Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200073

Backus FP Revisited: A Parallel Perspective on Modern Multicores

Alessandro DI GIORGIO [a] and Marco DANELUTTO [a] [1]
[a] *Dept. Computer Science, Univ. of Pisa*

Abstract. We discuss an open source implementation of Backus FP formalism in C++. Our implementation preserves all the nice formal properties of the original language. The implementation is fully C++17 compliant and leverages standard concurrency mechanisms. It provides linear scalability on state-of-the-art shared memory multi cores. By preserving the possibility to use all the rules of the associated "algebra of programs" described by Backus more that 40 years ago, the C++ FP implementation is a natural candidate to be used to introduce parallel programming concepts in core parallel computing courses.

Keywords. parallel patterns, algorithmic skeletons, code refactoring, structured parallelism

1. Introduction

Backus Turing award lecture [Bac78] dates back in the late '70 but provides different features that may be very interesting in these days characterized by the pervasive presence of different kind of parallel devices. Backus designed a programming framework (FP) aimed at relieving the programmers from the burden of explicitly controlling traffic through the Von Neumann bottleneck via memory references. Computations in FP are represented by compositions of functions that may be i) data combiners (just re-shaping data), ii) data transformers (e.g. arithmetic functions) and iii) higher order functional (an apply-to-all and an insert functions that represent kind of map and reduce computations).

As an example, **trans** represents the transpose data combiner: given a sequence of sequences it returns the sequence of sequences made by the corresponding items in the original inner sequences. The **distr** and **distl** data combiners get a sequence and an object and return the sequence of sequences made of the items of the original sequence paired (in a new sequence) with the object. The two variants represent distribution of the right object into the left sequence and vice-versa. Higher order functions include $[f,g]$, the higher order function that builds a sequence of two items obtained applying f and g respectively on the input item, α that is the *apply-to-all* higher order function, applying the function parameter to all items in the input sequence, and finally $/$ that is the *insert* higher order function, "summing up" all items in the input sequence by means of the parameter function (see Fig. 1 for the main FP function definition).

The typical code shown to illustrate FP features is the code implementing matrix multiplication. In FP data is represented in sequences, enclosed in angle brackets ($\langle \ldots \rangle$). A matrix will be therefore represented as a sequence of sequences (the matrix rows). In order to provide the matrix multiplication code, first we define the inner products as:

Higher order fuctions
$\alpha f : \langle x_1 \dots x_n \rangle \equiv \langle f(x_1) \dots f(x_n) \rangle$
$/\oplus : \langle x_1 \dots x_n \rangle \equiv \langle x_1 \oplus \dots \oplus x_n \rangle$
$[f,g] : x \equiv \langle f(x), g(x) \rangle$
$f \circ g : x \equiv f(g(x))$
Fuctions (data transformers)
$binop :< x,y >\equiv x\ binop\ y$
Data combiners
$\mathbf{distl} : \langle a, \langle x_1 \dots x_n \rangle \rangle \equiv \langle \langle a,x_1 \rangle \dots \langle a,x_n \rangle \rangle$
$\mathbf{distr} : \langle \langle x_1 \dots x_n \rangle, a \rangle \equiv \langle \langle x_1,a \rangle \dots \langle x_n,a \rangle \rangle$
$\mathbf{rotl} : \langle x_1 \dots x_n \rangle \equiv \langle x_2 \dots x_n, x_1 \rangle$
$\mathbf{rotr} : \langle x_1 \dots x_n \rangle \equiv \langle x_n, x_1 \dots x_{n-1} \rangle$
$\mathbf{trans} : \langle \langle < x_1 \dots x_n \rangle \langle y_1 \dots y_n \rangle \rangle \equiv \langle \langle x_1,y_1 \rangle \dots \langle x_n,y_n \rangle \rangle$
$i : \langle x_1 \dots x_i \dots x_n \rangle \equiv x_i$
All functions are $\perp$ preserving. Whenever x is or contains $\perp$ then $f : x = \perp$ for any f

Figure 1. Main FP components

$$IP \equiv (/+) \circ (\alpha \times) \circ \mathbf{trans}$$

The computation of the inner product applied to a sequence of two sequences representing the two vectors may be described by the following (rewriting) steps:

$$(/+) \circ (\alpha \times) \circ \mathbf{trans} : \langle \langle 1,2,3 \rangle, \langle 4,2,2 \rangle \rangle \to$$
$$(/+) \circ (\alpha \times) : \langle \langle 1,4 \rangle, \langle 2,2 \rangle, \langle 3,2 \rangle \rangle \to (/+) : \langle 1,4,6 \rangle \to 11$$

Then the matrix multiplication (input is a sequence of two matrices, each represented as a sequence of sequences (rows)) may be defined as follows:

$$\mathrm{MM} \equiv \underbrace{(\alpha\alpha IP)}_{compute\ code} \circ \underbrace{(\alpha\ \mathbf{distl}) \circ \mathbf{distr} \circ [1, \mathbf{trans} \circ 2]}_{data\ routing\ code}$$

In the MM code, the right part represents the computation needed to prepare the data for the actual computation part ($(\alpha\alpha IP)$, that is apply IP on all the sequences build of a row of the first matrix and a column of the second one, as prepared byu the data routing code from the initial pair of matrices). Actually, the definition of IP may be used in place of the IP call in MM, which leads to the expression:

$$\underbrace{(\alpha\alpha((/+) \circ (\alpha \times)}_{compute\ code} \circ \underbrace{\mathbf{trans})) \circ (\alpha\ \mathbf{distl}) \circ \mathbf{distr} \circ [1, \mathbf{trans} \circ 2]}_{data\ routing\ code}$$

with an even longer "data routing" part on the right and a correspondingly longer "computational" part on the left.

In his work, Backus stressed the fact FP may be used as an *algebra* of programs, with different rules that can be used to transform programs into functionally equivalent, syntactically different programs. Despite the fact parallel execution of programs was not considered in the paper, the higher order functions and the data combiners may be interpreted as parallel operations. Different researchers pointed out that FP programs naturally express parallel computations (e.g. [WB94,M1]). We claim that the idea of separating data combiners from actual computations may be useful to express different kind of optimizations.

In this paper, we discuss a framework (fpar) providing Backus FP as an embedded parallel DSL in C++. The implementation leverages the modern features recently included in C++ as well as different existing libraries for parallelism support (OpenMP) and to implement immutable data structures (see Sec. 2.0.3). We will show that programs written in fpar may be automatically parallelized achieving proper speedups on small size shared memory multi-cores using standard C++ mechanisms (threads) and state-of-the-art parallel programming frameworks (OpenMP). In addition, we will discuss how we may apply known and proven correct program transformations that actually improve application performance by coarsening parallel computations (map fusion rule) or improving the data combiner usage (zip rules).

The usage of refactoring rules preserving the functional semantics while changing/improving non functional properties of programs is of great importance. The availability of such refactoring rules has been proven to be a viable solution to explore alternative implementations of parallel applications even before actually starting their coding, especially in the framework of structured parallel programming [BHD$^+$13,GD18, MRR12]. The implementation of fpar preserves all the properties of Backus' FP framework and, in particular, it can be used to show how different refactoring rules may be applied to simple numerical computations such that the rules improve different kind of non functional properties (performance, as shown in Sec. 3, as well as data locality or load balancing, not covered in this paper). Overall this provides the possibility to run simple exercise in classroom whose complexity is far less than the complexity involved in running patterned applications such as those developed using different C++ based parallel programming frameworks [dRADFG17,Rei07].

Finally, we want to point out an additional argument in favour of the usage of FP, related to the utilization of data parallel accelerators. FP code exposes the data transformations needed to subsequently execute map and reduce functionals. This can be exploited while targeting GPU accelerators. In fact the composition of combiners may be used to optimize data transfers to and from GPU accelerators, and the apply-to-all and insert functionals naturally define proper and efficient GPU kernels. Despite we do not discuss explicitly in this paper these aspects, they may contribute to the development of automatic parallelization of programs targeting both CPU cores (as we demonstrated) and GPU cores.

The paper contribution can be summarized as follows.

- We introduce a modern C++ implementation of Backus' FP targeting shared memory multi-cores via OpenMP and we
- We discuss an experiment parallelizing a simple neural network training code. The parallelization comes for free after turning classical imperative code into FP code.
- We discuss how a trivial application of some "algebra of programs" transformation may be used to improve the performance of the original application FP code.
- Finally, we show how decently grained FP computations scale on state-of-the-art shared memory multicores.

2. Implementation

fpar is an implementation of Backus FP in C++17. The implementation is provided as an open source library and available on github[2]. The library is actually provided as an header only package as the compiler optimization techniques may greatly benefit from the simultaneous compilation of both library and business logic (user) code.

2.1. Data types

fpar aims at reproducing the key features of FP as faithful as possible. In particular, data types are implemented via a single variadic template class `Object` that encapsulates a `variant` type:

```
template <typename... Ts>
class Object {
  private:
    variant<monostate, Ts..., flex_vector<Object<Ts...>>> _obj;
  public:
    template <typename T> Object(const T& obj) : _obj(obj) {}
    template <typename T> operator T () const { return get<T>(_obj); }
};
```

This enables the possibility to express polymorphic objects that can assume values of one of the types specified in the instance of the template or, recursively, sequences of such objects. Finally, a further alternative is the empty type $\perp$ which is represented as an instance of `monostate` that is also conveniently denoted as the constant expression `Bottom`. The advantage of using this technique is twofold: on one hand the use of variant enforces type safety [std19], on the other hand the possibility of identifying alternatives of the possible types as variadic arguments frees the implementation from a fixed set of available types. However, programmer has to declare the types of the items eventually appearing in sequences before actually using them. As an example, if we want to have integers and floating point numbers in a sequence, we must use the following code:

```
using namespace fpar;
using Number = Object<int, double>;
Sequence<Number> X = {0.0, 42, 1.0, 23.0};
```

2.2. Functions

All of the basic arithmetic and logic operators and functions to manipulate and access sequences are implemented. In addition, higher order functions, called functionals, are also provided. It is worth pointing out that all of them are unary functions that take and return (constant references to) `Object` instances. Therefore, an n-ary function takes a sequence of n objects acting as multiple arguments (e.g., the `plus` operator takes the sequence of the two needed operands).

Since the available types are decided by the programmer, all of the functions and functionals provided by fpar are function templates that take as template parameter the instantiated `Object` class. Some of them also have an additional template parameter that specifies the kind of execution (parallel or sequential). As an example, the following code defines a function that squares all items in a sequence in parallel:

[2]`https://github.com/alessandrodgr/fpar`

```
auto square =
    apply_to_all<par_exec, Number>([](const Number& x) { return (x*x); });
```

In this case, an OpenMP parallel for is used to implement the apply_to_all. Parallelism degree used in parallel computations of fpar may be fixed through OMP_NUM_THREADS variable, as usual. If a seq_exec was used as first template parameter of the apply-to-all, the application of a square function on a sequence would have been performed sequentially. In other cases, such as the insert (foldr) and condition functionals, the mechanisms used for the parallel evaluation strategy are the ones provided by the standard library (thread, async, future).

2.3. *Immutability*

In order to respect the "functionality" of the FP framework, data managed by the fpar library are implemented using an immutable data structure implementation [Pue17], provided by the immer library[3].

An alternative version does not use immutable data structures requiring a little bit more attention while coding applications, but providing better performances when executing functionals in par_exec mode. It is worth pointing out fpar pays a penalty in terms of performance w.r.t. non fpar equivalent code, mainly due to data type boxing that, besides usual overhead, impairs automatic vectorization opportunities. We are currently working to overcome this limitation.

However, the usage of immutable data structures, in addition to the const-correctness enforced by the constant reference parameter passing, gives two main advantages: i) in many cases it rules out eventual data races that otherwise the programmer should take care of, and ii) it keeps the semantics of parallelized constructs unchanged.
This last property is a consequence of the fact that functions with no side effects naturally introduce independence among the tasks executing the constructs in parallel and, since these constructs are parallelized via embarrassingly parallel algorithms (parallel for) task independence is a prerequisite for the correctness of the results.

The only case where purity is not enough to guarantee the correctness of the result is the insert functional (foldr), where also commutativity and associativity of the reducing function is asked [MRR12].

Finally, the main consequence of keeping unchanged the semantics of these constructs is that all of the laws and theorems given by the "algebra of programs" also hold for parallel programs. Therefore, fpar programs performance can be optimized using two different, not necessarily disjoint, approaches:

- parallelization of constructs
- simplification of programs via algebraic laws

As an example, the first kind of optimization can be applied to the inner product function *IP* presented in Sec. 1, whose fpar implementation is:

```
auto ip =
  (insert<par_exec>(add, Number(0)) *
    (apply_to_all<par_exec, Number>(mul) *
      trans<Number>))(x);
```

[3] https://github.com/arximboldi/immer

As explained before, the parallelization happens for the `insert` and `apply_to_all` functionals, since their execution flag is set to `par_exec`. However, in this case we can further optimize the program by applying, for example, the rule for zip introduction ($\alpha\ f \circ \text{trans} \equiv \text{zip}\ f$):

```
auto ip =
  (insert<par_exec>(add, Number(0)) *
    (zip<par_exec, Number>(mul))(x);
```

Doing so, the amount of computation along with memory operations is drastically reduced, resulting in a better performing program equivalent to the original one. Also notice that in the original code there was a sequential part computing the transposition of the two input vectors (`trans`), while the transformed code is fully parallelized, except for the function composition.

3. Experimental validation

We first discuss an experiment aimed at demonstrating the applicability of the refactoring rules typical of the FP framework and the possibility to achieve notable performance increases through refactoring. In this experiment, we parallelized a simple Neural Network training code with fpar. The original code is written as a loop iterating steps that include matrix multiplications, matrix differences and matrix items transformations. The single iteration can be expressed in FP considering the application of the following steps, working on different input and temporary data sequences: a matrix multiplication, an α, a **zip**[4], another α, a second **zip**, a second matrix multiplication and eventually a final **zip**. These phases eventually result in the following excerpt of C++ code:

```
auto out = (apply_to_all<par_exec, Number>(sigmoid) *
            apply_to_all<par_exec, Number>(ip(W)))(X);

auto err = (apply_to_all<par_exec, Number>(sub_op<double, Number>) *
            trans<par_exec, Number> *
            construct<seq_exec, Number>({constant<Number>(Y), id<Number>}))(out);

auto delta = (apply_to_all<par_exec, Number>(mul_op<double, Number>) *
              trans<par_exec, Number> *
              construct<seq_exec, Number>({
                 constant<Number>(err), apply_to_all<par_exec, Number>(sigmoidder)
              }))(out);

auto WDelta = (apply_to_all<par_exec, Number>(ip(delta)) *
                trans<par_exec, Number>)(X);

W = (apply_to_all<par_exec, Number>(add_op<double, Number>) *
      trans<par_exec, Number> *
        construct<seq_exec, Number>({constant<Number>(W), id<Number>}))(WDelta);
```

The code computes the same results of the original, C++ only, sequential code and achieves decent speedups on single socket multi-core systems with 64bit Linux 4.15 and

[4]the **zip** combiner may be defined in FP as follows: **zip** $f \equiv (\alpha\ f) \circ$ **trans**

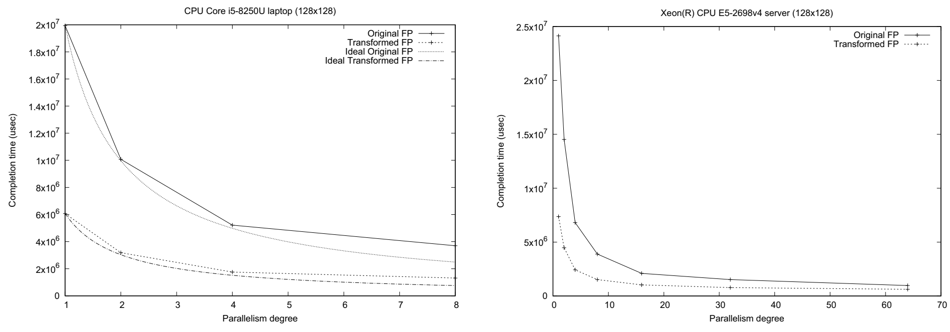

Figure 2. Original and transformed code T_C on a i5 laptop (4 core, 2 way hyper threading) left and on a dual Xeon(R) CPU E5-2698 v4 server (40 cores, 2 way hyper threading) (right). X axis: parallelism degree, Y axis: T_C in $\mu secs$

g++ 9.0.1. Leveraging on the FP formal background, the application represented by FP code may be rewritten using different rules of the algebra, namely:

- $(\alpha\ f) \circ (\alpha\ g) \equiv \alpha(f\ \circ\ g)$ (map fusion)
- $\alpha\ f \circ \text{trans} \equiv \text{zip}\ f$ (zip intro)
- $(\text{zip}\ f) \circ [\alpha\ g \circ 1,\ \alpha\ h \circ 2] \equiv \text{zip}\ (f \circ [g \circ 1, h \circ 2])$ (zip generalize)

In principle, the transformation in the C++ code may be performed automatically, may be following an approach such as the one proposed in [GD17] for more generic and high level parallel pattern applications.

By manually applying the rules mentioned above, we obtain the code listed below that turns out to compute the correct results (as expected, due to the proven correctness of the transformation rules used) but also to compute results with better performance w.r.t. the original code.

```
auto out =
  apply_to_all<par_exec, Number>(
    construct<seq_exec, Number>({id<Number>, sigmoidder}) *
    sigmoid * ip1(W)
  )(X);

auto delta = (zip<par_exec, Number>(sub_and_mul) *
          construct<seq_exec, Number>({constant<Number>(Y),
          id<Number>}))(out);

W = (zip<par_exec, Number>(
  add_op<double, Number> *
    construct<seq_exec, Number>({select<2, Number>,
    ip1(delta) * select<1, Number>})
) * construct<seq_exec, Number>({
        trans<par_exec, Number>,
        constant<Number>(W)
    }))(X);
```

Fig. 2 shows some results we achieved running our FP version of the neural network training code on different architectures. Fig. 2 (right) shows the completion time (T_C) with input matrix size 256x256 on a I7 laptop with 4 cores, with 2-way hyper-threading. Measured and ideal times are shown. Transformed code performs better both in absolute

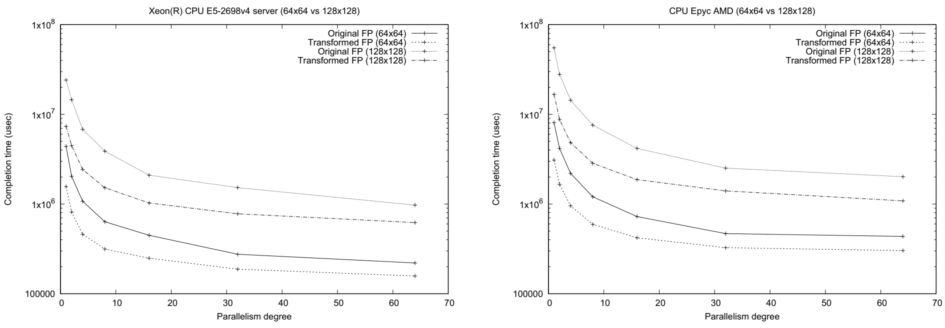

Figure 3. Effect of computational grain: T_C relative to different matrix sized (64x64 and 128x128) on a Xeon PHI KNL (64 cores, 4 way hyper threading) and on an AMD Epyc 7661 (2x32 core, 2-way hyper threading) original and transformed versions. Physical cores only have been used.

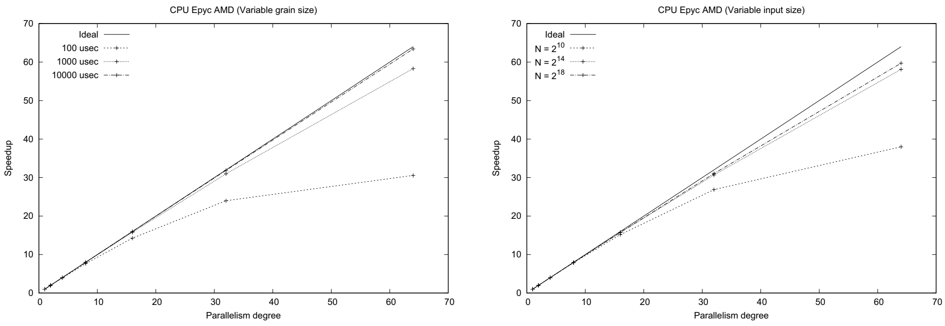

Figure 4. Speedup of typical FP code (AMD Epyc 7551, 64 cores, 2 way hyper-threading). Effect of variable business logic weight (left) and input size (right) in the computation of a synthetic FP application. Speedup computed w.r.t. plain C++17 sequential code.

times and in scalability w.r.t. original FP code. Fig. 2 (left) shows scalability results on a Xeon E5 v4 server relative to 1024x1024 matrix size, confirming the kind of results observed on smaller parallelism degrees on the I7 processor. Eventually, Fig. 3 shows impact of computational grain. The completion times show (log scale on the Y-axis) are relative to an experiment run on both an Intel Xeon PHI KNL [MMM$^+$17] and an AMD Epyc 7551 with different input matrix sizes: 256x256 and 1024x1024. In both cases, the transformed version performs much better than the original one, either stopping scaling after the original version (256x256 version) or even not stopping improving times with parallelism degree while the original version actually stops quite early.

The numbers shown here are good when comparing the two different versions of the code. However, in absolute they demonstrate quite an amount of overhead derived from the ”pure” implementation of FP. As an example, the usage of immutable data structures–while greatly simplifying the overall parallelism management–introduces a considerable overhead with respect to the very same computations implemented using plain `std::vector` data type. We therefore run a different set of experiments aimed at showing strong and weak scaling properties of fpar. Using synthetic applications, we looked for the typical computational grain needed to go closer to ideal speedups and to the effect of working on larger and larger data structures. The results are summarized in Fig. 4. The left plot shows that close to ideal speedup can actually be achieved when

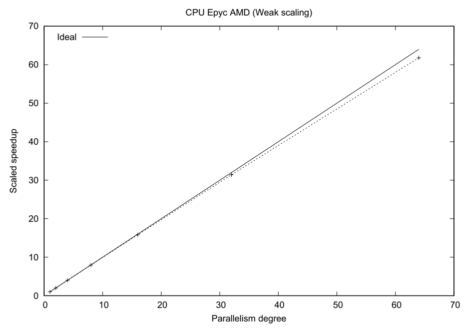

Figure 5. Weak scalability of typical FP code (AMD Epyc 7551, 64 cores, 2 way hyperthreading). Scaled speedup computed w.r.t. plain C++17 sequential code.

the grain of the parallel computation (time spent in a single parallel activity) is close or higher to the milliseconds. This allows to conclude that our FP implementation is definitely not fine grain but still can achieve decent strong scalability. The right plot shows speedups achieved when increasing the size of the processed data. Fig. 5 shows typical results achieved with the synthetic applications in terms of weak scalability.

4. Conclusions

In this paper we briefly discussed a modern open source implementation of Backus' FP exploiting pure C++17: fpar. We discussed how fpar implementation preserves all the nice properties of Backus algebra of programs and how fpar can be used to improve performance of parallel programs through the application of simple program refactoring rules from FP. fpar implementation demonstrated fairly good scalability on state-of-the-art shared multicore architectures. The experiments run with the synthetic applications also demonstrate that fpar implementation exploits medium to coarse grain parallelism pretty efficiently on the same state-of-the-art shared memory parallel architectures. Although other modern programming languages include some of the FP features discussed and exploited in this work, the clean and minimal design of FP supported the efficiency achieved by fpar. Other functional programming languages (Haskell and Erlang, just to mention two well know and widely adopted languages) also support parallelism at different levels and regularities. Refactoring techniques have also been designed to improve or introduce parallelism [BLH12]. Some of the advantages of these languages are that FP programs can be easily expressed via, for example, point-free programming and moreover, they natively support immutable data structures and purity without the need of external libraries. Therefore, they are probably more optimized than fpar with respect of these techniques and features that were artificially tuned inside of our C++17 implementation, as showed in Sec. 2. However the proper usage of these much more sophisticated languages requires different/more significant effort than the one required to understand and use fpar, especially for the parallelisation of programs that in our library is achieved just by setting a flag. Last but not least, the clean and easy to understand implementation of fpar makes it suitable to be adopted in parallel programming courses to demonstrate refactoring techniques for parallel programming.

Acknowledgements This work has been partially funded by Univ. of Pisa PRA_2018_66 DECLware: Declarative methodologies for designing and deploying applications

References

[Bac78] John Backus. Can programming be liberated from the von neumann style?: A functional style and its algebra of programs. *Commun. ACM*, 21(8):613–641, August 1978.

[BHD$^+$13] Christopher Brown, Kevin Hammond, Marco Danelutto, Peter Kilpatrick, Holger Schöner, and Tino Breddin. *Paraphrasing: Generating Parallel Programs Using Refactoring*, pages 237–256. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[BLH12] Christopher Brown, Hans-Wolfgang Loidl, and Kevin Hammond. Paraforming: Forming parallel haskell programs using novel refactoring techniques. In Ricardo Peña and Rex Page, editors, *Trends in Functional Programming*, pages 82–97, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[dRADFG17] David del Rio Astorga, Manuel F. Dolz, Javier Fernández, and José Daniel García. Supporting advanced patterns in grppi, a generic parallel pattern interface. In *Euro-Par 2017: Parallel Processing Workshops - Euro-Par 2017 International Workshops, Santiago de Compostela, Spain, August 28-29, 2017, Revised Selected Papers*, pages 55–67, 2017.

[GD17] Leonardo Gazzarri and Marco Danelutto. A tool to support fastflow program design. In Sanzio Bassini, Marco Danelutto, Patrizio Dazzi, Gerhard R. Joubert, and Frans J. Peters, editors, *Parallel Computing is Everywhere, Proceedings of the International Conference on Parallel Computing, ParCo 2017, 12-15 September 2017, Bologna, Italy*, volume 32 of *Advances in Parallel Computing*, pages 687–697. IOS Press, 2017.

[GD18] Leonardo Gazzarri and Marco Danelutto. Supporting structured parallel program design, development and tuning in fastflow. *The Journal of Supercomputing*, May 2018.

[MMM$^+$17] Nicholas Malaya, Damon McDougall, Craig Michoski, Myoungkyu Lee, and Christopher S. Simmons. Experiences porting scientific applications to the intel (knl) xeon phi platform. In *Proc.line of the Practice and Experience in Advanced Research Computing 2017 on Sustainability, Success and Impact*, PEARC17, pages 40:1–40:8, New York, NY, USA, 2017. ACM.

[MRR12] Michael McCool, James Reinders, and Arch Robison. *Structured Parallel Programming: Patterns for Efficient Computation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012.

[M1] Mihaela Maliţa and Gheorghe M. Ştefan. Backus language for functional nano-devices. In *CAS 2011 Proceedings (2011 International Semiconductor Conference)*. IEEE press, 2011.

[Pue17] Juan Pedro Bolívar Puente. Persistence for the masses: Rrb-vectors in a systems language. *Proc. ACM Program. Lang.*, 1(ICFP):16:1–16:28, August 2017.

[Rei07] James Reinders. *Intel Threading Building Blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edition, 2007.

[std19] std::variant — cppreference.com, 2019.

[WB94] Clifford Walinsky and Deb Banerjee. A data-parallel FP compiler. *J. Parallel Distrib. Comput.*, 22(2):138–153, 1994.