

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

June 6, 2014

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 [4].

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 [12], Cray's Chapel [5], and Sun's Fortress [2] are three prominent examples, that are based on the 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 and analyzing scalability are usually not considered a prominent design goal.

A promising approach for HPC comes from Delite [17], 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 rely on implementations using *lightweight modular staging* [16], an advanced design pattern utilizing a very minimal compiler plugin for Scala. Compared to FOOPAR, 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.

While the goal of our work is also a high productivity approach for HPC, our contribution is orthogonal to the before-mentioned approaches. Very recently, we developed FOOPAR [3], a parallel framework which is based on the functional object-oriented programming language Scala [14]. Functional languages have at most played a niche role in HPC, often both due to acceptance problems 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

#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.*

[3]). 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 [13].

In this paper, we see how higher order functions in FOOPAR encapsulate and modularize correctness, safety and running times. We see that, by choosing the right level of abstraction, one can achieve analyzability while still distancing oneself from verbose frameworks like MPI (through higher order functions, no explicit message passing, etc.) and less flexible schemes like Map-Reduce (e.g. by supporting One-to-All-Broadcasts, Prefix-Sum, etc.).

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 *without* disturbing the analyzability. Using functional programming concepts, the *Single Program Multiple Data* (SPMD) [7] can emulate *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).

While FOOPAR complements the *parallel collections* [15] 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 [10], Spark [18], and Scala’s own parallel collections [15] 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 sweet-spot between high performance computing and highly abstract, maintainable, and analyzable programming.

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 [8], 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 (albeit, inherently unscalable!).

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 [6]. Focusing on performance and programmability for next-generation architectures, novel languages like X10 and Chapel provide richer execution frameworks and allow also asynchronous creation of tasks [11]. 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.

Like other frameworks, FOOPAR abstracts away network programming and peer-to-peer communication leading to safer software in terms of deadlocks and race conditions.

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.

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

```

1 def pi(n: Int) = {
2   val f = (x: Double) => 4d / (1d + x * x)
3   val ff = (x: Int) => (x - 0.5d) / n
4   (1 to n).map(f compose ff).sum / n
5 }

```

Mapping the composition $f \circ ff$ on a range from 1 to n inclusive, we find the an approximation by dividing the sum by n with some precision n .

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:

```

1 def pi(n: Int) = {
2   val f = (x: Double) => 4d / (1d + x * x)
3   val ff = (x: Int) => (x - 0.5d) / n
4   DistSeq.ranged(1 to n).mapD(f compose ff).avgD
5 }

```

The sequential runtime T_S is $\Theta(n)$. 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 [9]).

Using the distributed parallel reduction, 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 $\Theta(n)$

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, λ , one can prove the correctness of f given some properties of λ . 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, λ . 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.

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.

Base step: Let $p = 2^n$, $n = 0$ and e_r the element initially contained by processing element r . The range 0 **until** $\lceil \log_2 1 \rceil - 1 = 0$ **until** 0 is empty, thus the loop is skipped, yielding simply the value e_0 .

```

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

```

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

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, $\text{rank} = 0$. Furthermore, we see that the remaining 2^n processing elements of the *upper half* complete the analogous algorithm at $\text{rank} = 2^n$, computing $\prod_{r=2^n}^{2^{n+1}-1} e_r$. At iteration $i = n$, $m = 2^n$ and thus, $\text{rank} = 0$ receives the reduction of the upper half from $\text{rank} = 2^n$. Since

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

$\text{rank} = 0$ now contains the reduction of the entire sequence $\square$.

The proof of safety in terms of liveness can be constructed in a similar way to that of the correctness proof.

While the combination of pseudo-code, correctness/safety proofs, and parallel asymptotic runtime and scalability analysis can take us far, it is useless if the theory and implementation remain disconnected. Specification testing [13] is a natural next step in Unit Testing. Using this, FOOPAR provides a high level of code-trust, solidifying the relationship between proofs and properties found theoretically and the actual code.

```

1 property("maxD") = forAll(sizedLists) { xs: List[Int] =>
2   val dseq = xs.toDistSeq
3   dseq.maxD shouldEqual xs.max
4 }
5 property("reduceD with concat") = forAll(sizedLists) { xs: List[Int] =>
6   val dseq = xs.toDistSeq.mapD(_.toString)
7   dseq.reduceD (_+_) shouldEqual xs.mkString
8 }

```

The above code shows how specification testing can programmatically provide arbitrary test-cases (i.e., automatic edge-case construction) without huge amounts of boiler-plate code. Correctness is proven easily by showing that integer addition (string concatenation respectively) is associative. Since we proved the reduce operation correct and safe for any associative binary operation (implicitly present in the 1st property, explicitly as the anonymous function $(_+_)$ in the 2nd property), we need only to assert our belief that the implementation does in fact reflect the pseudo-code in Algorithm 1. Since Scala is a general purpose language, there is no straight forward way to programmatically proof the correctness of the code itself, however, due to the support of imperative programming we can do a direct translation of the pseudo-code in the language. This encapsulation of correctness proofs and running times through higher order functions in FOOPAR coupled with specification testing provides a basis for building rigid, analyzable programs without performance regrets.

The `shouldEqual` method is a small FOOPAR extension that allows for idiomatic use of the option monads returned from the DPD methods. The code below shows how the method `shouldEqual` utilizes the `Option` monad to provide a fallback making sure that processing elements that do not

receive do not yield false **negatives** in properties.

```
1 implicit class Should[T](opt: Option[T]) {  
2   def shouldEqual[U](x: U) = opt.map(_ == x).getOrElse(true)  
3 }
```

References

- [1] Carver. <http://nersc.gov/users/computational-systems/carver/>.
- [2] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., and Sam Tobin-Hochstadt. The Fortress Language Specification. Technical report, Sun Microsystems, Inc., 2007.
- [3] AUTHOR and AUTHOR. FooPar: A Functional Object Oriented Parallel Framework in Scala. In *PPAM*, LNCS, 2013. To be published.
- [4] David A. Bader, Kamesh Madduri, John R. Gilbert, Viral Shah, Jeremy Kepner, Theresa Meuse, and Ashok Krishnamurthy. Designing scalable synthetic compact applications for benchmarking high productivity computing systems. *CTWatch Quarterly*, 2(48), November 2006.
- [5] B. L. Chamberlain, D. Callahan, and H. P. Zima. Parallel programmability and the chapel language. *IJHPCA*, 21(3):291–312, 2007.
- [6] Cristian Coarfa, Yuri Dotsenko, John Mellor-Crummey, François Cantonnet, Tarek El-Ghazawi, Ashrujit Mohanti, Yiyi Yao, and Daniel Chavarría-Miranda. An evaluation of global address space languages: Co-Array Fortran and Unified Parallel C. In *PPoPP*, pages 36–47. ACM, 2005.
- [7] F. Darema. The SPMD model : Past, present and future. In *EuroPVM/MPI Conference*, LNCS 2131, page 1. Springer, 2001.
- [8] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy 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.
- [9] A. Grama, G. Karypis, V. Kumar, and A. Gupta. *Introduction to Parallel Computing*. Pearson, Addison Wesley, 2003.
- [10] Rita Loogen, Yolanda Ortega-Mallén, and R. Peña. Parallel functional programming in Eden. *JFP*, 15:431–475, 2005.
- [11] Ewing Lusk and Katherine Yelick. Languages for high-productivity computing: the DARPA HPCS language project. *PPL*, 89(17), 2007.
- [12] Josh Milthorpe, V. Ganesh, Alistair Rendell, and David Grove. X10 as a parallel language for scientific computation: Practice and experience. *Parallel and Distributed Processing Symposium*, pages 1080–1088, 2011.
- [13] Martin Odersky. Contracts for Scala. In *Runtime Verification (RV)*, LNCS 6418, pages 51–57. Springer, 2010.
- [14] Martin Odersky. The Scala language specification, 2011.

- [15] Aleksandar Prokopec, Phil Bagwell, Tiark Rompf, and Martin Odersky. A generic parallel collection framework. In *Euro-Par 2011 Parallel Processing*, LNCS 6853, pages 136–147. Springer, 2011.
- [16] Tiark Rompf, Arvind K. Sujeeth, Nada Amin, Kevin J. Brown, Vojin Jovanovic, HyoukJoong Lee, Manohar Jonnalagedda, Kunle Olukotun, and Martin Odersky. Optimizing data structures in high-level programs: New directions for extensible compilers based on staging. *SIGPLAN Not.*, 48(1):497–510, January 2013.
- [17] Tiark Rompf, Arvind K. Sujeeth, HyoukJoong Lee, Kevin J. Brown, Hassan Chafi, Martin Odersky, and Kunle Olukotun. Building-blocks for performance oriented dsls. In Olivier Danvy and Chung chieh Shan, editors, *DSL*, volume 66 of *EPTCS*, pages 93–117, 2011.
- [18] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *USENIX conference on Hot topics in cloud computing*, page 10. USENIX Association, 2010.