Placement Strategies: Structured Skeleton Composition with Location Aware Remote Data

Lukas Immanuel Schiller
- research student -

Philipps-Universität Marburg
Fachbereich Mathematik und Informatik
`schiller@mathematik.uni-marburg.de`

Abstract. The additional complexity of dividing the computations of parallel programs into parallelizable parts and mapping these parts onto the available processors asks for a structured solution. Functional programming seems like an auspicious approach to control the additional complexity and numerous functional high-level approaches that reduce complexity exist. Of all the different approaches to parallel programming the concept of algorithmic skeletons is one of the most promising. Not least because skeletons are a known technique to achieve modularity. In this paper we extend algorithmic skeletons with *Placement Strategies*: a functional, structured mechanism to organize coordination of parallel computation placement. Placement Strategies allow to access explicit and semi-explicit placement in a functional style. By doing so we increase the flexibility and clarity of algorithmic skeletons. Example skeletons are implemented using an extension of Eden's Remote Data, that allows for simple skeleton composition and drop-in parallelization of sequential programs. Nevertheless the scheme of Placement Strategies is transferable to other functional languages that use explicit placement. In experimental evaluation we will show the effectiveness of the new skeletons and that the overhead caused by the additional location information is marginal.

Keywords: parallel · functional · language design · algorithmic skeleton composition.

1 Introduction

The additional layer of complexity that parallelism adds to a computation can be reduced to the questions "*What* should be computed *where*?" and "How are the necessary communications organized?". When designing a programming language we have to address these questions from different perspectives. The first perspective opens the question:

"Where and by whom should be decided where a parallel computation is placed?"
The question of the location where a parallel computation is placed is often subordinated to the question of what should be computed in parallel. However,

at some point the decision where a certain computation is actually placed has to be made. The mechanisms to do so are as numerous as parallel systems are. This decision depends on the targeted hardware, the concepts of the programming language, the options of the operating system and the program itself. This decision is for example made by the compiler (Futhark [13]), the OS (POSIX threads [14]), a VM (JVM threads [12]), the RTS (GpH [24]), a library (PFunc [16]), the programmer (MPI[11]) or various combinations of those. From the programmers point of view the different approaches can be categorized in three groups: no influence on the placement is possible, the programmer needs to provide different hints that assist in the decision process, or programmers needs to decide explicitly by themselves where the computation is placed. The question that differentiates the first two groups is:

Should the programmer be able to manipulate the placement of parallel processes? While some concepts try to hide as much as possible of the additional complexity parallelism adds to a program, experience has shown that this is at most feasible for certain restricted classes of problems. On one hand there are situations where the class of problems the programming language aims at is clear-cut and in this case for example a highly optimized compiler could result in better outputs than a manual optimization of the program. Whenever the parallel task and the targeted hardware fit well enough or the knowledge about a specific class of similar problems is sufficient, an adequate placement can often be chosen without interference of the programmer. Good examples for this can be found in the advanced mechanisms used in the various systems for data parallel programming or GPGPU programming (e.g. Futhark [13], accelerator [5]). However, with a wider range of targeted hardware or complexer problems a prespecified analysis is often not sufficient and additional information about the specific problem needs to be passed to the decision-making mechanism (compare for example Repa's fine tuning [17], an example where even in the quite specific domain of data parallel programming good improvements can be gained by hints from the programmer). In general, the programmer's knowledge about the characteristics of a particular program is often a necessary ingredient for a successful parallelization. This is especially the case if the programming language does not aim at a clearly restricted class of problems. Therefore our goal is a programming language that finds a good balance between simplicity and the possibilities of addressing a wide range of programming problems. In the Chapel [6] community a programming language aiming at different kinds of parallelism (data and task parallelism) and stages of parallel hardware (co-processors, multicore processors, distributed computing, . . .) is sometimes called a "multifunctional" programming language. To avoid confusion with the term "functional" we will call such a programming language "multifaceted".

In our opinion in a good multifaceted programming language the programmer should be able to manipulate the placement – but this manipulation can be restricted by the other goals of the language (e.g. functional purity).

Should the programmer be able to determine a specific *parallel process placement?* The missing possibility to express an explicit placement may be considered as a limitation itself. Not only in terms of fine-tuning but also as for some algorithms placement is an elementary component of the algorithm. On the other hand, the possibilities to express explicit placement are often kept basic or they struggle with functional paradigms.

A multifaceted parallel programming language should provide high-level constructs to express parallel problems independent of the specific hardware as well as allow for (hardware or problem specific) fine-tuning. In our experience this fine-tuning is – if the programming language aims on a wide field of application – substantially simplified by the possibility to express an explicit placement, at best in a well structured manner.

Where should the placement decision mechanism be placed and how can the programmer communicate with it? With the balance between simplicity and expressiveness in mind our goal should be a programming language where default implementation of parallel structures exist but the possibility to influence the placement (up to explicit placement) is possible. For this, it is paramount where to place the different parts of the placement mechanism and which possibilities of communication these parts have between each other. In existing languages the hints provided by the programmer can for instance express related tasks, either directly (e.g. architecture aware GpH [1], where boundaries for placements distances in a virtual architecture are hinted by the programmer) or through the data the tasks work on (e.g. HPF's alignment [22], where data fields in different arrays are linked with hints and then placed on the same processor element). But most of these approaches are motivated by a specific hardware setup or again by a specific class of problems and therefore quite restricted in their expressiveness.

In this paper we will use algorithmic skeletons [7] for a structured approach connecting the precise computational control of explicit placement with a high level of abstraction. They implement common computation or communication patterns of parallel algorithms and are both suitable for data and task parallelism. They are a known technique to achieve modularity and many parallel algorithms can be expressed as an instance of an algorithmic skeleton. By doing so the programmer can focus on the algorithm itself, leaving the details of the parallel implementation to the skeleton. They offer the possibility to compose different skeletons to complexer ones and thus allow for different levels of access, suitable for different levels of fine-tuning. Another benefit of algorithmic skeletons is the separation of computation and coordination and skeleton composition unfolds the full potential of this high-level approach.

Furthermore (at least for functional programming languages) a pure functional solution to express parallelism *and* placements is desirable.

Separating the algorithm itself from its coordination of parallel communication has a long tradition in functional programming, with evaluation strategies [21] as the prime example. Yet, the major part of functional parallel algorithmic skeleton concepts use a combination of semi-explicit parallelism and a scheduler for the placement of the parallel computations. But when it comes to parallel fine-tuning the possibilities are limited. In some cases this is answered by the possibility to use different schedulers (e.g. the ParMonad) or the combination of explicit placement with a scheduler [15]. But sometimes good hints for the scheduler result in almost completely restricted schedulers [1]. Especially in the case of high communication costs between processor elements (e.g. if some of the processor elements are located on different computers connected by a relatively slow connection like Ethernet) a good placement is often obvious to the programmer but seldom to the scheduler.

In this paper a concept is presented that enables for flexible and elegant explicit placement. This is done by passing functions that determine the explicit placement – called *"Placement Strategies"* – to algorithmic skeletons as arguments. A specialized data type that carries some information of the current location of an arbitrary piece of data, called "Location Aware Remote Data", is used. Placement Strategies build the bridge between a user-friendly high-level approach and the possibility of expressive fine-tuning. They open the possibility to change the placement systematically when different skeletons with different strategic profiles are combined. These simple ingredients allow for surprising possibilities: by locating the predefined structures of parallelism into a library and allowing to pass even complexer functions we get a solution with functional pureness (no need to use potentially unpure elements like `selfPe`), a high level of abstraction, where the Placement Strategy has more information to determine a placement than many other systems that use hinting. This allows for structured nesting of algorithmic skeletons. In combination with the Eden programming language it will be even possible to combine work-pulling algorithmic skeletons (like a workpool) with work-pushing ones (like a map). With Placement Strategies it is possible to catch the non-determinism in the placement of the work-pulling skeleton and work systematically with the actual placement in a subsequent skeleton.

We will present different illustrative map and divide-and-conquer skeletons with Placement Strategies. The realization of the concept is in the Eden programming language [19,18], a parallel Haskell[20] dialect. However, the conceptual idea itself is independent of a specific programming language. An evaluation section shows the usefulness of this approach.

2 Introductory Example

In the following example a sequential implementation of a merger based on the bitonic sorting network [2,23] shall be parallelized. The structure of the network can be found in Figure 1. Arrow boxes depict comparison elements and lines depict the data flow. We can assume that the computation of a comparison element is expensive enough to justify the parallelization. It is obvious that this algorithm can be parallelized as every comparison element in the same column works on independent data.

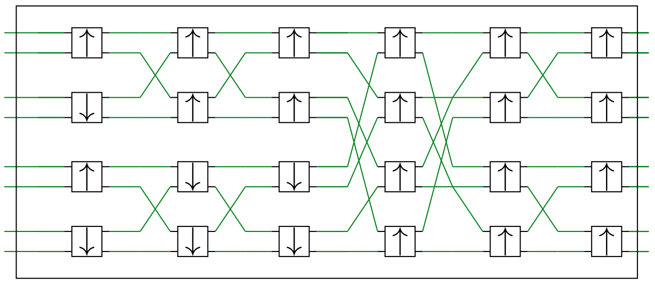


Fig. 1: bitonic sorter of order 8

In the given implementation a map function is applied several times. In every stage some permutations are performed on the input list and then a comparator element is mapped over the permuted list. For this example it is neither relevant which permutations are behind the different functions (and how the permutation functions are implemented) nor is it relevant to understand the structure of the sorting network completely. It is sufficient to know that the comparison element, a function with type `[[a]]` → `[[a]]` is mapped several times. The implementation is given as:

```
res = merger map cElem inp
[...]
merger mapSkel cElem = s4 ∘ s4 ∘ s3 ∘ s2 ∘ s2 ∘ s1 where
    ac = mapSkel cElem
    s1 = perm1out ∘ ac ∘ perm1in
    s2 = perm2out ∘ ac ∘ perm2in
    s3 = perm3out ∘ ac ∘ perm3in
    s4 = perm4out ∘ ac ∘ perm4in
```

For a sequential version of the merger Haskell's `map` function for lists can be used. At best the parallelization is done by simply changing the `map` in the call of the `merger` function to a parallel map. But which requirements would this parallel map have to fulfill?

The map function is one of the most basic algorithmic skeletons. Consequently parallel variants of map exist in almost all parallel Haskell variants. There are several ways to parallelize `map f [a1,..,an]`. Even in the most simple case if we want to create a process for each computation `f a1,..,f an` the strategies to place these processes are numerous. The best strategy to do so depends heavily on the context:

- Where are the elements `a1,..,an` located?
- Where do we need them and/or the computational results next?
- How expensive is the communication between (different) processor elements?

With this in mind we can decide which strategy is the most suitable:

1. In some cases we want to distribute the computations along the available processor elements (e.g. all elements are on the same processor element and we want the computation to be spread across all available processor elements).
2. In some cases we know exactly where we want the computation to take place (e.g. if communication of the computational results is more expensive than the communication of the elements itself, the communication is an inherent aspect of the skeleton or a follow up computation is supposed to be co-located on specific processor elements.
3. In some cases we want the computations to be placed where the data is located (e.g. every element `ai` is already on a different processor element and we do not want to move them because of high communication costs).
4. In some cases a more complex strategy is needed (e.g. "minimize the communication between processor elements but do not place more than three computations on the same processor element").

The first two ways to parallelize the map function are already feasible in Eden. In this paper we will present a safe (in terms of functional pureness) way to realize the last two ways using Placement Strategies. Especially in the fourth case when an arbitrary complex strategy is needed a good interface to express and change the strategy is essential.

We will see that in the above merger example a naive parallel map does not co-locate processes and an explicit placement requires a complete understanding of the different permutations and can result in an inflexible solution or might need deeper changes in the program while a data and strategy guided placement is possible solving the parallelization problem with ease.

3 Eden in a Nutshell

Eden [1] extends Haskell with explicit parallel function application via parallel *processes* with implicit communication.

A process can be instantiated with explicit or implicit placement. A parallel process abstraction can be created and instantiated with explicit placement by the function `instantiateFAt` (read "instantiate function at") where the first argument denotes a processor element (PE) on which the process is instantiated. Processor elements are also called (logical) machines and are numbered from 1 to number of PEs. They usually correspond with the number of CPUs in the system. The process output is in the parallel action monad, thus it can be combined to a larger parallel action.

```
instantiateFAt :: (Trans a, Trans b)
    ⇒ Place                    -- ^PE number
    → (a → b)                  -- ^function for process
    → a                        -- ^process input
    → PA b                     -- ^process output
```

[1] url: `http://www.mathematik.uni-marburg.de/~eden`

The class `Trans` consists of *transmissible* values. `Place` is a type synonym for `Int`. If the first argument of instantiateFAt is 0 the processes are placed round robin on all available processor elements.

`runPA $ instantiateFAt 0 f expr` with some function `f :: a → b` will create a (remote) child process. The expression `expr` will be evaluated (concurrently by a new thread) in the parent process and the result `val` will be sent to the child process. The child process will evaluate `f $ val` (cf. Figure 2).

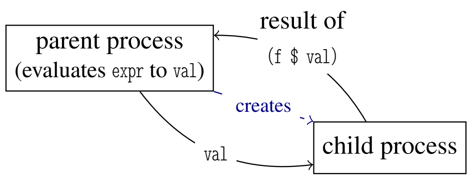


Fig. 2: The scheme of process instantiation. Source: [18]

With `spawnFAt` a version of `instantiateFAt` for structures exists. It expects PE numbers for the placement, functions to be instantiated and input data. `spawnFAt` is a data centric function, thus the PE numbers and functions are cycled and therefore the number of created processes equals the number of elements in the data input.

```
spawnFAt :: (Trans a, Trans b,
    Traversable f, Traversable t1, Traversable t2)
    ⇒ f Place       -- ^PE numbers
    → t1 (a → b)   -- ^function input
    → t2 a          -- ^data input
    → t2 b          -- ^data output
```

For example `spawnFAt [2,6] [(+3), (*5)] [4,7]` will compute `((+3) 4)` on PE 2 and `((*5) 7)` on PE 6. The functions, the input data and the results will be communicated implicitly.

With these basic constructs it is possible to build simple algorithmic skeletons and combine them to more complex ones. A simple parallel map (as our first and most basic skeleton) can be defined using `spawnFAt`. We structure the class of parallel map functions with our ParFunctor class which is a parallel version of the Functor class. Instances of ParFunctor should satisfy the same laws as the `fmap` function from Functor:

```
class ParFunctor f where
    -- | parallel version of @'fmap'@.
    pmap :: (Trans a, Trans b) ⇒ (a → b) → f a → f b
```

Additionally we define a `pmap` with explicit placement:

```
    pmapAt :: (Traversable t, Trans a, Trans b)
        ⇒ t Place   -- ^places for instantiation
        → (a → b) -- ^worker function
        → f a       -- ^tasks
        → f b       -- ^results
```

And the instance for lists fulfills this requirement.

```
instance ParFunctor [] where
  pmap = pmapAt [0]
  pmapAt pos f tasks = spawnFAt pos (repeat f) tasks
```

With these two parallel map functions the merger from the example can be parallelized; `merger pmap cElem` and `merger (pmapAt [1..4]) cElem` would result in the same placement with four processes placed on PE 1, 2, 3 and 4. However the hidden communication of the processes would result in a repeated aggregation of the data on the PE on which the merger function is called between every two stages of the algorithm (cf. Figure 3).

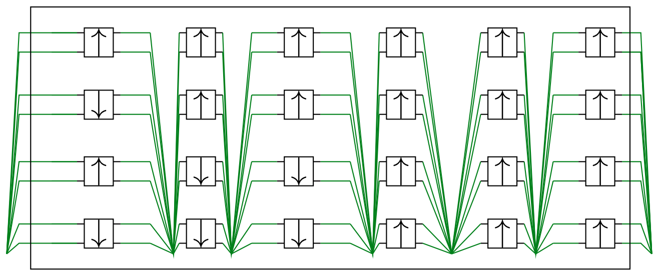


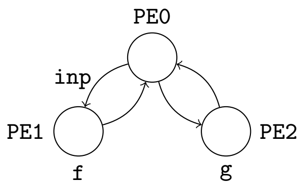
Fig. 3: Actual communication scheme of the parallelized algorithm.

When composing skeletons an efficient connection between output and input is often essential for performance. The Eden programming language provides a sophisticated yet simple and effective concept for a division of computational and coordinational communication which is called Remote Data. The Remote Data [10] concept uses data handles to lighten the data volume of intermediate communication steps by enabling direct communication between both ends of a communication chain, therefore allowing for efficient skeleton composition. Between the skeletons the smaller handle is transmitted instead of the computational data. The actual data is transmitted directly, without the detour. The involved functions converting local data into corresponding Remote Data and back again are:

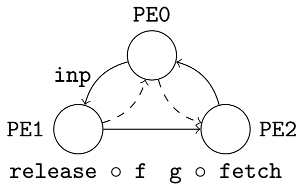
```
release :: Trans a ⇒ a → RD a
fetch :: Trans a ⇒ RD a → a
-- list variants
releaseAll :: Trans a ⇒ [a] → [RD a]
```

```
fetchAll :: Trans a ⇒ [RD a] → [a]
```

In Figure 4 the communication scheme of a Remote Data connection is shown. Two functions, `f` and `g`, are instantiated in succession from the same parent process. Without Remote Data the intermediary result is communicated through the PE where the parent process is located.



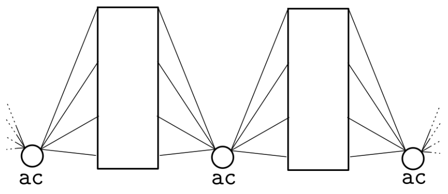
(a) Indirect connection.



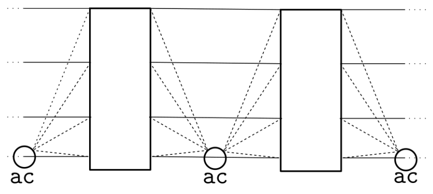
(b) Direct connection.

Fig. 4: Remote Data scheme. Source: [18].With RD a handle is generated on `PE1` and transferred via `PE0` to `PE2`, the actual result is transferred directly.

The separation of the computational and the coordinational communication results in a largely intuitive coordination of the communication while the algorithms stays unchanged. This adaption is often particularly simple. In the introductory example we solely need to wrap the mapped function into `fetchAll` and `releaseAll` (cf. Figure 5).



(a) Intermediary communication without Remote Data using `pmap cElem`.



(b) And with Remote Data using `pmap (releaseAll ∘ cElem ∘ fetchAll)`.

Fig. 5: Introductory Example with and without Remote Data.

If we use this modified version of a parallel map all comparison elements on the same row are placed on the same PE (cf. Figure 6). The function `ac` which calls the parallel map is still located on the PE where the merger is located.

Although this seems like a very good solution it is not an optimal one. If for example the location of the results is not relevant (as it usually is not in distributed computing) a better placement is possible (cf. Figure 7).

Even though this does look like only a small improvement, it is to be noted that this sorting network was already created to be parallelized efficiently with a

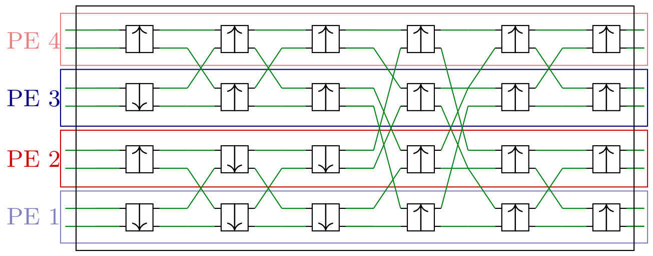


Fig. 6: bitonic sorter with rowise placement

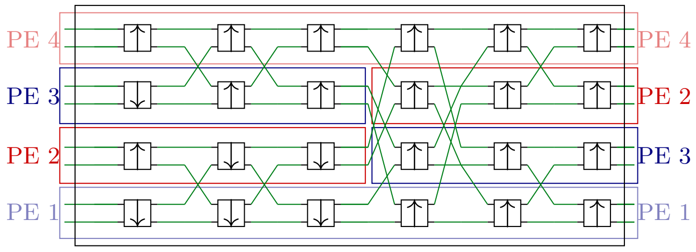


Fig. 7: bitonic sorter with communication minimized placement

row-wise placement. The placement is partially tied to the structure of data (thus hidden in the permutations). If different design considerations are more important (e.g. semantic provability) it is desirable to separate the additional layers of potential parallelization and suitable placement. One could for example think of another sorting network evolved from this one by altered permutations with the same semantic properties but extremely inadequate for a row-wise placement.

Hence, we want to emphasize the usefulness of the disengagement of the different layers of communication. While the Remote Data concept decouples the computational and the data communication a third layer of (independent) placement communication is needed. This is the case whenever the placement is not (completely) given by the data structure. The improved placement could be achieved with the explicit placement of `pmapAt` but at the cost of deep changes to the program. The other possible solution – to change the permutations and represent the new location inside the data structure – is also not preferable because again of its deep intervention into the structure of the program.

Another, more intuitive way to solve this problem is to write algorithmic skeletons which carry the location of the results along. But this solution might be oversized and cumbersome as it changes the type of the results of a skeleton. The additional information could be hidden inside a monadic structure but this again would cause bigger changes.

In this paper we will address this problem by carrying the location of the data along the Remote Data handle and using specialized algorithmic skeletons which benefit from this additional information.

4 Extended Implementation of Remote Data

Dieterle [8] introduced the idea of tagging a remote data with its location. The previous definition

```
type RD a = ChanName (ChanName a)
```

is therefore changed to

```
data RD a = RD {place :: Place,
                rd :: ChanName (ChanName a) }
```

where place is a data field which contains the current position of the Remote Data. This location is the location where the Remote Data is created. We must ensure that the data selector `place` is not misused as it is not purely functional. We are free to use it for process placement but we must not use it somewhere else. One simple solution to do so is to use a hidden data field which is not exported and therefore restrict its use to skeletons only.

The new field can be used for co-located function application in a natural way. For this we will define an instance of ParFunctor for Remote Data as a simple example:

```
instance ParFunctor RD where
    pmap f rd = runPA $
          instantiateFAt (place rd) (liftRD f) rd
    pmapAt places f rd = runPA $
          instantiateFAt (head (toList places)) (liftRD f) rd
```

`liftRD :: (Trans b, Trans a) ⇒ (a → b) → RD a → RD b` lifts a function to Remote Data. Let `rdd = release d` for some data `d`. Then `pmap f rdd` computes `f $ d` on the PE where the Remote Data handle was created.

Analogous to the previously introduced `pmap` for lists we can define a parallel map for functors containing Remote Data with a co-locational placement.

```
rdmap :: (Trans a, Trans b, Functor t, ParFunctor t)
    ⇒ (a → b) -- ^map function
    → t (RD a) -- ^inputs
    → t (RD b) -- ^outputs
rdmap f = fmap (pmap f)
```

The mapped function is placed where the inputs are located. Therefore the inputs need to be distributed already. A possible solution for this is the following function:

```
releaseAt :: (Trans a, Traversable f, ParFunctor t)
    ⇒ f Place       -- ^target locations
    → t a           -- ^input data
    → t (RD a)      -- ^Remote Data handle output
releaseAt places inp = pmapAt places release inp
```

So, `rdmap (+5) ∘ releaseAt [2,7] [3,6]` will compute `((+5) 3)` on PE 2 and `((+5) 6)` on PE 7. Even though this function is extremely useful it is not sufficient for the problem introduced by the introductory example. In the example the input for the map function is a list of lists containing two elements each. So, the optimal placement depends on a nested data structure and is more complex than the simple mapping `rdmap` uses.

5 Example Skeletons and Placement Strategies

In this section we will explore the possibilities of skeletons equipped with Placement Strategies. The Placement Strategy that is passed to the skeleton is a function with three arguments:

```
(ParFunctor f, Traversable t, Trans a)
   ⇒ ((RD a1 → Place)      -- place selector
        → t Place          -- valid target places
        → f a              -- data input
        → t Place)         -- location output
```

The third argument is the data structure containing the Location Aware Remote Data. The places where the data is located can be extracted by the Placement Strategy with the help of RD's data selector "place". The place function itself is (as mentioned before) a hidden field. Thus it needs to be passed to the Placement Strategy (as the first argument) by the skeleton as it is visible to the skeleton but not to the programmer. Note that the data has type '`a`' which might be the same as '`RD a1`' but could also be a nested data structure (with elements of type `RD a1` inside a deeper level). The second argument is a Traversable containing the places ("valid target places") from which the Placement Strategy should select a subset and return it to the skeleton as the placement. This is necessary for a more complex nested parallelism. With this we can define skeletons with a semi-explicit placement that can often be used as drop-in replacements in sequential programs. Many useful strategies are possible and they can become arbitrarily complex.

5.1 Parallel (nested) Map Skeletons

A parallel map with Placement Strategy support is given by the following definition:

```
rdmapPStratT
    :: (ParFunctor f, Traversable t, Trans a, Trans b)
   ⇒ ((RD a1 → Place)      -- placement strategy
        → t Place
        → f a
        → t Place)
   → t Place               -- valid target places
   → (a → b)               -- map function
   → f a                   -- input
```

```
    → f b                        -- output
rdmapPStratT pstrat targets f xs = pmapAt places f xs where
    places = pstrat place targets xs
```

With `rdmap` we introduced a map where the computation follows the data. If we have a list `l = [a1,..,a4] :: [RD a]` and `map place l = [1,3,5,7]` then `rdmap f l` results in four processes located on the PEs 1, 3, 5 and 7. But if we have a different list `l2 = [b1,..,b4]` with `map place l2 = [1,1,2,2]` we could possibly want to locate the four processes of a parallel map again on four different PEs. This can be achieved by explicit relocation with the function:

```
moveTo :: (Trans a, Traversable f, ParFunctor t)
    ⇒ f Place       -- ^target locations
    → t (RD a)      -- ^input Remote Data
    → t (RD a)      -- ^relocated Remote Data
moveTo places inp = pmapAt places (release ∘ fetch) inp
```

The same could be achieved in a more elegant and universal way with a simple strategy. This motivates to articulate the placement in a declarative style which usually results in a solution that is simpler adaptable to changes. We can easily imagine a function `minfreeidx` that returns the balanced list of places for this problem (e.g. `minfreeidx [1,1,2,2] = [1,3,2,4]`). A linear time algorithm to choose the smallest index can be found in [3].

The strategic solution for the introductory example is now obvious. The appropriate strategy takes the smallest non-colliding index for every sublist. If, for example, the Remote Data is located at `[[1,2],[3,4],[1,2],[3,4]]` then the chosen placement is `[1,3,2,4]`. For sorting networks this strategy results in a perfect placement. This means that in every stage at most one process is placed on the same PE (if the number of PEs is at least the width of the sorting network) and communication is minimized. The needed changes to the sequential program are limited to the replacement of the `map` in the function call. A definition of the Placement Strategy for this distribution is given in Appendix A. The potency of this approach becomes visible when fundamentally different strategies are combined. For example work pushing and work pulling becomes compatible: the `workpoolSorted` skeleton [9] of Eden's skeleton library can be perceived as a work pulling implementation of a parallel map. Work pulling is especially suited for heterogeneous problems whose computations are difficult to predict. Placement Strategies can pick up the nondeterminism in the placement introduced by the work pool skeleton.

5.2 Divide-and-Conquer Skeletons

The idea of Placement Strategies is naturally transferable to other (complexer) skeletons. All (explicit) skeletons from Eden's skeleton library can be updated. The necessary adjustments are straightforward and usually the previous skeletons are an instance of the derived skeletons. As an example a distributed divide-and-conquer skeleton with placement strategies has the following type signature:

```
rdPStratDC
  :: (Traversable f, Traversable t, Trans a, Trans b)
    ⇒ ((RD a1 → Place)
         → f Place
         → t a
         → t Int)                    -- placement strategy
      → f Place                       -- tickets
      → (a → Bool)                    -- trivial?
      → (a → b)                       -- solve
      → (f Place → a → t a)           -- split
      → (f Place → a → t b → b)       -- combine
      → a → b                         -- input / output
```

The introductory example can simply be expressed through this skeleton as it is a nested divide-and-conquer algorithm [23]. The Placement Strategy from the previous section can be reused. The following evaluation section shows that the new skeleton is superior to the existing algorithmic skeletons of the Eden skeleton library.

With placement strategies even complex decision mechanisms are possible. One can think of placement strategies that are based on cost models or ones that take the specific hardware situation into account (e.g. heterogeneous hardware).

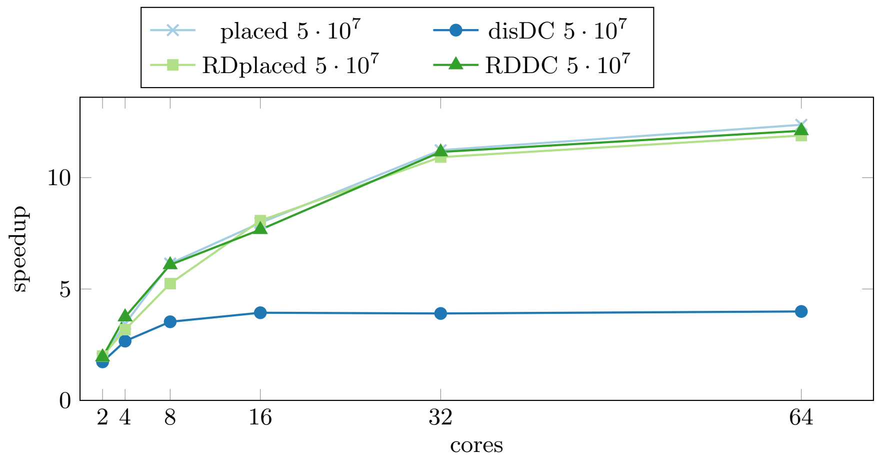
6 Experimental Evaluation

The main goal of algorithmic skeletons is to reduce complexity by condensing recurring algorithmic pattern into a reusable structure. This means that if everything is done right, algorithmic skeletons with Placement Strategies are as fast as the equivalent direct explicit placement, yet easier and faster to use. Their main advantage lies in the structuring of the different layers of parallelism and their possibility to increase the flexibility of skeletons.

In this section we will compare a manually tuned version of an divide-and-conquer implementation of the introductory example with different skeletons. We tested the algorithms on a multicore computer, equipped with an AMD Opteron CPU 6378 (64 cores) and 64 GB memory.

As the following figure shows the manually tuned version called `placed` (with "perfect" placement) shows approximately the same parallel speedup as the manually tuned version that fetches its placement information from the Remote Data (called `RDplaced`) and the new algorithmic skeleton equipped with the previously described Placement Strategy (called `RDDC`). The differences in the speedups are within the usual scattering and the overhead is marginal. On the other hand, the distributed divide-and-conquer skeleton from the Eden skeleton library with the name `disDC` does choose a different placement, which is hard coded into the skeleton and can not be changed without defining a new skeleton. This placement is, in general, a good placement for many different divide-and-conquer algorithms. But for our example the fine-tuning of the Placement Strategy is clearly superior.

In particular, the transferability of the Placement Strategy is a great advantage.



7 Related Work

This section discusses different forms of placement organization and their relation to Placement Strategies, as well as concepts that are structurally connected to the idea of Placement Strategies. Some are linked directly or indirectly to the ideas presented in this paper while others are not connected but compatible. Beside the different functional placement concepts some non-functional concepts use sophisticated forms of placement organization.

Not only the explicitness of expressing placement is treated differently, even the location of the decision-making mechanism is very different in the various parallel systems. The kind and the amount of information accessible by the decision-making mechanism differ as well.

7.1 Implicit Placement

Implicit placement is superior whenever the usefulness of the compiler's knowledge about the specific hardware outweighs the programmer's knowledge about the program. In this case explicit placement becomes uneconomic. This has led to a well justified coexistence of both concepts. While explicit placement is especially popular in distributed computing (with heterogeneous hardware), implicit placement is, for example, very popular in functional data-parallel languages. It is not unusual for modern parallel systems to use a mixture of distributed computing and hardware acceleration, this rises the urge to benefit from both research areas. While explicit and implicit placement are intrinsically opposed concepts, some data-parallel languages organize their parallel expressiveness in specialized functions that are close to the conceptual idea of algorithmic skeletons (e.g. Futhark's Second-Order Array Combinators [13]). This means a good connecting between the concepts with a closely related syntax is possible, where parallelism (and indirect parallel placement) is coordinated with separate, higher-order functions as the organizational units.

7.2 Annotation-based Semi-Explicit Placement

In the area of annotation-based parallelism, attempts to affect the placement exist as well. Closest related to Placement Strategies are approaches where a structured influence on the placement is the goal of the annotation. A common way to influence the placement is through the expression of co-location. For example in High Performance Fortran [22] the keyword "align" can be used to align the data distribution of a data structure with another already distributed data structure. Similar effects can be achieved with Location Aware Remote Data and Placement Strategies. It is possible to adapt the placement to the distribution of one or more distributed data structures. Often the alignment is the gratis result of the Remote Data concept and no additional efforts must be made to achieve co-location.

Another interesting idea is to express (relative) locality, which is very useful if the hardware is inhomogeneous. In an architecture aware variant of GpH [1] the (maximal) relative distance between two computations can be expressed. The explicit equivalent would be a Placement Strategy with a representation of the hardware structure as a function argument. Obviously all kinds of cost-functions and metrics can be used in Placement Strategies.

Despite the different basic principle of annotation-based and explicit concepts, the annotated information can be used in explicit systems as well and Placement Strategies are a high-level access to the underlying reasoning.

7.3 Explicit Placement

Functional Explicit Placement There is a variety of functional languages supporting explicit placement. The usual constructs are the possibility to express a place for a specific computation (Clean's "`@`", Eden's "`instantiateFAt`", Erlang's "`spawn/4`") and a function that returns the current location (Clean's "`self`", Eden's "`selfPE`", Erlang's "`self/1`"). The use of algorithmic skeletons to organize complexer parallel patterns is popular [19,4]. Yet, the placement is usually decided exclusively by the skeleton and the only way to manipulate the placement is to choose a different skeleton or reorganize the data input.

Non-Functional High-Level Explicit Placement Explicit placement is very common in non-functional languages. Apart from a direct placement similar to the explicit placement discussed in the previous paragraph, some concepts for structured placement exist.

MPI [11] provides a rich set of options for process selection and therefore determining placement. However, the focus is on communication models and differs fundamentally from the function-based communication model used in Eden and other functional languages.

Designed for high performance computing Chapel [6] provides a so-called "multifunctional" programming approach. While we agree to almost all of the conceptual ideas presented as the basis of the Chapel programming language the chosen implementations differ. `Domain Maps` are used to express data distribution in a wide-ranging and compositional way.

7.4 Evaluation Strategies

Evaluation strategies [21,24] are a well known technique to control dynamic parallel behaviour through controlling the evaluation degree of an expression. The goal of evaluation strategies is therefore the *"how?"* and not the *"where?"*, the placement is usually delegated to the runtime system. Conceptually evaluation strategies and placement strategies are related in the manner their goals are pursued. Both are written in the same language as the algorithm and therefore extensible by the user and they both try to separate the algorithm from the organization of parallel behaviour.

8 Conclusion and Future Work

The proposed idea of this paper does not preclude other promising approaches. It is desirable to parallelize programs without major adjustments while maintaining the computational communication and the (potentially diverging) parallel coordination separated. In the best case this happens in an functional and elegant way. The vast majority of parallel functional concepts use either scheduler (in the runtime system) or parallel data structures. This paper presents a middle course with the opportunity to change the point of view depending on the used skeleton. By doing so a precise parallelism control is possible with the expressiveness of explicit placement with improved elegance. Whenever a strategy that solely requires the location of the data is appropriate this approach fits best. Even thought all concepts that contribute to this solution are well known, the resulting solution fulfills many different goals at once:

1. functional pureness / functional programming style: in our opinion Placement Strategies fit perfectly into a functional programming style. They encourage the programmer to express a placement as a function and the use of the potentially unpure `selfPe` function becomes unnecessary.
2. high level of abstraction: the combination of algorithmic skeletons and Placement Strategies divide an algorithm into its computational and organizational part
3. high level of manipulation possible: Placement Strategies in combination with Location Aware Remote Data can determine a parallel placement relatively to the actual position and therefore not only co-location of tasks is possible but more sophisticated placement is possible

4. nesting algorithmic skeletons: it is possible to nest algorithmic skeletons in every manner, while Placement Strategies manages the different domains the algorithmic skeletons can work on
5. skeleton composition / catching non-determinism: by locating the organisation of the parallelism into algorithmic skeletons it is possible to combine work-pulling skeletons which result in non-deterministic placement with work-pushing skeletons. A Placement Strategy can cope with the non-deterministic placement either directly by applying a semi-explicit placement which is relative to the non-deterministic placement or transform it into a deterministic one by applying an explicit placement.

This generates the need for a versatile skeleton library with suitable Placement Strategies for different situations.

Appendix A Placement Strategy for Sorting Networks

```
pstratni place targets tasks = nextfreeidx targets pl where
    pl = fmap (fmap place) tasks

nextfreeindex targets xs = fill targets xs' where
  xs' = foldl (λx → nidx targets x x) init (transpose xs)
  init = map (const 0) xs

nidx _  _    _      []     = []
nidx ts used (x:xs) (y:ys) = r:nidx ts (r:used) xs ys where
    r = if x == 0 then r2 else x
    r2 = if y `notElem` used && y `elem` ts then y else 0

fill ts xs = f' (cycle ts) xs' where
    xs' = f' (ts \\ xs) xs
    f' _  [] = []
    f' [] xs = xs
    f' (t:ts) (x:xs) =
      if x == 0 then (t:(f' ts xs)) else (x:f' (t:ts) xs)
```

Appendix B Local Aware Remote Data Definition

Converts local data into corresponding remote data. The result is in the parallel action monad and can be combined to a larger parallel action.

```
releasePA :: Trans a
            ⇒ a              -- ^ The original data
            → PA (RD a)      -- ^ The Remote Data handle
releasePA val = PA $ do
  (cc, Comm sendValC) ← createComm
  fork (sendValC val)
  return RD {rd = cc, place = selfPe}
```

References

1. Aswad, M.K., Trinder, P., Loidl, H.: Architecture aware parallel programming in glasgow parallel haskell (gph). Procedia Computer Science **9**, 1807 – 1816 (2012). https://doi.org/https://doi.org/10.1016/j.procs.2012.04.199, http://www.sciencedirect.com/science/article/pii/S1877050912003201, proceedings of the International Conference on Computational Science, ICCS 2012
2. Batcher, K.E.: Sorting networks and their applications. In: Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference. pp. 307–314. AFIPS '68 (Spring), ACM, New York, NY, USA (1968). https://doi.org/10.1145/1468075.1468121
3. Bird, R.: Pearls of Functional Algorithm Design. Cambridge University Press, New York, NY, USA, 1st edn. (2010)
4. Brown, C., Danelutto, M., Hammond, K., Kilpatrick, P., Elliott, A.: Cost-directed refactoring for parallel erlang programs. International Journal of Parallel Programming **42**(4), 564–582 (Aug 2014). https://doi.org/10.1007/s10766-013-0266-5, https://doi.org/10.1007/s10766-013-0266-5
5. Chakravarty, M.M., Keller, G., Lee, S., McDonell, T.L., Grover, V.: Accelerating haskell array codes with multicore gpus. In: Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming. pp. 3–14. DAMP '11, ACM, New York, NY, USA (2011). https://doi.org/10.1145/1926354.1926358
6. Chamberlain, B., Callahan, D., Zima, H.: Parallel programmability and the chapel language. The International Journal of High Performance Computing Applications **21**(3), 291–312 (2007). https://doi.org/10.1177/1094342007078442, https://doi.org/10.1177/1094342007078442
7. Cole, M.: Algorithmic Skeletons: Structured Management of Parallel Computation. MIT Press, Cambridge, MA, USA (1991)
8. Dieterle, M.: Structured Parallelism by Composition. Ph.D. thesis, Philipps-Universität Marburg (2016). https://doi.org/10.17192/z2016.0107
9. Dieterle, M., Berthold, J., Loogen, R.: A skeleton for distributed work pools in eden. In: Blume, M., Kobayashi, N., Vidal, G. (eds.) Functional and Logic Programming, 10th International Symposium, FLOPS 2010, Sendai, Japan, April 19-21, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6009, pp. 337–353. Springer (2010). https://doi.org/10.1007/978-3-642-12251-4_24, https://doi.org/10.1007/978-3-642-12251-4_24
10. Dieterle, M., Horstmeyer, T., Loogen, R.: Skeleton composition using remote data. In: Carro, M., Peña, R. (eds.) Practical Aspects of Declarative Languages, Lecture Notes in Computer Science, vol. 5937, pp. 73–87. Springer Berlin Heidelberg (2010). https://doi.org/10.1007/978-3-642-11503-5_8
11. Forum, T.M.: MPI: A Message-Passing Interface Standard. Tech. rep., Knoxville, TN, USA (2012), https://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf
12. Gosling, J., Joy, B., Steele, G.L., Bracha, G., Buckley, A.: The Java Language Specification, Java SE 8 Edition. Addison-Wesley Professional, 1st edn. (2014)
13. Henriksen, T., Serup, N.G.W., Elsman, M., Henglein, F., Oancea, C.E.: Futhark: Purely functional gpu-programming with nested parallelism and in-place array updates. SIGPLAN Not. **52**(6), 556–571 (Jun 2017). https://doi.org/10.1145/3140587.3062354, http://doi.acm.org/10.1145/3140587.3062354
14. IEEE: IEEE Std 1003.1-2001 Standard for Information Technology — Portable Operating System Interface (POSIX) Rationale (Informative) (2001), revision of

IEEE Std 1003.1-1996 and IEEE Std 1003.2-1992) Open Group Technical Standard Base Specifications, Issue 6.

15. Jones, Jr., D., Marlow, S., Singh, S.: Parallel performance tuning for haskell. In: Proceedings of the 2Nd ACM SIGPLAN Symposium on Haskell. pp. 81–92. Haskell '09, ACM, New York, NY, USA (2009). https://doi.org/10.1145/1596638.1596649, http://doi.acm.org/10.1145/1596638.1596649
16. Kambadur, P., Gupta, A., Ghoting, A., Avron, H., Lumsdaine, A.: Pfunc: Modern task parallelism for modern high performance computing. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis. pp. 43:1–43:11. SC '09, ACM, New York, NY, USA (2009). https://doi.org/10.1145/1654059.1654103, http://doi.acm.org/10.1145/1654059.1654103
17. Lippmeier, B., Chakravarty, M., Keller, G., Peyton Jones, S.: Guiding parallel array fusion with indexed types. SIGPLAN Not. **47**(12), 25–36 (Sep 2012). https://doi.org/10.1145/2430532.2364511, http://doi.acm.org/10.1145/2430532.2364511
18. Loogen, R.: Eden — Parallel Functional Programming with Haskell. In: Zsók, V., Horváth, Z., Plasmeijer, R. (eds.) Proceedings of the 4th Summer School Conference on Central European Functional Programming School, CEFP 11, Lecture Notes in Computer Science, vol. 7241, pp. 142–206. Springer-Verlag, Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32096-5_4
19. Loogen, R., Ortega-Mallén, Y., Peña-Marí, R.: Parallel functional programming in Eden. J. Funct. Program. **15**(3), 431–475 (May 2005). https://doi.org/10.1017/S0956796805005526
20. Marlow, S.: Haskell 2010 language report (2010), http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.179.2870
21. Marlow, S., Maier, P., Loidl, H.W., Aswad, M.K., Trinder, P.: Seq no more: Better strategies for parallel haskell. In: Proceedings of the Third ACM Haskell Symposium on Haskell. pp. 91–102. Haskell '10, ACM, New York, NY, USA (2010). https://doi.org/10.1145/1863523.1863535, http://doi.acm.org/10.1145/1863523.1863535
22. Rice University, C.: High performance fortran language specification. SIGPLAN Fortran Forum **12**(4), 1–86 (Dec 1993). https://doi.org/10.1145/174223.158909, http://doi.acm.org/10.1145/174223.158909
23. Schiller, L.I.: An agglomeration law for sorting networks and its application in functional programming. In: Schwarz, S., Voigtländer, J. (eds.) Proceedings 29th and 30th Workshops on (Constraint) Logic Programming and 24th International Workshop on Functional and (Constraint) Logic Programming, Dresden and Leipzig, Germany, 22nd September 2015 and 12-14th September 2016. Electronic Proceedings in Theoretical Computer Science, vol. 234, pp. 165–179. Open Publishing Association (2017). https://doi.org/10.4204/EPTCS.234.12
24. Trinder, P.W., Hammond, K., Loidl, H.W., Jones, S.P.: Algorithm + strategy = parallelism. Journal of functional programming **8**(1), 23–60 (1998)