# Group Communication Patterns for High Performance Computing in Scala

Felix P. Hargreaves Daniel Merkle Peter Schneider-Kamp

Department of Mathematics and Computer Science, University of Southern Denmark

{hargreaves,daniel,petersk}@imada.sdu.dk

## Abstract

We developed a Functional Object-Oriented PARallel framework (FOOPAR) for high-level high-performance computing in Scala. Central to this framework are Distributed Memory Parallel Data structures (DPDs), i.e., collections of data distributed in a shared nothing system together with parallel operations on these data.

In this paper, we first present FOOPAR's architecture and the idea of DPDs and group communications. Then, we show how DPDs can be implemented elegantly and efficiently in Scala based on the *Traversable/Builder* pattern, unifying Functional and Object-Oriented Programming.

We prove the correctness and safety of one communication algorithm and show how specification testing (via ScalaCheck) can be used to bridge the gap between proof and implementation. Furthermore, we show that the group communication operations of FOOPAR outperform those of the MPJ Express open source MPI-bindings for Java, both asymptotically and empirically.

FOOPAR has already been shown to be capable of achieving close-to-optimal performance for dense matrix-matrix multiplication via JNI. In this article, we present results on a parallel implementation of the Floyd-Warshall algorithm in FOOPAR, achieving more than 94% efficiency compared to the serial version on a cluster using 100 cores for matrices of dimension 38000 × 38000.

***Categories and Subject Descriptors*** D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel Programming; D.3.3 [*Programming Languages*]: Language Constructs and Features—Frameworks

***General Terms*** Performance, Design, Languages

***Keywords*** High-Performance-Computing, Scala

## 1. Introduction

Building scalable systems in the presence of several thousands, even millions, of computational cores asks for new programming paradigms in order to meet obvious goals such as performance exploitation, correctness, portability, fault tolerance, and usability. It is well foreseeable that the programming model underlying the Message Passing Interface (MPI), the current de-facto standard for programming distributed memory systems, will not be able to address the challenges for the necessary next steps in High Performance Computing (HPC). Reasons for this include that MPI ignores memory hierarchies and that reaching a high productivity (i.e., performance, expressivity, portability, and robustness) is challenging based on the error-prone low-level programming model [3].

It is no surprise that new programming languages for simplifying the programming on peta- and exascale parallel systems are currently developed and studied. IBM's X10 [17], Cray's Chapel [5], and Sun's Fortress [2] are three prominent examples, that are based on the partitioned global address space (PGAS) programming model. While PGAS aims at combining the advantages of using an SPMD programming model on distributed memory systems with the advantages of using referencing semantics of shared memory systems, this approach also comes with some drawbacks. To reach high performance users still need to specify and reason about the placement of data and tasks, e.g., in Chapel a *local* type is used for that and in X10 user-defined *places* encapsulate binding of activities and globally addressable data. Also, proving correctness or analyzing scalability is usually not considered a prominent design goal.

A promising approach for HPC comes from Delite [24], a framework and runtime for the definition of high performing DSLs, encompassing techniques such as loop-fusion, common subexpression elimination, term rewriting, modular code emitters (support for SIMD instructions), automatic parallelization of expressions based on analysis of side-effects, and much more. The syntactical part of the DSLs then relies on implementations using *lightweight modular staging* [25], an advanced design pattern utilizing a very minimal compiler plugin for Scala. This approach solves many problems, but is ultimately not analyzable from the end user perspective, i.e., it can become difficult to gauge the running time of programs written in these DSLs. Additionally, while this approach is by far more productive than building external DSLs from scratch, it requires a big skill set from the DSL author.

While the goal of our work is also a high productivity approach for HPC, our contribution is orthogonal to the aforementioned approaches. In this paper we describe FOOPAR, a parallel framework which is based on the functional object-oriented programming language Scala [20]. Functional languages have at most played a niche role in HPC, often both due to acceptance problems[29] and due to an inability to perform on-par with hand-optimized C code close to the theoretical limits of the hardware. FOOPAR heavily employs distributed memory data structures especially suited for HPC, enabling it to achieve excellent performance. It offers a balance between conciseness and performance, achieving scalability and efficiency close to that of hand-optimized C code from a handful of lines of Scala code combined with native interface computational kernels (cf. Figure 1, data from [12]). However, FOOPAR does not only allow for high performance from an empirical point of view, but also allows for a theoretical scalability analysis. In addition, our framework allows for correctness proofs of parallel code as well as specification testing [19].

The paper is structured in the following way. Section 2 describes the rationale behind the framework and compares it with further related work. Then, Section 3 introduces the concept of Distributed memory Parallel Data structures (DPDs) and their associated group communication operations including information about parallel runtimes useful for analysing code written in FOOPAR. The

| #cores | $k$ | optimized C | FOOPAR |
|---|---|---|---|
| 216 | 30240 | 26.72 | 27.07 |
| 343 | 30240 | 17.27 | 17.58 |
| 512 | 30240 | 11.50 | 12.51 |
| 512 | 40000 | 25.21 | 26.40 |

Figure 1: *Runtimes in seconds for $k \times k$ matrix-matrix multiplication using* **FOOPAR** *with Open MPI Java bindings and* ***optimized C*** *with native Open MPI on the Carver cluster at NERSC [1]. For $k = 40000$,* FOOPAR *reaches 4.84 TFLOP/s, i.e., 88.8% efficiency w.r.t. the theoretical peak performance.*

implementation of these data structures based on Scala's functional object-oriented features is presented in Section 4. In Section 5 we demonstrate how correctness can be proven for the algorithms used in FOOPAR and how specification testing can be used to ensure reliability of their implementation. In Section 6 we provide empirical evidence for the efficiency and scalability of our framework. Finally, we conclude briefly in Section 7.

## 2. The FOOPAR Framework

The main goal of FOOPAR is to avoid common challenges of distributed memory parallel programming and High Performance Computing (HPC) through the use of high-level abstractions while maintaining analyzability:

- Using functional programming concepts, the *Single Program Multiple Data* (SPMD) [8] concept can be combined with *Single Instruction Multiple Data (SIMD)* at a data structure level. This allows algorithms to be formulated in virtually the same way as their serial versions (see Example 1 below).
- We abstract away peer-to-peer message passing by introducing a set of group communication operations appropriate for HPC use. In this way, we can avoid deadlocks, starvation, race conditions, and other common concurrency issues.
- We avoid the many pitfalls of manual memory management through the use of a managed programming language running on top of the Java virtual machine. In addition, this provides platform independence.

### 2.1 Design and Related Work

While FOOPAR complements the *parallel collections* [22] introduced in Scala 2.8, it is not meant as an extension. This is due to multiple reasons. First, the parallel collections use workload-splitting strategies leading to communication bottle-necks in distributed memory settings. Second, they employ an implicit master-slave paradigm unsuitable for massively distributed HPC. Third, the SPMD paradigm requires launching multiple copies of the process as opposed to branching internally into threads.

FOOPAR differs from other functional programming frameworks for parallel computations in some key aspects. While frameworks like Eden [15], Spark [30], and Scala's own parallel collections [22] try to maximize the level of abstraction, this is mostly done through strategies for data-partitioning and distribution which in turn introduce network and computation bottlenecks. Furthermore, these tools lend themselves poorly to parallel runtime analysis hindering asymptotic guarantees that might otherwise be achieved. To unaware users, "automagic" parallel programming can easily lead to decreased performance due to added overhead and small workloads. With this in mind, FOOPAR aims at the sweetspot between high performance computing and highly abstract, maintainable and analyzable programming. This is achieved by focusing on user-defined workload distribution and deemphasizing fault tolerance. In this way, the performance pitfalls of both *dynamic workload allocation* and the *master-slave* paradigm can be avoided and FOOPAR can provide HPC parallelism with the conciseness, efficiency and generality expected from mature Scala libraries, while nicely complementing the existing parallel collections of Scala's standard API for shared memory use.

While Scala's parallel collections are limited to shared memory systems, FOOPAR works both in shared nothing as well as shared memory architectures. Taking some inspiration from MPI [9], FOOPAR implements most of the essential operations found in MPI in a more convenient and abstract level as well as expanding upon them. As an example, FOOPAR supports reductions with arbitrary types and variable sizes, e.g. reduction by list or string concatenation is entirely possible and convenient in FOOPAR (however inherently unscalable). As an addition, performance impact from the use of concatenation or other size-increasing operations is directly visible through the provided asymptotic runtime analysis for operations on the Distributed Memory Parallel Data Structures (cf. Section 3).

FOOPAR shares goals with the partitioned global address space (PGAS) programming model in the sense that the reference semantics of shared memory systems is combined with the SPMD style of programming. Prominent examples of PGAS are Unified Parallel C, or Co-Array Fortran among others [7]. Focusing on performance and programmability for next-generation architectures, novel languages like X10 and Chapel provide richer execution frameworks and also allow asynchronous creation of tasks [16]. All these languages either resemble and extend existing languages or are designed from scratch; their features are usually accessed via syntactic sugar. FOOPAR, in contrast, is more oriented towards abstraction by employing distributed data structures and combining this with the mathematical abstraction inherently integrated in functional languages like Scala. This approach is somewhat similar to that of STAPL [4], however, the combination with functional programming has the potential to be more productive and produce more analyzable code.

Finally, comparing to frameworks based on Multiple Program Multiple Data (MPMD), the SPMD paradigm used in FOOPAR emphasizes rank-data mapping, where ranks are the IDs of the processing elements in an execution. Section 3 shows how rank-data mappings play a major role in FOOPAR and how it can abstract over serial and parallel programming.

### 2.2 Communication Groups

Instead of explicit message passing, we use the notion of *groups*, a collection of *processes* with a set ofoup communication operations. We view the instantiation of a DPD of type $T$ as a projection from the set of all processing elements, $P$, defined as $P \xrightarrow{\text{subgroup}} P \xrightarrow{\text{data}} T$. Subgroup mappings allow for communication algorithms to work independently on a multitude of topologies and sets of processing elements without cluttering the implementations with special cases. Figure 2 shows a parallel reduction algorithm following the pattern of the *recursive doubling algorithm* [6]. Here, associativity of the reduction operator is assumed. This group communication pattern can be directly mapped to the reduction operation on a DPD as shown in Section 4. Using FOOPAR, complete parallel computations can be described concisely as a chain of such operations indirectly invoked through operations on DPDs.

### 2.3 Architecture

Figure 3 shows the three layers of FOOPAR's architecture: on top the DPDs, in the middle the group communication operations, and on the bottom the backends that abstract from network and communication specifics.

At the beginning of an execution, to facilitate this abstraction, a *provider* of communication groups and processing elements is chosen and instantiated. In this way, both industry-proven libraries,

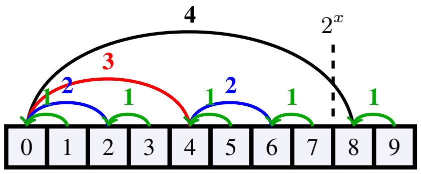

Figure 2: *10-node reduction by inverse recursive doubling; numbers on the edges indicate the time step of the reduction.*

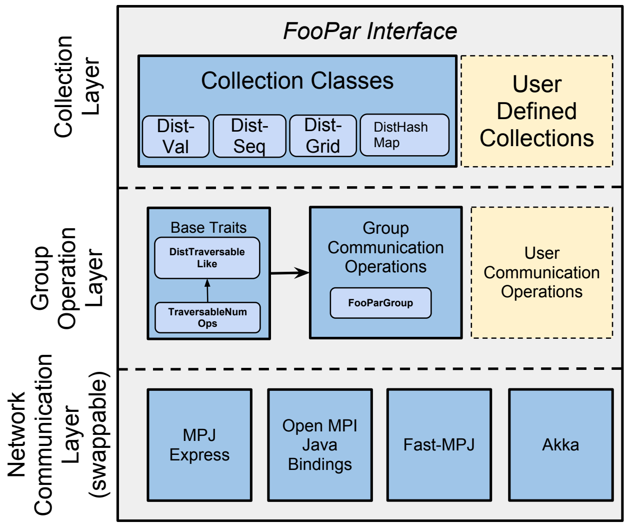

Figure 3: FOOPAR*'s layered architecture*

such as *MPI* [9, 26, 27], and an abstract implementation provided by FOOPAR with arbitrary network backends (e.g. *Akka* [23]) can be used. The communication patterns work across multiple communication backends due to the *process* interface, which abstracts the message-passing functionality.

Finally, the user is presented with an SPMD programming model using DPDs with SIMD-like [8] operations. This combination provides a powerful abstraction of parallelism while maintaining enough control for writing efficient and analyzable programs.

**Example 1.** *Consider the following* embarrassingly parallel *algorithm for approximating the transcendental constant $\pi$ with arbitrary precision, here expressed as a Scala function approximating the integral $\int_0^1 \frac{4}{1+x^2}dx$.*

```
def pi(n: Int) = {
  val f = (x: Double) => 4d / (1d + x * x)
  val ff = (x: Int) => (x - 0.5d) / n
  (1 to n).map(f compose ff).sum / n
}
```

*Line 1 defines a function* `pi` *taking an integer (the number of "samples" from the integral), Line 2 defines the function* `f` *to integrate, and Line 3 defines the function* `ff` *generating the sample positions. Line 4 uses a Scala built-in to create the list* `[1,2,...,n]`, *maps the composition of* `f` *and* `ff` *to each element of this list, and reduces the resulting list to its average, which is returned by* `pi` *as its value.*

*In order to obtain a parallel version, we simply replace Scala's built-in lists in Line 4 with* FOOPAR*'s DPD for sequences,* DistSeq, *and use the DPD methods* mapD *and* avgD *which correspond to* map *and* sum / n, *respectively:*

```
def pi(n: Int) = {
  val f = (x: Double) => 4d / (1d + x * x)
  val ff = (x: Int) => (x - 0.5d) / n
  DistSeq.ranged(1 to n).mapD(f compose ff).avgD
}
```

*The sequential runtime $T_S$ is obviously in $\Theta(n)$.*

*For the parallel runtime $T_P$, assuming $p$ processing elements, we*

*have $T_P \in \Theta(n/p + T_C)$ where $T_C$ is the communication time.*

*A naive implementation of the sum would require linear communication overhead (i.e., $T_C \in \Theta(p \cdot (t_s + t_w \cdot m))$) where $t_s$ is the communication start-up time, $t_w$ is the per-word transfer time, and $m$ is the message size in words (alpha-beta cost model in the notation of [10]).*

*Using the reduction based on inverse recursive doubling (cf. Figure 2), we obtain $T_C \in \Theta(\log p \cdot (t_s + t_w \cdot m))$ and, thus, $T_P \in \Theta(n/p + \log p \cdot (t_s + t_w \cdot m))$. If our parameter $n$ is at least $p \cdot \log p$, then the* cost *$pT_p$, is in $\Theta(n)$ and we call our parallel algorithm* cost optimal. *In other words, if each processing element gets to compute at least $\log p$ elements of the sum, the overhead of the parallel version does not dominate the serial runtime asymptotically and therefore all processing elements can be used efficiently.*

The above example also demonstrates the importance of efficient communication operations. This is the focus of the next section, where we introduce our parallel data structures and

the computation and communication operations on them.

## 3. Distributed Memory Parallel Data Structures

At the heart of FOOPAR lie the Distributed Memory Parallel Data Structures (DPDs). The DPDs provide data-abstractions, but they rely on the user to define partitions of the data to processing elements either directly through rank-data mappings or indirectly through lazy data wrappers. The former case can be understood as a mapping of processing elements to the data they store locally. To understand the latter, consider the situation where a user might define a two-dimensional grid representing a large matrix. Since this structure would probably not fit in the memory available for a single processing element, each entry is wrapped in a lazy container. Scala has language support for this and thus writing a lazy matrix-class can be achieved simply by adding the *lazy* keyword in front of the actual data field. In this way, each processing element will *inflate* only the required matrices at execution. While user-defined partitioning provides a lower level of abstraction than a dynamic workload balancing scheme would, it provides a good balance between abstraction and analyzability. Alternatively, one of the available DPDs can be used to conveniently map into data partitions, e.g. for a given array `a`, we could use `(0 to a.length-1).toDistSeq.mapD(a)`, where `a` is used as a partial function.

Once a DPD is instantiated, algorithms can be implemented directly through chains of parallel transformations and communication methods directly on the DPD. As usual in functional programming, in FOOPAR, DPDs are treated as immutable data structures and, consequently, all operations on DPDs return new data structures. FOOPAR currently offers four DPDs: Distributed Values, Distributed Sequences, Distributed Grids, and Distributed HashMaps. Due to the modularity of the communication operations in FOOPAR, the collections can easily be extended and expanded via user defined DPDs. Such new distributed collections can either make use of the built-in group communications or obtain direct peer-to-peer message passing capabilities (at the price of potential deadlocks, etc.). In this paper, we focus on the Distributed Sequence DPD as the most common case. We also briefly introduce the Distributed Values DPD and the Distributed Grid DPD.

A constant or variable is the simplest form of a data structure, as it is unstructured. FooPar supports unstructured parallel data in the form of the **Distributed Value DPD**. A Distributed Value is shared among all processing elements of some communication group. As there is a local value for each processing element, typical sequence operations like `map`, `reduce` etc. can still be used, albeit without any sense of order or of sequence length. In other words, all processing elements are participating in all operations and the reduction operators have to also be commutative

For Distributed Values, the processing element's rank is mapped to the local data given as an argument at the point of instantiation. While this mapping is trivial, it is still useful for operations that include all processing elements in an execution.

**Example 2.** *Agreeing on a dynamic random seed can be done in an unstructured way by taking the current unix time at each node and agreeing and communicating the minimum of all these timestamps, thereby agreeing on a global seed.*

```
val now: Long = System.currentTimeMillis
val min: Option[Long] = DistVal(now).allMinD
val rnd: Option[Random] = min.map(new Random(_))
```

*Here,* `allMinD` *computes the minimum of the values in all processing elements and broadcasts it to all processing elements in $\Theta(\log p)$. Alternatively, one could use the indexing method* `apply(0)` *to get the current time from some arbitrary processing element with index $0$. It is even possible to share the random number generator itself. The reason for the type* `Option[Long]` *is that for other DPDs, not all processing elements necessarily take part in this computation. At the end, all processing elements have access to a random generator initialized with the global seed. We use the suffix* `D` *on parallel methods of DPDs in order to clearly distinguish between sequential and parallel operations. This is relevant especially since* `Option` *supports similar higher order methods, e.g.* `dseq.reduceD(_+_).map(_+10)` *would be a valid expression using a map on the* `Option` *result.*

The **Distributed Sequence DPD** distributes a sequence of a certain length to processing elements. In other words, a distributed sequence is a one-to-one mapping of indices to ranks, i.e., an index $i$ of the sequence is mapped to the numerical identifier of the processing element that stores the $i$-th element of the sequence.

Such a rank-data mapping can be generated from any existing indexed Scala sequence using `toDistSeq` as exemplified in the first paragraph of this section or from a symbolic range such as `(1 to n)` as seen in Example 1. While creating distributed sequences from indexed sequences require each processing element to contain the entire sequence, in combination with lazy data elements, this restriction is lifted and actually turned into a powerful advantage. Consider for example a Distributed Sequence DPD containing lazy references to matrices. The runtime overhead of dealing with this symbolic indexed sequence will be dominated by the matrix-matrix multiplication used to reduce the sequence.

Operations on the Distributed Sequence DPD are parallel versions of the typical sequence operations such as `map`, `reduce` etc. Figure 5 shows how the SIMD principle is mimicked in a parallel map operation. In contrast to the Distributed Values DPD, order and length of the sequence are respected and only associativity is required for reduction operators.

The **Distributed Grid DPD** offers a more involved rank-data mapping than the other DPDs, the motivation being a lack of efficient means for nested traversals of distributed sequences representing multidimensional data. It offers an efficient rank-data mapping while providing convenience methods for advanced communication patterns like subdimension partitioning. The mapping itself is a generalization of positional notation for integers called *mixed radix* with the addition of a possible transposition. In a nutshell, integers can be converted to and from a list of subindices for the individual dimensions. Section 6.2 shows how to use this to model a distributed matrix in a Floyd-Warshall parallelization.

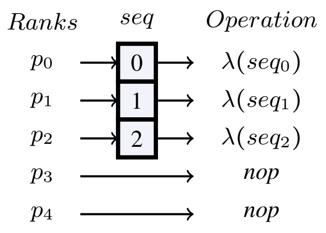

Figure 5: *Example of SIMD principle in mapD method invocation on Distributed Sequence of size 3.*

## 3.1 Group Communication Algorithms

Central to the DPD operations of FooPar are the *group communication algorithms*. These operations serve as the basis for *all* network communication in FooPar, but can be viewed as a completely encapsulated module. All the operations work via *asynchronous message passing*, i.e., nonblocking sends coupled with blocking receives. FooPar introduces the notion of `FooPar Processes`, abstract processes which encapsulate *integer ranks* as well as *send* and *receive* operations, and, thus, can be treated as abstract processing elements in group communication algorithms.

FooPar adopts a topology-oblivious approach to the distribution of processing elements in groups, i.e., tasks are currently distributed in a round-robin fashion with no regard to cache coherence. Section 6 shows that this is efficient in practice. However, FooPar's design allows for future topology-aware heuristics to improve distribution of processing elements to optimize communication.

The group communication algorithms center around communication via message passing. Non-blocking sends and blocking receives are made possible through the process interface. We define the methods as `send(destination, message)` and `receive(origin)` where `destination` and `origin` are local ranks within a communication group. This level of abstraction allows enough flexibility for a vast amount of algorithms to be implemented with no regard for actual network implementation or topology.

Arguably, the most trivial group communication operation is a **Circular Shift** by $d$ elements for a sequence of length $n$, where each process with rank $r$ sends its element to $(r + d) \bmod n$ and receives an element from $(r - d) \bmod n$.

In Figure 2 we have seen how to use inverse recursive doubling to implement the **Reduction** group communication operation for associative reduction operators. This operation is a special case of the higher-order *fold* algorithm without an identity element. The typical sequential fold comes in ordered left-to-right (`foldLeft`), right-to-left (`foldRight`) or unordered (`fold`). To support these, data is required to be a *monoid* structure, i.e., a semi-group with zero element. While the sequential fold imposes more restrictions on the structure of the data, it offers more flexibility by supporting ordered folds on empty sequences as well as supporting arbitrary typed zero elements. While left- and right-associative folds are

| **Operation** | **Parallel Running Time** $T_P$ | **Communication** |
|---|---|---|
| `mapD` $\lambda$, `foreach` $\lambda$ | $\Theta(T_\lambda)$ | None |
| `apply` $i$ | $\Theta((t_s + t_w \cdot m) \log p)$ | Broadcast |
| `reduceD` $\lambda$ | $\Theta((t_s + t_w \cdot m + T_\lambda) \log p)$ | Reduce |
| `scan1D` $\lambda$ | $\Theta((t_s + t_w \cdot m + T_\lambda) \log p)$ | Prefix Sum |
| `shiftD` $d$ | $\Theta(t_s + t_w \cdot m)$ | Circular Shift |
| `sumD`, `productD`, `minD`, `maxD`, `avgD` | $\Theta((t_s + t_w \cdot m) \log p)$ | Reduce |
| `allSumD`, `allProductD`, `allMinD`, `allMaxD`, `allAvgD` | $\Theta((t_s + t_w \cdot m) \log p)$ | All-Reduce |

Figure 4: *Table over operations supported by* FooPar *DPDs.* $m$ *is message size in words,* $p$ *is # of processing elements*

inherently sequential, the reduction operation can be parallelized as discussed before.

Finally, the elementary group communication **Broadcasting** which communicates a value from one processing element to all others, can be performed by recursive doubling, i.e., the inverse of Figure 2, in $\Theta(\log p)$.

To profit from the group communication patterns, high level data structure operations are mapped to group operations. Scala sequences support second-order operations like for example `map` and `reduce` as well as element retrieval using the indexing operator `apply`. Analogously, FooPar offers variants of these on distributed sequences. The remainder of this section describes how and at what (parallel) running time cost `reduceD`, `mapD`, and `apply` can be implemented.
For performing computations on the local data of the processing elements, we do not need communication, but we need to apply a function to each element of the distributed sequence. A `mapD` with a function $\lambda$ of running time $T_\lambda$ can be performed independently in parallel, yielding a parallel running time of $\Theta(T_\lambda)$, as opposed to $\Theta(n \cdot T_\lambda)$ in the serial case. Here, we assume $n \leq p$, i.e., the existence of sufficiently many processing elements. This assumption works because each element in the DPD can represent partitioned data.

The indexing operation, `apply`, needs communication. If each process needs a copy of the $i$-th element, a Broadcast communication operation based on recursive doubling can be used to perform this operation in $\Theta((t_s + t_w \cdot m) \log p)$ parallel time. Here, $t_s$ and $t_w$ are the message startup time and per-word transfer time respectively. Similarly, `reduceD` can be performed in $\Theta((t_s + t_w \cdot m + T_\lambda) \log p)$ for any reduction operation $\lambda$.

For further operations, consult Figure 4. There, two more communication operations are used. All-Reduce is just like Reduce, except that all process elements receive the result of the reduction, while Prefix Sum is a reduction where each processing element receives the partial reduction result up to its index. We use the versions from [10], except that we correct an obvious bug for Prefix Sum in order to avoid commutativity as a requirement for the operation.

**Example 3.** *The following code multiplies two matrices $A$ and $B$ represented by 2-dimensional Scala `Arrays` of lazy matrix-wrappers named `A` and `Bt`, where the latter represents $B^T$. Using* `mapD` *and* `reduceD`*, this runs in parallel time* $\Theta\left(\frac{n^3}{p} + \frac{n^2}{p^{2/3}} \log p\right)$.

```
for (i <- 0 until M; j <- 0 until N)
 A(i) zip Bt(j) mapD {case (a,b) => a*b} reduceD (_ + _)
```

## 4. FooPar Implementation in Scala

The mix between functional programming and object oriented features brings interesting new possibilities for program design. In this section, the language concepts needed to implement DPDs and their use in FooPar are presented. In particular, our framework builds on the *builder/traversable pattern* [21] in order to obtain reusable and maintainable code by reducing code duplication and boiler plate code as well as introducing a natural unification of functional concepts.

### 4.1 Option monad

Like Haskell, the standard Scala library also contains a *maybe monad* [28] structure in the shape of the parameterized `Option[+A]` trait. Simply put, a monad is an abstract concept that can be used for convenient control-flows in functional programming. FooPar uses Scala's `Option` monad in order to enable the use of SIMD instructions for group operations where not all processing elements participate.

FooPar also relies on the `Option` monad for robustness. In the battle against *NullPointerExceptions* known from *Java*, Option makes the possibility of a None value explicit, thus moving the problem to compile time. It is a huge advantage for this framework to push as many errors as possible into compile time to disallow erronous runs on expensive HPC-hardware with limited access to computation time. One of the biggest advantages comes from the way Option supports SIMD operations. If we consider the method `map`, `nop` instructions can be simulated elegantly. From the definition of `Option` in the *Scala* API 2.9.2 we have:

```
@inline final def map[B](f: A => B): Option[B] =
  if (isEmpty) None else Some(f(this.get))
```

This nicely encapsulates `nop` as a special case of the map operation on `Option` elements. Given a list `xs` of options, we can nest a map operation to simulate `SIMD`:

```
val xs = (1 to 4).map(i => if (i > 2) Some(i) else None)
def simd[T,U](f: T => U) = (o:Option[T]) => o.map(f)
def double(i:Int) = i*2
xs.map(simd(double)) //
    Vector(None,None,Some(6),Some(8))
```

Note that all the above code is sequential. However, FooPar generalizes this pattern to parallel map operations supporting arbitrary SIMD operations in the form of $\lambda$ expressions. Lists in Scala are also monads, the difference being that a list can be empty (`Nil`), or contain *one or more* elements where Option supports none or *exactly one* element.

To break it down, FooPar uses options mainly in 2 cases: **1)** When not all processing elements participate in a group operation, the SIMD model enforces that they still make the method invocation, safely returning the type `Option[T]`, and **2)** when constructing a DPD, every processing element creates a symbolic structure with their local part stored in an `Option[T]` value, i.e., processing elements that are not a part of the communication group contain the `None` singleton but can continue to invoke methods safely.

## 4.2 Implicits

In Scala, implicits allow the compiler to choose appropriate values for expressions at compile time. Coupled with type-inference, this allows for advanced design patterns. FooPar uses implicits extensively for its typeclasses and for the Traversable-Builder Pattern employed by its DPDs.

Listing 1: *Generic base trait*

```scala
trait A[T] {
  def filter(f: T => Boolean) = ??? //Need type B[T]
  def map[U](f: T => U) = ??? //Need type B[U]
}
class B[U](val x: U) extends A[U]
```

Consider the problem depicted in Listing 1 : Given two classes, `A[T]` and its subclass, `B[U]`, how can a method of `A[T]` have `B[U]` as a return type? This problem can be solved through implicit parameter resolution. If we define a generic method (`map`, Line 3 in Listing 2) we allow the compiler to choose the generic type parameter based on available implicit values in companion objects (`builder`, Line 8 in Listing 2). Now, super-classes can implicitly work with sub-types through their definition of implicit builders. In turn, this allows for generic implementations of functions that return concrete types, i.e., `map[U,That]` is implemented with no knowledge of subclasses A and B, however, if we map an `A` into a container of `String`, we get a B.

Listing 2: *Simple builder pattern*

```scala
1  trait Cont[T] {
2    def elem: T
3    def map[U, That](f: T => U)(implicit builder: U =>
         That): That = builder(f(elem))
4  }
5  case class A(elem: Int) extends Cont[Int]
6  case class B(elem: String) extends Cont[String]
7  object B { //Companion object for B
8    implicit def bldr:(String => B) = (s:String) => B(s)
9  }
10 object Impl extends App {
11   val a:A = A(42)
12   val b:B = a.map(x => x.toString)
13 }
```

## 4.3 Type-Classes through Implicits

FooPar uses type-classes to provide numeric distributed operations on the DPDs, e.g. `sum, product, max, minBy` etc. Type-classes are flexible, and thus, FooPar uses the `Numeric` type-class directly from the standard library of Scala, making numeric operations available for all implementing classes of the standard library. FooPar goes even further and offers methods like `average` which are unavailable in the standard library.

A type-class [11] defines features for a set of types in a weaker sense than interfaces or traits. In Scala, type-classes are powered by implicit arguments to methods providing a looser coupling between classes than other interface-constructs. Using the type-class pattern, a class can impose constraints on a generic type, `T`, on a per-method basis rather than a per-class basis as provided by interfaces and type-bounds. This concept is called *context bounds* in Scala, for example:

```scala
def sumL[T:Numeric](l:List[T]) = l.sum
```

This unfolds to the following more verbose definition:

```scala
def sumL[T](l:List[T])(implicit num:Numeric[T]) = l.sum
```

Now we have a function definition which works only for types `T` which provide an implicit numeric parameter, usually supplied by the companion object for `T`. To explain how this is different than using an *upper type bound* directly on `T`, consider the following example:

```scala
case class NumList[T <: Complex](xs: Complex*) {
  def sum = (xs fold new Complex(0, 0))(_ + _)
  def map[U <: Complex](f: Complex => U): NumList[U] =
      NumList(xs.map(f): _*)
  override def toString = "[" + xs.mkString(", ") + "]"
}
```

We define a numeric list based on lists of type `T` with upper bound `Complex`. In this way, `NumList` gets access to numeric operations like summation over the list. As a result, the list can be used for example like this:

```scala
val r = new Real(2)
val n = new Natural(10)
val comps = NumList(r, n, r, n)
println(comps)
println("sum: " + comps.sum)
println("sum * 2: " + comps.map(x => x + x).sum)
```

While this provides added functionality for numeric types, it comes at the expense of an additional class definition and a bound on `T`. Furthermore, we were forced to provide a template definition of addition (i.e., the method from `Complex`). By introducing a `numeric` type-class, we unify the concept of a generic list and a numeric list. With a numeric type-class, the list class can be rewritten as follows:

```scala
case class GenList[T](xs: T*) {
  def sum(implicit num: Numeric[T]) = xs.sum
  def map[U](f: T => U) = GenList(xs.map(f): _*)
  override def toString = "[" + xs.mkString(", ") + "]"
}
```

Now the list can be used for any type `T` while sum works for any type `T` providing implicit definitions of `Numeric[T]`. In Scala, many of the subclasses of `AnyVal` support this operation, so we have successfully provided a numeric operation with a *least knowledge* principle about the generic type `T`. Type-classes allow us to use the smallest set of constraints for a type and provide us with very fine-grained control.

Using implicit classes in combination with type-classes, we can extend existing implementations of classes without modifications or additional interfaces. As an example, we can add the `average` method to sequences of the Scala API:

```scala
implicit class AvgSeq[T](s: Seq[T]) {
def average(implicit num: Numeric[T]) =
  num.toDouble(s.sum) / s.size
}
println(1 to 9 average) //result: 5.0
```

We have seen a use of this already in Section 3, where a method `toDistSeq` was added to Scala sequences.

Furthermore, FooPar makes use of typeclasses to provide convenience methods for numeric types. As an example, consider the following extension to a Matrix class which can be considered a numeric type. We implement a subclass of the generic `Numeric[T]` trait for the type `Matrix`.

```scala
class MatrixIsNumeric extends Numeric[Matrix] with
    Ordering[Matrix] {
  def plus(x: Matrix, y: Matrix): Matrix = x + y
  def times(x: Matrix, y: Matrix): Matrix = x * y
  def negate(x: Matrix): Matrix = x * -1
...
}
```

By making an instance of this class implicitly available for the `Matrix` companion object, it will be accessible at compiletime for the FooPar DPD classes.

**Example 4.** *As an example, consider the reduction of a distributed sequence containing matrices of equal dimensions. By introducing the implicit numeric instance for matrices, the distributed sequence can be reduced via summation without imposing any constraints on the remaining methods of the distributed sequence.*

```scala
1 val x = new Matrix(Seq(Seq(1, 2), Seq(3, 4)))
2 val dSeq = Array.fill(size)(x).toDistSeq
3 for (res <- dSeq.sumD) {
4   pprintln(res, " size = " + size, res == x * size)
5 }
```

*Line 1 creates an instance of the matrix class. Line 2 creates an array of processing elements in this FooPar execution and then converts it to a distributed sequence. Line 3 calls the distributed* `sumD` *method, which requires an implicit instance of the* `Numeric[Matrix]` *class. Using the for-comprehension, only the root process prints and checks the result against $x \cdot size$. Note that the for-comprehension can be used because Scala's* `Option` *monad supports the higher order method* `foreach`.

*Running the above FooPar program with 8 processing elements yields the following output as expected:*

```
Rank0: ([8.0,16.0]
[24.0,32.0], size = 8,true)
```

## 4.4 Builder/Traversable Pattern

For FooPar's implementation, a main goal was to avoid reimplementing the distributed operations such as `reduceD` and `mapD` for each DPD. With the *Builder/Traversable* pattern we achieve this and implement them once and for all, while maintaining the specific types of the DPDs.

One of the main benefits of Scala's implicits comes from the way the compiler resolves implicit arguments at compiletime. *"If there are several eligible arguments which match the implicit parameter's type, a most specific one will be chosen using the rules of static overloading resolution"* [20]. Using this feature, type information can be pushed *upwards* in a type hierarchy via generics and an adaptation of the *Factory Pattern* [21], i.e., the *Builder/-Traversable Pattern*, effectively solving the problem depicted in Listing 1.

```scala
trait DistTraversable[+T] {
  def elem: Option[T]
  def foreach(f: T => Unit): Unit
  def group: FooParGroup
}
```

The `DistTraversable` trait presented above resides at the base of the type hierarchy (a simplified overview can be seen in Figure 6). It defines methods for retrieving a process' local element as well as means of traversing it. The group is used to supply network communication operations and follows a DPD through chains of consecutive transformations.

FooPar utilizes Scala's standard library support for the `Numeric[T]` type-class [11]. In combination with the group communication operations the user gains convenient access to distributed versions of operations like `sum`, `average`, `min`, `max` and `product` for types `T`, which provide implicit `Numeric[T]` values. This allows user-defined algebraic types (as well as standard primitives) to work with the built in DPDs in FooPar. The distributed numerical operations are available through the `TraversableNumOps` trait.
Both for the type-class support of numeric operations as well as for applying the *Traversable/Builder* pattern, FooPar makes heavy use of *context bounds*.

In the following we are going to take a look at the implementation of distributed variables. The implementation of e.g. distributed sequences is analogous except for more complex builders due to the mapping of indices to ranks.

As shown below `DistVal[+T]` is basically just the composition of the traits `DistTraversableLike` and `TraversableNumOps` and thus the most basic implementation of a DPD.

```scala
class DistVal[+T]
(val elem: Option[T], val group: FooParGroup)
(implicit val fpapp: FooParApp)
extends DistTraversableLike[T, DistVal[T]]
  with TraversableNumOps[T, DistVal[T]] {
  def size = 1
}
```

The real work is performed in `DistTraversableLike` as well as in the implicit builders of the companion object:

```scala
object DistVal {
  type DTB[T] = DistTraversableBuilder[T]
  implicit def canBuildFrom[T, U]
  (implicit fpapp: FooParApp): CBF[DistVal[T], U,
      DistVal[U]] =
    new CBF[DistVal[T], U, DistVal[U]] {
      def apply(): Builder[U, DistVal[U]] =
          newBuilder(None)
      def apply(from: DistVal[T]): Builder[U,
          DistVal[U]] =
        newBuilder(Some(from.group))
    } ...
}
```

When the companion objects of an extending class supplies an implicit builder through the generic trait `CanBuildFrom[From,Elem,To]`, super classes can access this builder at compile-time using explicit self-typing. The companion object `DistVal` implements such an implicit definition of the `CanBuildFrom` trait.

Finally, the trait `DistTraversableLike[T,Repr]` defines the collection operations associated with a distributed traversable by using an implicit builder as a parameter to the respective methods as seen in the below code.

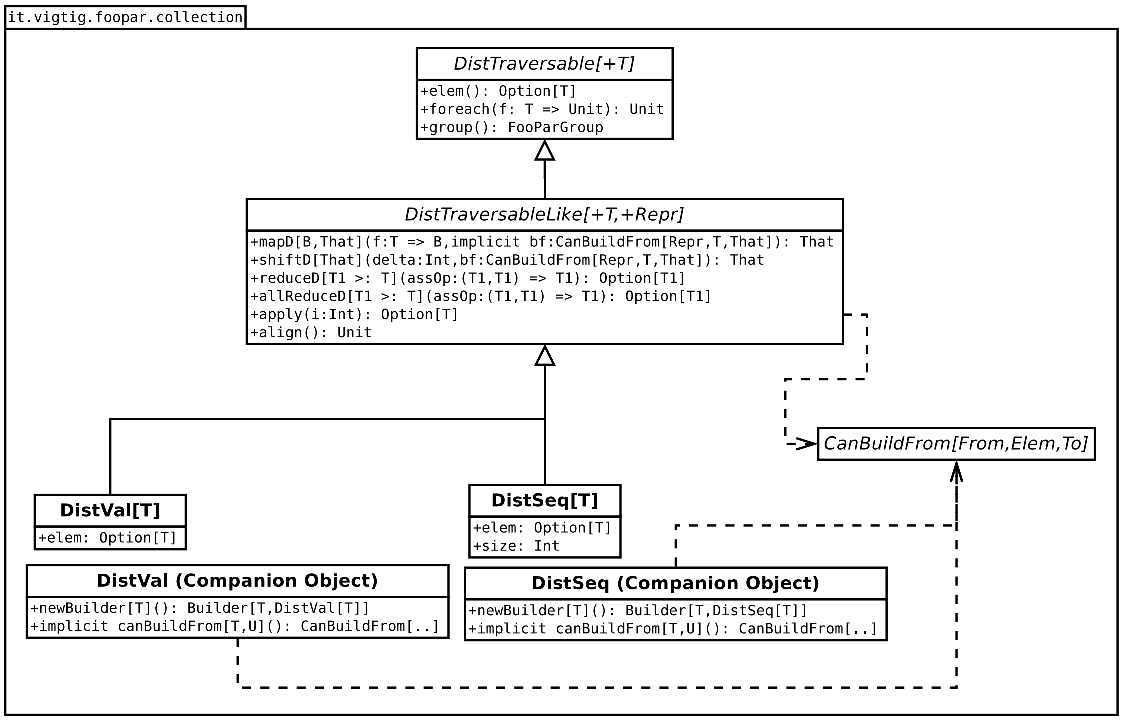
it.vigtig.foopar.collection
DistTraversable[+T]
+elem(): Option[T]
+foreach(f: T => Unit): Unit
+group(): FooParGroup
DistTraversableLike[+T,+Repr]
+mapD[B,That](f:T => B,implicit bf:CanBuildFrom[Repr,T,That]): That
+shiftD[That](delta:Int,bf:CanBuildFrom[Repr,T,That]): That
+reduceD[T1 >: T](assOp:(T1,T1) => T1): Option[T1]
+allReduceD[T1 >: T](assOp:(T1,T1) => T1): Option[T1]
+apply(i:Int): Option[T]
+align(): Unit
CanBuildFrom[From,Elem,To]
DistVal[T]
+elem: Option[T]
DistSeq[T]
+elem: Option[T]
+size: Int
DistVal (Companion Object)
+newBuilder[T](): Builder[T,DistVal[T]]
+implicit canBuildFrom[T,U](): CanBuildFrom[..]
DistSeq (Companion Object)
+newBuilder[T](): Builder[T,DistSeq[T]]
+implicit canBuildFrom[T,U](): CanBuildFrom[..]

Figure 6: *Simplified UML diagram of collection package architecture. The companion objects of `DistSeq` and `DistVal` provide type information to `DistTraversableLike` through a loose coupling with implicit definitions of `CanBuildFrom`.*

```scala
trait DistTraversableLike[+T, +Repr] extends
    DistTraversable[T] {
  self: Repr =>
  ...
  def reduceD[T1 >: T](assOp: (T1, T1) => T1):
      Option[T1] = {
    if (group.partOfGroup)
      for (x <- this) group.allOneReduce(x,
          assOp).foreach(y => return Some(y))
    return None
  }

  def mapD[B, That](f: T => B)(implicit bf:
      CanBuildFrom[Repr, B, That]): That =
  {
    val b: Builder[B, That] = bf(this)
    if (group.partOfGroup)
      for (x <- this) b += f(x)
    b.result
  } ...
}
```

As shown in Figure 6, the same approach is used for distributed sequences (and other DPDs supported by FooPar).

## 5. Verification and Testing

Using core features of functional programming like *higher-order functions* and *immutable data structures*, FooPar allows for analyzable and verifiable programs to be designed and implemented. Given a higher order function, $f$, which takes a function value parameter, $\lambda$, one can prove the correctness of $f$ given some properties of $\lambda$. Furthermore, higher order functions can abstract away message passing, including deadlock-safety in their proofs. As an example, FooPar provides a distributed method, `reduceD`($\lambda$), which is proven to be correct and deadlock-safe for any associative binary operator, $\lambda$. Once a program can be modeled purely in terms of proven higher-order functions, the correctness and safety of that program follows from properties of the function-parameters. In this section, we give a short proof of *All-to-One Reduction* and show how specification testing can be used to bridge the gap between proof and practice.

### 5.1 Correctness Proof of All-to-One Reduction

```
function REDUCE($\lambda, size, acc, rank$)
    for $i \leftarrow 0$ until $\lceil \log_2 size \rceil - 1$ do
        m $\leftarrow 2^i$
        if $rank \bmod 2m = 0$ then
            $acc \leftarrow \lambda(acc, \mathrm{rcv}(rank + m))$        ▷ Side-effect
        else if $rank \bmod m = 0$ then
            send($rank - m, acc$)        ▷ Side-effect
    end for
    if $rank = 0$ then return Some(value)
    else return None
end function
```

Algorithm 1: *Functional All-to-One Reduction in* FooPar.

We prove the correctness of the All-to-One Reduction in FooPar (Algorithm 1) for powers of two by induction on the number of processing elements $p$. Without loss of generality, we use the associative operator $\lambda := (\cdot)$ and $rank = 0$ as root element. The case for $p = 1$ is trivial in that no communication takes place. For illustration purposes, we choose $p = 4$ as the base-case instead.

**Base step:** Let $p = 2^n, n = 2$ and $e_r$ the element initially contained by processing element $r$.

| $i$ | $m$ | $rank$ | recieve/send | $acc$ |
|---|---|---|---|---|
| 0 | 1 | 0 | $receive(1)$ | $(e_0 \cdot e_1)$ |
| | | 1 | $send(0, e_1)$ | $e_1$ |
| | | 2 | $receive(3)$ | $(e_2 \cdot e_3)$ |
| | | 3 | $send(2, e_3)$ | $e_3$ |
| 1 | 2 | 0 | $receive(2)$ | $((e_0 \cdot e_1) \cdot (e_2 \cdot e_3))$ |
| | | 1 | $nop$ | $e_1$ |
| | | 2 | $send(0, (e_2 \cdot e_3))$ | $(e_2 \cdot e_3)$ |
| | | 3 | $nop$ | $e_3$ |

We see that after 2 iterations, $rank = 0$ contains $\prod_{r=0}^{p-1} e_r$, concluding the base step. Let the induction hypothesis state that, for $p = 2^n$, the processing element with $rank = 0$ will contain $\prod_{r=0}^{2^n-1} e_r$ after the $n$th iteration.

**Induction step** $(n+1)$**:** If $p = 2^{n+1}$, at the beginning of iteration $i = n$, by the induction hypothesis, the first $2^n$ elements complete the partial reduction at the root node, $rank = 0$. Furthermore, we see that the remaining $2^n$ processing elements of the *upper half* complete the analogous algorithm at $rank = 2^n$, computing $\prod_{r=2^n}^{2^{n+1}-1} e_r$. At iteration $i = n$, $m = 2^n$ and thus, $rank = 0$ receives the reduction of the upper half from $rank = 2^n$. Since

$$\prod_{r=0}^{2^{n+1}-1} e_r = \left(\prod_{r=0}^{2^n-1} e_r\right) \cdot \left(\prod_{r=2^n}^{2^{n+1}-1} e_r\right)$$

$rank = 0$ now contains the reduction of the entire sequence □.

To prove that this algorithm is safe, we follow the same pattern as the correctness proof. To go from the case of $p = 2^n$ to the case $p = 2^{n+1}$, we complete the final step of the reduction by sending a message from $rank = 2^n$ to $rank = 0$, unlocking the resource necessary for $rank = 0$ to complete its receive call.

Note that the order of computation respects associativity and thus works for any binary associative function. By the fact that Recursive Doubling is the exact inverse operation of All-to-One Reduction using an appropriate $\lambda$ function, we conclude that both the correctness proof and the safety property hold for Recursive Doubling. Finally, the algorithm and proof can (with only minor additions) be generalized to arbitrary roots [1] and arbitrary number of processing elements.

### 5.2 Testing Properties

While the combination of pseudo-code, correctness/safety proofs, and parallel asymptotic running time and scalability analysis can take us far, it is useless if the theory and implementation remain disconnected. Even in a high-level language like Scala, it can be extremely tough to show that the implementation adheres to a strict contract between the pseudo-implementation and the code. Specification testing [19] is a natural next step in Unit Testing. At its core, it is an abstraction over test-case construction, using generators for core types that can be extended to support new data structures. Using *ScalaCheck*, FooPar provides a high level of code-trust, solidifying the relationship between proofs and properties found theoretically and the actual code. Since Scala supports imperative programming, detailed pseudo-code can be translated directly into program code. If imperative code pertains to inherently stateful parts of a program (e.g. message passing in group communication patterns), and is interfaced only through stateless functional programming, the disadvantages of imperative programming do not bleed into user programs. In addition, we can modularize proofs elegantly through proper use of higher order functions and immutability.

[1] Reduction to other roots can obviously be achieved by xor-ing all ranks in the algorithm with the rank of the target root.

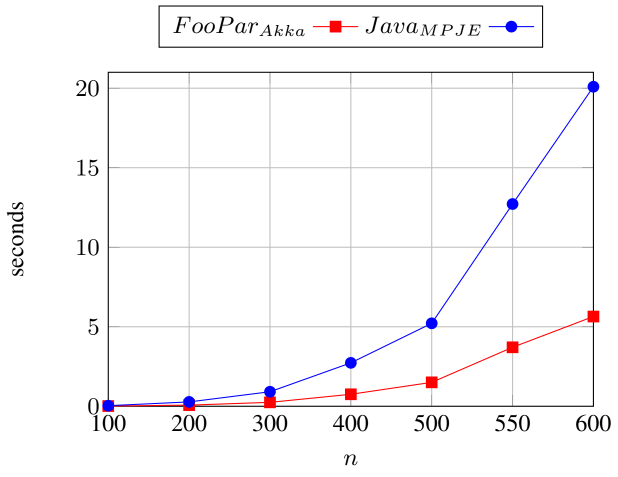

Figure 7: *Average walltime in seconds for matrix reduction with multiplication.* FooPar *with Akka backend and Java with MPJ Express on 16 cores spread across 2 machines.*

Listing 3: *Specification Testing in FooPar*

```
property("maxD") = forAll(sizedLists) { xs: List[Int] =>
  val dseq = xs.toDistSeq
  dseq.maxD shouldEqual xs.max
}
property("reduceD with concat") = forAll(sizedLists) {
    xs: List[Int] =>
  val dseq = xs.toDistSeq.mapD(_.toString)
  dseq reduceD (_+_) shouldEqual xs.mkString
}
```

Listing 3 shows how specification testing can programmatically provide arbitrary test-cases (i.e., automatic edge-case construction) without huge amounts of boiler-plate code. The `shouldEqual` method is a small FooPar extension that allows for idiomatic use of the option monads returned from the DPD methods.

Listing 4: *Implicit extension with shouldEqual method*

```
implicit class Should[T](opt: Option[T]) {
  def shouldEqual[U](x: U) = opt.map(_ ==
      x).getOrElse(true)
}
```

Listing 4 shows how the method `shouldEqual` utilizes the `Option` monad to provide a fallback making sure that processing elements that do not receive a result always answer `true` to specification testing.

## 6. Empirical Results

FooPar's ability to reach close-to-optimal performance in real world HPC settings has already been demonstrated extensively for dense matrix-matrix multiplication (cf. Figure 1) in [12]. Thus, in this section, we focus on two other aspects. First, we compare the performance of a FooPar reduction to that of the MPJ Express implementation. Second, we show the framework's ability to scale by presenting a parallel implementaiton of the Floyd-Warshall algorithm in FooPar and comparing it to a sequential implementation.

### 6.1 Comparison to MPJ Express

We conducted a test of computational scalability reducing $p$ matrices of varying size with the associative operation of matrix-matrix multiplication. By varying the size of the matrices we show the per-

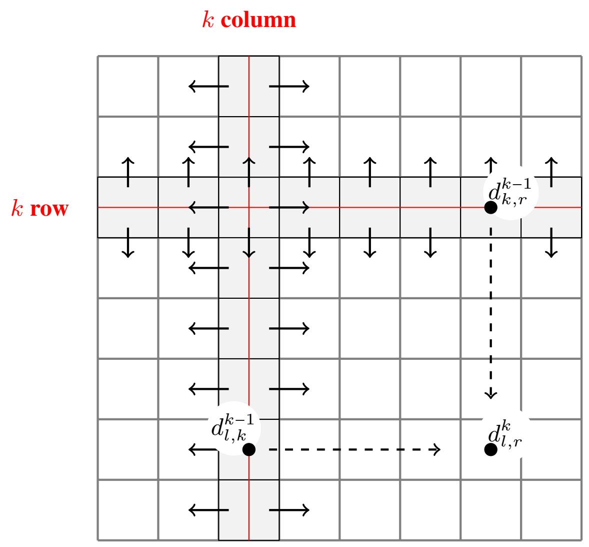

Figure 8: *Communication pattern employed in parallel Floyd-Warshall: in iteration k, messages of size $n/\sqrt{p}$ are broadcast along $\sqrt{p}$ processing elements (in rows and columns).*

formance impact of the computation part in a reduction in FooPar compared to Java+MPJ Express. Both use the same naïve Java implementation of matrix multiplication utilizing a 1-dimensional `double` array matrix representation.

An analysis of the source of MPJ Express reveals an inefficient implementation of the reduce operation. For matrices of size $n$, it runs in $T_P \in \Theta((p-1)(T_s + T_w(n^2) + n^3))$, whereas FooPar implements this operation in $T_P \in \Theta(n(\log p(T_s + T_w(n^2) + n^3))$. Note that for these running times we assume sufficient bandwidth between nodes at each stage of the respective algorithms, i.e., they are network topology oblivious [10].

Even when using MPJ Express as a backend, FooPar can still achieve the latter parallel running time because it adds an algorithmic layer for group communication operations directly above message passing, thereby diminishing the relevance of backend supplied group operations. Figure 7 shows the expected tendency of running times. Notice that, even for small sizes of $n$, FooPar dominates Java with MPJ Express.

## 6.2 Floyd-Warshall Parallelization

The Floyd-Warshall algorithm solves the all-pairs shortest path problem between all nodes $v_1, \dots, v_n$ in a weighted graph. Let $d_{i,j}^k$ be the weight of the *minimum-weight path* between $v_i$ and $v_j$ among vertices in the set $\{v_1, \dots, v_k\}$. The weight of an edge between nod $v_i$ and $v_j$ is denoted as $w(v_i, v_j)$.

The dynamic programming formulation can be expressed as follows, where the shortest path from $v_i$ to $v_j$ is given by $d_{i,j}^n$:

$$d_{i,j}^k = \begin{cases} w(v_i, v_j) & , k = 0 \\ \min\left\{d_{i,j}^{k-1}, d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}\right\} & , k \geq 1 \end{cases}$$

We follow the parallelization approach from [14] (communication pattern presented in Figure 8) and present a scalable version of the parallel Floyd-Warshall algorithm that employs FooPar's Distributed Grid (size $p = q^2$, i.e., $q$ processing elements per dimension):

| n | p | Speedup | | Running time in seconds | |
|---|---|---|---|---|---|
| | | MPJ | Akka | MPJ | Akka |
| 10080 | 16 | 14.65 | 12.76 | 408.52 | 469.06 |
| | 25 | 22.56 | 19.93 | 265.3 | 300.33 |
| | 36 | 26.86 | 27.40 | 222.85 | 218.5 |
| | 49 | 39.07 | 33.81 | 151.25 | 177.05 |
| | 64 | 46.17 | 36.81 | 129.66 | 162.63 |
| | 81 | 56.42 | 45.07 | 106.09 | 132.83 |
| | 100 | 63.54 | 60.55 | 94.21 | 98.86 |
| **38000** | **100** | **94.28** | 87.39 | 3401.68 | 3669.86 |

Figure 9: *Floyd-Warshall parallel benchmark with input size* $n$ = 10080 *and varying numbers of processing elements p. The row for* $n$ = 38000 *and* $p$ = 100 *exemplifies speedups on large-scale inputs.*

Listing 5: *Floyd-Warshall implementation in FooPar*

```
1  def update(row: Vector, col: Vector)(mat: Matrix) = {
2    for (i <- 0 until mat.size; j <- 0 until mat(0).size)
3      mat(i)(j) = math.min(mat(i)(j), row(j) + col(i))
4    mat
5  }
6  def floyd(blocks: LazyMatrix, BS: Int) = {
7    val dim = blocks.size
8    val R = 0 until dim
9    val N = dim * BS
10   var grid = DistGrid(R, R) mapD { case i :: j :: Nil
         => blocks(i)(j).data }
11   for (k <- 0 until N) {
12     val ik = grid.ys.mapD(_(k % BS)).apply(k / BS).get
13     val kj = grid.xs.mapD(_.map(_(k % BS))).apply(k /
            BS).get
14     grid = grid.mapD(update(ik, kj))
15   }
16   grid
17 }
```

Briefly explained, Lines 7-10 initialize the 2-dimensional grid and Line 12 is the inherent sequential loop of the algorithm, which is safely modeled as a standard for loop. Line 13 gets the row $(k \bmod BS)$ of block $\lfloor k/BS \rfloor$ in the *column* of the calling process. Similarly, Line 14 gets the column $(k \bmod BS)$ of the block $\lfloor k/BS \rfloor$ in the *row* of the calling process. Line 15 transforms the grid into the next iteration by updating each block in parallel.

Figure 9 shows scalability result for this implementation. We reach efficiencies of $\approx 0.94$ and $\approx 0.87$ with MPJ-Express and Akka respectively, i.e., we see that the algorithm is scalable even for large numbers of processing elements. Furthermore, while Akka is mostly dominated by MPJ-Express, the backends behave similarly and the difference in performance are likely caused by differences in constants like $t_s$ or $t_w$, i.e., startup and per-word transfer times.

While this example showcases the power of the abstractions in FooPar, no work has been done to optimize the computational kernel for the update function (Lines 1-5 of Listing 5). We note, however, that a highly rewarding aspect of using higher-order functions is that computational kernels are naturally separated from the overall algorithm and, more importantly, completely disconnected from the communication code.

## 7. Conclusion

We have presented FooPar, a novel Scala framework for massively parallel distributed memory computing in Scala, which allows for concise and elegant high-level formulations of parallel algorithms by abstracting the underlying communication into high-level group communication operations.

A FOOPAR implementation of the Floyd-Warshall algorithm using distributed grids achieves a near-linear speed-up of more than 94 on a cluster using 100 processing elements for a matrix of dimension $38000 \times 38000$ demonstrating FOOPAR's potential for expressing scalable algorithms, in this example reducing a computation of nearly 4 days to less than 1 hour.

We have also shown that, independently of the backend, FOOPAR can achieve extremely competitive performance due to its judicious implementation of the high-level abstractions based on the *Builder/Traversable* pattern.

### 7.1 Relation to Previous Work

FOOPAR was very recently introduced in [12], wich focuses on parallel algorithms, isoefficiency analysis, and absolute performance. In contrast, in this paper we focus on its architecture and programming model, its implementation in Scala, its comparative performance w.r.t. MPJ Express, and its real-world scalability.

### 7.2 Future Work

First, we observe that workload partitioning represents a threshold point of abstraction for parallel programming. This problem is less pronounced in shared memory architectures, as the partitioning of data can be symbolic. In distributed memory settings, the communication cost of workload partitioning can easily become a bottleneck of an algorithm. Furthermore, parallel programs using automated workload partitioning are often harder to analyze. While automated workload partitioning does not fit directly into FOOPAR, we plan to explore how FOOPAR can be integrated with a separate dynamic workload allocation module, in particular for local shared memory workload partitioning in multi-core nodes.

Thirdly, FOOPAR's mix of the SPMD and SIMD paradigms makes it ideal for use in connection with large scale SIMD hardware. CUDA [18] and OpenCL [13] offer interesting takes on SPMD/SIMD programming, especially w.r.t. GPU based systems. GPU executions of DPD methods could offer performance boosts to single-node operations, in combination with the distributed algorithms already possible in FOOPAR.

## Acknowledgments

We acknowledge the support by the Danish Council for Independent Research, the Innovation Center Denmark, the Lawrence Berkeley National Laboratory, and the Scientific Discovery through Advanced Computing (SciDAC) Outreach Center.

## References

[1] Carver. `http://nersc.gov/users/computational-systems/carver/`.

[2] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. Steele Jr., and S. Tobin-Hochstadt. The Fortress Language Specification. Technical report, Sun Microsystems, Inc., 2007.

[3] D. A. Bader, K. Madduri, J. R. Gilbert, V. Shah, J. Kepner, T. Meuse, and A. Krishnamurthy. Designing scalable synthetic compact applications for benchmarking high productivity computing systems. *CT-Watch Quarterly*, 2(48), November 2006.

[4] A. Buss, Harshvardhan, I. Papadopoulos, O. Pearce, T. Smith, G. Tanase, N. Thomas, X. Xu, M. Bianco, N. M. Amato, and L. Rauchwerger. Stapl: Standard template adaptive parallel library. In *Proceedings of the 3rd Annual Haifa Experimental Systems Conference*, SYSTOR '10, pages 14:1–14:10, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-908-4. . URL `http://doi.acm.org/10.1145/1815695.1815713`.

[5] B. L. Chamberlain, D. Callahan, and H. P. Zima. Parallel programmability and the chapel language. *IJHPCA*, 21(3):291–312, 2007.

[6] N. Chen, R. K. Karmani, A. Shali, B.-Y. Su, and R. Johnson. Collective communication patterns. In *ParaPLOP*, 2009.

[7] C. Coarfa, Y. Dotsenko, J. Mellor-Crummey, F. Cantonnet, T. El-Ghazawi, A. Mohanti, Y. Yao, and D. Chavarría-Miranda. An evaluation of global address space languages: Co-Array Fortran and Unified Parallel C. In *PPoPP*, pages 36–47. ACM, 2005.

[8] F. Darema. The SPMD model : Past, present and future. In *EuroPVM/MPI Conference*, LNCS 2131, page 1. Springer, 2001.

[9] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *European PVM/MPI Users' Group Meeting*, pages 97–104, 2004.

[10] A. Grama, G. Karypis, V. Kumar, and A. Gupta. *Introduction to Parallel Computing*. Pearson, Addison Wesley, 2003.

[11] C. V. Hall, K. Hammond, S. L. Peyton Jones, and P. L. Wadler. Type classes in haskell. *ACM TOPLAS*, 18(2):109–138, 1996.

[12] F. P. Hargreaves and D. Merkle. FooPar: A Functional Object Oriented Parallel Framework in Scala. In *PPAM*, number 8385 in LNCS, pages 118–129, 2014. preprint:`http://arxiv.org/abs/1304.2550`.

[13] K. Karimi, N. G. Dickson, and F. Hamze. A performance comparison of CUDA and OpenCL. *CoRR*, abs/1005.2581, 2010.

[14] V. Kumar and V. Singh. Scalability of parallel algorithms for the all-pairs shortest path problem. *JPDC*, 13(2):124–138, 1991.

[15] R. Loogen, Y. Ortega-Mallén, and R. Peña. Parallel functional programming in Eden. *JFP*, 15:431–475, 2005.

[16] E. Lusk and K. Yelick. Languages for high-productivity computing: the DARPA HPCS language project. *PPL*, 89(17), 2007.

[17] J. Milthorpe, V. Ganesh, A. Rendell, and D. Grove. X10 as a parallel language for scientific computation: Practice and experience. *Parallel and Distributed Processing Symposium*, pages 1080–1088, 2011.

[18] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, 2008.

[19] M. Odersky. Contracts for Scala. In *Runtime Verification (RV)*, LNCS 6418, pages 51–57. Springer, 2010.

[20] M. Odersky. The Scala language specification, 2011.

[21] M. Odersky and A. Moors. Fighting bit rot with types (experience report: Scala collections). In *FSTTCS*, LIPIcs 4, pages 427–451, 2009.

[22] A. Prokopec, P. Bagwell, T. Rompf, and M. Odersky. A generic parallel collection framework. In *Euro-Par 2011 Parallel Processing*, LNCS 6853, pages 136–147. Springer, 2011.

[23] R. Roestenburg and R. Bakker. *Akka in Action*. Manning, 2012.

[24] T. Rompf, A. K. Sujeeth, H. Lee, K. J. Brown, H. Chafi, M. Odersky, and K. Olukotun. Building-blocks for performance oriented dsls. In O. Danvy and C. chieh Shan, editors, *DSL*, volume 66 of *EPTCS*, pages 93–117, 2011.

[25] T. Rompf, A. K. Sujeeth, N. Amin, K. J. Brown, V. Jovanovic, H. Lee, M. Jonnalagedda, K. Olukotun, and M. Odersky. Optimizing data structures in high-level programs: New directions for extensible compilers based on staging. *SIGPLAN Not.*, 48(1):497–510, Jan. 2013. ISSN 0362-1340. . URL `http://doi.acm.org/10.1145/2480359.2429128`.

[26] A. Shafi and J. Manzoor. Towards efficient shared memory communications in MPJ express. In *IPDPS*, pages 1–7, 2009.

[27] G. L. Taboada, J. Touriño, and R. Doallo. F-MPJ: Scalable Java Message-passing Communications on Parallel Systems. *Journal of Supercomputing*, 60(1):117–140, 2012.

[28] P. Wadler. The essence of functional programming. In *Principles of Programming Languages*, pages 1–14. ACM, 1992.

[29] P. Wadler. Why no one uses functional languages. *SIGPLAN Not.*, 33(8):23–27, Aug. 1998. ISSN 0362-1340. . URL `http://doi.acm.org/10.1145/286385.286387`.

[30] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *USENIX conference on Hot topics in cloud computing*, page 10. USENIX Association, 2010.