation [26].

Reducers in The Static Scheduler

Cilk lazily creates and reduces views as a new view is required only upon work-stealing, which is far less than the number of **spawn** statements executed. As such, laziness results in economy. In the case of static scheduling, the same design results in runtime overhead, which is important for fine-grain loops. As such, we implement reducers more efficiently but without changing the programming interface.

We make the simplifying assumption that it is known ahead of time what reducers are used in the loop. Hereto, the compiler needs to perform a static analysis that uniquely idenfies all reducers referenced in the loop. This is generally possibly, as discussed below, in part because we focus on fine-grain loops.

The fine-grain parallel loop ABI method pre-allocates a view for every reducer used in the loop and for every thread. The views are aligned to cache-line boundaries to avoid cache line sharing between processors, which may lead to ping-pongs in the coherence protocol. Each thread initializes its own views prior to executing the loop body. Views are reduced and destroyed as part of pairwise thread synchronization in the join phase of the tree barrier.

The runtime allocates one array of views for each reducer used in a parallel loop (Figure 13). All arrays of views are stored in a *hyperarray* to facilitate lookup of views. Cient code looks up a view through a new ABI call, `__cilkrts_hyper_array_lookup`, that takes as argument a hyperarray (an array of pointers to reducers) and the index in of the requested reducer in the hyperarray. The index is assigned by the compiler. Figure 10 shows the construction of the hyperarray (Line 14) and the lookup of views (Line 7).

By pro-actively creating instances of the reduction variable for each thread, we perform exactly $\lceil \log_2 P \rceil$ reduction operations for P threads. In contrast, Cilk performs as many reductions as successful work steal events [52], which may be significantly

higher than P.

This technique applies to Cilk's non-commutative reducers provided as every thread is assigned a contiguous range of loop iterations.

Restrictions to Fine-Grain Loops

It is required that the compiler can list all hyperobjects accessed within a fine-grain loop. This requires that:

- The compiler can inspect all code executed in the loop body. A sufficient solution is that all functions called from the loop body can be disambiguated at compile-time (they are not called through function pointers) and that they are part of the current compilation unit.
- Reducers may not be created or destroyed during execution of the loop.
- All addresses of hyperobjects are loop-constant. This condition implies that the hyper-lookup calls can be hoisted out of the loop [40].
- All (pointers to) hyperobjects referenced in the loop concern distinct hyperobjects. In practice, most loops have only one reducer.

These constraints are tested at compile-time.

2.5 Combining NUMA and Fine-Grain Annotations

The annotations for NUMA-aware scheduling and fine-grain loops can be used in the same program. It makes however no sense to place both annotations on the same loop as a loop suitable for NUMA-aware scheduling will typically contain a nested loop to distribute work over the attached CPU socket. As such, the NUMA-aware loop is typically not fine-grain itself. The nested loop, however, may be annotated as a fine-grain loop, as in the following example:

```
    cilk::reducer<cilk::op_add<int>> r;
    int chunk = (n + num_numa_domains − 1) / num_numa_domains;
#pragma cilk numa(strict)
    cilk_for (int d=0; d < num_numa_domains; ++d) {
#pragma cilk finegrain
        cilk_for (int i=d∗chunk; i < std::min((d+1)∗chunk,len); ++i)
            ∗r += a[i];
    }
```

In this case, the fine-grain scheduler selects one master thread on each sockets and enables this master to schedule work within its socket. In this case, the same scheduling tree is used as in Figure 12(a). The per-socket master threads, however, utilise only the per-socket sub-trees one level down from the root of the tree.

2.6 Implementation

We have implemented Swan in commercial-grade software systems. The Swan runtime system extends the Intel Cilk runtime system and is publically available on GitHub[1]. The Swan language extensions are implemented in the clang compiler[2] and require minor extensions to LLVM[3]. When setting up the compiler and runtime, the repository containing test cases and documentation will be helpful[4].

3 Case Study: Map/Reduce Applications

In this Section, we explore the applicability of the Cilk programming model for data analytics. This exploration was presented at the Third Workshop on Advances in Software and Hardware for Big Data to Knowledge Discovery, co-located with IEEE Big Data [4]. In the following Sections, we show how the extensions in Swan improve performance for data analytics workloads.

Our analysis assumes large, high-end compute servers. We study applications executing on a single server. Conclusions may be extrapolated to clusters of computers and data centers as similar computations are repeated across all nodes in such systems.

The design of a data analytics programming environment must meet two constraints that are often contradictory. First, performance must support the timely processing of large data sets. Second, programmability must be high since data analysts, who are likely not HPC experts, program the systems. The map-reduce system and its API [20] achieve both objectives for distributed memory systems (clusters): the API hides key performance aspects, such as data access and movement, load balancing and fault-tolerance. while supporting efficient processing.

Map-reduce on shared memory machines is different. Data movement is vertical between levels of the memory hierarchy instead of horizontal between compute nodes. Load balancing can be achieved at a much finer granularity of work items and shared memory map-reduce runtime systems do not usually provide fault tolerance [76, 105, 92, 59, 57].

Our work explores how to structure the map-reduce API and runtime on shared memory systems to maximize performance and ease-of-programming. We derive our solution from an analysis of the main short-comings of existing map-reduce systems.

Framework overheads exist with all systems. Map-reduce systems, however, aggravate them by the need to fit applications to the map-reduce API. The class of programs that can be represented in the map-reduce model is limited theoretically [24] and also from a practical point of view, e.g., in graph analysis [55, 50]. Programmers thus need to overcome hurdles in order to fit their program to the map-reduce model.

[1] `https://github.com/project-asap/swan_runtime`
[2] `https://github.com/project-asap/swan_clang`
[3] `https://github.com/project-asap/swan_llvm`
[4] `https://github.com/project-asap/swan_tests`

Appropriate intermediate data structures are necessary to maximize performance [92]. However, map-reduce systems require the use of lists of key-value pairs that lose the structure of the data, which must be sorted and grouped by key, increasing the time to retrieve individual data elements. In contrast, our map-reduce system admits arbitrary data structures. Programmers can use the most appropriate data structure, including hash maps or arrays.

We build our map-reduce runtime on top of the Cilk programming language [28, 39] and call it *CilkMR*. Cilk offers a simple means to express parallel loops (using the `cilk_for` syntax) and reductions (using generalized reducer hyperobjects [26]). These two concepts have the same conceptual programming complexity as other map-reduce systems [76, 105, 92, 59, 57]. CilkMR, however, retains the structure of the sequential code, which is in stark contrast to previously proposed map-reduce frameworks. Our design choices are not specific to Cilk but could be repeated using other efficient parallel programming languages with similar functionality.

We have also analyzed the the performance and programmability of OpenMP [2] for map-reduce workloads [5]. OpenMP user-defined reductions allow programmers to express complex and application-specific reduction operations. The mechanism, however, assumes a parallel tree reduction pattern, as reduction operators are defined on two arguments. In this study we found that a better way to execute reductions on containers in parallel is to assign keys to threads and to make each thread reduce all values for its keys. This way there are no data dependences between threads during the reduction phase. This is however at odds with the way OpenMP and Cilk express reduction operations.

3.1 Related Work

Several research projects have investigated efficient map-reduce runtime systems for shared memory systems. The first paper on the Phoenix system [76] compared it against a POSIX threads implementation of the benchmarks. Phoenix matched the performance of the POSIX threads codes that fit the map-reduce model but performed less well on the few that did not. We demonstrate that CilkMR outperforms the latest Phoenix++ runtime system on all but two benchmarks. Because CilkMR can compose map-reduce and other parallel code, it outperforms Phoenix++ by up to 4x on programs that do not match the map-reduce programming model well.

Yoo *et al* [105] improved the scalability of Phoenix for a 256-thread SPARC T2 machine. They found that the internal data structures that store intermediate data are critical for performance. They re-designed the data structures to reduce pointer indirection, fragmentation of data structures and to improve memory allocation.

Talbot *et al* [92] specialized the internal data structures to the applications. This specialization, for instance, replaces generic key-value maps with arrays when appropriate. Further, Phoenix++ replaces Phoenix's C function pointer-based implementation with C++ templates and code inlining to reduce function call overhead.

TiledMR [16] further enhances the memory locality of the map-reduce runtime sys-

tem. TiledMR splits the input data set and runs small map-reduce jobs in succession while recycling the intermediate data structures efficiently. Many others use this principle, such as in the resilient distributed data sets (RDDs) of SPARK [107].

Lu *et al* [57] optimize map-reduce for the Xeon Phi. As in TiledMR, they pipeline map and reduce to reduce the memory footprint. They also try to vectorize the map task, which only worked for numerical applications such as Black-Scholes and Monte Carlo. Alternatively, they vectorize computation of hash table indices. The programmer must specify which form of vectorization, if any, should be applied, which further burdens the programmer with performance optimization. In our case, vectorization can be enabled by using the array notation of Cilkplus, which is a concise and auto-vectorizable notation for operations that are repeated over all array elements.

Mao *et al* [59] exploit huge page support in the kernel. Huge page sizes require fewer entries in the CPU's translation look-aside buffer (TLB), which reduce TLB misses for large data sets. Mao *et al* also advocate the use of NUMA-aware memory allocators, which Yoo *et al* [105] also investigated.

In an alternate approach to map-reduce, Jiang *et al* [42] focus on the reduction stage. Their work interleaves operations from the map and reduce phase instead of rigidly separating the map and reduce phases. They extended their approach to page the reduction data to disk for (too) large data sets [41].

Several authors analyze the characteristics of map-reduce workloads. Talbot *et al* [92] report the task multiplicity (how many keys may be emitted per task), the number of values per key, and the amount of computation in the map task as key characteristics. De Kruijf *et al* [19] follow a numeric approach and measure the amount of computation performed in the partition step, map, reduce or sort. They develop a micro-benchmark that may be dominated by one of these steps. However, their model is incomplete as the appearance of common keys between map tasks may significantly impact performance [92].

While the map-reduce model is conceptually simple, a subtly aspect is the commutativity of reductions. This aspect of the programming model is often undocumented, for instance in the Phoenix systems [76, 105, 92]. However, executing non-commutative reduction operations on a runtime system that assumes commutativity can lead to program bugs [18] even in extensively tested programs [101]. We use Cilk reducers [26] to perform reductions. Unlike many map-reduce models [92, 16, 1], Cilk reducers do not require commutativity. Thus, they are a safe programming construct that will not lead to subtle programming bugs.

3.2 Programming Map-Reduce Workloads

The map-reduce programming model typically assumes that key-value pairs represent data. For instance, the links between internet sites may be represented with a source URL as the key and a list of target URLs as the value. This representation exposes high degrees of parallelism through independent operations on different key-value pairs.

Computations on key-value pairs consist of a map function and a reduce function.

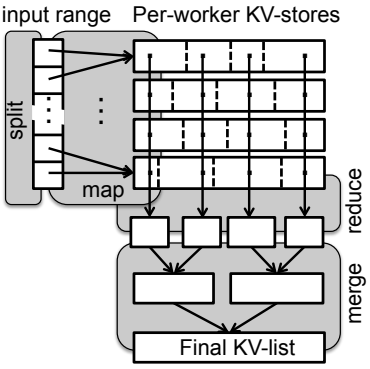


Figure 14: Schematic overview of Phoenix++ runtime system

The map function transforms a single input item (typically a key-value pair) into a list (which may be empty) of key-value pairs. The reduce function combines all values for the same key. Many computations fit this model [20, 54] or can be converted to fit it [55, 50].

3.2.1 The Phoenix++ Map-Reduce API and Runtime

The Phoenix++ shared-memory map-reduce system has several steps: splitting input data; map-and-combine; reduce; and sort-and-merge (Figure 14). The split step splits the input data into independent chunks upon which map tasks can operate. The input data may be a list of key-value pairs read from disk, but can also be other data such as a set of documents. The map-and-combine step breaks each 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. Performing an initial reduction improves performance by reducing the intermediate data set size [92].

Phoenix++ optimizes the storage of intermediate key-value pairs [92]. While a naive implementation would simply use lists, Phoenix++ allows programmers to select intermediate data structures, called *containers*, that are tuned to application properties. Supported containers include hash-maps indexed by key (e.g., for the word count application) and arrays (e.g., when the key is in a densely used integer range). The values in the containers are instances of the *combiner* data structure and hold the (aggregated) associated values, which could be a list or a sum of values.

Every thread produces one instance of the container during the map phase. To facilitate parallel reductions, all instances can be split by key ranges. Each thread reduces the key-value pairs that lie within a key range across all containers. The split is straightforward when the container is a fixed-size array but is more involved for dynamic data structures like a hash map. Finally, the resulting key-value lists are optionally sorted by key and merged into a single key-value list.

Phoenix++ extends the basic map-reduce API. The programmer must select (possibly write) container and combiner data structures that are tuned to the application. This API extension increases the performance of the runtime [92].

```
int64_t prod(int64_t mi, int64_t ri, int64_t mj, int64_t rj) {
    return (ri −mi) ∗ (rj −mj);
}
void cov(int const∗ matrix, int64_t const∗ means,
         int64_t∗ cov, int N, int length) {
    cilk_for (int i = 0; i < N; i++) {
        int64_t row_mean = means[i];
        int64_t const∗ v1 = matrix + i ∗ N;
        cilk_for (int j = i; j < N; j++) {
            int64_t col_mean = means[j];
            int64_t const∗ v2 = matrix + j ∗ N;
            int64_t sum=0;
            for(int64_t k=0; k< length; k++)
                sum+= (v1[k] − row_mean) ∗ (v2[k] − col_mean);
            cov[i∗N+j] = sum / (length−1);
        }
    }
}
```

Figure 15: The calculation of co-variance (PCA algorithm) expressed in Cilk.

3.2.2 Performance Limitations

The design of map-reduce systems like Phoenix++ creates many performance limitations. We identify two limitations, which our example application, Principal Components Analysis (PCA), illustrates. The PCA algorithm (Figure 15) calculates a co-variance matrix where position (i, j) lists a correlation metric between items i and j. The Cilk version (Figure 15) exhibits the expected characteristics of the code: two nested loops iterate over the pairs (i, j), calculate a correlation metric and store the value in the output matrix `cov`. In contrast, the map-reduce version (Figure 16) separates out the map and split parts of the computation. The co-variance matrix is represented as key-value pairs by associating the correlation metric (value) to the pair of indices (i, j) (the key).

The map-reduce version is long and tedious. It obfuscates both functionality and performance since the Phoenix++ code (Figure 16) focuses on the mechanics of the computation: how the data is split and processed in parts and how results are reduced.

Selection of Data Structures A list of key-value pairs supports little structuring of the data, which is often rich in structure. Appropriate data structures can improve performance significantly. Phoenix++ provides the option to select a pre-defined data structure, a *container*, to hold the output of the map tasks. An appropriate combiner (reduction operation) must be supplied to combine values with a common key. These issues require a deep understanding of the internals of the map-reduce runtime.

Tuning intermediate data structures breaks the map-reduce abstraction. While key-value pairs often do not support high-performance computation, the programmer must now think about appropriate data structures and how to map key-value pairs to them.

Framework Overheads While many problems match the computational and data organization patterns of map-reduce, others do not. Iterative algorithms with multiple rounds of map and reduce phases, in particular show inefficiencies due to the repeated input, output, shuffling and sorting of key-value pairs. This process has significant computational redundancies, especially if the data has a richer structure than that captured by its representation as key-value pairs.

```
#define MAX_DIM 2000
typedef struct {
    int row_num;
    int col_num;
} pca_cov_data_t;
typedef common_array_container<int64_t, int64_t, one_combiner, MAX_DIM*MAX_DIM>
    cov_container;

class CovMR : public MapReduceSort<CovMR,
     pca_cov_data_t,  // map input type
     int64_t,         // key type
     int64_t,         // value type
     cov_container    // intermediate key-value container
 > {
    int64_t const* matrix, * means;
    int N, length;
    mutable int row;
    mutable int col;

public:
    CovMR(int64_t const* _matrix, int64_t const* _means,
          int N, int length) : matrix(_matrix), means(_means),
      N(N), length(length), row(0), col(0) {}
    void map(data_type const& data, cov_container& out) const {
        int const* v1 = matrix + data.row_num*N;
        int const* v2 = matrix + data.col_num*N;
        int64_t m1 = means[data.row_num];
        int64_t m2 = means[data.col_num];
        int64_t sum = 0;
        for(int i = 0; i < length; i++)
            sum += (v1[i] - m1) * (v2[i] - m2);
        sum /= (length-1);
        emit_intermediate(out, data.row_num*N + data.col_num, sum);
    }
    int split (pca_cov_data_t& out) {
        if (row >= N) {
            return 0; // End of data reached
        } else {
            out.row_num = row;
            out.col_num = col;
            col++;
            if (col >= N)
                col = ++row; // only calculate triangle as Cov is symmetric
            return 1; // Valid chunk produced
        }
    }
};
```

Figure 16: Co-variance calculation (PCA) expressed as a map-reduce algorithm in Phoenix++.

K-means clustering is common iterative map-reduce algorithm that demonstrates this issue. This machine learning algorithm summarizes large data sets by assigning points in a multi-dimensional space to one of K clusters. The clusters are tuned to the data by iterating two steps until convergence: (i) assign each point to the closest cluster center; and (ii) update the cluster center based on the assigned points.

In the map-reduce model, each iteration of the two steps is a map-reduce algorithm with its own input and output key-value pairs. Every iteration thus requires data serialization and de-serialization, in particular the updated cluster centers. In contrast, an efficient implementation incurs no cost to communicate cluster centers from one iteration to the next as the data structures are reused across iterations.

Summary The map-reduce model does not support two key performance properties. Instead the programmer must use work-arounds or third-party solutions that are hard to compose efficiently. Phoenix++ addresses some of these issues by extending the map-reduce API such that programmers must deeply understand the internals of the runtime. In the following section, we define CilkMR, a map-reduce runtime that presents the reduction operation differently and, thus, is easier to use.

3.3 Map-Reduce using Cilk

We use appropriate programming interfaces to support scalable implementations of map-reduce workloads. Our map-reduce runtime builds on the Cilk language [28] because of its support for generalized reductions [26] and Intel's Cilkplus array notation that facilitates auto-vectorization [39].

3.3.1 Map-Reduce Code Templates

Cilk is a task-oriented parallel programming model that supports expression of (map) task parallelism (`cilk_for` and `cilk_spawn`) and reduction operations (`cilk::reducer`).

Balanced Template Figure 17 shows one variant of the CilkMR map-reduce API, which uses the `cilk_for` keyword to express that iterations of the loop may execute in parallel (line 7). The map task `mapfn` can be applied in parallel to all items in the data set described by the `ibegin` and `iend` iterators. The template is "balanced" as the directed acyclic graph that describes the parallel activities assigns a comparable number of loop iterations to each processor. Work stealing is minimal during execution of this template if each loop iteration has a comparable amount of work.

Unbalanced Template Figure 18 shows an unbalanced template for map-reduce. The user defines a functor `splitfn` that splits the input into work items. The `cilk_spawn` statement indicates that the `mapfn` functor may be applied in parallel to the work items. The `map_reduce` method blocks at the `cilk_sync` statement until all spawned tasks

```
template<class Monoid, class InputIterator, class MapFunctor>
void __attribute__ (( flatten ))
map_reduce( InputIterator ibegin, InputIterator iend,
            MapFunctor mapfn,
            typename Monoid::value_type & output ) {
    cilk::reducer<Monoid> imp_;
    cilk_for ( InputIterator I=ibegin, E=iend; I != E; ++I )
        mapfn( *I, imp_.view() );
    std::swap( output, imp_.view() );
}
```

Figure 17: CilkMR map-reduce API call with balanced spawn tree.

```
template<class Monoid, class SplitFunctor, class MapFunctor>
void __attribute__ (( flatten ))
map_reduce( SplitFunctor splitfn, MapFunctor mapfn,
            typename Monoid::value_type & output ) {
    cilk::reducer<Monoid> imp_;
    typename SplitFunctor::value_type value;
    while( splitfn ( value ) ) {
        cilk_spawn [&]( typename SplitFunctor::value_type v ) {
            mapfn( v, imp_.view() );
        }( value );
    }
    cilk_sync;
    std::swap( output, imp_.view() );
}
```

Figure 18: CilkMR map-reduce API call with unbalanced spawn tree.

complete. This code template is “unbalanced” as the underlying directed acyclic graph that describes the parallel activities is highly skewed. Work stealing will be frequent during its execution.

The structure of the code dictates the choice between the balanced and unbalanced templates. The balanced template can be used when the map-reduce template is over a known range. In other cases, such as text parsing problems, the programmer may need to define a split function to divide the input into independent chunks.

Notes While the Phoenix++ runtime strictly separates map, reduce, sort and merge phases, the CilkMR `map_reduce` templates overlap these activities in time. Thus, load imbalance can potentially impact the Phoenix++ runtime much more.

CilkMR returns a container, e.g., a hash map, instead of a list of key-value pairs. An additional step must serialize the hash map when a list of key-value pairs is desired. This separation clearly portrays the cost of this conversion, which may is often unnecessary, to the programmer. Only one of our benchmarks strictly requires it.

```
template<class map_type>
struct map_monoid : cilk::monoid_base<map_type> {
    static void reduce(map_type * left, map_type * right) {
        for(typename map_type::const_iterator
              I=right->cbegin(), E=right->cend(); I != E; ++I)
            (* left )[I->first] += I->second;
        right->clear();
    }
    static void identity (map_type * p) const {
        new (p) map_type();
    }
};
```

Figure 19: Example of a hash-map monoid for counting occurrences of words.

3.3.2 Generalized Reductions

The application-specific `Monoid` class defines the reduction through three components [26]: a data type; an associative operation; and an identity value. Cilk reductions do not need to be commutative. Thus, Cilk reducers can support reductions like concatenation of lists.

Figure 19 shows the definition of a monoid for a hash-map data type. The template parameter `map_type` defines the underlying non-concurrent hash-map type. Hash-maps are assumed to be reduced by taking the join of all keys and that the values for common keys are further reduced using an `operator +=` (Line 6). The identity value is an empty hash-map as indicated in the initialization function (Line 9).

The runtime system dynamically creates copies of the reduction variable, and reduces those copies as needed. The creation and reduction of these copies, or *views*, aligns with work-stealing activities in the scheduler. Views are created only after a work stealing event. They are reduced when the stolen task completes. Work stealing activities are rare in highly parallel programs because the design of the Cilk scheduler executes most spawn statements as if they are sequential function calls. In these cases, the same view is used across tasks. Views are reduced under conditions of mutual exclusion. Thus, synchronization rarely impacts application-specific reducer code.

3.3.3 Performance Characterization

The *balanced* and the *unbalanced* templates have different parallel scalability. Cilk scheduling overhead is bound by the span of the spawn tree [65]. The *balanced* template uses the `cilk_for` loop, which recursively divides the iteration range in half until a fine granularity is reached. Spawning each half of the range results in a balanced spawn tree. No more than $O(\log n)$ work steals are required for n data items. In contrast, the span of the spawn tree of the *unbalanced* template is $O(n)$.

Reduction operations are proportional to steals [26]. Views are reduced off the

```
struct Monoid : cilk::monoid_base<uint64_t[768]> {
    typedef uint64_t value_type[768];
    static void reduce( value_type *left, value_type *right ) {
        for( size_t i=0; i< 768; i++)
            (*left)[i] += (*right)[i];
    }
};
struct histogram_map {
    void operator() ( const char * pix, uint64_t & histogram[768] ) {
        histogram[(size_t)pix[0]]++;
        histogram[256+(size_t)pix[1]]++;
        histogram[512+(size_t)pix[2]]++;
    }
};
uint64_t result[768];
cilkmr::map_reduce<Monoid>( byte_array, byte_array_length/3,
                            histogram_map(),
                            result );
```

Figure 20: The fixed-length histogram algorithm expressed in CilkMR.

critical path and are amortized with steals [26]. Thus, they have no overhead if they take constant time [52].

3.3.4 Example: Histogram

Figure 20 shows an algorithm that constructs a fixed-size histogram to demonstrate the use of CilkMR. The code has three parts: the definition of the Monoid (Line 1); the definition of the map task (Line 8); and the call of the map-reduce routine (Line 15). The monoid reduces two histograms, adding up all elements pair-wise. The map function adds 3 successive byte values to appropriate elements of the histogram. The call statement uses a variation of the map-reduce template with a begin iterator and a count. This template is a convenience short-hand to define the range using a begin and end iterator (Figure 17).

3.3.5 Addressing the Performance Limitations

We discuss how CilkMR addresses performance issues of prior map-reduce systems.

Selection of Data Structures While existing map-reduce systems expose an API to reduce two key-value pairs, the CilkMR runtime exposes reductions on data containers. Thus, the programmer controls the type of container that the program uses. In contrast, Phoenix++ pre-defines several containers and associated reduction operators. Adding a new container in Phoenix++ requires an extension to the runtime. The generic CilkMR approach exposes the selection of data structures in its API.

Table 1: Phoenix++ codes: map task multiplicity, combiner data type, sorting, merge and reduction operation.

	map	**combiner**	**sort**	**reduction**
histogram	*:768	array	Y	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	Y	array add
strmatch	*:0	n/a	N	n/a

Framework Overheads Prior map-reduce systems serialize the data and return a list of key-value pairs. Successive map-reduce operators must convert data back and forth between the serialized representation and the efficient representation. The CilkMR map-reduce templates return the data set stored in the selected container type. Successive operators are performed without unnecessary data transformations.

3.4 Benchmarks

We implement all 7 Phoenix++ benchmarks in CilkMR. Table 1 describes their Phoenix++ properties. 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 a map task can emit. Previous reports [92] on these properties are inconsistent with the distributed code. For **matmul** and **strmatch**, the map tasks emit no key-value pairs so the multiplicity is **:0*. Instead, they use shared memory operations to produce output results, which is inconsistent with the spirit of the map-reduce model. The second column shows the intermediate key-value data structure. In most cases, a generic key-value list is optimized to an array indexed by an integer key. For word count, intermediate key-value pairs are stored in a hash map indexed by a character string key. The CilkMR implementations are similar except:

- For **wc**, we use the same hash table as Phoenix++ but we reduce hash table instances following Cilk's schedule of reduction operations, which is markedly different from the separation of map and reduce phases under Phoenix++;
- For **histogram**, we avoid sorting a list of key-value pairs by storing it as an array while Phoenix++ generates a list of key (index)-value pairs that it then sorts;
- For **matmul**, we use a tried-and-tested matrix multiply implementation with good parallel scalability and locality;
- For **pca**, we partition the co-variance matrix among threads and use a scalar Cilk reducer to aggregate the total co-variance;

Table 2: CilkMR codes: balanced parallelism (bal), using map-reduce API (mr), nested parallelism (nest), vectorization (vec) and sorting (sort).

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

- For **kmeans**, we use a Cilk reducer object to merge partial results in the computation of cluster averages, which is more efficient than a key-value pair representation;
- For **strmatch**, we make no significant changes to the Phoenix++ distribution, which is simply a parallel for-loop over the input data.

The CilkMR codes (Table 2) use similar reduction data structures as the Phoenix++ versions. In some cases, we further specialize to the benchmarks, e.g., a struct of scalars vs. an array for **lreg**. Some benchmarks use the balanced (B) vs. the unbalanced (U) template, nested parallelism, vectorization or sorting. The label (Y) indicates that vectorization is possible, but did not improve performance as expected.

Cilk codes that do not require a reduction are not implemented using the map-reduce API but are implemented using parallel for loops. In the case of **matmul** we used the MIT Cilk-5 implementation, which is known to perform well. The Phoenix++ **strmatch** implementation does not perform a reduction as it produces no output. Thus, we parallelized the loop without using the map-reduce API. These choices are possible as our runtime composes with other parallel code written in the same language. Thus, the programmer may choose not to use the map-reduce API when appropriate without increasing programming complexity.

3.5 Evaluation

We evaluate the programming systems on a quad-socket 2.6GHz Intel Xeon E7-4860 v2, totaling 48 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 three input dataset sizes for the Phoenix++ benchmarks (Table 3). Three input sizes for the Phoenix++ benchmarks correspond to those used by Talbot *et al* [92]. We create another input through further scaling. For **kmeans** we search for 100 clusters in a 100-dimensional space. The 'wc' dataset contains few large files with sizes varying from 10MB to 800MB. Reported results are averaged over 15 executions.

Table 3: Input dataset sizes

	medium	**large**	**huge**
histogram	400MB	1.4GB	11.2GB
lreg	100MB	500MB	4GB
wc	50MB	100MB	800MB
kmeans	50,000	75,000	100,000
matmul	512x512	768x768	1024x1024
pca: items	1000	1500	1500
vector length	1000	1500	4500
strmatch	100MB	500MB	4GB

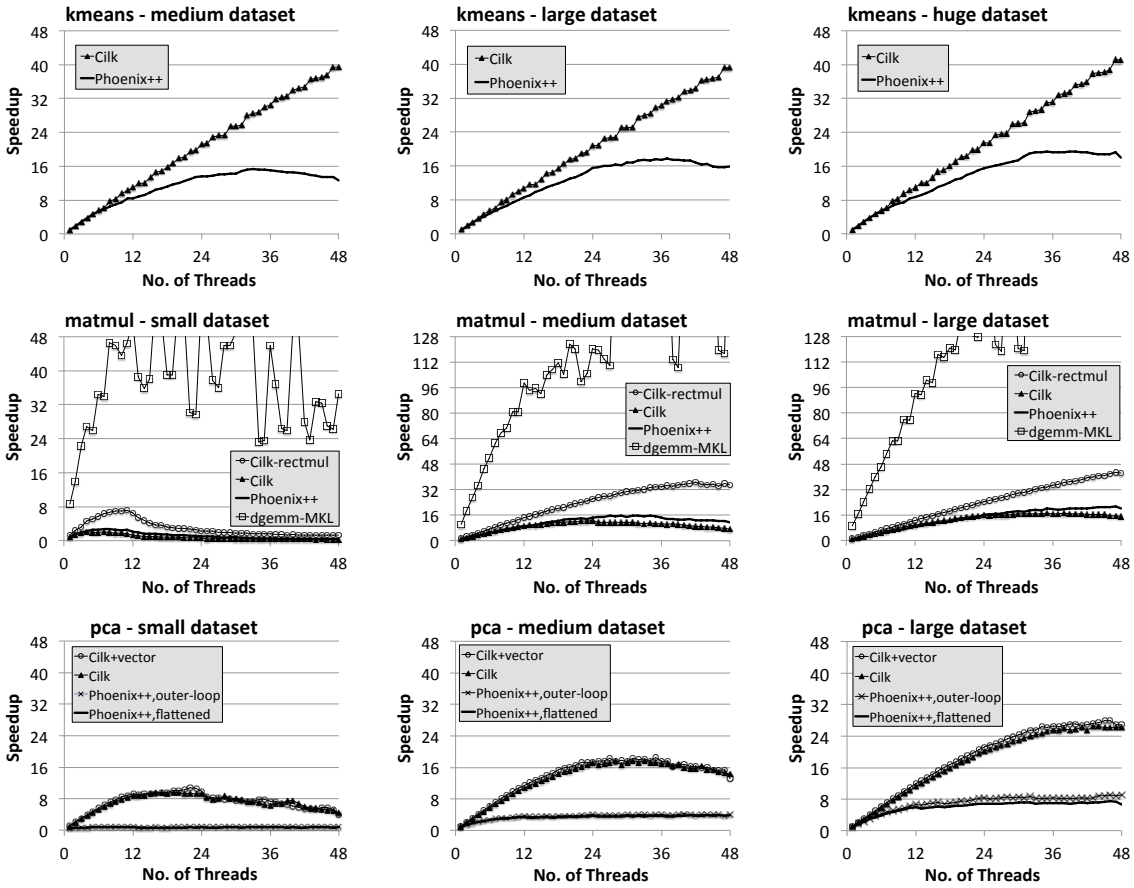


Figure 21: Results (a): Applications dominated by map time (compute-bound).

We experimented with various multi-threaded memory allocators including Hoard [7], SSMalloc [56] and TCMalloc [70]. The resulting performance is similar to that with the default system allocator. They achieve slightly higher performance for some benchmarks but sometimes introduce performance anomalies. For example, **lreg** did not

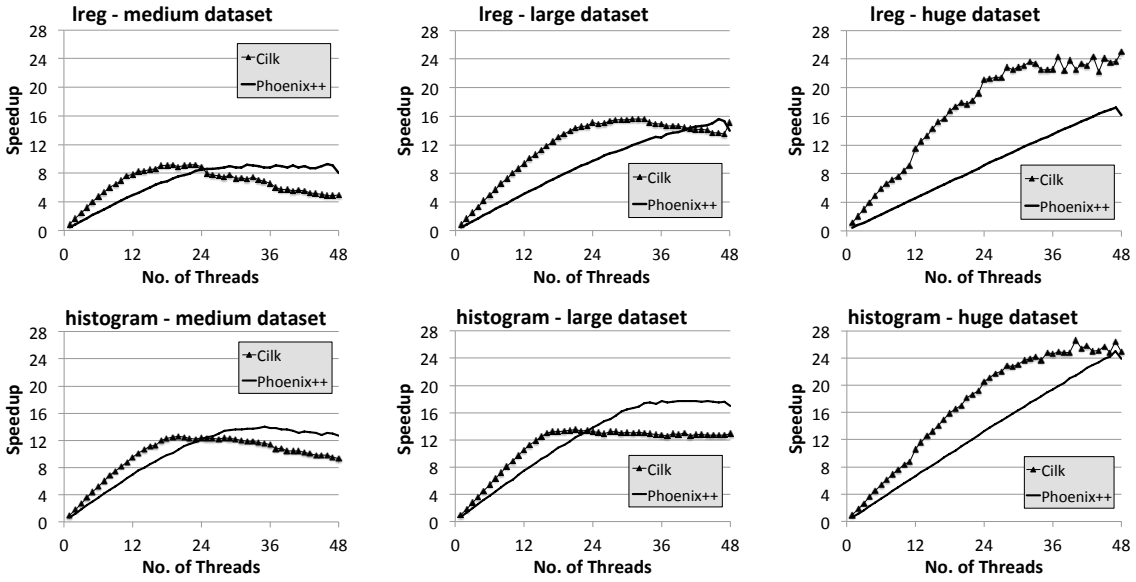


Figure 22: Results (b): Memory-bound applications.

scale well with TCMalloc. The memory allocator does not affect our conclusions as it cannot make up for algorithmic inefficiencies. Thus, we report results for the default allocator.

3.5.1 Performance Evaluation: Speedup

Figures 21–23 present the speedup using CilkMR, Cilk and Phoenix++ over the sequential version of the benchmarks. Figure 21 shows the benchmarks dominated by computation in the map phase: **matmul**, **pca** and **kmeans**. Phoenix++ repeatedly serializes and de-serializes the centers to key-value lists for **kmeans**, which reduces scalability. **matmul** and **pca** do not strictly require a reduction operation as each map task deposits its results in distinct locations of an array. Phoenix++ uses this observation for **matmul** (Table 1). As previously discussed, we use *plain* Cilk for **matmul** and **pca**.

For **matmul** we use two matrix multiply implementations distributed with MIT Cilk [28]. The **matmul** version uses recursive decomposition 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. Also, its leaf task, a 16x16 block multiply, is highly optimized. Figure 21 normalizes performance to the sequential version of **matmul**. We also present the performance of `dgemm` from the Intel MKL library. These results show that specifically optimized codes clearly outperform a generic map-reduce framework like Phoenix++, which shows that blindly applying the map-reduce concept to every problem is not sensible. Further, the map-reduce runtime is used inappropriately for **matmul** as the map

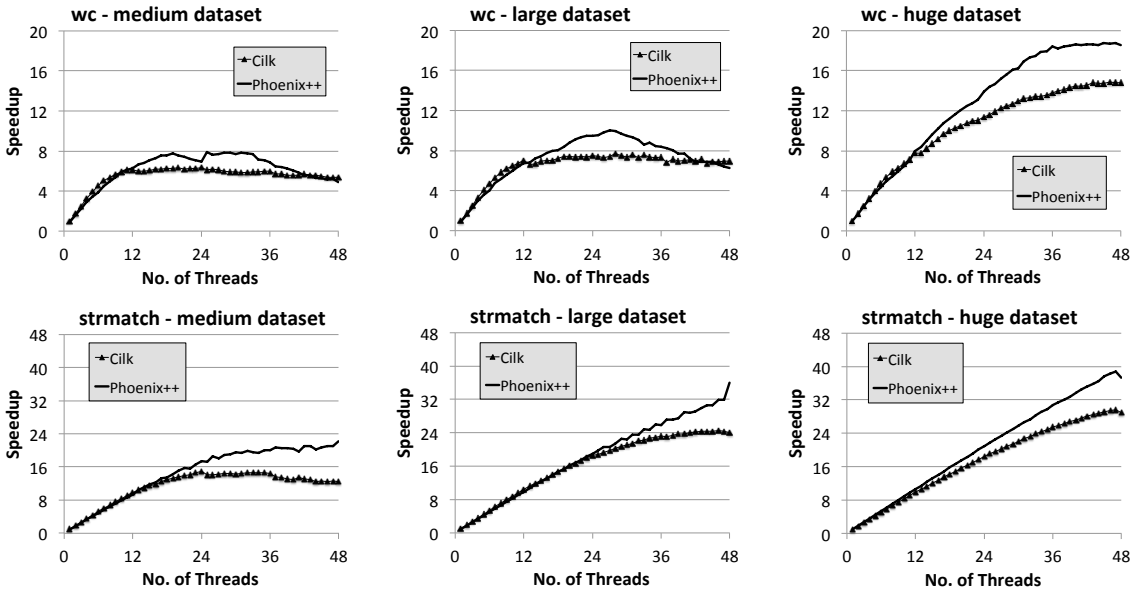


Figure 23: Results (c): Applications with unbalanced spawn trees.

task accesses shared memory and does not emit key-value pairs.

The memory-bound benchmarks **histogram** and **lreg** show good scalability with both map-reduce systems (Figure 22). Both benchmarks perform few operations per input byte – 4 integer operations for **histogram** and 7 for **lreg**. The CilkMR version accelerates faster with increasing thread counts but eventually saturates. Saturation occurs at lower thread counts with smaller inputs, which suggests that the Cilk scheduler carries a higher burden than the Phoenix++ scheduler. Interestingly, the specialized map-reduce system performs better on smaller inputs.

The benchmarks **wc** and **strmatch** use the unbalanced CilkMR template (Figure 23). The performance of these benchmarks is nearly identical to that of the Phoenix++ version up to around 8–24 threads, depending on the problem size, after which the inefficiency of the unbalanced template limits scalability.

3.5.2 Addressing Performance Limitations

Internal Data Structures Our initial CilkMR implementation of **wc** performed poorly because we used the default C++ STL `unordered_map`. This hash map data structure performs badly for map-reduce applications because it balances performance against space. In map-reduce applications, however, insert operations dominate execution time, so the performance trade-off does not arise.

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. Despite improvements, the Phoenix++ hash table still outperforms it since the STL code dynamically allocates memory for each element of the hash table.

Table 4: Peak heap memory usage (MB) in excess of data set size when using the large data set.

		Memory usage (MB) for thread count.			
		1	**16**	**32**	**48**
histogram	CilkMR	0.06	0.95	1.67	2.50
	Phoenix++	0.04	0.43	0.86	1.23
lreg	CilkMR	0.06	0.69	1.39	2.08
	Phoenix++	$<$0.01	0.06	0.11	0.17
wc	CilkMR	11.70	28.10	34.30	34.00
	Phoenix++	15.10	60.60	98.30	117.00
pca	Cilk	25.82	26.44	27.13	27.82
	Phoenix++	159.90	161.50	160.00	160.10
kmeans	CilkMR	39.81	41.66	42.98	44.55
	Phoenix++	68.62	502.00	963.2	1423.4
strmatch	CilkMR	0.06	0.69	1.38	2.07
	Phoenix++	0.56	0.58	0.61	0.73
matmul	Cilk	4.06	4.69	5.39	5.35
	Phoenix++	4.06	4.16	4.27	4.39

We conclude that map-reduce applications are sensitive to the performance of the data structures due to the generally low amount of computation per data structure access. Thus, appropriate data structure selection is essential. This observation holds across programming models.

Memory Consumption Low memory consumption is important as map-reduce workloads tend to be applied to large data sets and main memory is limited. Table 4 shows the peak heap memory when using the large input. We use the valgrind tool "massif" [66] to collect this data. We subtract the memory required to store the input data set for clarity. For many benchmarks, the runtime requires little additional space over the data set. Nonetheless, the internal data structure size grows moderately as the thread count increases by about 18 KB per thread for CilkMR and about 1 KB per thread for Phoenix++. This difference arises because CilkMR requires a varying number of stacks depending on how work stealing progresses [51].

Three benchmarks consume significant additional memory: **wc**, **pca** and **kmeans**. They store large volumes of data in the intermediate data structures that support the reduction. This space increases rapidly with thread count for Phoenix++, while the memory utilization remains fairly constant for CilkMR. Phoenix++ collects all key-value pairs during the map phase prior to initiating the reduction phase, resulting in large intermediate data sets. In contrast, CilkMR repeatedly merges small data sets throughout the computation, which keeps the footprint small. The excessive memory consumption of **kmeans** arises from a deliberate memory leak in Phoenix++ that was created for performance reasons.

3.6 Conclusion

We have presented a scalable and composable map-reduce runtime system. Our runtime system, called CilkMR, builds on the Cilk parallel programming language in order to reap opportunities for performance optimization that are out of scope of state-of-the-art specialized map-reduce systems. Ease of programming with CilkMR is similar to other map-reduce systems. CilkMR supports composition of multiple, potentially nested, map-reduce kernels. In contrast, state-of-the-art map-reduce frameworks require redesign of the parallel structure from first principles when composing codes.

We evaluated several map-reduce benchmarks implemented in the Cilk parallel programming language and in Phoenix++, a state-of-the-art shared-memory map-reduce runtime. Our evaluation shows that on a 48-core workstation, the Cilk codes perform 1.5x–4x better, although performance is reduced by up to 30% for two applications where the Cilk versions use an less efficient parallel code structure.

Our performance evaluation demonstrates that CilkMR offers a better parallel implementation of the map-reduce pattern than Phoenix++. It differs from the Phoenix++ approach by not representing data as key-value pairs when appropriate. Viewing our map-reduce templates as a library extension to a generic parallel programming language, they also avoid inappropriate map-reduce-inspired algorithm design.

Some algorithms have been extensively studied and high-performance implementations have been constructed for them. From a viewpoint of programmability, it is advisable to re-use these implementations, e.g., through standardized libraries. As shown in our evaluation of matrix multiply, K-means and PCA, forcing these applications to match the map-reduce API hinders performance. CilkMR supports composing map-reduce code with existing code.

This work first applies to large-scale shared memory servers, which can scale to terabyte-sized main memory. An important avenue for future work is to investigate how the representation of map-reduce programs affects the design of distributed map-reduce systems, in particular whether these benefit equally from representing the reduction operation over containers as opposed to individual key-value pairs.

4 Case Study: NUMA-Aware Graph Analytics

In this Section we investigate the utilisation of Swan's extension for NUMA-aware scheduling. We do this through a specific use-case, graph analytics, which has been shown to be sensitive to NUMA-aware scheduling [108]. Many important problems in social network analysis, artificial intelligence, business analytics and computational sciences can be solved using graph-structured analysis. There is increasing evidence that large-scale shared-memory machines with terabyte-scale main memory are well-suited to solve these graph analytics problems as they are characterized by frequent and fine-grain synchronization [3, 86, 108, 67, 49, 80].

Recently, graph partitioning has been proposed to isolate memory accesses to specific parts of the graph data. Graph partitioning allows to stage graph data in main

memory from backing disk [49] and allows to direct memory accesses to the locally-attached memory node in Non-Uniform Memory Access (NUMA) machines [108]. Moreover, graph partitioning is essential in distributed memory systems to spread the computation evenly across all nodes [30].

Several studies have proposed efficient heuristic partitioning techniques for social network graphs [30, 49], as near-optimal partitioning is excessively time-consuming. A common approach is to partition the edge set with the aim to place an equal number of edges in each partition. This results in balanced computation per partition as many graph analyses perform work proportional to the number of edges [30].

While graph partitioning is a crucial building block for graph analytics, little is known about the various ways in which it affects performance. We analyze heuristic graph partitioning in detail and identifies side effects that limit achievable performance. In particular, we show that graph partitioning incurs an innate performance overhead, which stems from increased control flow and from the decreased connection density of the partitions.

Moreover, we find that partitioning the edge set results in an imbalance in the number of vertices appearing in each partition. Alternatively, partitioning the vertex set results in an imbalance in the number of edges. The net result is that significant load imbalance exists between partitions, either for loops iterating over vertices, or for loops iterating over edges.

We make the following contributions:

- We analyze the characteristics of graph partitions and identify how these limit performance.
- We present GraphGrind, a NUMA-aware graph analytics framework that reduces the performance impact of graph partitioning. Key highlights of GraphGrind are an improved graph representation, tuning the partitioning to the characteristics of the algorithm and improving the NUMA memory mapping of key data structures.
- We apply Swan's extension for expression of NUMA affinity for parallel loops to graph analytics. Our extension simplifies the design of GraphGrind and is generally applicable to enfore NUMA-aware scheduling in parallel programs.
- We experimentally evaluate the performance of GraphGrind on 6 real-world graphs and 3 synthetic graphs. We show that GraphGrind improves performance by up to 82% over Polymer and up to 326% over Ligra.

4.1 Motivation

Graph analytics provide abstract, *vertex oriented* and/or *edge oriented* programming models that iteratively calculate a value associated to a vertex. The two key data structures are graphs and frontiers. A graph $G = (V, E)$ has a set of vertices V and a set of directed edges $E \subset V \times V$ represented as pairs of end-points. A frontier is a

ALGORITHM 1: Partitioning by destination

input : Graph $G = (V, E)$; number of partitions P
output : Graph partitions $G_i = (V, E_i)$ for $i = 0, \ldots, P-1$

```
1 avg = |E|/P;                                   // target edges per partition
2 i = 0;
3 for v : V do
4 |   if |E_i| >= avg and i < P - 1 then
5 |   |   ++i;                                   // i has exceeded target edges
6 |   E_i = E_i ∪ in-edges(v);                    // i is home partition of v
```

subset of the vertices which are active. Graph algorithms visit the destination vertices of the active edges ($\{v \in V : (u, v) \in E \wedge u \in F\}$) and apply an algorithm-specific function to update the value computed for v taking into account the current value for u. This operation is repeated until all values have converged.

A low-overhead partitioning algorithm is listed in Algorithm 1 [49, 108]. It partitions the edge set and produces partitions with similar properties as the algorithm of [30]. The graph is partitioned as $G_i = (V, E_i)$ where E_i is a partitioning of E: $\cup_i E_i = E$ and all E_i are non-overlapping. The algorithm assigns each vertex to a home partition such that (i) each partition is home to a range of subsequent vertex IDs and (ii) an edge $(u, v) \in E$ is assigned to the home partition of v. It follows that $E_i \subset V \times V_i$: each partition only has edges pointing to its own home vertices, but the sources may be any vertex.

An often-used criterion for balancing CPU load is to place the same number of edges in each partition, as many graph analytic algorithms perform an amount of work that is proportional to the number of edges.

Figure 24 graphically illustrates a graph traversal over a graph with highly skewed degree distribution. The graph is shown at the top, together with its representation in the Compressed Sparse Rows (CSR) format [81]. The CSR format stores two arrays: an edge array with IDs of the destionation vertices and an index array storing for each vertex the index into the edge array where the destinations of its edges are recorded. The graph is partitioned in two parts by Algorithm 1. Partition 0 contains 7 edges and is home to vertices 0, 1, 2 and 3. Partition 1 also contains 7 edges and is home to vertices 4 and 5. Each graph traversal now needs to visit each vertex twice to find the out-going edges of the vertex in each partition.

We refer to this partitioning technique as *partitioning by destination* as edges are assigned to the home partition of the destination vertex. Alternatively, *partitioning by source* assigns an edge (u, v) to the home partition of u. Both algorithms achieve nearly the same number of edges in each partition [108].

4.1.1 Extra Work Induced by Partitioning

When partitioning the edge set, the list of edges of a vertex is split with parts of the list appearing in different partitions. As such, the edges for some vertices are stored in

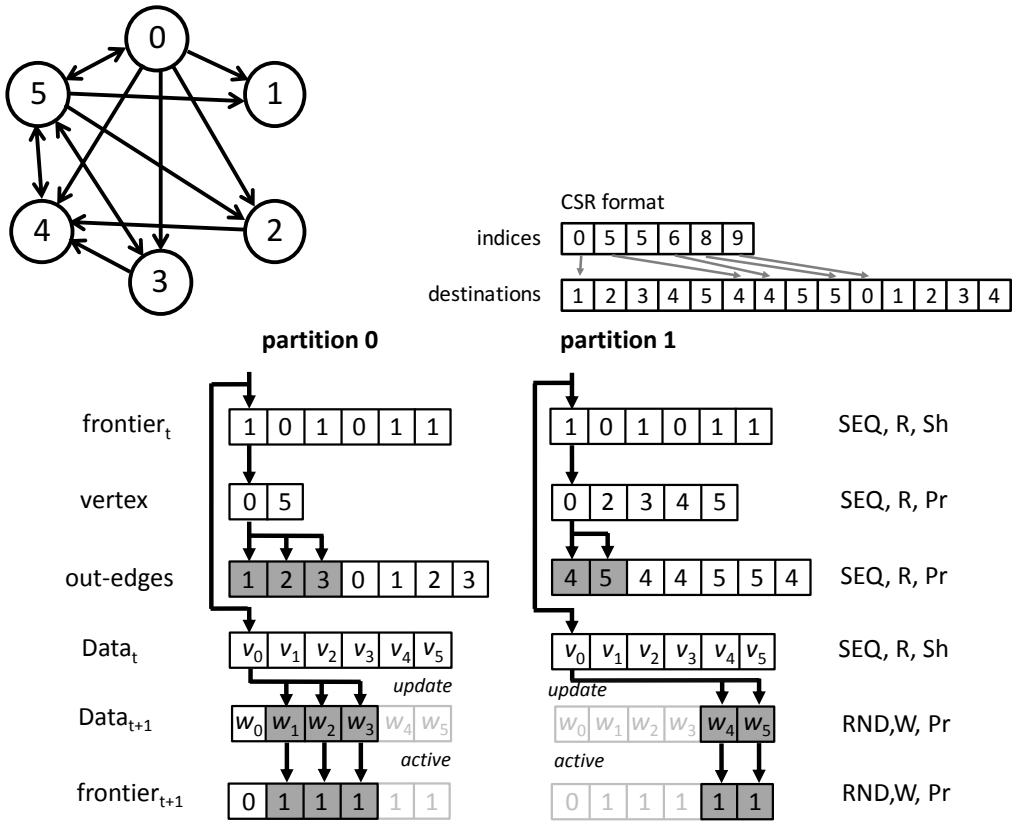


Figure 24: Traversal of a graph partitioned by destination.

distinct partitions. Graph traversal must thus visit the vertex once for each replication. The additional cost of this is a small amount of control flow, lookups in the graph representation and checking whether the vertex is active. While these actions require only a few dozen assembly instructions, it is important to keep in mind that graph analytics perform little computation, typically less than a dozen assembly instructions per edge. Moreover, the overhead involves several main memory accesses as these algorithms are memory intensive.

Figure 25 shows the average replication factor of vertices for various degrees of partitioning. The graphs are described in Section 4.3.6. We show data for 6 of the 9 graphs as the remaining 3 behave similarly. Graphs with few edges per vertex (USAARoad and Friendster) have the lowest replication factors while highly skewed graphs (Twitter and Orkut) have the highest. Assuming 4 partitions, replication factors are often in the range 2–3, which implies that the control flow overhead of graph traversal is repeated 2 to 3 times. This results in an instruction count increase of up to 18%.

Figure 25 moreover shows that the graph partitioning algorithm studied in this paper achieves a comparable replication factor as the more elaborate algorithm in [30]. We may thus assume that the conclusions of this paper are independent of the partitioning algorithm used.

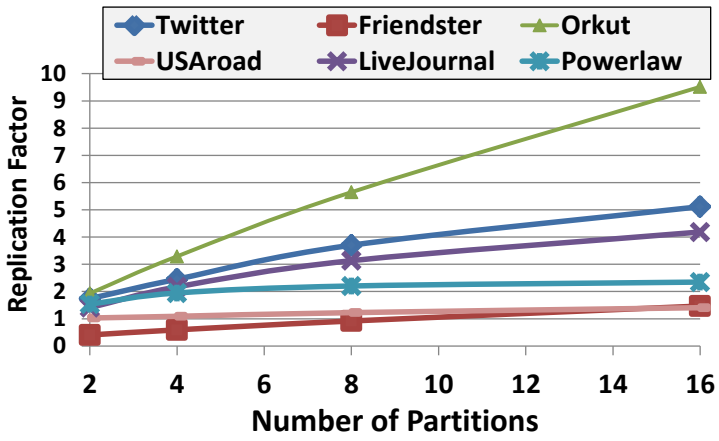


Figure 25: Compressed vertices replication factor varying partition number

4.1.2 Sparsity of Graph Partitions

If vertices are not replicated across all partitions, then by necessity vertices will not have incoming or out-going edges in several of the partitions. Figure 26 (left) shows the average number of vertices with zero degree for varying degrees of partitioning by destination. Similar results hold for partitioning by source. The fraction of vertices with zero out-going edges shoots up quickly as more partitions are introduced, exceeding in many cases 50% for 4 partitions. Moreover, real-world social networks have strongly imbalanced partitions (Figure 26 (right)). In contrast, the partitions of synthetic graphs, intended to model real-world graphs, have equal numbers of unconnected vertices in each partition. Interestingly, the Friendster graph has fairly equal partitions.

The sparsity of graph partitions leads to an opportunity: if we can avoid iterating over the absent vertices in a partition, then the instruction count increase for these vertices can be restricted only to the partitions where the vertex occurs. To this end, GraphGrind uses a variation of the CSR representation where zero-degree vertices are not recorded.

4.1.3 Balancing Edges vs. Vertices

It is hard to partition a social network graph in a balanced way due to its skewed degree distribution. Figure 27 shows the relative number of vertices per partition for various graphs and numbers of partitions. Social network graphs like Twitter and Friendster have highly different numbers of vertices per partition when balancing the number of edges.

The imbalance of the number of vertices per partition has an important impact on performance. First, many graph algorithms make passes over vertices apart from passes over the edges. As such, the work performed per graph partition is not only proportional to the number of edges, but also depends on the number of vertices.

Secondly, not all algorithms perform a fixed amount of work per edge. Instead, al-

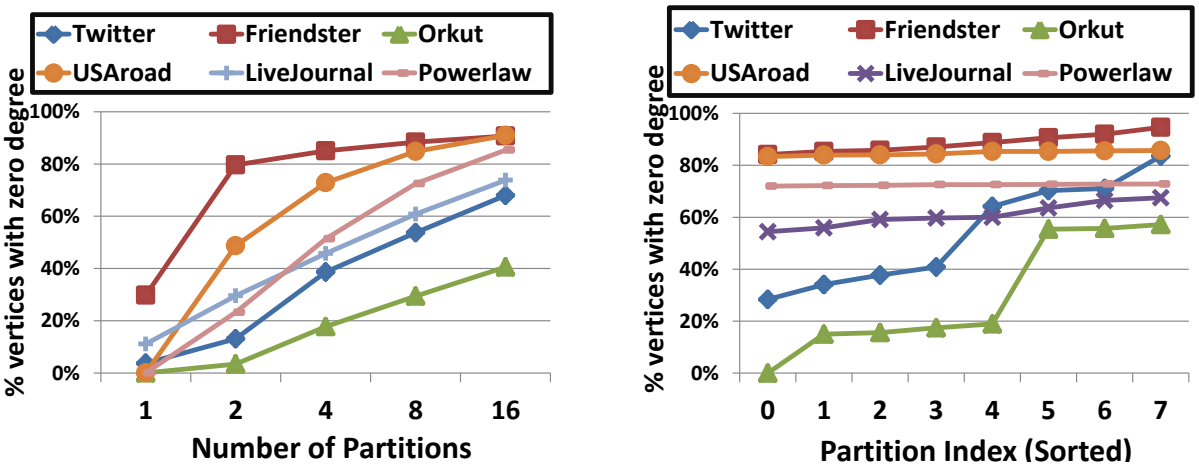


Figure 26: Percentage of vertices with zero out-degree averaged across all partitions (left) and variation across each of 8 partitions (right).

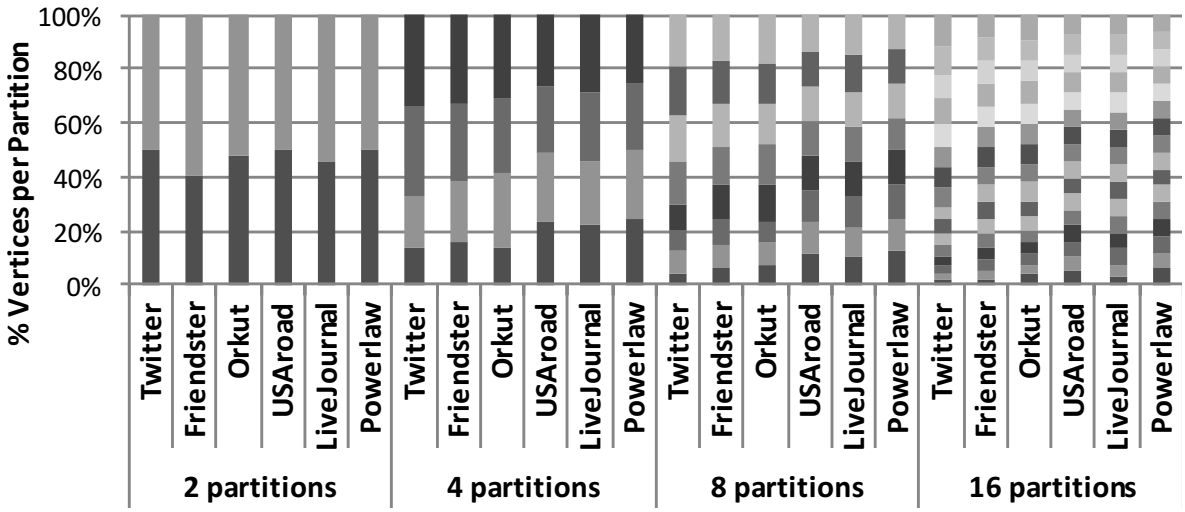


Figure 27: Relative sizes of partitions for varying degree of partitioning.

gorithms such as BFS, betweenness-centrality, Bellman-Ford and K-Core visit at most one active edge per active vertex. For them, balancing the edges between partitions does not result in a balanced CPU load.

Thirdly, an imbalance in the number of vertices per partition results in a skewed utilization of memory and creates hotspots for certain partitions. This unnecessarily drives to scale-out distributed systems to higher degrees of parallelism to drive the worst-case partition size down, even if the computation does not warrant scaling out. In shared memory systems the memory imbalance may be combated by storing data in a sub-optimal NUMA node, which results in the lesser evil of remote NUMA accesses.

Increasing the number of partitions may seem to avoid skewed partitions. This

is however not true. As Figure 27 shows, the presence of highly-connected vertices remains an issue with higher degrees of partitioning as some partitions have twice as many vertices as others. We conclude that the graph partitioning needs to balance CPU load and should be adapted to characteristics of the algorithm.

4.2 GraphGrind: Design and Implementation

We designed GraphGrind, a graph analytics framework for NUMA shared memory machines that builds on the characteristics of graph partitions to optimise the memory layout of graphs and to reduce load imbalance.

GraphGrind contains all the required features of graph analytics systems, including hierarchical parallel decomposition of the computation, NUMA-aware data placement and code scheduling [108], balanced vertex-cut partitioning [30] and adapting data structures [37] and search direction [6] to the size of the frontier. We discuss its key features below.

4.2.1 Application Programming Interface

GraphGrind is compatible with the Ligra programming model. It provides two data types: graphs and frontiers. A frontier is a subset of the vertices in a graph. The key functions apply operations to edges or vertices and calculate new frontiers in the process. They are defined as follows:

- *size()*: For a frontier F, $size(F)$ returns $|F|$.
- The *edge-map()* operator is the main work-horse. It applies an algorithm-specific function to every active vertex in the graph. Its arguments are a graph $G = (V, E)$, a frontier F, a function Fn and a condition C. An edge $(u, v) \in E$ is active if $u \in F$ and $C(v) =$ *true*. The argument Fwd determines whether a forward or a backward traversal is likely to be faster. *Edge-map* returns a new frontier consisting of all visited vertices v for which $F(u, v)$ returned a true value.
- *vertex-map()* applies a function Fn to every vertex in the frontier F. It returns a new frontier consisting of all visited vertices u for which $F(u)$ returned a true value.

We extend the programming interface with a *cache* for ***backward*** *edge-map* traversals. While *edge-map* may execute in parallel, it traverses the incoming edges of a vertex sequentially when the number of vertices is not very large (less than 1000). Compilers should, in principle, be able to hold the intermediate updates for the destination vertex's value in registers. However, the complexity of control flow and pointer aliasing prohibits this in practice. GraphGrind allows the programmer to specify how to cache intermediate updates for the function Fn. This explicit notation allows compilers to allocate them to registers.

4.2.2 Edge Traversal

The efficient implementation of graph algorithms is sophisticated and requires deep knowledge of the characteristics of the algorithms. First, the frontier is a set of vertices and may be implemented either as a bitmap or as an array storing vertex IDs. The most efficient implementation depends on the *density* of the frontier [37]. In a *dense frontier* more *edges* are active, while a *sparse frontier* has few active *edges*. The threshold is typically set at 5% active edges.

Secondly, edges may be traversed in *forward* or *backward* manner. In each case, the goal is to traverse the destination vertices of active edges. A *forward* traversal first traverses source vertices $u \in V$ and checks if they are active ($u \in F$). If they are, then their out-going edges are traversed. A *backward* traversal iterates over destination vertices $v \in V$ as well as their incoming edges $(u, v) \in E$. Only then can it check that the source vertex u is active.

Some algorithms execute faster with forward traversal, while others with backward traversal. The distinction is to a large extent motivated experimentally [86]. Beamer et al.motivate the distinction by the number of visited edges [6].

The graph representation is designed for efficient forward *and* backward iteration. Hereto, a dual representation is used for directed graphs (incoming and out-going edges are equal for undirected graphs). These use the compressed sparse rows (CSR) and compressed sparse columns (CSC) formats for sparse matrices [81]. These formats use two arrays to encode a graph. In CSR, an *edge array* stores the vertex IDs of edge destinations while an *index array* stores for every vertex the start index of its edges in the edge array. CSC records the edges in inverse direction: the edge array stores the sources of the edges.

4.2.3 Frontier Representation

We adapt the representation of frontiers between bitmaps and arrays of vertex IDs on-the-fly, depending on their density [37]. Frontiers are created either by constructors, or by the *edge-map* and *vertex-map* functions. From the users point of view, frontiers are immutable. One of the constructors creates a frontier containing all vertices. We explicitly record this property in the frontier to in order to omit checks of the frontier and speed up graph traversal. Remember that graph analytics typically perform little work per edge. As such, any reduction in instruction count has a measurable impact.

This optimization affects both the forward and backward traversal for dense frontiers (a frontier containing all vertices is by definition not sparsely populated). The backward traversal benefits much more from this optimization as it performs more lookups in the frontier, namely once per edge vs. once per vertex in the case of the forward traversal. We similarly optimize the *vertex-map* operation and any auxiliary loop iterating over the frontier.

4.2.4 Graph Representation

The compressed sparse rows (CSR) format and the compressed sparse columns (CSC) format [81] are often-used to store graph data as they efficiently support both sequential traversal over all edges and random access to the edges of a specific vertex. These representations, however, waste time on vertices with zero degree, which are very common in partitioned graphs (see Section 4.1.2).

GraphGrind uses a modified CSR and CSC representation in order to combat efficiency issues with zero-degree vertices. We compress the index array by storing only information for vertices with non-zero degree. In order to know which vertices are present, we also store the vertex ID in each element of the index array. Overall, this reduces the size of the index array due to the high number of zero-degree vertices. The main benefit, however, is that a sequential edge traversal becomes more efficient as iteration over the index array automatically skips all zero degree vertices.

GraphGrind stores each individual graph partition in the CSR and CSC representations. This representation is, however, not efficient for traversals with sparse frontiers as these require random access. Moreover, sparse traversals are slowed down significantly by graph partitioning as they are by nature dominated by control flow, which is only made worse by the replication of vertices. As such, we retain a *non-partitioned* copy of the original CSR representation of the graph specifically for sparse traversal.

4.2.5 Partition Balancing Criterion

We have argued that balancing the number of edges across partitions does not necessarily result in the best balancing of CPU time. Instead, some algorithms observe better CPU load balancing when the number of vertices in each partition is about equal. GraphGrind adds a parameter to the algorithm specification that shows its preference for a balanced edge partitioning vs. a balanced vertex partitioning. This parameter is checked during graph ingress in order to select the balancing criterion for graph partitioning. Our balanced vertex partitioning is similar to Algorithm 1, except that we strive for $|V|/P$ destination vertices in each partition.

Balancing vertices is appropriate for 3 of the 8 algorithms that we use in the experimental evaluation. The algorithms are commonly used in prior work. As such, this property is sufficiently important to ask programmers to record it. The property is easily derived from the algorithm specification.

4.2.6 NUMA Optimization

The state-of-the-art in NUMA-aware programming requires two coordinated actions: (i) data placement and (ii) thread placement. Common data placement strategies are to allocate data in a specific NUMA node or to distribute the data across nodes. Thread placement is optimized such that the thread has a low latency/high bandwidth connection to the NUMA domain holding its most frequently accessed data. This two-pronged strategy allows for many optimizations, such as co-locating threads with data

Table 5: NUMA allocation and binding strategy

Data structure	**NUMA allocation**
full graph	interleaved
graph partition	allocate on one node
vertex arrays	match home partition
Operation	**NUMA binding**
edge-map (sparse)	none
edge-map (dense)	bind to holding node
vertex-oriented loops (e.g., vertex-map)	equally distribute loop iterations over NUMA nodes

and spreading data and threads across NUMA domains to enhance memory bandwidth.

Graph partitions can enforce NUMA-local access as each partition can be stored and processed within the confines of one NUMA node. Prior work has advocated to replicate frontiers and algorithm-specific data arrays on each NUMA node [108]. Accordingly, memory accesses are NUMA-local, except when interchanging data across nodes.

GraphGrind follows a different route, which is summarized in Table 5. The full graph is stored in an interleaved fashion over the NUMA nodes. As the full graph is used with sparsely populated frontiers only, the memory accesses are few and hard to schedule optimally. Interleaved allocation provides a good compromise.

Graph partitions are spread over NUMA nodes in such a way that each partition is stored on one NUMA node and all NUMA nodes hold the same number of partitions. A graph traversal over a partition is scheduled on the NUMA node that holds that partition. This ensure that the majority of memory accesses are issued against the local NUMA node.

We distribute *vertex arrays* over NUMA nodes, storing the element for each vertex on the same NUMA node as its home partition. As such, the *edge-map* operation that is *writing* data to a vertex element performs NUMA-local accesses. This placement incurs some *false sharing*, as NUMA placement works on the granularity of virtual memory pages. As such, a small fraction of the vertices will be placed on a remote NUMA node. E.g., assuming 1 M vertices, at most 1 in 10,000 will be stored in a different node.

The distribution of vertex arrays may be highly skewed due to the imbalance of vertices in each partition. Loops iterating over the vertex arrays, such as *vertex-map* and loops that analyze frontiers, are however scheduled such that the loop iterations are equally spread across NUMA nodes. While this induces some remote NUMA accesses, it is far more important to load-balance these loops than it is to optimize NUMA-awareness.

An alternative strategy is to replicate the vertex arrays on each NUMA node [108]. We found this to be sub-optimal due to the additional memory traffic that is required to

Table 6: Graph algorithms and their characteristics. Frontiers: S=sparse, D=dense.

Apps	Description	Edge traversal	Frontiers	Cache	Balance
BC	betweenness-centrality [86]	backward	SDS	Yes	Vertices
CC	connected components using label propagation [86]	backward	DS	Yes	Edges
PR	simple Page-Rank algorithm using power method (10 iterations) [71]	backward	D	Yes	Edges
BFS	breadth-first search [86]	backward	SDS	No	Vertices
PRD	optimized Page-Rank forwarding delta-updates between vertices [86]	forward	DS	No	Edges
SPMV	sparse matrix-vector multiplication (1 iteration)	forward	D	No	Edges
BF	Bellman-Ford algorithm for single-source shortest path [86]	forward	SDS	No	Vertices
BP	Bayesian belief propagation [108] (10 iterations)	forward	D	No	Edges

replicate and to merge vertex arrays. In contrast, our NUMA placement and scheduling rules guarantee that an *edge-map* operation on a graph partition only writes to vertex array elements stored on the local NUMA node. Read operations may be remote, but these have lower impact on performance. As such, we obtain good NUMA locality without incurring the overhead of replicating data.

The algorithm is robust against anomalous conditions such as absence of active threads on a NUMA domain and a mismatch between the number of NUMA domains specified by the program and those in hardware. In both cases, pending iterations are executed by threads who have completed their work. Performance may be sub-optimal in both instances, but correctness is guaranteed. Thread pinning can be applied in order to guarantee performance.

The NUMA-aware extension supports non-commuting reductions [26] and pedigrees [53]. Both constructs depend on the execution order of function calls, which the helper function disrupts. Although it takes minimal effort to support these constructs, it is beyond the scope of this paper to explain the mechanics.

4.3 Experimental Evaluation

We evaluate GraphGrind experimentally on a 4-socket 2.6GHz Intel Xeon E7-4860 v2 machine, totaling 96 threads. It has 256 GB of DRAM. We compile all codes using our modified version of the Clang compiler which implements the NUMA extension to Cilk. We evaluate 8 graph analysis algorithms, described in Table 6, using 9 widely used graph data sets, described in Table 7. All reported results are averaged over 5 executions.

Table 7: Characterization of real-world and synthetic graphs used in experiments.

Graph	Vertices	Edges	Type
Twitter [48]	41.7M	1.467B	directed
Friendster [103]	125M	1.81B	directed
Orkut [64]	3.07M	234M	undirected
LiveJournal [103]	4.85M	69.0M	directed
Yahoo_mem [100]	1.64M	30.4M	undirected
USAroad [108]	23.9M	58M	undirected
Powerlaw ($\alpha = 2.0$)	100M	1.5B	directed
RMAT24	16.8M	168M	directed
RMAT27	134M	1.342B	directed

4.3.1 Performance Comparison

We compare the performance of GraphGrind against leading graph analytics systems for shared-memory, namely Ligra [86], Polymer [108] and Galois [67] (Table 8). GraphGrind and Polymer both use 4 partitions to match the NUMA characteristics of our hardware. All systems use 96 threads. We show the backward PageRank algorithm for Polymer as the forward version, presented in [108], contains errors. The absolute execution times are different than reported in other papers as we use different hardware, a different compiler and have different randomly generated graphs. Moreover, some algorithms are sensitive to the start vertex, which in our experiments is vertex 100 for all graphs. The trends, however, match previously reported results.

Overall, GraphGrind outperforms the other systems for all algorithms and all graphs, except for CC and BF on the USAroad graph. In these cases, Galois is faster. This results from using different algorithms [67, 108]. Nonetheless, GraphGrind makes progress over Polymer and Ligra for these cases. In a few cases, GraphGrind performs on par with other systems. These are labeled in bold-face as well.

The performance improvements are significant: up to 326% faster than Ligra (SPMV with Orkut graph) and up to 82.2% faster than Polymer (BP with USAroad graph). The smallest speedups appear for BFS, as there is already little computation going on. The superior performance of GraphGrind results from a combination of optimizations. Next, we will tease out the main contributing factors.

4.3.2 Graph Representation

GraphGrind's graph data structure prunes vertices with zero degree from the representation. We will show later that this saves significant spaces compared to the CSC and CSR representations used by Polymer. Moreover, by not storing these vertices, *edge-map* traversals no longer need to visit them. Figure 28 shows the speedup resulting from the graph representation for 5 algorithms, which ranges between 2% and 16%. Twitter and LiveJournal benefit most due to the high sparsity of graph partitions.

Table 8: Runtime in seconds of GraphGrind, Polymer, Ligra and Galois. The fastest results are indicated in bold-face. Execution times that differ by less than 1% are both labeled. Missing results occur as not all systems implement each algorithm.

Apps	Graph	GG	Polymer	Ligra	Galois
CC	Twitter	**1.810**	2.580	2.878	16.660
	Friendster	**5.924**	8.030	7.330	6.210
	Orkut	**0.122**	0.180	0.138	0.311
	LiveJournal	**0.111**	0.177	0.125	0.206
	Yahoo_mem	**0.042**	0.049	0.063	0.046
	USAroad	35.348	36.730	38.910	**20.110**
	Powerlaw	**1.168**	2.110	1.680	3.113
	RMAT24	**0.455**	0.522	0.601	1.440
	RMAT27	**2.305**	3.220	2.444	10.120
BC	Twitter	**1.771**		4.130	4.160
	Friendster	**3.394**		5.490	6.110
	Orkut	**0.149**		0.160	0.178
	LiveJournal	**0.197**		0.334	0.388
	Yahoo_mem	**0.091**		0.110	0.150
	USAroad	**4.402**		5.174	6.010
	Powerlaw	**2.118**		2.300	2.860
	RMAT24	**0.482**		0.503	1.110
	RMAT27	**2.073**		2.360	15.110
PR	Twitter	**15.979**	20.400	23.660	20.120
	Friendster	**38.249**	41.8	43.300	61.200
	Orkut	**1.596**	1.660	2.240	2.120
	LiveJournal	**0.652**	0.688	0.708	0.700
	Yahoo_mem	**0.234**	0.262	0.278	0.255
	USAroad	**0.933**	1.220	1.582	1.180
	Powerlaw	**10.394**	12.716	13.600	11.614
	RMAT24	**2.730**	2.970	3.660	3.110
	RMAT27	**17.517**	23.21	28.600	30.220
BFS	Twitter	**0.254**	0.298	0.319	0.449
	Friendster	**0.896**	**0.899**	1.210	1.330
	Orkut	**0.039**	0.043	0.044	0.051
	LiveJournal	**0.050**	0.068	0.078	0.103
	Yahoo_mem	**0.025**	0.026	0.033	0.363
	USAroad	**1.750**	1.855	2.009	5.180
	Powerlaw	**0.595**	0.601	**0.599**	0.993
	RMAT24	**0.104**	0.119	0.118	**0.104**
	RMAT27	**0.412**	0.421	0.429	0.631

4.3.3 Adapating Graph Partitioning

We remove CPU load imbalance through selecting an appropriate criterion to balance the graph partitions. We identified through code inspection that 3 of the evaluated algorithms (BFS, BC and BF) prefer an equal number of vertices in each partition. The others prefer a uniform number of edges. Figure 29 shows the speedup obtained by balancing vertices over balancing edges for these 3 algorithms and PR. We show results for a subset of the graphs, the remaining graphs behave similar to the

Table 8: Continued.

Apps	Graph	GG	Polymer	Ligra	Galois
PRD	Twitter	**20.560**	24.120	29.890	
	Friendster	**36.097**	36.600	62.100	
	Orkut	**1.244**	1.310	3.472	
	LiveJournal	**1.013**	1.110	1.138	
	Yahoo_mem	**0.831**	1.094	1.640	
	USAroad	**2.124**	2.260	2.905	
	Powerlaw	**10.659**	14.100	16.900	
	RMAT24	**1.845**	2.230	2.911	
	RMAT27	**8.645**	12.120	14.500	
SPMV	Twitter	**2.251**	2.860	4.610	
	Friendster	**3.624**	5.220	9.010	
	Orkut	**0.148**	0.208	0.630	
	LiveJournal	**0.060**	0.096	0.151	
	Yahoo_mem	**0.033**	0.045	0.063	
	USAroad	**0.077**	0.128	0.166	
	Powerlaw	**0.655**	**0.661**	0.707	
	RMAT24	**0.197**	0.221	0.288	
	RMAT27	**1.963**	2.210	2.830	
BF	Twitter	**1.489**	1.618	2.213	12.810
	Friendster	**6.498**	7.193	7.690	9.220
	Orkut	**0.213**	0.310	0.354	2.100
	LiveJournal	**0.258**	0.293	0.284	0.530
	Yahoo_mem	**0.146**	0.200	0.173	0.288
	USAroad	21.992	24.110	26.310	**16.330**
	Powerlaw	**10.326**	11.112	12.600	15.110
	RMAT24	**1.366**	1.390	1.410	1.880
	RMAT27	**1.665**	1.933	2.180	5.310
BP	Twitter	**38.896**	**38.900**	56.980	
	Friendster	**58.704**	66.210	129.000	
	Orkut	**2.223**	3.110	5.538	
	LiveJournal	**1.026**	1.420	1.940	
	Yahoo_mem	**0.448**	0.455	1.124	
	USAroad	**1.024**	1.660	1.462	
	Powerlaw	**15.264**	15.530	19.500	
	RMAT24	**4.788**	7.030	9.310	
	RMAT27	**32.994**	43.320	58.230	

ones shown. The partitioning has negligible impact for Friendster and PowerGraph, which have a balanced number of vertices per partition in either case (see Figure 26). Graphs with unbalanced partitions see important improvements with vertex-balanced partitions, with up to 37% speedup for LiveJournal.

Vertex-balanced partitioning is appropriate only for algorithms with fixed amount of work per vertex. Other algorithms, like PR, have a strong preference for edge-balanced partitioning. We conclude that it is crucial to balance partitions appropriately to the algorithm.

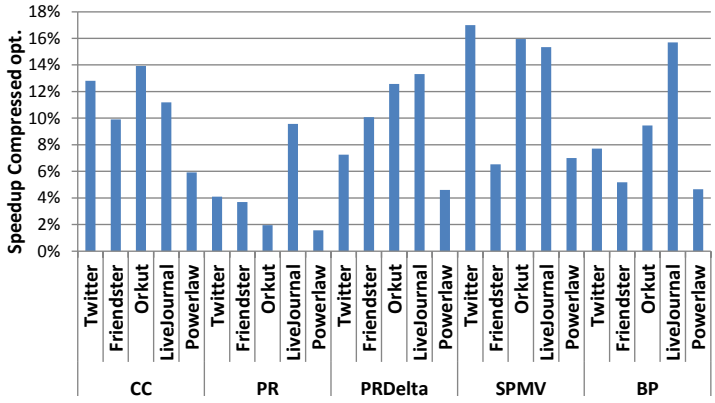


Figure 28: Speedup of compressed graph compared to visit zero-degree vertices.

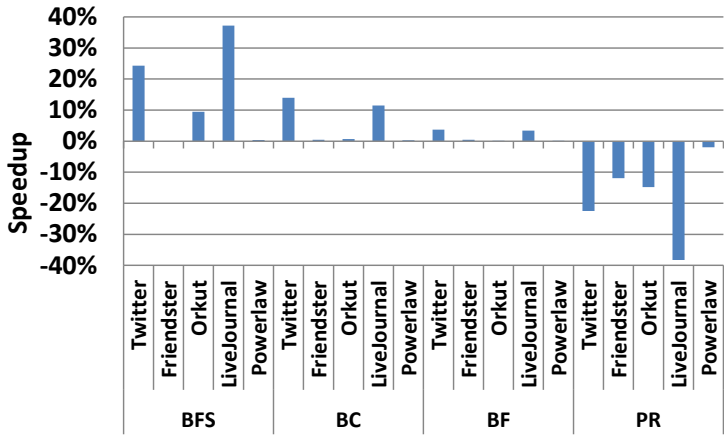


Figure 29: Speedup of balancing vertices compared to balancing edges in graph partitions.

4.3.4 NUMA Optimization

Various choices can be made for the placement of vertex arrays, i.e., arrays storing frontiers or per-vertex application-specific data. GraphGrind places the vertex arrays such that each vertex is co-located with its home partition. Vertex-oriented loops, such as those in *vertex-map*, are typically short and have well-balanced work per iteration. As such, GraphGrind distributes the iterations equally across threads, even though this results in remote NUMA accesses.

We compare two variations on the NUMA policy (Figure 30): (i) placing vertex data and scheduling iterations on their home partition; (ii) equally spreading vertex data and iterations across all NUMA nodes. Option (i) aims to avoid remote NUMA access during vertex-oriented loops. This is however uniformly worse than GraphGrind's policy. It shows that CPU load balance is simply more important than NUMA locality for the vertex-oriented loops.

Option (ii) load-balances vertex-oriented loops and tries to minimize remote NUMA

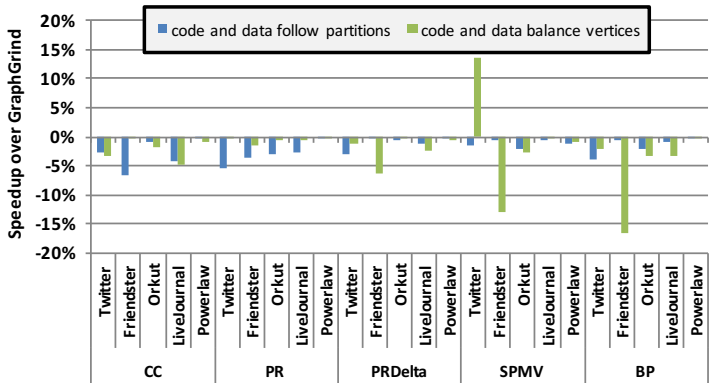


Figure 30: Impact of NUMA decisions for vertex arrays. GraphGrind may be described as data follow partitions, code balances iterations.

accesses by spreading vertex arrays to match the distribution of iterations. This results in worse performance in nearly all cases as the placement decision is sub-optimal for the *edge-map* operator. This operator performs the majority of main memory accesses and will incur excess remote memory accesses when vertices are not co-located with their home partition.

An interesting effect occurs when SPMV processes the Twitter graph, as in this case an increase in remote memory accesses during *edge-map* results in improved performance. We contrast this against Friendster, where the same effect results in performance degradation. We measured the local and remote memory accesses incurred and observe that both GraphGrind and option (ii) incur the same total number of memory accesses and that option (ii) incurs an increased number of remote accesses for both graphs.

The performance difference between the graphs, however, results as Twitter has highly skewed partitions: The number of elements of vertex arrays accessed on one NUMA node is much higher than on other NUMA nodes. Where GraphGrind directs those accesses to the local NUMA node, option (ii) spreads them across nodes. This way, option (ii) can share the unused memory bandwidth on one NUMA node with the computation on another node. On Friendster, GraphGrind is faster than option (ii) because Friendster has relatively uniform partitions and performs more memory accesses per unit of time. As such, all NUMA nodes are equally stressed and there is no benefit in making remote accesses.

These results show that a careful trade-off is required to optimize NUMA placement, as option (i) incurs fewer remote memory accesses than GraphGrind, yet has worse performance. In rare cases can remote accesses result in performance improvement due to imbalance in memory traffic.

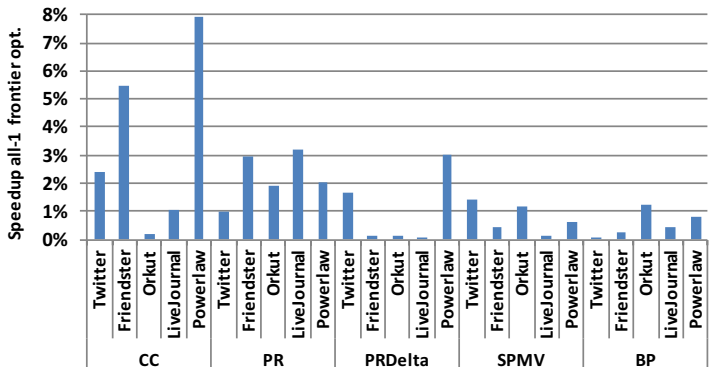


Figure 31: Speedup due to all-vertex frontier optimization.

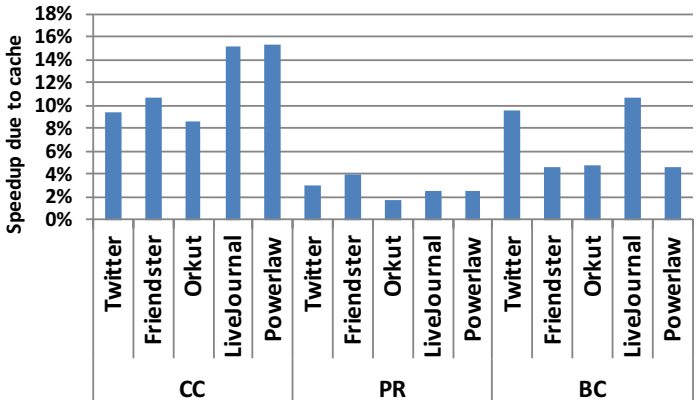


Figure 32: Speedup due to holding intermediate values in registers.

4.3.5 Peephole Optimizations

GraphGrind labels frontiers that are initialized to contain all vertices such that an optimized version *edge-map* can avoid memory accesses and control flow related to frontier access. Figure 31 shows the speedup resulting from this optimization for all algorithms that initialize frontiers this way. Only algorithms that initialize frontiers this way can benefit. The algorithms using backward traversal (CC and PR) benefit most, up to 8%, as the backward traversal queries the frontier once for every edge, while the forward traversal queries it only once per vertex. The speedup is modest, but consistently positive. It moreover requires no user intervention.

GraphGrind allows programmers to define a cache, which allows the compiler to store intermediate values in registers (the cache) and avoid memory accesses. This optimization is relevant only during dense, backward traversal. Figure 32 shows that for the relevant algorithms, the cache results in a speedup between 2 and 15%.

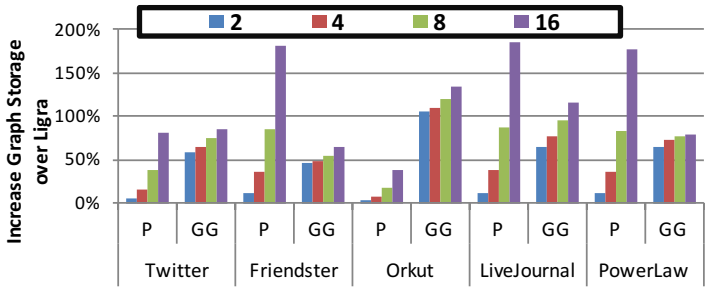


Figure 33: Increase of graph storage for Polymer (P) and GraphGrind (GG) compared to Ligra.

4.3.6 Memory Usage

The premise of graph processing on scale-up shared-memory machines is that sufficient main memory can be provided to store the graph data. Thus, memory usage should remain under control. Figure 33 shows the additional memory used on graph data for Polymer and GraphGrind compared to Ligra. Polymer stores each graph partition in CSR and CSC format (as in Ligra) using index arrays of length $|V|$. Because of this, the memory consumption of Polymer grows as $P|V|$ for P partitions. As GraphGrind stores only vertices with non-zero degree in the index arrays, its memory usage grows more slowly and follows the vertex replication factor (Figure 25). However, as GraphGrind stores an additional copy of the graph for sparse traversal, it starts at a 50% increase compared to Ligra for directed graphs, and 100% increase for undirected graphs. Overall, GraphGrind's memory consumption is more scalable than Polymer's.

4.4 Further Related Work

It has been documented that generic tools such as METIS [44] to partition graphs by vertex or edge cut do not produce good partitions for social network graphs. Moreover, they take much more time to compute than many graph algorithms. Sheep [60] is a distributed graph partitioner that produces high quality edge partitions an order of magnitude faster than METIS. Alternatively, linear-time heuristics have been proposed. The vertex cut is a greedy edge partitioning algorithm that minimizes the number of cut vertices [30].

GraphChi [49] is designed to stream graph data in from disk. It uses partitioning to obtain small vertex sets that fit in the main memory. It uses partitioning by destination and aims for an equal number of edges per partition. The number of vertices must be made to fit in memory by tuning the number of partitions.

X-Stream [80] uses what we call partitioning by source, but does not required edges to be pre-sorted. It aims for a uniform number of vertices per partition as it wants to keep only vertex data in fast memory (e.g., CPU cache), whereas edges are streamed

in from slower memory (e.g., main memory).

GraphX [102] is a Spark [89] library for graph analytics. It partitions edge lists using Spark's resilient distributed datasets (RDD) and supports user-defined partitioning schemes.

Our observations are relevant for each of the systems discussed above. E.g., the reduced connectivity of partitions implies that memory locality is poor in a system like X-stream. A large variation in vertices per partition implies that partitions with few vertices will leave a large portion of main memory unutilized in GraphChi.

Frasca et al. [25] design NUMA-aware work queues for betweenness centrality. The work queues first execute locally generated work prior to stealing work from other queues. Work queues are visited in order of increasing NUMA distance. They demonstrate a 51.2% performance improvement compared to an OpenMP implementation.

Agarwal et al. [3] study the execution of breadth-first-search on NUMA systems. They too organize the computation around work queues, spread over multiple sockets. Moreover, they use efficient spinning locks and lock-free channels to synchronize threads. They also introduce peephole optimizations such as avoiding atomic operations by first checking if they will fail.

Graph compression can significantly reduce memory requirements and with it memory bandwidth. Shun *et al* [87] compress the destination IDs of vertices stored in the edge array of the CSR and CSC representations. They reduce memory usage up to 56%. These techniques are orthogonal to the compressed representation of the CSC and CSR index arrays proposed in this work, as they pertain to edges only.

4.5 Conclusion

Graph partitioning is an important technique to efficiently orchestrate the execution of graph analytics in the context of NUMA-aware data placement and code scheduling. We apply Swan's NUMA-aware scheduling extension to improve the performance of Ligra, a graph analytics framework based on Cilk with a clean API that hides the framework details from users. Prior work on NUMA-aware graph analytics has proposed Polymer, where parallelism is hard-coded using POSIX threads and explicit barrier synchronization. Using this approach, Polymer is unable to keep the user code separated from the framework, which results in highly complex and error-prone code. In this context, GraphGrind leverages the NUMA-aware loop scheduling extension of Swan to retain the cleanliness and separation of concerns achieved with Ligra's API.

In order to achieve performance improvement, however, we needed to study and address several issues that arise from graph partitioning, including load imbalance, increased work per vertex, and a significantly reduced connection density. Combined, these problems imply that graph partitioning is inherently unscalable to large partition counts.

We propose several techniques to counter-act the identified performance issues and implement these in GraphGrind, a novel NUMA-aware graph analytics framework that is compatible with the Ligra API.

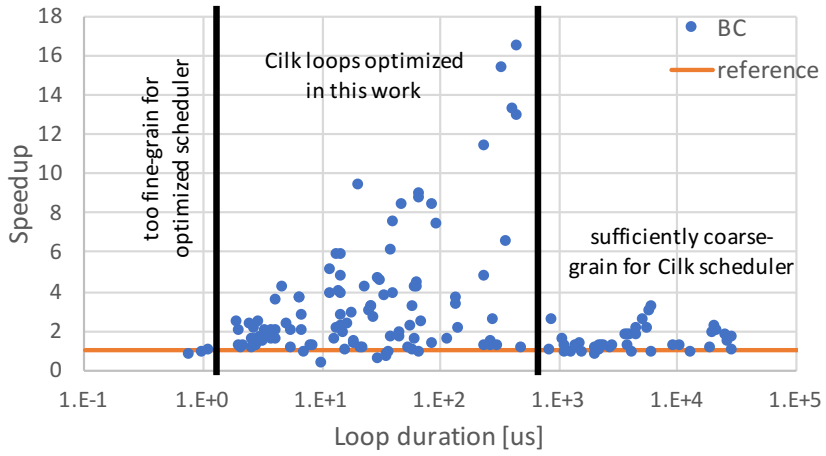


Figure 34: Speedup of fine-grain parallel loops as a function of loop duration in betweenness-centrality (BC) computation on the Twitter graph.

GraphGrind achieves significant speedup compared to prior work, out-performing Polymer, the most recent contender, by as much as 82%. We moreover show that fully minimizing remote memory accesses is not optimal in irregular computations. Instead, one needs to strike a careful trade-off between remote accesses and CPU load balancing.

5 Case Study: Fine-Grain Scheduling in Data Analytics

While Moore's Law remains active, every new processor generation has an increasing number of CPU cores. Highly-parallel processors such as Intel's Xeon Phi Knights Landing [88] provide a high number of less powerful but energy-efficient cores. Moreover, scale-up shared memory machines such as the SGI UV line serve tightly synchronized workloads. Scheduling and distributing work load on large scale shared-memory machines becomes increasingly important in order to make efficient use of the hardware.

Scheduling and work distribution induce a run-time overhead, called *burden* [35]. The burden includes the time taken to make scheduling decisions, send the work to other processors and synchronize on the completion status. The scheduler burden has not been widely documented or studied. It has been reported that creating tasks in Cilk has "about an order of magnitude" overhead compared to a normal function call [10]. However, this does not yet involve distributing the task to other processors.

In order to understand the scale of the problem, Figure 34 shows the duration of fine-grain parallel loops that occur in the betweenness-centrality benchmark taken from Ligra [86]. These loops perform operations such as reductions, filtering, and packing of

array elements. The loops already execute on 48 threads (see Section 5.1 for details on the platform) using the Cilk runtime [28]. The dynamic range of loop duration is very high, ranging from sub-microsecond to 10s of milliseconds. This relates to the loop iteration count as well as the amount of work performed per iteration. The vertical axis (Figure 34) shows the speedup obtained by reducing the scheduler burden using Swan's fine-grain scheduling annotation, which can result in as much as a 16-fold speedup for some loops. We will demonstrate that other schedulers, such as OpenMP, are also significantly impacted by their burden.

An immediate consequence of the burden is that some parallel code is *too fine-grain* to make parallel execution worthwhile. What "too fine-grain" means precisely depends strongly on the hardware, the scheduling algorithm and its implementation. The existence of the burden has important implications on the scalability of the scheduler as the burden may grow with increasing degrees of parallelism. Morever, the burden affects performance portability as the acceptable lower-bound on the granularity of parallel loops differs between hardware.

In a bid to reduce the burden, a significant body of research has investigated how to reduce the computational complexity of scheduling algorithms. Near-optimal execution times can be obtained with *greedy* scheduling techniques. These always execute available work and execute it in the order it arrives [11]. Alternatively, *static* schedulers apply an easy-to-calculate work distribution but leave no room for adapting the schedule. While this limits run-time overhead, the burden remains significant.

Other research has investigated ways to reduce the synchronization delay through specialized hardware support [47, 23]. Hard-coding a scheduler in hardware, however, removes the option of tuning schedulers to application areas or application-specific properties. Numerous scheduling algorithms have been proposed in the literature, none of which performs unanimously best across a wide range of applications. It is thus important to be able to select scheduling algorithms and retain a programmable implementation. Alternatively, hardware support for message queues is more general [75, 83]. The key benefit of message queues is that they use dedicated on-chip networks to circumvent the cache coherence protocol [83]. The intra-socket ping-pong delay between two threads using the cache coherence protocol on a modern processor is, however, around 60 ns (measured on a 2.6 GHz Intel Xeon E7-4860 v2). As such, the potential benefit of hardware support for message passing is much smaller than the burden of state-of-the-art software schedulers.

Contributions: This work proposes new techniques for scheduling and work distribution of fine-grain parallel loops and presents compiler support to enhance their efficiency. Reducing the minimum granularity of loops that can be efficiently scheduled on a large-scale shared memory machine addresses many problems in parallel computing. It minimizes the need to tune parallel loops to hardware characteristics. It also minimizes the need to question if parallelizing a particular code region pays off, a question that every parallel programmer faces. Moreover, fine-grain work distribution mechanisms improve the scalability of schedulers and execute parallel programs efficiently on large-scale machines.

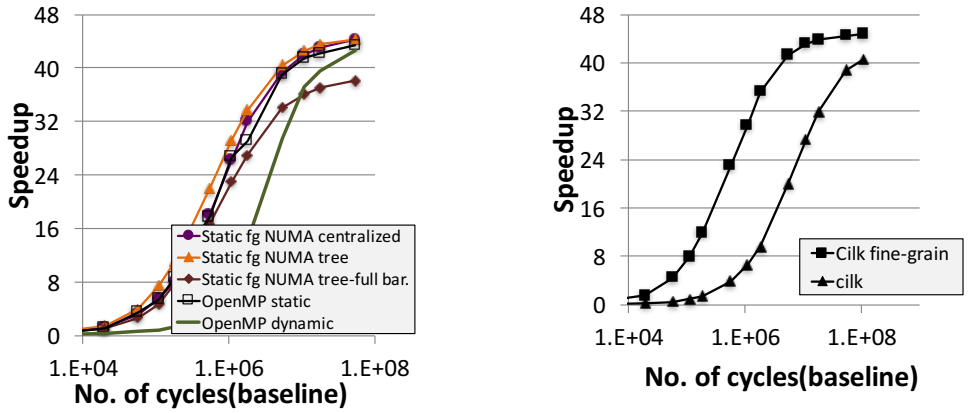


Figure 35: Speedup of **uforall** micro-benchmark for the OpenMP (left) Cilk (right) runtimes

5.1 Experimental Evaluation

We evaluate our scheduler on a 4-socket 2.6 GHz Intel Xeon E7-4860 v2 machine with 12 physical cores (plus hyperthreading) and 30 MB L3 cache per socket. We pin threads to cores so that only one thread per core is used. The operating system is CentOS 7.0. We have implemented the fine-grain scheduling technique in both OpenMP and Swan in order to demonstrate that it is generally applicable. We use the Intel C/C++ compiler v. 14.0.0 for OpenMP applications and Clang version 3.4.1 for Swan applications.

We use two microbenchmarks (**uforall**, a parallel loop micro-benchmark, and **STREAM** [61]), and six benchmarks: 3 map/reduce benchmarks **histogram**, **lreg** and **pca** from the Phoenix++ suite [92], SPEC CPU2006 **mcf**, PARSEC **streamcluster** [8] and the GraphGrind graph analytics framework (see Section 4). For the map-reduce benchmarks we use the small, medium and large inputs [92]. For mcf we use the SPEC test, train and reference inputs. For streamcluster we use the simsmall, simmedium and simlarge inputs.

We report results averaged over 15 executions of the benchmarks. We calculate speedups against the sequential version of the benchmark where no parallel constructs appear. This method correctly penalizes parallel runtimes for any overhead they may induce. Moreover, we exclude the runtime startup time in all speedup results.

5.1.1 Scheduling Burden

The **uforall** micro-benchmark is designed to measure loop scheduling overhead. By varying the amount of work in the parallel loop, we can emulate loops of different granularities. Figure 35 shows the speedup obtained for varying granularity (sequential execution time) of the loop for the OpenMP- and Cilk-based runtimes. Note that the

Table 9: Characterizing the burden of the schedulers

	d (μs)
Fine-grain–NUMA tree	5.67
Fine-grain–NUMA centralized	7.55
Fine-grain–NUMA tree with full-barrier	12.00
OpenMP static	8.12
OpenMP dynamic	31.94
Cilk	68.80

horizontal axis is displayed on log-scale. The curves do not extend to 48-way speedup due to measuring overhead in the micro-benchmark.

The micro-benchmark results visualize the burden of the scheduler. The OpenMP static scheduler has smaller burden than OpenMP dynamic scheduler, which in turn has smaller burden than the Cilk scheduler. Note that these are in part properties of the implementation and may very between implementations of OpenMP.

The micro-benchmark allows us to estimate the overhead of scheduling. We assume that Amdahl's Law applies:

$$S = \frac{T}{d + T/48}$$

where T is the sequential execution time, d is the work distribution time and S is the resulting speedup. We perform a least-squares fit between the experimental data and the model to estimate d. The burden is several micro-seconds (Table 9). The dynamic schedulers have a significantly higher burden than the static ones. These estimates confirm that using a half barrier in the fine-grain scheduler reduces the scheduling delay by about half compared to a full barrier.

Note that a burden of 69 μs for Cilk implies that a loop can achieve decent speedup only when its parallel execution time approaches 1 ms. For instance, speedup is 12 when $T = 16\, d = 1.1$ ms This is in agreement with Figure 34 where the fine-grain scheduler can deliver an extra 4x speedup.

5.1.2 STREAM

We use the STREAM benchmark [61] to demonstrate that memory-bound loops are sensitive to efficient scheduling. In a NUMA system it is important to schedule loops such that memory accesses are directed to the locally attached memory. The remote memory, attached to other CPU sockets, typically has significantly lower memory bandwidth. As such, schedulers like Cilk and OpenMP dynamic achieve poor memory bandwidth, on our system no more than half of peak memory bandwidth. Static schedulers, in contrast, consistently issue loop iterations to the same threads. In combination with a first touch policy, this results in NUMA-local accesses.

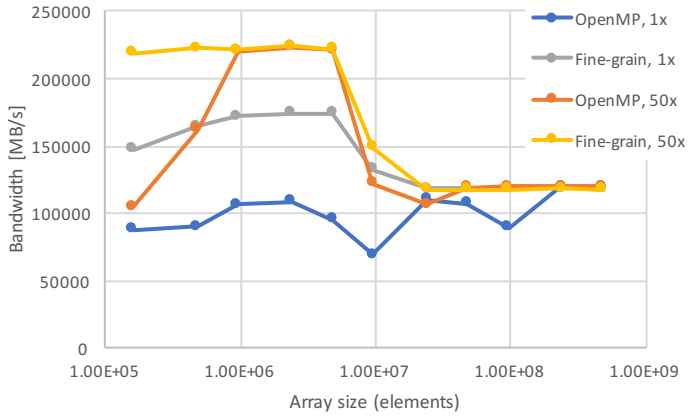


Figure 36: Performance of STREAM triad for varying array sizes and repeating work items

We demonstrate the importance of scheduling efficiency using OpenMP static scheduling and the Cilk-based fine-grain scheduler. As these use different compilers (icc vs. clang), with different optimization quality, we use the same, highly tuned assembly code in the kernels.

Figure 36 shows the throughput obtained with the triad kernel (curves labelled "1x"). The array size is varied along the horizontal axis. The amount of work performed in the triad loop is proportional to the array size. The largest array sizes used do not fit in the on-chip caches and show that the sustainable memory bandwidth is about 120 GB/s. Smaller array sizes fit in the on-chip caches and result in markedly higher bandwidth with the fine-grain scheduler. For the smallest array shown, the kernel completes in 28 μs, which leaves little time for synchronisation. The fine-grain scheduler achieves on average 60% higher bandwidth than OpenMP, which does not succeed in exploiting the on-chip bandwidth at all.

We demonstrate that the OpenMP runtime achieves less bandwidth for small array sizes due to scheduling overhead. Hereto, every thread repeats its task 50 times (curves labelled "50x"). As such, the overhead of synchronisation is reduced 50-fold. This markedly improves the bandwidth achieved with both runtimes and allows them to achieve nearly the same bandwidth for array sizes over 1 million. For smaller arrays, and finer-grain tasks, the OpenMP runtime again performs worse. This demonstrates that the performance difference is due to task granularity.

5.1.3 Speedup of Fine-Grain Scheduler

We evaluate the speedup achieved by the fine-grain scheduler when integrated in the OpenMP and Cilk runtimes, and compare its performance against the baseline runtimes. We focus on the NUMA-tree version from now, as it gives best performance. Figure 37 shows the speedup obtained by fine-grain scheduling across a number of

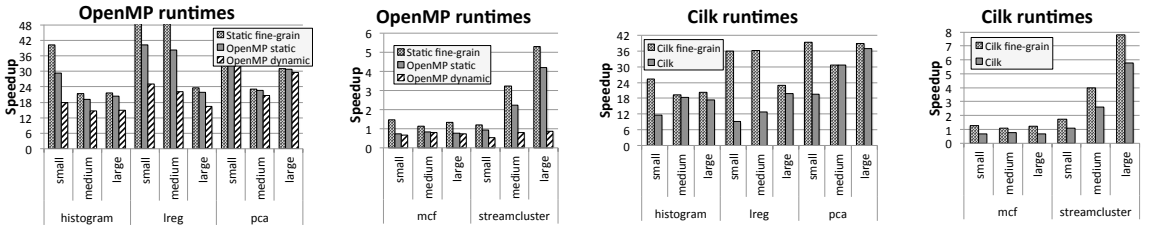


Figure 37: Speedups obtained by the fine-grain scheduler compared to the baseline OpenMP static and Cilk runtimes for various input data sets. We compare 48-thread execution times for all benchmarks, except for mcf where we compare 12-thread execution times.

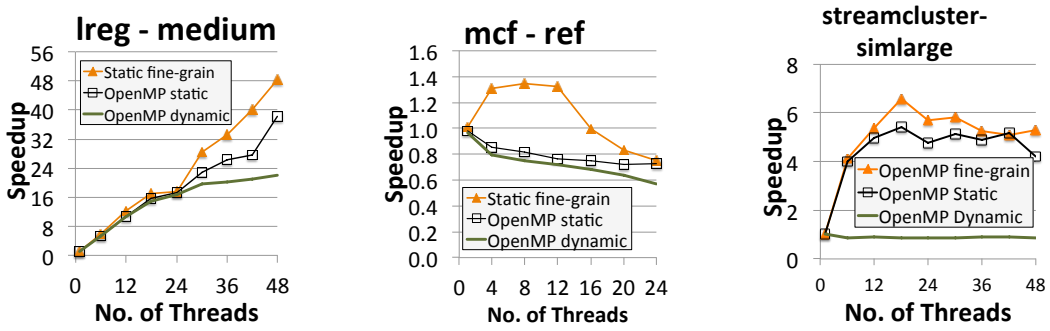


Figure 38: Speedup of benchmarks executing on the OpenMP-based runtimes.

input data set sizes. Note that the OpenMP and Cilk results are compiled with different compilers. As such, the two sets of results are normalized against slightly different sequential baselines and we analyze them separately.

We have analyzed the sensitivity of the dynamic schedulers to the grain size of scheduled work items. We found no important impact of this parameter on performance in a wide range of values.

The fine-grain scheduler out-performs the Cilk and OpenMP runtimes on all applications. Improvements are generally higher for smaller input data sets. This is not unexpected, as the scheduler burden is more exposed on small data sets. Speedups over the Cilk and OpenMP schedulers range up to 3.9x when executing the program end-to-end.

Figure 38 and Figure 39 show details on the scalability of a few benchmarks. The other benchmarks are in line with these results. In many cases, the speedup of the baseline schedulers reduces as the degree of parallelism is increased. In contrast, the fine-grain scheduler is able to squeeze incremental performance gains out of additional threads. In the case of mcf, it is impossible for the baseline schedulers to obtain speedup over sequential execution. This is a very hard benchmark. The fine-grain scheduler consequently cannot achieve parallel speedup when using a second socket.

We have moreover measured execution time of the Phoenix++ runtime [92], which was specifically constructed to achieve high performance on map-reduce workloads

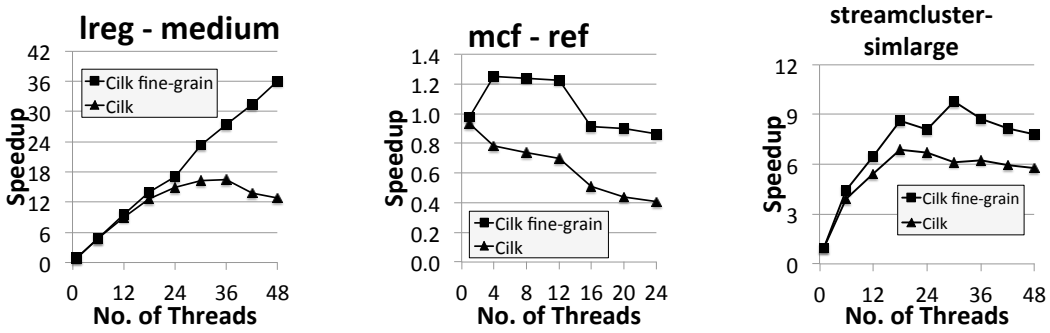


Figure 39: Speedup of benchmarks executing on the Cilk-based runtimes. same for OpenMP graphics

Table 10: Performance of graph analytics workloads and statistics on duration of fine-grain loops.

	fine-grain loops		time per fine-grain loop			Cilk		Hybrid		speedup	speedup
	nb.	< 0.1ms	min	avg	max	total	fine-gr.	total	fine-gr.	total	fine-gr.
BFS	73	48	1.84 μs	1.27 ms	21 ms	0.387	0.093	0.376	0.076	2.75%	21.6%
BC	148	33	0.77 μs	2.06 ms	27.2 ms	1.59	0.304	1.53	0.250	4.35%	21.9%
CC	118	93	0.44 μs	1.32 ms	21.6 ms	2.13	0.155	2.10	0.129	1.48%	29.6%
PR	82	17	45.9 μs	11.1 ms	19.9 ms	13.7	0.907	13.4	0.711	2.73%	27.5%

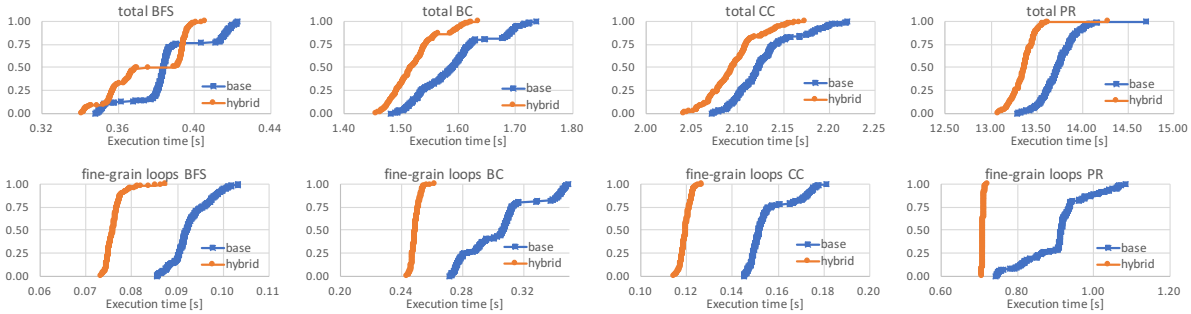


Figure 40: Cumulative distribution of execution time in fine-grain loops and total execution time for GraphGrind workloads using the Cilk/Swan scheduler as a baseline, and using the hybrid scheduler.

like lreg, histogram and pca. Although not shown, the performance of Phoenix is largely in line with the OpenMP versions of these codes. We attribute the reason to the use of coarse-grain locking on work queues and shared data structures in the runtime. We conclude that the problem posed by fine-grain parallelism indiscriminately affects general-purpose and special-purpose parallel runtimes.

5.1.4 Hybrid Scheduling

We apply the hybrid static/dynamic scheduler to GraphGrind, a graph analytics system. GraphGrind contains phases of coarse-grain parallelism seperated by fine-grain parallel operators such as prefix-scan and reduce over arrays of highly varying length, ranging from 10s to millions of elements. We use it to analyse the ability of the scheduler to switch efficiently between coarse-grain and fine-grain parallelism.

Table 10 summarizes the execution times of 4 graph analytics kernels processing the Twitter graph [48]. We execute each workload 200 times and separate out the time spent in the fine-grain parallel loops and compare against an execution using the baseline Cilk runtime. The fraction of total execution time covered by fine-grain loops varies between 6.6% and 24%, depending on the graph analytics kernel. The experiments demonstrate that low-overhead scheduling speeds up fine-grain loops by 21.6% to 29.6%, and speeds up the whole application by 1.5%–4.4%. The only intervention required is to label the fine-grain parallel loops.

Fine-grain loops in GraphGrind are very short (Table 10, columns 2–6): the shortest loops take less than a microsecond (2600 CPU cycles on our system) and almost half of the fine-grain loops take less than 0.1 millisecond. A few loops take 20–30 ms, which skews the average loop duration statistic.

Figure 40 shows the cumulative distribution function of the time spent in fine-grain loops and total application time across 200 executions. The fine-grain loops are clearly accelerated as all, or nearly all, of the measurements on the hybrid scheduler are smaller than those on the baseline scheduler. Moreover, the variance of execution time on the static scheduler is less as the slope of the curves is larger.

We use the Welch two-sample t-test to test the equality of the averages of total execution times. This test is appropriate as the distributions are approximately normal. We assume a 5% confidence level. The difference in execution time between the hybrid and baseline schedulers is statistically significant for BFS (μ = 0.376s and σ = 0.195s for the hybrid scheduler vs. μ = 0.387 and σ = 0.198s for Cilk) with $t = 5.268$ and 397.97 degrees of freedom gives $p = 2.3e - 7$. The other cases yield even lower p-values. These statistics show that the improvement of the hybrid scheduler exceeds the variation on the execution time.

We have applied the one-sided F-test to check that variance is reduced. Here, we find that the hybrid scheduler likely does not reduce variance for BFS ($F = 1.021$ with 199 degrees of freedom and $p = 0.44$). However, for the other kernels we find that the hybrid scheduler does reduce the variance of the total execution time, e.g., for BC, $F = 2.423$ with 199 degrees of freedom and $p = 4.2e - 10$.

The distributions moreover demonstrate that the static scheduler does not suffer from stragglers. In fact, the Cilk scheduler is more susceptible to long delays. We believe this results from random work stealing in the Cilk scheduler, as the granularity of stolen work depends on the selected victim, and with it the number of work stealing events and ensuing runtime management.

5.2 Conclusion

We have demonstrated the importance of efficient and scalable scheduling and work distribution for fine-grain parallel programs. We have designed a loop scheduler for fine-grain parallelism with efficient, atomics-free work distribution using a half-barrier loop pattern. We integrated the fine-grain scheduler in the Intel OpenMP and Cilkplus runtimes and designed compiler support to enhance the performance on reductions. Experimental evaluation demonstrates up to a 3.2x performance improvement over the OpenMP and Cilk schedulers. The fine-grain scheduler, moreover, achieved speedup on loops where the baseline schedulers slow down compared to the sequential baseline.

We have moreover proposed metrics to assess the supported granularity of schedulers. Our fine-grain scheduler supports 2.1x finer-grain parallelism than the OpenMP static scheduler and 21.4x fine-grain parallelism than Cilk.

This work has practical implications to parallel computing. Programmers need to constantly decide whether it is worthwhile to parallelize individual code fragments. This choice depends on a combination of factors, among which the data set size, runtime system implementation and hardware specifics. Making a wrong decision may lead to performance degradation when code is in production. The present work significantly reduces the impact of such decisions and improves the performance portability of parallel programs.

6 Case Study: Text Analytics

This section demonstrates the application of Swan to text analytics. This work was presented at IEEE Big Data [96]. Text analytics are an important class of data analytics, differentiated from analytics in general by extracting information from *textual data*. While exact data is hard to obtain, it is claimed that 80% of all big data is unstructured, hence textual in nature [31].

Analyzing data at high speed is immensely important given that data volumes are consistently growing. The dominant approach to scaling up analytics capabilities consists of using increasing numbers of servers. Single-thread performance, i.e., the time it takes an individual server to process its part of the work, is generally neglected [62]. This approach is not scalable in the long term due to operational costs of the high number of components involved and the diminishing returns that result from scaling out. In contrast, improving the single-thread performance of analytics can reduce operational costs even in the face of growing data volumes.

The data analytics literature generally pays little attention to single-thread performance. There is a good motivation for this: single-thread code is typically written by data analysts and it is not desirable to require high-performance computing expertise from data analysts. In contrast, performance-critical code is encapsulated in libraries and frameworks, although the performance of these is under scrutiny [62, 69, 73, 34, 84]. However, to the best of our knowledge, there exist no frameworks, nor good prac-

tice, for manipulating textual data *at high speed* for general algorithms. The goal of this work is to fill this gap in the literature and provide guidelines for achieving *high-performance text analytics*. Moreover, we present HPTA, a library that implements these guidelines in a reusable way.

Here we presents three guidelines towards high-performance text analytics:

Memory management: text analytics will deal with a large number of text fragments. These fragments are often short, e.g., words in a natural language. Traditional memory management, involving independent allocation of each fragment, is not scalable due to the performance overhead of fine-grain dynamic memory management and the resulting fragmentation. Nonetheless, popular systems such as Hadoop [1] and Spark [89] follow this approach. We investigate techniques to circumvent this problem and experimentally characterize their effectiveness.

Parallelism and associative data structures: associative data structures track the computed values for each text fragment. It is well known that the choice of associative data structure, e.g., hash table versus map, affects performance. Data analytics frameworks have settled on lists of key-value pairs as the main associative data structure [1, 107]. Few would argue that this is optimal in single-threaded applications, yet it seems to work well for parallel execution, in particular for data partitioning and reduction. In contrast, frameworks that use more complex data structures are restricted to single-threaded execution [74, 33]. We argue that parallel execution is possible using any associative container and we present methods for partitioning and reduction. Experimental validation shows that the use of appropriate data structures outperforms the list-based representation.

Moving data is faster: We demonstrate that different phases in text analytics applications utilize the data structures in different ways. As such, phases require different data structures, which leads to the counter-intuitive result that moving the data to different data structures during the computation reduces execution time, even though data volumes are large. Note however that we re-organize data within a node.

6.1 Related Work

Research into high-performance text analytics often involves approximate algorithms and acceleration using Graphics Processing Units (GPUs).

The Term Frequency-Inverse Corpus Frequency (TF-ICF) algorithm [78] approximates the IDF scores using document relevance metrics that are pre-calculated over a reference corpus. The assumption is that the document frequencies are constant for a large enough corpus. TF-ICF scores can be calculated in a single pass, as opposed for two passes for TF-IDF. While this work demonstrates our techniques on the TF-IDF algorithm, they are equally applicable to TF-ICF.

Erra *et al* [22] present a GPU implementation of an approximate streaming version of TF-IDF. The TF-IDF metric is approximated by counting occurrences of a pre-set number of terms only in order to meet the memory limitations of GPUs. Our approach in contrast produces exact counts.

Zhang *et al* [109, 110] study document clustering on clusters of computers equipped with GPUs. They pre-compute TF-ICF scores [78]. on the CPU and accelerate a *flock clustering* algorithm on the GPU. They demonstrate a 10x speedup when using 16 high-end NVIDIA GPUs compared to executing on a single desktop.

Szaszy *et al* accelerate document *stream* clustering where they assume that a stream of documents needs to be continuously clustered [91]. They use sparse matrix-vector multiply (SpMV) techniques to compute the similarity between the TF-IDF of a document and the reference clusters. The SpMV calculation is performed on the GPU. They do not study the issue of text parsing and assume that a document is converted to TF-IDF form prior to entering their system.

Each of the above works investigate acceleration of the numerical aspect of document clustering algorithms. Numerical computations are however well understood. In contrast, this work focuses on the text processing aspect of text analytics.

An important component of this work is concerned with parallel operation on associative data structures. Several works have investigated scalable parallel data structures. The Standard Template Adaptive Parallel Library (STAPL) [77, 93] distributes data structures across a cluster by partitioning the key space. Accesses to data structures are transparently forwarded to the appropriate machine. The Parallel Standard Template Library (PSTL) [43] is a similar, older project. The parallel-STL approach has limited scalability in comparison to this work as it aims to parallelize individual operations on associative data structures. This work, in contrast, is concerned only with parallel iteration.

PDQCollections [99] processes associative data structures in a map-reduce-like model. The authors consider functions on the data such that the data may be split (e.g., by key range), operated on independently and then merged as in a reduction operation. PDQCollections is more akin to the approach taken in this work due to the reduction of associative data sets. An important distinction is that our approach is not specific nor limited to map-reduce programs.

6.2 Text Analytics: TF-IDF Case Study

To simplify the exposition, we will study term frequency-inverse document frequency (TF-IDF) [82] as a guiding example of text analytics. While the TF-IDF operation is simple enough to understand in detail, it exposes important reoccurring properties of text analytics operations.

TF-IDF assigns a weight to each term-document combination. The weights reflect the importance of the term within the document and across the set of documents. Fig. 41 shows a pseudo-code for TF-IDF. This code uses a number of associative containers that store information on each encountered term. These operations are described in Table 11. Firstly, the code uses an associative container per input file to store the term frequency within that document. I.e., the container associates every term (key) to its frequency of occurrence (value). Secondly, a single associative container is used to calculate the document frequency across the collection. This con-

```
procedure TFIDF(documents[0..n-1]) {
    // term frequency per document
    associative_container(string -> int) term_freq[n];
    // document freq. and ID
    associative_container(string -> (int,int)) doc_freq;
    parallel_for (i : 0..n-1) {
        // Calculate term frequency in i-th document
        parallel_for (term : documents[i])
            modify(term_freq[i], term,+,1)
        // Update document frequencies for term in i-th document
        // Increment counter for each term ignoring term frequency
        // Value of ID is irrelevant at this time
        merge(doc_freq,term_freq[i],
                f=(k,(dfl, idl), tfr)->(k,(dfl+1, idl)),
                g=(k, tfr)->(k,(1,0)))
    }
    // Assign unique IDs to each term. The terms can be optionally
    // sorted alphabetically. Sorting here affects the order of
    // terms in the TF-IDF matrix and output.
    // Store IDs in second element of value pair in doc_freq.
    sort-by-key(doc_freq)
    ID = 0;
    parallel_for (term : doc_freq) {
        modify(doc_freq,term,f=((tf,old_ID),ID)->(tf,ID))
        ID += 1
    }
    // Construct TF-IDF (sparse) matrix
    parallel_for (i : 0..n-1) {
        for((term, tf) : term_freq[i]) {
            // Calculate TF-IDF score for term in i-th document
            (df, id) := lookup(doc_freq,term)
            tfidf [i, id] := tf * log((df+1)/(n+1))
        }
    }
    return tfidf
}
```

Figure 41: Code structure for term frequency–inverse document frequency calculation for a collection of documents. The operations on associative containers are defined in Table 11.

tainer stores two integer values for each term encountered in each of the documents: the number of documents where the term occurs (document frequency) and a unique sequence number that is determined only when all files have been read. The latter number is important for sorting the output data alphabetically.

The algorithm has three distinct phases. In the first phase (term count phase), all documents are read and the per-document term frequency is determined. Moreover, all terms from all documents are added to the document frequency container and the document frequency is updated. The containers are mostly updated during the term

Table 11: Common operations on associative containers.

Operation	Description
insert(c, k, v)	insert value v for key k in collection c
modify(c, k, f, v)	modify collection c to store value v_0 for key k as $v_0 = f(v_0, v)$ or insert v if key k absent
lookup(c, k)	lookup what value is stored for key k in collection c
iterate-seq(c, k, v)	retrieve the next stored key-value pair
iterate-par(c, k, v)	as *iterate-seq*(k, v) but can be used as iterator in a parallel for-loop
merge(c_l, c_r, f, g)	merge collection c_r into c_l by storing the value $f(v_l, v_r)$ if $(k, v_l) \in c_l$ and $(k, v_r) \in c_r$ for a key k, or by inserting $(k, g(v_r))$ for $(k, v_r) \in c_r$.
sort-by-key(c)	sort all entries of collection c by key
sort-by-value(c)	sort all entries of collection c by value

count phase. The access pattern consists thus of random accesses.

The second phase of the algorithm assigns a unique ID to each term. This is helpful to build the TF-IDF matrix, i.e., to index it by numeric ID rather than by string. Assigning IDs is however also critical in order to produce an alphabetically sorted output.

The third phase computes the TF-IDF scores and stores them in a matrix. Although the pseudo-code depicts a dense matrix, a sparse matrix is used as non-stop-words typically occur in only a fraction of the documents. The matrix is built up by rows, where rows can be easily constructed in parallel. Each row corresponds to a document and is constructed by iterating over all elements of the corresponding per-document term frequency container. Each term in this container is looked up in the term frequency container to obtain the corresponding document frequency.

6.3 Optimization of Text Analytics

We have identified optimization opportunities that are applicable to text analytics operations in general, and to TF/IDF specifically. We will experimentally demonstrate their impact in Section 6.5.

6.3.1 Memory Management

Text analytics operate on a large collection of text fragments. A common goal is to map these text fragments into a numeric multi-dimensional space [82], but until that mapping is achieved, text analytics operations process individual text fragments. The text fragments can be created and represented in multiple ways:

Text fragments are individually allocated as they are read in or discovered. Memory allocators typically round allocated memory sizes up to frequently occurring sizes. This will incur significant internal fragmentation as text fragments have highly

Table 12: Time complexity of operations on associative containers assuming the container holds n elements.

Operation	**Time Complexity**			
	List	**Sorted List**	**Hash Table**	**Map**
insert(k, v)	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$
modify(k, f, v)	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$[a]	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$
lookup(k)	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$
iterate-seq(k, v)	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
iterate-par(k, v)	$\mathcal{O}(1)$	$\mathcal{O}(1)$	n/a	n/a
merge(c_l, c_r, f, g)	$\mathcal{O}(n_l n_r)$	$\mathcal{O}(n_l + n_r)$	$\mathcal{O}(n_r)$	$\mathcal{O}(n_l + n_r)$[b]
sort-by-key(c)	$\mathcal{O}(n \log n)$	$\mathcal{O}(1)$	n/a	$\mathcal{O}(1)$
sort-by-value(c)	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	n/a	n/a

[a] $\mathcal{O}(n)$ for new keys due to moving elements in the array.

[b] Assumes usage of the C++'11 insertion hint indicating that the element is inserted in the immediate neighborhood of an iterator. The iterator is assumed to be the position where the previous element was inserted.

varying lengths. Alternatively, systems using garbage collection will incur significant garbage collection overheads when all text fragments are separately allocated. The garbage collector must analyze these objects upon each collection pass, adding to the overhead of garbage collection [90]. However, it can be expected that large groups of text fragments have equal life-times in text analytics applications.

The input files are retained integrally. A fast solution results when reading in input files integrally into working memory [76], e.g., using `mmap` on UNIX-based systems. This avoids separate memory allocations for each fragment in the input. However, it will result in large memory overhead and bad memory locality. In particular, when terms repeat many times in the same document, each repetition of the word will be held in memory while a *bag-of-words* model requires that only one copy of each word is stored. This is the case, e.g., in the TF-IDF example. More importantly, retaining full input files requires that sufficient main memory is available.

Region-based memory allocators aim to maximize performance by eradicating internal fragmentation and by efficiently de-allocating a large number of items in bulk [29, 38, 90]. Region-based memory allocation is effective when individually allocated items go out of scope at the same time, implying that their memory can be reclaimed at the same time. Region-based memory management is more sophisticated than retaining all input files in memory, but results in similar performance benefits.

Region-based memory management is generally provided using application-specific code [38, 29]. Language support has been proposed [32, 85] but is not widely available. As such, we have selected a library implementation.

Table 13: Conversion cost of associative containers holding n elements.

Source	Target Container			
	List	**Sorted List**	**Hash Table**	**Map**
List	–	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$	$\mathcal{O}(n \log n)$
Sorted List	$\mathcal{O}(1)$	–	$\mathcal{O}(n)$	$\mathcal{O}(n)$[a]
Hash Table	$\mathcal{O}(n)$	$\mathcal{O}(n \log n)$	–	$\mathcal{O}(n \log n)$
Map	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	–

[a]Assumes usage again of the C++'11 insertion hint. If absent, time complexity is $\mathcal{O}(n \log n)$.

6.3.2 Reference Associative Containers

A myriad associative containers have been proposed in the literature, each making distinct trade-offs in the time complexity of various operations, in average-case vs. worst-case time complexity, in memory efficiency, in raw performance, etc. The goal of this work is not to identify the best possible container for text analytics or for TF-IDF. Rather, we aim (i) to demonstrate that text analytics are sensitive to the properties of the containers, (ii) identify the opportunity for moving data from one container type to another during an algorithm and (iii) to set out guidelines how to select container types.

We consider four different classes of associative containers: lists of key-value pairs, lists of key-value pairs sorted by key, hash tables and hash maps. These are different enough to warrant their study. Table 12 shows the average-case time complexity of the common operations for these 4 data structures. Time complexity is a good predictor of execution time given that analytics typically concerns large data sets. For sorted lists of key-value pairs we assume that value lookup uses a binary search algorithm [46]. The time complexity of the hash table is based on the unordered map described in the C++ standard [15], while the map is based on the C++ map, which always stores its elements in sorted order.

Table 12 shows that a hash table provides best time complexity on a range of operations. However, it is not possible to sort the contents of a hash table. In order to do that, it is necessary to move the data to a different container, either a list of key-value pairs or a map. However, once the data has been moved over, all operations have higher time complexity. It is now more expensive to access the data. Hence, a careful trade-off is necessary to decide on conversions.

For completeness, we show the time complexity of conversion in Table 13, and of merge operations in Table 14.

6.3.3 Container Selection

As indicated above, containers must be selected with care, but when done right, there is opportunity for switching containers. In this Section we outline our methodology to select containers. Referring back to the TF-IDF algorithm (Figure 41), we observe that each container is used in different ways throughout the algorithm. The per-document

Table 14: Cost of merging associative containers of different types.

Right (m)	**Left Argument** (n)			
	List	**Sorted List**	**Hash Table**	**Map**
List	$\mathcal{O}(mn)$	$\mathcal{O}(mn)$	$\mathcal{O}(m)$	$\mathcal{O}(m \log n)$
Sorted List	$\mathcal{O}(mn)$	$\mathcal{O}(m+n)$	$\mathcal{O}(m)$	$\mathcal{O}(m)$
Hash Table	$\mathcal{O}(mn)$	$\mathcal{O}(mn)$	$\mathcal{O}(m)$	$\mathcal{O}(m \log n)$
Map	$\mathcal{O}(mn)$	$\mathcal{O}(m+n)$	$\mathcal{O}(m)$	$\mathcal{O}(m)$

term catalogs are used in line 15 only with the *modify* operation. At lines 9 and 29, the catalogs are traversed sequentially, either in a *merge* operation or using *iterate-seq* during the construction of the TF-IDF matrix. The *modify* operation is clearly most efficient on a hash table (Table 12). Iteration over all elements has $\mathcal{O}(1)$ time complexity for lists and the hash table. Detailed measurement has shown however that iteration through an array-based list is more efficient than through a hash table. As such, we consider that there is opportunity to change the container type for the term catalogs prior to line 15.

Similarly, we analyze the usage of the document frequency container. This container is updated using *merge* at line 15. The merge operation is most efficient when the left-hand-side argument (`doc_freq`) is a hash table (Table 14). In fact, the hash table is the only data structure where the time complexity of *merge* is independent of the size of `doc_freq`. At line 21, however, the document frequency container must be sorted by key, which is impossible with a hash table. A change in container is thus necessary due to the functionality. At line 23, the document frequency container is traversed, preferably in parallel. This is most efficient with a list-based data structure. The subsequent modification is, however, $O(1)$ in all cases as *modify* can be performed through a pointer into the container. Finally, at line 31, a *lookup* of the document frequency is performed, which is again more efficient with a hash table. We have thus identified four code regions accessing the document frequency container. Each code region has a distinct preference for the container type, which can be distinct from that of the preceding code region.

Note that data structure conversions are a non-obvious choice when working with potentially large data sets. In fact, the leading data analytics platforms have designed their parallel execution exclusively around lists: Hadoop [1] operates exclusively on key-value lists while Spark [89] is organized around resilient distributed data sets (RDDs), which, like our key-value lists, are essentially arrays.

6.3.4 Parallelization

Parallelism occurs naturally in data analytics due to the possibility to traverse data sets in parallel. While it is clear that an array-based list can be traversed in parallel, so too can any iterable collection. In the worst case, parallel traversal may require additional computation in order to get each parallel thread started. Concretely, for data structures

Table 15: Characterization of input data sets.

Data set	Description	Size	Files	Unique words
Various	"Classic3" and "Classic4" data sets (CISI, CRAN, MED and ACM) and Reuters press releases (Reuters-21578)	62.8 MB	23 K	192 K
NSF Abstracts	NSF research award abstracts 1990–2003 [94]	311 MB	101 K	268 K
Gutenberg	A selection of e-books from Project Gutenberg, covering multiple languages	20.00 GB	52,361	259 M
Artificial	Phoenix++ [92] word count data set. 4th and 5th file repeat 3rd file 4 times, respectively 8 times	1.33 GB	5	144 K

providing a C++ *random access iterator*, such as arrays, we divide the iteration range among the threads. Each thread can jump directly to the appropriate position due to the random access nature of the iterator. For data structures that provide a C++ *input iterator*, we can divide the iteration range similarly on the basis of the number of elements to iterate over. However, finding the appropriate starting point requires repeated increments of the iterator to traverse from the beginning of the collection to the desired point. This can be done, e.g., using `std::advance()` in C++, which is a linear-time operation for input iterators.

Apart from traversing data sets in parallel, we also need to construct associative data structures in parallel. One approach is to use concurrent or parallel data structures where multiple threads can insert or modify elements [93, 43]. This approach potentially has limited scalability due to the need to synchronize threads when accessing the shared data structure. The approach chosen in this work is to construct private data structures within each thread and to *merge* these data structures in pairs as threads complete. We demonstrate that this approach results in a high degree of scalability.

If we apply the above obersvations to TF-IDF, we observe that *all* of the loops in Fig. 41 can be executed in parallel. Some loops are trivial to parallelize, while others require more work. For instance, the loop at Line 8 can be parallelized by dividing the document in large chunks, split at a word boundary [76]. Distinct associative containers are computed for each chunk. These are reduced in pairs using a tree reduction at the end of the loop. Moreover, the loop at Line 23 can be parallelized using a prefix sum [9].

6.4 HPTA Library

The key component of HPTA is a *word bank*, a data structure that implements region-based memory allocation of strings. The word bank consists of a list of large chunks

of memory that are allocated using the system-supplied memory allocator. Words that are added to the word bank are allocated at the end of the last chunk using bump-pointer allocation [45]. When all memory in the last chunk has been used, or the next string is too long to fit in the chunk, a new chunk is appended to the list. The chunk size can be tuned by the programmer. In general, using larger chunk sizes results in less system overhead.

HPTA furthermore couples each associative data structure with a word bank. As such, the associative data structure and its word bank are created and destroyed together. The main advantage of this approach is that it is safe to store pointers to strings in the associative data structure where the pointers point into the word bank.

HPTA furthermore implements auxiliary data structures such as sparse and dense vectors and matrices and methods for reading and writing the WEKA file format [33].

6.5 Experimental Evaluation

We analyze the proposed optimizations for text analytics experimentally on a quad-socket 2.6GHz Intel Xeon E7-4860 v2, totaling 48 threads. The operating system is CentOS 6.5 with the Intel C compiler version 14.0.1. We have implemented HPTA in C++ and parallelized key operations with Cilk [28], using Intel Cilkplus. Reported results are averaged over 10 executions. We use 4 public data sets with varying sizes in the evaluation (Table 15). The "artificial" data set has few documents. As such, its execution time is dominated by word frequency computation.

We assume that documents are encoded in the UTF-8 format [104] with unique representations for all strings. I.e., a choice is made between "á" and the diacritic "a' " to represent the accented character a. Under this assumption lexicographic ordering of UTF-8 strings can be determined by comparing character by character without decoding the content.

The evaluation below focuses on the TF-IDF algorithm for words. We have also evaluated the effectiveness of the optimizations when calculating TF-IDF for 3-grams. The results are qualitatively the same. As such we present results only for single-word terms (1-grams).

6.5.1 Memory Management

The memory management policy has an important impact on the performance of text analytics. Fig. 42 shows parallel speedup using the system memory allocator, region-based memory management and retaining all input files in-memory. All associative data structures are hash tables in this experiment. Note that parallel speedups range from 4$\times$ to 24$\times$ and correlate strongly to the data set size.

The system allocator has the lowest performance across the board. The performance of per-word memory allocation could be improved on by using NUMA-aware memory allocators [59], However, NUMA-awareness is not the only issue. Analyzing

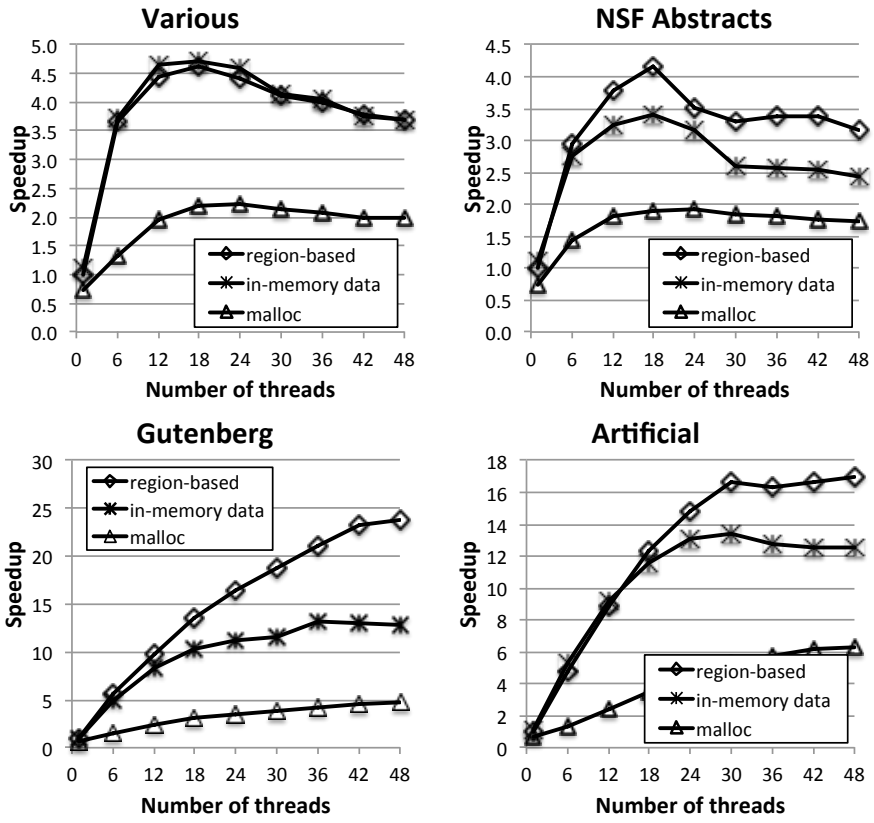


Figure 42: Parallel speedup dependent on the memory management policy. Speedups are normalized against region-based memory management.

the single-thread execution time demonstrates that per-word memory allocation also incurs overhead due to extra work performed.

Phoenix [76, 92] retains all input files in-memory. This avoids memory allocation as each word can point directly to the input file buffer. This technique is fastest for the smaller input sets. However, it has poor parallel scalability due to increased working set size (Fig. 42). We consider only the region-based allocator from this point on.

6.5.2 Exploration of Container Types

We first analyze performance assuming only one data structure is used throughout the computation. Fig. 43 shows the single-threaded execution time normalized to using a hash table. We omit the execution times for the sorting stage as this is marginal or not applicable in the case of the hash table.

Using key-value lists throughout the computation performs up to 20x slower than hash tables for the NSF Abstracts data set. This is interesting to note as the key-value list abstraction is fundamental to the operation of Hadoop [1] and lies at the heart of Spark's RDDs [89].

The main performance bottleneck in our list-based implementation is the *merge*

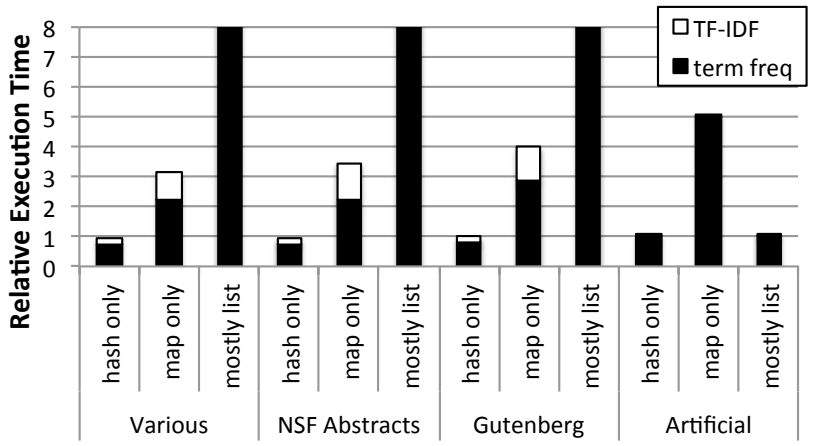


Figure 43: Execution time when storing the term frequency data in a hash table, a key-value list or a map, normalized to the hash table case.

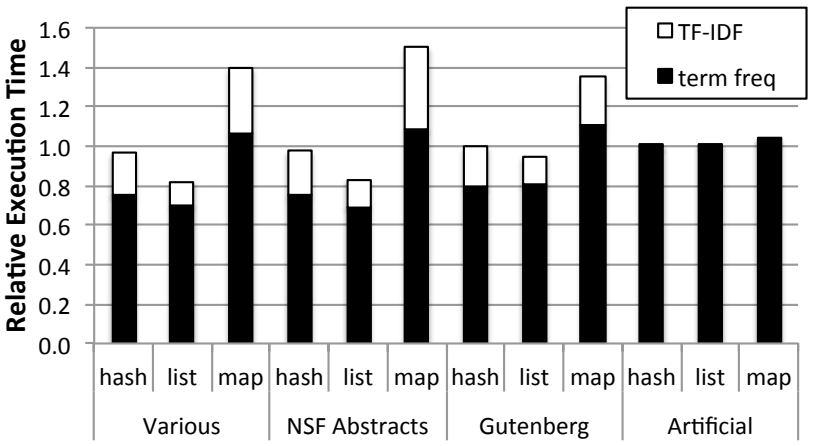


Figure 44: Execution time when retaining the term frequency data in a hash table, or when converting it to a key-value list or a map, normalized to the hash table case. Document frequencies are stored in a hash table.

operation, which has time complexity $\mathcal{O}(m+n)$ to merge collections of n and m elements. Note that *merge* is called once per document and that the size of the target container is continuously growing throughout execution. Assume for the sake of argument that d documents contain m unique words each, then the time complexity of *merge* is $\mathcal{O}(d^2m)$. A Hadoop-like sorting solution could perform better with a time complexity of $\mathcal{O}(dm \log dm)$, assuming a list of dm words is first constructed by concatenation and subsequently sorted.

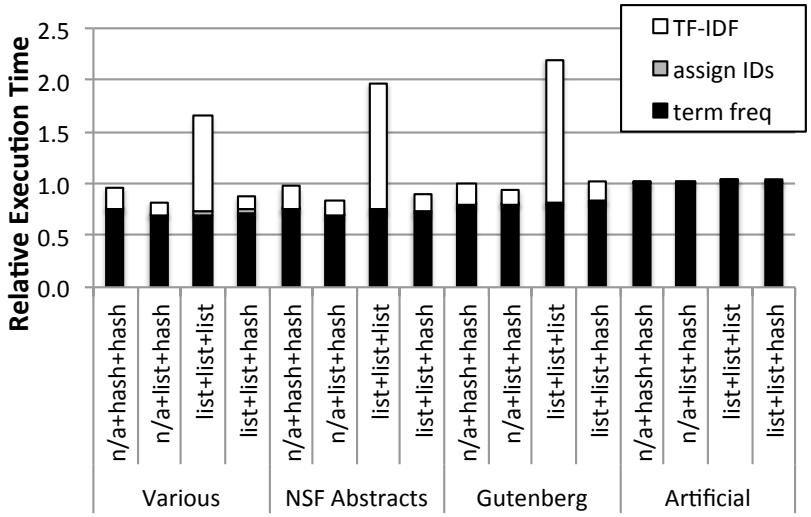


Figure 45: Execution times. Format: sort+iterate+lookup, where sort is the container used to sort words, iterate is the container type iterated during term catalog and lookup is the container type used for document frequency lookup. The remaining operations are performed on hash tables.

6.5.3 Unsorted Output

We will first consider the case where the corpus need not be sorted alphabetically. In this case, the sorting step can be omitted and document frequencies can be stored in a hash table throughout the algorithm. We have however observed that execution time can be reduced by converting the term frequency container to a sorted list. Term frequencies are stored in a hash table during construction (Lines 8– 9, Fig. 41) and converted to a list prior to Line 13. Fig. 44 shows performance when using a hash table, a key-value list or a map for the *merge* and *iterate-seq* operations. Converting the data to key-value list is worthwhile as it is much faster to iterate through a list vs. a hash table. Overall execution time is reduced by up to 19% for the “Various” data set. The “Artificial” data set is slowed down marginally ($< 1\%$) as the conversion takes time and does not lead to significant gains due to the low number of documents.

6.5.4 Sorted Output

Sorting the output alphabetically is best achieved by converting of document frequencies to a sorted container which, in practical terms, implies a sorted key-value list (results with a map are invariably worse). The TF-IDF phase performs lookups on the document frequencies. These can be performed either using binary search on the list, or on a hash table provided the data is converted back to a hash table. Fig. 45 shows these options. The first bar corresponds to using only hash tables. The second bar corresponds to converting term frequencies to a list, the best case for unsorted output. The third bar shows execution time performing lookups using binary search on a

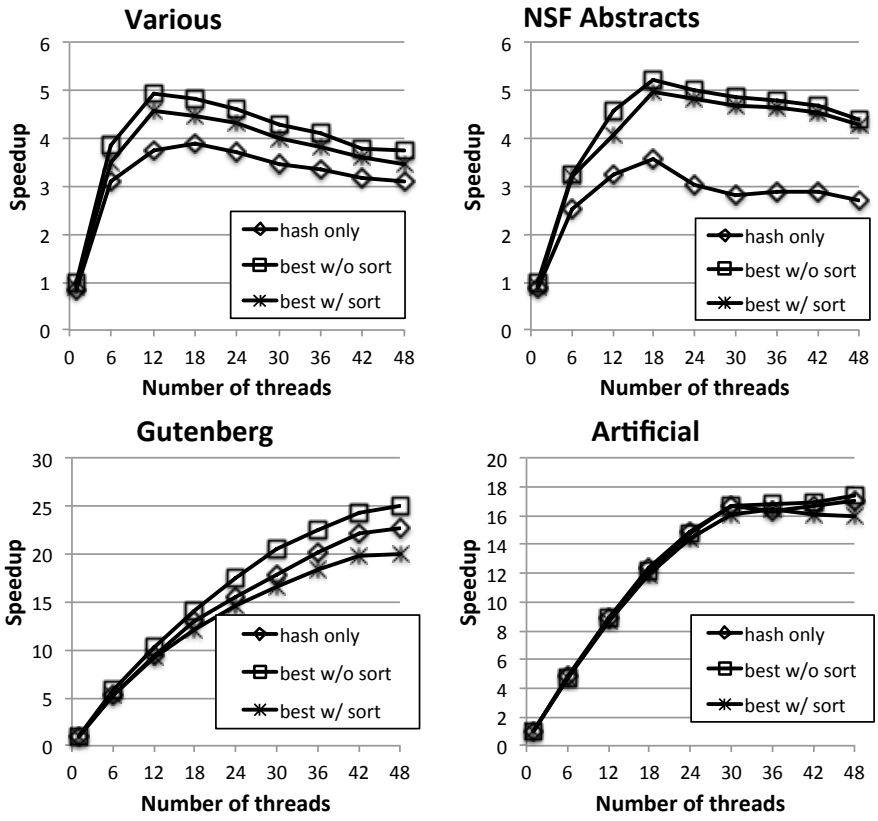


Figure 46: Parallel speedup of TF-IDF normalized against using a hash table for lookup-intensive code regions and a list for iteration-intensive code regions.

sorted key-value list. This is unacceptably slow. The fourth bar shows that converting the document frequencies back to a hash table for fast lookup results in performance competitive with that of the unsorted case, and often out-performs the solution with only hash tables. Yet, the output is alphabetically sorted.

6.5.5 Parallel Scalability

Using lists rather than hash tables has additional advantages for parallel execution as it is easier and more efficient to parallelize accesses to (array-based) lists. The best version with unsorted output achieves higher speedup than the hash table-only version (Fig. 46). This furthermore depends on the data set: data sets with few files observe less benefit from converting the term frequencies to lists.

The best algorithm for sorted output can achieve better speedup than the hash table-only version when the number of files is large. It performs poorly on the Gutenberg data set as the number of unique words is very large, which implies more time is spent sorting the corpus.

Table 16: TF/IDF execution time (seconds) with HPTA, Phoenix++ and SciKit-Learn. T_1 shows single-thread execution time; T_{48} is execution time for 48 threads; $S_{48} = T_1/T_{48}$.

	HPTA			Phoenix++			SciKitLearn
Data set	T_1	T_{48}	S_{48}	T_1	T_{48}	S_{48}	T_1
Various	1.8	0.5	3.7	1.9	1.2	1.7	13.4
NSF	7.5	1.7	4.4	7.7	2.7	2.9	44.5
Gutenberg	385.7	15.4	25.1	398.6	53.5	7.5	4448.0
Artificial	16.2	0.9	17.4	11.0	6.9	1.6	267.6

6.5.6 Comparison Against Single-Node Systems

We compare HPTA against Phoenix++ [92] and SciKit-Learn [74], two state-of-the-art single-node systems.

Phoenix++ [92] is a shared memory runtime system for map-reduce workloads. We have implemented TF-IDF in Phoenix++ with one map/reduce round. Each map task processes one document and produces a list of *(term, frequency, document-id)* tripples. The reduce tasks merge tripples into a list of *(document-id, TF-IDF)* pairs per term. The map task uses a hash table internally as a performance optimization. Note that there is no parallel processing of large files. This could be addressed by splitting the work over multiple map/reduce pipelines, which precludes usage of hash table within a map task and does not bring substantial added parallelism for the data sets with a large number of files.

Phoenix++ achieves limited scalability in comparison to HPTA (Fig. 47, Table 16). The "Artificial" workload is a special case as the map phase handles each document in a sequential manner. As such, speedup is limited to 5. The performance of Phoenix++ is limited due to sub-optimal handling of the three optimization opportunities identified in this work: (i) Phoenix++ uses memory mapping of input files and keeps the full files in memory throughout the execution. We have shown this is sub-optimal to region-based memory allocation. (ii) Phoenix++ limits the choices of data structures. While we have made the optimal choice for a hash table within the map task, the code is otherwise restricted to use specific intermediate containers and specific access patterns. In contrast, HPTA supports freely structured programs whereby the programmer can manipulate the resulting word–frequency map without restrictions. (iii) In order to associate word–frequency pairs with their document, Phoenix++ requires that each pair is annotated individually. In contrast, HPTA allows to associate an entire hash table to its document, which is significantly more space-efficient.

The final comparison is against SciKit-Learn, a popular single-threaded machine learning library [14]. We compare against SciKit-Learn rather than Gensim [79] because of its use of the fast NumPy library. Our setup uses Python 2.7.5, SciKit-Learn 0.17.1 and NumPy 1.7.1. HPTA is one order of magnitude faster than SciKit-Learn (Table 16). Analysis of the source code shows that SciKit-Learn is prone to the limitations addressed by HPTA. It does not switch data structures throughout the computation. Moreover, as Python is a managed language using garbage collection, memory man-

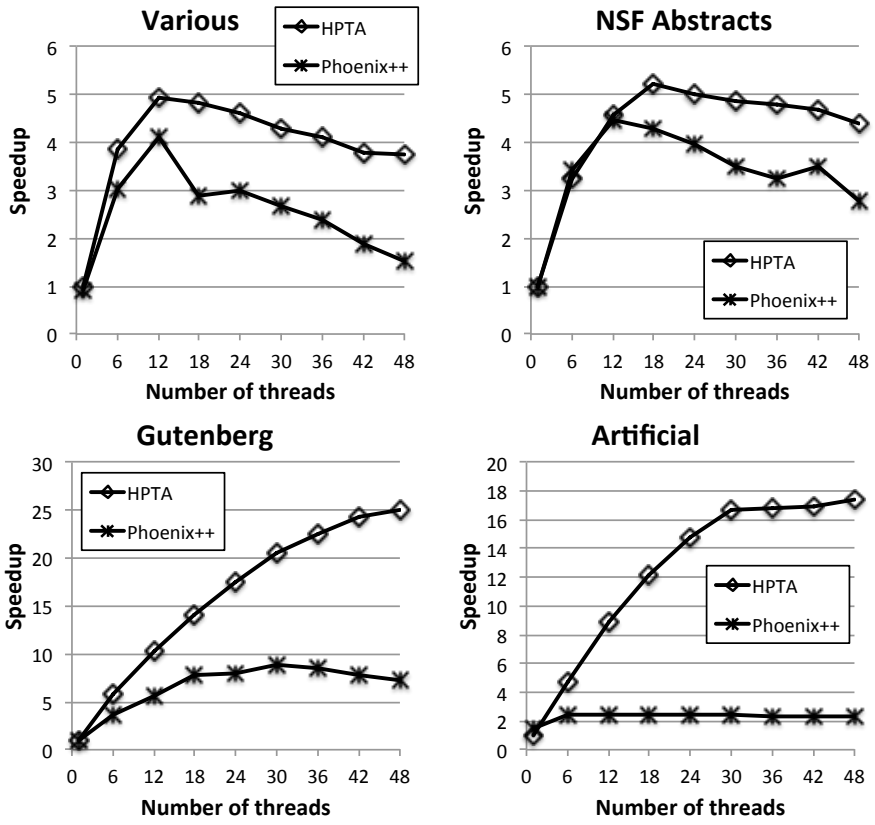


Figure 47: Parallel speedup of TF-IDF comparing the optimized solution against a map/reduce solution using Phoenix++. Speedup is normalized against HPTA.

agement is hard to control.

6.6 Conclusion

Text analytics are an important type of data analytics. We address the unexplored issue of manipulating text fragments *at high speed*, which is orthogonal to achieving speed-up by scaling-out analytics processing. The goal of this work is to formulate guidelines for optimizing text analytics and to demonstrate that they can be implemented in a reusable library. We have identified three performance optimizations: (i) region-based memory management, (ii) selection of associative data structures and (iii) transferring between associative data structures throughout the computation. We note that these optimizations are not implemented in leading data analytics platforms such as Hadoop and Spark. Our experimental evaluation however shows significant performance improvements, up to 5$\times$ for region-based memory management, up to 20$\times$ for data structure optimization and up to 19% for changing data structures during the computation.

The techniques presented significantly boost the performance of leading data an-

alytics frameworks, which will reduce hardware requirements and improve time-to-solution and energy-efficiency.

7 Memory Management in Spark

Spark [89] overcomes inefficiencies in Hadoop by creating Resilient Distributed Dataset (RDD) [107] 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. Transformations are applied on existing RDDs with operations map, filter, reduce and join. ASAP leverages Spark to overcome the limitation of iterative processing in MapReduce.

However, Spark can suffer from I/O inefficiency due to the manner in which data is read from disk. Spark reads data into an RDD from disk before it can subsequently partition the RDD to optimise further transformations of data on worker nodes. In Spark there is a limited choice of API's for reading in large datasets and unfortunately the options require that all data is read into memory at once before processing may begin even though processing of each individual text fragment is not processed until a later stage by worker nodes. It therefore results in IO blocking at the input stage which can be pronounced when the volume of data is large as is often the case with datasets of a textual nature.

Spark also suffers inefficiencies in its handling of memory for large text datasets. Documents are typically long, and there are many words in each document. Words on average are short and there exists much repititon of words across a corpus. Spark uses a managed memory system based on the Java Virtual Machine (JVM). When large datasets are read each individual text fragment must be separately allocated. This means that each time the JVM initiates a garbage collection phase all individual references to text fragments, of which there are many, must be separately examined to determine if an object is still in scope.

The upfront nature of input and overworked garbage collection can cause memory paging and workers to freeze. Spark workers sometimes detach from the master due to exceeding time-out intervals waiting for 'still alive' signals.

In this work, firstly, our aim was to implement library utilities for Spark to optimize the task of inputting large textual datasets. Secondly, we provide a memory management solution which caters to the particular demands of storing large amounts of small and replicated textual fragments which tend to come into scope at the same time and also go out of scope at the same time.

```
class ASAPWordsIterable[String](val dname: java.lang.String, val fname: java.lang.String) extends
    Iterable[java.lang.String] {
        def iterator = ASAPWordsIterator(dname, fname)
    }

class ASAPWordsIterable[String](val dname: java.lang.String, val fname: java.lang.String) extends
    Iterable[java.lang.String] {
        def iterator = ASAPWordsIterator(dname, fname)
    }

}
```

Figure 48: Iterables defined for a File and a Document.

7.1 TF/IDF

We studied Term Frequency Inverse Document Frequency (TF/IDF) as a guiding example of text analytics. It exposes important recurring properties of text analytics operations around the area of memory management and input. For a description of TF/IDF, see Section 6.2.

7.2 Opimized input/output for Spark

Spark provides *wholeTextFiles* API to enable the reading of a directory containing multiple small text files, and returns each of them as (filename, content) pairs [89]. We studied the performance slowdown for the processing of large textual datasets using the TF/IDF algorithm and observered that the call to *wholeTextFiles* took a disproportionate amount of time in relation to the overall running time of analytics queries.

To overcome the bottleneck of IO in Spark we implemented a scheme of lazy tokenization where workers tokenize terms as they progress through the documents instead of requiring that all data is read into memory upront. This was achieved in Spark by defining Scala *FileIterables* over files and a corresponding *WordIterable* over words within files (Fig. 48). For each iterable we specialised an iterator and for each iterator we specialised the *next* and *hasNext* member functions to effect the retrieval of the next file or word in the input stream (Fig. 49). The default version using *wholeTextFiles* passes a prepopulated nested list of documents and containing words to mllib's Term Frequency function. In our implementation we pass the new *FileIterable* object to Term Frequency and on this iterable *next* is called to cause the next input fragment to be read from file. This drip-feeding of data occurs in an interleaved fashing with the actual processing of fragments at the worker node site. This approach shares the work of inputting the dataset between worker nodes, avoids IO blocking and improves data locality.

```
class ASAPWordsIterator(val dname: String, val fname: String) extends Iterator[String] {

    var dirname = dname
    var filename = fname
    var fullname :String = dirname+"/"+filename

    val fileSrcMap = scala.collection.mutable.Map[String,Source]()
    var lineIterator = managedFromFile(fullname) { s => s.getLines }
    var wordIterator = lineIterator.next.split("\\s+").iterator

    def next = {
        if (!wordIterator.hasNext) {
            while ((!wordIterator.hasNext) && (lineIterator.hasNext))
                wordIterator = lineIterator.next.split("\\s+").iterator
        }
        if (wordIterator.hasNext)
            wordIterator.next
        else
            throw new Exception(Call to next on empty Word Iterator)
    }

    def hasNext = {
        if ((wordIterator.hasNext) || (lineIterator.hasNext))
            true
        else {
            fileSrcMap(fullname).close()
            false
        } } }
```

Figure 49: Iterator's specialisation of next and hasNext.

7.3 Optimized memory management for Spark

To achieve a better memory management for Spark we implemented a library for off-heap allocation of words from a corpus for TF/IDF. We used *ByteBuffers* to *mmap* the contents of large files from disk to main memory, and operate directly on *word* buffers which are effectively demarcated subsections of the global buffer. The Term Frequency associative data structure containing the words stores for each word a set of pointers to mark the start and end of the word in the global byte buffer. In this way no allocation is made on the heap. This avoids use of the JVM memory allocator and most importantly avoids expensive garbage collection passes where each allocated word has to be examined separately before determining if it can be removed. The main disadvantage of this approach is that it requires a machine to have sufficient main memory to make mapping large files from disk to memory possible. But we argue that for the purposes of selecting the best backend engine this optimization should be evaluated as an alternative materializations if the memory specifications of the machine is adequate.

In the code we achieve the mapping from file to in memory *ByteBuffer* by acquiring a read only channel from an input file strream:

```
FileInputStream.getChannel.read (globalByteBuffer)
```

7.4 Experiments

We ran benchmarks for TF/IDF on 4 datasets shown in Table 15. We evaluated the iterative and off-heap libraries against the default wholeTextFile API of Spark on a quad-socket 2.6GHz Intel Xeon E7-4860 v2, 256GB RAM, totaling 48 threads, in a shared memory setup. Results in 50 show a 13 times speedup of the off-heap library over wholeTextFiles on gutenberg20GB dataset. The speed-up achieved with off-heap over iterators can be as much as 1.2 showing that at least for some datasets off-heap is more efficient than the straight iterator library. For example, iterators achieves an 11 times speedup over wholeTextFiles on guten20GB dataset versus 13 times for off-heap. For word count a speedup of 3.5 is seen for off-heap over wholeTextFile.

The results show that drip feeding input tokens to Spark for processng, on shared memory, is worthwhile. Additionally allocating memory off-heap is also beneficial as in the results show that we can achieve a further 1.2 speedup over the iterator optimization alone.

Figure 50: Parallel speedup comparisons with iterator vs wholeTextFile input methods and off-heap memory management.

We are currently conducting similar experiments to evaluate the merits of Iterative input patterns and Off-heap memory optimization on distributed implementations of Spark.

8 Conclusion

We have described the Swan language and its features that were designed specifically for the analytics workloads that we have analyzed. Swan extends the Cilk parallel programming language with three features: data-flow dependencies, a fine-grain scheduling hint and a NUMA-aware scheduling hint. We have implemented Swan in the clang compiler, part of the LLVM toolsuite and have made it publically available. We have demonstrated that Swan out-performs state-of-the-art data analytics systems in the same space.

Several avenues for future research remain open. On the one hand, scaling out Swan is possible using the data-flow annotations. These annotations allow tasks to express the data that they require to operate on and allow the runtime system to virtualize the memory, moving data to other nodes in the data center as it see fit without programmer intervention.

Another avenue, which we have embarked upon but not completed, is to apply the observations we have made in the context of Swan to other analytics frameworks. In particular, we are investigating the optimizations identified for text analytics in the context of Spark. Our preliminary results have been reported and are promising.

References

[1] Hadoop 2015. Apache Hadoop, 2015. `http://hadoop.apache.org`.

[2] OpenMP application programming interface, version 4. 2015.

[3] V. Agarwal, F. Petrini, D. Pasetto, and D. A. Bader. Scalable graph exploration on multicore processors. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[4] M. Arif, H. Vandierendonck, D. S. Nikolopoulos, and B. de Supinski. A scalable and composable map-reduce system. In *Proceedings of Third Workshop on Advances in Software and Hardware for Big Data to Knowledge Discovery (ASH)*, page 10, December 2016.

[5] M. Arif and H. Vandierendonck. *A Case Study of OpenMP Applied to Map/Reduce-Style Computations*. In *In IWOMP'15*, pages 162–174, 2015.

[6] S. Beamer, K. Asanović, and D. Patterson. Direction-optimizing breadth-first search. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 12:1–12:10, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[7] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IX, pages 117–128, New York, NY, USA, 2000. ACM.

[8] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[9] G. E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, November 1990.

[10] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. 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, 1995.

[11] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, pages 356–368, Washington, DC, USA, 1994. IEEE Computer Society.

[12] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra. Dague: A generic distributed dag engine for high performance computing. *Parallel Comput.*, 38(1-2):37–51, January 2012.

[13] E. D. Brooks. The butterfly barrier. *International Journal of Parallel Programming*, 15(4):295–307, 1986.

[14] L. Buitinck, G. Louppe, M. Blondel, F. Pedregosa, A .Mueller, O. Grisel, V .Niculae, P. Prettenhofer, A .Gramfort, J. Grobler, R. Layton, J. VanderPlas, A .Joly, B. Holt, and G. Varoquaux. API design for machine learning software: experiences from the scikit-learn project. In *Proceedings of the ECML/PKDD Workshop: Languages for Data Mining and Machine Learning*, page 15, September 2013.

[15] International standard ISO/IEC 14882:2014(E) programming language C++, 2014.

[16] R. Chen and H .Chen. Tiled-mapreduce: Efficient and flexible mapreduce processing on multicore with tiling. *ACM Transactions on Architecture and Code Optimization (TACO)*, 10(1):3, 2013.

[17] Cilkplus documentation. `https://www.cilkplus.org`.

[18] C. Csallner, L. Fegaras, and C. 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.

[19] M. de Kruijf and K. Sankaralingam. Mapreduce for the cell broadband engine architecture. *IBM J. Res. Dev.*, 53(5):747–758, September 2009.

[20] J .Dean and S .Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[21] R. Dolbeau. Address selection for efficient barriers on the intel xeon phi. `http://www.dolbeau.name/dolbeau/publications/barrierphi.pdf`, 2014.

[22] U. Erra, S. Senatore, F. Minnella, and G. Caggianese. Approximate TFIDF based on topic extraction from massive message stream using the GPU. *Information Sciences*, 292:143 – 161, 2015.

[23] Y. Etsion, F. Cabarcas, A. Rico, A. Ramirez, R. M. Badia, E. Ayguade, J. Labarta, and M. Valero. Task superscalar: An out-of-order task pipeline. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 89–100, Washington, DC, USA, 2010. IEEE Computer Society.

[24] B. Fish, J. Kun, A. D Lelkes, L. Reyzin, and G. Turán. On the computational complexity of mapreduce. In *International Symposium on Distributed Computing*, pages 1–15. Springer, 2015.

[25] M. Frasca, K. Madduri, and P. Raghavan. Numa-aware graph mining techniques for performance and energy efficiency. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 95:1–95:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[26] M. Frigo, P. Halpern, C. E. Leiserson, and S. Lewin-Berlin. Reducers and other cilk++ hyperobjects. In *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, pages 79–90, New York, NY, USA, 2009. ACM.

[27] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, FOCS '99, pages 285–, Washington, DC, USA, 1999. IEEE Computer Society.

[28] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pages 212–223, New York, NY, USA, 1998. ACM.

[29] D. Gay and A. Aiken. Language support for regions. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, pages 70–80, New York, NY, USA, 2001. ACM.

[30] J. E Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, volume 12, page 2, 2012.

[31] S. Grimes. Unstructured data and the 80 percent rule. `http://breakthroughanalysis.com/2008/08/01/unstructured-data-and-the-80-percent-rule/`, August 2008.

[32] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 282–293, New York, NY, USA, 2002. ACM.

[33] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. Witten. The WEKA data mining software: An update. *SIGKDD Explor. Newsl.*, 11(1):10–18, November 2009.

[34] M. Han, K. Daudjee, K. Ammar, M. T. Özsu, X. Wang, and T. Jin. An experimental comparison of Pregel-like graph processing systems. *Proc. VLDB Endow.*, 7(12):1047–1058, August 2014.

[35] Y. He, C. E. Leiserson, and W. M. Leiserson. The cilkview scalability analyzer. In *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, pages 145–156, New York, NY, USA, 2010. ACM.

[36] D. Hensgen, R. Finkel, and U. Manber. Two algorithms for barrier synchronization. *Int. J. Parallel Program.*, 17(1):1–17, February 1988.

[37] S. Hong, T. Oguntebi, and K. Olukotun. Efficient parallel graph exploration on multi-core cpu and gpu. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 78–88. IEEE, 2011.

[38] H. Inoue, H. Komatsu, and T. Nakatani. A study of memory management for web-based applications on multicore processors. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 386–396, New York, NY, USA, 2009. ACM.

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

[40] Intel Corporation. *Intel Cilk Plus Application Binary Interface Specification*, 2011. Revision 1.1. Document number 324512-002US.

[41] W. Jiang and G. Agrawal. Ex-mate: data intensive computing with large reduction objects and its application to graph mining. In *Proceedings of the 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 475–484. IEEE Computer Society, 2011.

[42] W. Jiang, V. T. Ravi, and G. Agrawal. A map-reduce system with an alternate api for multi-core environments. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, CCGRID '10, pages 84–93, Washington, DC, USA, 2010. IEEE Computer Society.

[43] E. Johnson and D. Gannon. HPC++: Experiments with the parallel standard template library. In *Proceedings of the 11th International Conference on Supercomputing*, ICS '97, pages 124–131, New York, NY, USA, 1997. ACM.

[44] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48(1):96 – 129, 1998.

[45] D. E. Knuth. *The Art of Computer Programming Volume 1: Fundamental Algorithms*. Addison Wesley, 1997.

[46] D. E. Knuth. *The Art of Computer Programming Volume 3: Sorting and Searching*. Addison Wesley, 1998.

[47] S. Kumar, C. J. Hughes, and A. Nguyen. Carbon: Architectural support for fine-grained parallelism on chip multiprocessors. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 162–173, New York, NY, USA, 2007. ACM.

[48] H. Kwak, C. Lee, H. Park, and S. Moon. What is twitter, a social network or a news media? In *Proceedings of the 19th international conference on World wide web*, pages 591–600. ACM, 2010.

[49] A. Kyrola, G. Blelloch, and C. Guestrin. GraphChi: Large-scale graph computation on just a pc. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 31–46, Hollywood, CA, 2012. USENIX.

[50] S. Lattanzi, B. Moseley, S. Suri, and S. 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.

[51] I. T.A., S. Boyd-Wickizer, Z. Huang, and C. E. Leiserson. Using memory mapping to support cactus stacks in work-stealing runtime systems. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 411–420, New York, NY, USA, 2010. ACM.

[52] C. E. Leiserson and T. 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.

[53] C. E. Leiserson, T. B. Schardl, and J. Sukha. Deterministic parallel random-number generation for dynamic-multithreading platforms. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 193–204, New York, NY, USA, 2012. ACM.

[54] J. Leskovec, A. Rajaraman, and J. D. Ullman. *Mining of massive datasets*. Cambridge University Press, 2014.

[55] J. Lin and M. 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.

[56] R. Liu and H. Chen. Ssmalloc: A low-latency, locality-conscious memory allocator with stable performance scalability. In *Proceedings of the Asia-Pacific Workshop on Systems*, APSYS '12, pages 15:1–15:6, New York, NY, USA, 2012. ACM.

[57] M. Lu, L. Zhang, H. P. Huynh, Z. Ong, Y. Liang, B. He, R. S. M. Goh, and R. Huynh. Optimizing the mapreduce framework on intel xeon phi coprocessor. In *2013 IEEE International Conference on Big Data*, pages 125–130, 2013.

[58] B. D. Lubachevsky. Synchronization barrier and related tools for shared memory parallel programming. *Int. J. Parallel Program.*, 19(3):225–250, March 1991.

[59] 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.

[60] D. Margo and M. Seltzer. A scalable distributed graph partitioner. *Proc. VLDB Endow.*, 8(12):1478–1489, August 2015.

[61] J. D. McCalpin. STREAM: Sustainable memory bandwidth in high performance computers. Technical report, University of Virginia, 1995. A continually updated technical report. `http://www.cs.virginia.edu/stream/`.

[62] F. McSherry, M. Isard, and D. G. Murray. Scalability! but at what COST? In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, Kartause Ittingen, Switzerland, May 2015. USENIX Association.

[63] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9:21–65, February 1991.

[64] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee. Measurement and Analysis of Online Social Networks. In *Proceedings of the 5th ACM/Usenix Internet Measurement Conference (IMC'07)*, San Diego, CA, October 2007.

[65] and C. G. Plaxton N S. Arora, R. D. Blumofe. *Thread Scheduling for Multiprogrammed Multiprocessors*. In *In SPAA '98*, pages 119–129, 1998.

[66] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM, 2007.

[67] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 456–471, New York, NY, USA, 2013. ACM.

[68] OpenMP application programming interface, version 4.5. http://www.openmp.org/, November 2015.

[69] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun. Making sense of performance in data analytics frameworks. In *Proc. of the 12th USENIX Conf. on Networked Systems Design and Implementation*, NSDI'15, pages 293–307, 2015.

[70] S. Ghemawat and P. Menage. Tcmalloc: Thread-caching malloc, 2015.

[71] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120.

[72] V. Papaefstathiou, M. G.H. Katevenis, D. S. Nikolopoulos, and D. 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.

[73] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *Proc. of the 2009 ACM SIGMOD Intl. Conf. on Management of Data*, SIGMOD '09, pages 165–178, 2009.

[74] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and É. Duchesnay. Scikit-learn: Machine learning in python. *J. Mach. Learn. Res.*, 12:2825–2830, November 2011.

[75] R. Rangan, N. Vachharajani, M. Vachharajani, and D. I. August. Decoupled software pipelining with the synchronization array. In *PACT '04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 177–188, 2004.

[76] 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.

[77] L. Rauchwerger, F. Arzu, and K. Ouchi. Standard templates adaptive parallel library (STAPL). In *LCR '98: Selected Papers from the 4th International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, pages 402–409, 1998.

[78] J. W. Reed, Y. Jiao, T. E. Potok, B. A. Klump, M. T. Elmore, and A. R. Hurson. TF-ICF: A new term weighting scheme for clustering dynamic data streams. In *5th International Conference on Machine Learning and Applications (ICMLA'06)*, pages 258–263, Dec 2006.

[79] R. Rehurek and P. Sojka. Software framework for topic modelling with large corpora. In *The LREC 2010 Workshop on new challenges for NLP frameworks*, pages 45–50. University of Malta, 2010.

[80] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 472–488. ACM, 2013.

[81] Y. Saad. SPARSKIT: A basic tool for sparse matrix computations. Technical Report NASA-CR-185876, NASA, May 1990.

[82] G. Salton and M. J. McGill, editors. *Introduction to Modern Information Retrieval*. Mcgraw-Hill, 1983.

[83] D. Sanchez, R. M. Yoo, and C. Kozyrakis. Flexible architectural support for fine-grain scheduling. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 311–322, New York, NY, USA, 2010. ACM.

[84] N. Satish, N. Sundaram, Md. M. A. Patwary, J. Seo, J. Park, M. A. Hassaan, S. Sengupta, Z. Yin, and P. Dubey. Navigating the maze of graph analytics frameworks using massive graph datasets. In *Proc. of the 2014 ACM SIGMOD Intl. Conf. on Management of Data*, SIGMOD '14, pages 979–990, 2014.

[85] D. Shabalin and M. Odersky. Region-based off-heap memory for Scala. Technical report, École Polytechnique Fédérale de Lausanne, February 2015.

Available as https://infoscience.epfl.ch/record/213469/files/Regions%20report.pdf.

[86] J. Shun and G. 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.

[87] J. Shun, L. Dhulipala, and G. E Blelloch. Smaller and faster: Parallel processing of compressed graphs with ligra+. In *Data Compression Conference (DCC), 2015*, pages 403–412. IEEE, 2015.

[88] A. Sodani, R. Gramunt, J. Corbal, H. S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y. C. Liu. Knights landing: Second-generation intel xeon phi product. *IEEE Micro*, 36(2):34–46, 2016.

[89] Apache spark. http://spark.apache.org, 2016.

[90] C. Stancu, C. Wimmer, S. Brunthaler, P. Larsen, and M. Franz. Safe and efficient hybrid memory management for Java. In *Proceedings of the 2015 International Symposium on Memory Management*, ISMM '15, pages 81–92, New York, NY, USA, 2015. ACM.

[91] M. J. Szaszy and H. Samet. Document stream clustering using GPUs. Available at http://wwwold.cs.umd.edu/Grad/scholarlypapers/papers/Szaszy.pdf, 2013.

[92] 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.

[93] G. Tanase, C. Raman, M. Bianco, N. M. Amato, and L. Rauchwerger. Languages and compilers for parallel computing. chapter Associative Parallel Containers in STAPL, pages 156–171. Springer-Verlag, Berlin, Heidelberg, 2008.

[94] NSF research awards abstracts 1990–2003. archive.ics.uci.edu/ml/machine-learning-databases/nsfabs-mld/nsfawards.html, November 2003. Consulted: February 2016.

[95] H. Vandierendonck. Efficiently scheduling task dataflow parallelism: A comparison between swan and quark. In *Exascale Applications and Software Conference (EASC)*, page 6, April 2015.

[96] H. Vandierendonck, K. Murphy, M. Arif, and D. S. Nikolopoulos. HPTA: High-performance text analytics. In *Proceedings of the IEEE International Conference on Big Data*, page 8, December 2016.

[97] H. Vandierendonck, G. Tzenakis, and D. S. Nikolopoulos. A unified scheduler for recursive and task dataflow parallelism. In *PACT '11: Proceedings of the 20th international conference on Parallel architectures and compilation techniques*, pages 1–11, Washington, DC, USA, October 2011. IEEE Computer Society.

[98] H. Vandierendonck, G. Tzenakis, and D. S. Nikolopoulos. Analysis of dependence tracking algorithms for task dataflow execution. *ACM Trans. Archit. Code Optim.*, 10(4):61:1–61:24, December 2013.

[99] M. Varshney and V. Goudar. PDQCollections: A data-parallel programming model and library for associative containers. Technical Report 130004, Computer Science Department, Universtiy of California, Los Angeles, April 2013.

[100] Y. Vigfusson. *Affinity in distributed systems*. PhD thesis, Cornell University, 2010.

[101] T. Xiao, J. Zhang, H. Zhou, Z. Guo, S. McDirmid, W. Lin, W. Chen, and L. 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*, pages 44–53. ACM, 2014.

[102] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica. GraphX: A resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems*, GRADES '13, pages 2:1–2:6, New York, NY, USA, 2013. ACM.

[103] J. Yang and J. Leskovec. Defining and evaluating network communities based on ground-truth. *CoRR*, abs/1205.6233, 2012.

[104] F. Yergeau. *UTF-8, a transformation format of ISO 10646*. The Internet Society, November 2003. RFC-3629.

[105] 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.

[106] K. Yotov, T. Roeder, K. Pingali, J. Gunnels, and F. Gustavson. An experimental comparison of cache-oblivious and cache-conscious programs. In *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '07, pages 93–104, New York, NY, USA, 2007. ACM.

[107] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. 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'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[108] K. Zhang, R. Chen, and H. Chen. NUMA-aware graph-structured analytics. In *Proc. of the 20th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 183–193, 2015.

[109] Y. Zhang, F. Mueller, X. Cui, and T. Potok. Large-scale multi-dimensional document clustering on GPU clusters. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–10, April 2010.

[110] Y. Zhang, F. Mueller, X. Cui, and T. Potok. Data-intensive document clustering on graphics processing unit (GPU) clusters. *J. Parallel Distrib. Comput.*, 71(2):211–224, February 2011.

FP7 Project ASAP
Adaptable Scalable Analytics Platform

End of ASAP D2.3

Program analysis and transformation

WP 2 – A Unified Analytics Programming Model

Nature: Report

Dissemination: Public