Handling Parallelism in a Concurrency Model

Mischael Schill, Sebastian Nanz, and Bertrand Meyer

ETH Zurich, Switzerland
firstname.lastname@inf.ethz.ch

Abstract. Programming models for concurrency are optimized for dealing with nondeterminism, for example to handle asynchronously arriving events. To shield the developer from data race errors effectively, such models may prevent shared access to data altogether. However, this restriction also makes them unsuitable for applications that require data parallelism. We present a library-based approach for permitting parallel access to arrays while preserving the safety guarantees of the original model. When applied to SCOOP, an object-oriented concurrency model, the approach exhibits a negligible performance overhead compared to ordinary threaded implementations of two parallel benchmark programs.

1 Introduction

Writing a multithreaded program can have a variety of very different motivations [1]. Oftentimes, multithreading is a functional requirement: it enables applications to remain responsive to input, for example when using a graphical user interface. Furthermore, it is also an effective program structuring technique that makes it possible to handle nondeterministic events in a modular way; developers take advantage of this fact when designing reactive and event-based systems. In all these cases, multithreading is said to provide *concurrency*. In contrast to this, the multicore revolution has accentuated the use of multithreading for improving performance when executing programs on a multicore machine. In this case, multithreading is said to provide *parallelism*.

Programming models for multithreaded programming generally support either concurrency or parallelism. For example, the Actor model [2] or Simple Concurrent Object-Oriented Programming (SCOOP) [3,4] are typical concurrency models: they are optimized for coordination and event handling, and provide safety guarantees such as absence of data races. Models supporting parallelism on the other hand, for example OpenMP [5] or Chapel [6], put the emphasis on providing programming abstractions for efficient shared memory computations, typically without addressing safety concerns.

While a separation of concerns such as this can be very helpful, it is evident that the two worlds of concurrency and parallelism overlap to a large degree. For example, applications designed for concurrency may have computational parts the developer would like to speed up with parallelism. On the other hand, even simple data-parallel programs may suffer from concurrency issues such as data races, atomicity violations, or deadlocks. Hence, models aimed at parallelism

could benefit from inheriting some of the safety guarantees commonly ensured by concurrency models.

This paper presents a library-based approach for parallel processing of shared-memory arrays within the framework of a concurrency model. To achieve this, the data structure is extended with features to obtain *slices*, i.e. contiguous data sections of the original data structure. These data parts can be safely used by parallel threads, and the race-freedom guarantee for the original data structure can be preserved.

The approach is applied to SCOOP [3,4], a concurrency model implemented on top of the object-oriented language Eiffel [7]. A performance evaluation using two benchmark programs (parallel Quicksort and matrix multiplication) shows that the approach is as fast as using threads, and naturally outperforms the original no-sharing approach. While SCOOP lends itself well to our approach, the basic idea can be helpful for providing similar extensions to Actor-based models.

The remainder of the paper is structured as follows. Section 2 describes the problem and the rationale of our approach. Section 3 presents the slicing technique. Section 4 provides the results of the performance evaluation. Section 5 describes related work and Section 6 concludes with thoughts on future work.

2 Performance issues of race-free models

To help conquer the complexity of nondeterministic multithreading, programming models for concurrency may provide safety guarantees that exclude common errors by construction. In Erlang [8] for example, a language famous for implementing the Actor model [2], there is no shared state among actors; hence the model is free from data races.

In a similar manner, the object-oriented concurrency model SCOOP [3,4] does not allow sharing of memory between its computational entities, called *processors* (an abstraction of threads, processes, physical cores etc). More specifically, every object in SCOOP belongs to exactly one processor and only this processor has access to the state of the object. A processor can however be instructed to execute a call on behalf of another processor, by adding the call to the processor's *request queue*. Also this regime offers protection from data races.

Unfortunately, the strict avoidance of shared memory has severe performance disadvantages when trying to parallelize various commonplace computational problems. As an example, Listing 1 shows an in-place Quicksort algorithm written in SCOOP. Every time the array is split, a new worker is created to sort its part of the array. The workers `s1` and `s2` and the array `data` are denoted as **separate**, i.e. they reference an object that may belong to another processor. By creating a separate object, a new processor is spawned. Calls to a separate object are only valid if the processor owning the separate object is *controlled* by the current processor, which is guaranteed if the separate object appears in the argument list, hence the `separate_sort`, `get`, and `swap` features. Each proces-

```
data: separate ARRAY[T]
lower, upper: INTEGER

make (d: separate ARRAY[T]; n: INTEGER)
  do
    if n > 0 then
      lower := d.lower
      upper := d.lower + n - 1
    else
      upper := d.upper
      lower := d.upper + n + 1
    end
    data := d
  end

sort
  local i, j: INTEGER; s1, s2: separate SORTER[T]
  do
    if upper > lower then
      pivot := get (data, upper)
      from i := lower; j := lower until i = upper loop
        if get(data, i) < pivot then
          swap(data, i, j)
          j := j + 1
        end
        i := i + 1
      end
      swap (data, upper, j)
      create s1.make (data, j - lower)
      create s2.make (data, j - upper)
      separate_sort(s1, s2)
    end
  end

get (d: separate ARRAY[T]; index: INTEGER): T
  do Result := data[index] end

swap (data: separate ARRAY[T]; i, j: INTEGER)
  local tmp: T do tmp := d[i]; d[i] := d[j]; d[j] := tmp end

separate_sort (s1, s2: separate SORTER[T])
  do s1.sort; s2.sort end
```

Listing 1. SORTER: In-place Quicksort in SCOOP

sor can only be controlled by one other processor at a time, thereby ensuring freedom from data races.

The execution of this example exhibits parallel slowdown: a sequential version outperforms the algorithms for most workloads. This has two main reasons:

1. Every call to the data array involves adding the call to the request queue, removing the call from the request queue, and sending back the result; this creates a large communication overhead.
2. Only one of the workers at a time can execute the `get` and `swap` features on the array because they require control of the processor handling the array; this serialization prohibits the algorithm from scaling up.

The same issues occur in a broad range of data-parallel algorithms using arrays. Efficient implementations of such algorithms are impossible in race-

protective concurrency models such as SCOOP, which is unacceptable. Any viable solution to the problem has to get rid of the communication overhead and the serialization. There are two general approaches to this problem:

1. Weaken the concurrency model to allow shared memory without race protection, or interface with a shared memory language. The programmers are responsible to take appropriate synchronization measures themselves.
2. Enable shared memory computations, but hide it in an API that preserves the race-freedom guarantees of the concurrency model.

The first approach shifts the burden to come up with solutions for parallelism to the programmer. Unfortunately, it also forfeits the purpose of race-protection mechanisms in the first place. Still, it is the most prominent approach taken. This paper presents a solution based on the second approach, in particular offering both race protection and shared memory performance.

3 Array slicing

To allow the implementation of efficient parallel algorithms on arrays, the following two types of array manipulation have to be supported:

- *Parallel disjoint access:* Each thread has read and write access to disjoint parts of an array.
- *Parallel read:* Multiple threads have read-only access to the same array.

The array slicing technique presented in this section enables such array manipulations by defining two data structures, *slices* and *views*, representing parts of an array that can be accessed in parallel while maintaining race-freedom guarantees.

Slice Part of an array that supports read and write access of single threads.

View Proxy that provides read-only access to a slice while preventing modifications to it.

In the following we give a specification of the operations on slices and views.

3.1 Slices

Slices enable parallel usage patterns of arrays is where each thread works on a disjoint part of the array. The main operations on slices are defined as follows:

Slicing Creating a slice of an array transfers some of the data of the array into the slice. If shared memory is used, the transfer can be done efficiently using aliasing of the memory and adjusting the bounds of the original array.

Merging The reverse operation of slicing. Merging two slices requires them to be adjacent to form an undivided range of indexes. The content of the two adjacent slices is transferred to the new slice, using aliasing if the two are also adjacent in shared memory.

Creation procedures (constructors)	
`make`(`n`: **INTEGER**)	Create a new slice with a capacity of `n`
`slice_head`(`slice`: `SLICE`; `n`: **INTEGER**)	Slice off the first `n` entries of `slice`
`slice_tail`(`slice`: `SLICE`; `n`: **INTEGER**)	Slice off the last `n` entries of `slice`
`merge`(`a`, `b`: `SLICE`)	Create a new slice by merging `a` and `b`
Queries	
`item`(`index`: **INTEGER**): `T`	Retrieve the item at `index`
`indexes`: `SET`[**INTEGER**]	Indexes of this slice
`lower`: **INTEGER**	Lowest index of the index set
`upper`: **INTEGER**	Highest index of the index set
`count`: **INTEGER**	Number of indexes: `upper` − `lower` + 1
`is_modifiable`: **BOOLEAN**	Whether the array is currently modifiable, i.e. `readers` = 0
`readers`: **INTEGER**	The number of views on the slice
Commands	
`put`(`value`: `T`; `index`: **INTEGER**): `T`	Store `value` at `index`
Commands only accessible to slice views	
`freeze`	Notifies the slice that a view on it is created by incrementing `readers`
`melt`	Notifies the slice that a view on it is released by decrementing `readers`
Internal state	
`area`: **POINTER**	Direct unprotected memory access
`base`: **INTEGER**	The offset into memory

Table 1. API for slices

Based on this central idea, an API for slices can be defined as in Table 1. Note that we use the letter `T` to refer to the type of the array elements. After creating a new slice using `make`, the slice can be used like a regular array using `item` and `put` with the `indexes` ranging from `lower` to `upper`, although modifying it is only allowed if `is_modifiable` is true, which is exactly if `readers` is zero. Internally, the attribute `area` is a direct pointer into memory which can be accessed like a 0-based array. The `base` represents the base of the slice, which is usually 1 for Eiffel programs, but may differ when a merge results in a copy. The operations `freeze` and `melt` increment and decrement the `readers` attribute which influences `is_modifiable` and are used by views (see section 3.2).

Slicing. Like any other object, a reference to the slice can be passed to other processors. A processor having a reference to a slice can decide to create a new slice by slicing from the lower end (`slice_head`) or upper end (`slice_tail`). By doing this, the original slice transfers data to the new slice by altering the bounds and referencing the same memory if possible. Freedom of race conditions is ensured through the exclusive access to the disjoint parts.

Listing 2 shows an implementation of the `slice_head` creation procedure, taking advantage of shared memory. It copies the `lower` bound, the `base` and the memory reference of the slice `a_original`. It also sets the upper bound

```
slice_head (n: INTEGER; a_original: separate SLICE[T])
  require -- Precondition
    within_bounds: n > 0 and n <= a_original.count
    a_original.is_modifiable
  do
    lower := a_original.lower
    upper := a_original.lower + n - 1
    base := a_original.base
    area := a_original.area
    a_original.lower := a_original.lower + n
  ensure -- Postcondition
    a_original.count = old a_original.count - n
    a_original.lower = old a_original.lower + n
    a_original.upper = old a_original.upper
    lower = old a_original.lower
    upper = a_original.lower - 1
    count = n
    -- "forall i in indexes : item(i) = old a_original.item(i)"
  end
```

Listing 2. Slicing

according to the size `n` of the new slice and increases the lower bound of the original by `n`.

We use Eiffel for our implementation. Eiffel provides preconditions and post-conditions, which we use to make sure only modifiable arrays are altered.

Merging. If a processor has two *adjacent* slices (the lower index of the one equals the upper index of the other plus one), calling `merge` creates a new combined slice. This transfers all the data from the old slices to the new one, making the old ones empty. If the two slices are located next to each other in memory, the transfer simply adjusts the bounds; otherwise, it copies the data into a new slice.

The implementation of merging (see Listing 3) sets the bounds according to the arguments. It then checks whether the two parts are actually next to each other in memory by checking whether the `area` and the `base` are the same. In this case, it copies the base and the memory reference. Otherwise, it allocates new memory and copies all the data of the two arguments. In the end, it empties the two arguments, setting their `count` to 0 by making `lower` = 1 and `upper` = 0.

Strategies for slicing. The most common choice for disjoint index subsets are sets with contiguous indexes. Those subsets can be identified by their lower and upper index and resemble a normal array. A rarer case is to create the disjoint subsets according to another principle. This warrants a different implementation, which is possible by using inheritance. However, current cache architectures limit the usefulness of slices with a size smaller than a cache line.

3.2 Views

Views enable read-only access on arrays. The main operations on views are defined as follows:

```
merge (a_one, a_another: separate SLICE[T])
  require
    a_one.is_modifiable
    a_another.is_modifiable
    one.is_adjacent (a_another)
  do
    lower := a_one.lower.min(a_another.lower)
    upper := a_another.upper.max(a_one.upper)
    if  a_one.area = a_another.area and a_one.base = a_another.base then
      area := a_one.area
      base := a_one.base
    else
      base := lower
      -- "Copy data from the a_one and a_another to area"
    end
    a_another.empty; a_one.empty
  ensure
    lower = old a_one.lower.min(a_another.lower)
    upper = old a_one.upper.max(a_another.upper)
    a_one.count = 0 and a_another.count = 0
    -- "forall i in old a_one.indexes : item(i) = old a_one.item(i)"
    -- "forall i in old a_another.indexes : item(i) = old a_another.item
        (i)"
  end
```

Listing 3. Merging

Creation procedures (constructors)	
`make(slice: SLICE)`	Create a new view on `slice`
Queries	
`original: SLICE[T]`	Slice this view references
`indexes: SET[INTEGER]`	Indexes of this view
`item(index: INTEGER): T`	Retrieve the item at index
`lower: INTEGER`	Lowest index of the index set
`upper: INTEGER`	Highest index of the index set
Commands	
`free`	Disconnects the view from the slice
Internal state	
`area: POINTER`	Direct unprotected memory access
`base: INTEGER`	Offset into memory

Table 2. API for slice views

Viewing Creating a view from a slice copies the bounds and the memory reference into the view. The original slice is no longer modifiable.

Releasing The reverse operation of viewing. If no other views on the same slice exist, it is modifiable again. Also, the view is no longer usable.

The API for views is shown in Table 2. A processor is able to read the slice in parallel by creating a view using the `make` creation procedure. The original slice is then available as the `original` query. This also prevents all further modifications of the array unless the view is released with the `free` procedure. All the other features of views behave exactly like their counterparts in the slices.

```
make (a_original: separate SLICE[T])
  do
    a_original.freeze
    original := a_original
    lower := a_original.lower
    upper := a_original.upper
    base := a_original.base
    area := a_original.area
  ensure
    lower = a_original.lower
    upper = a_original.upper
    not a_original.is_modifiable
    -- "forall i in indexes : item(i) = a_original.item(i)"
  end
```

Listing 4. Viewing

Viewing. Creating a view basically copies the bounds and the memory reference into the view. By increasing the view counter (`readers`) using the `freeze` operation of the slice `a_original`, the original slice is no longer modifiable (see Listing 4). By calling `free` on a view, the view loses its reference to the memory of the slice and the original slice is notified through `melt` that there is one less `reader`.

Releasing. The `free` procedure redirects the `area` to 0 and sets `lower` to 1 and `upper` to 0. Therefore no access is possible at any index. In addition, the number of readers of the original decremented by a call to `melt`. This causes the original to be modifiable again if the number of readers falls to zero. Because of its simplicity, the code is omitted.

4 Performance evaluation

To assess the performance of our approach, we apply it to two benchmark problems: to determine how well our approach works in a divide-and-conquer scenario, we choose a parallel in-place Quicksort algorithm; to determine the raw performance, we use parallel matrix multiplication. In both cases, the extension of SCOOP with the slicing technique is compared with implementations in Eiffel using only threads and without synchronization except a join at the end.

For the performance tests we use a server with four 8-core Intel Xeon E7-4830 processors and 256 GB of RAM. We ran every program 20 times and report the mean value of the running times in Table 3. The source code of the benchmarks is available online[1] as well as the support for slicing in SCOOP[2]. In the following we discuss both benchmarks and their results in detail.

[1] `https://bitbucket.org/mischaelschill/array-benchmarks`
[2] `https://bitbucket.org/mischaelschill/scoop-library`

		Number of cores					
		1	2	4	8	16	32
Quicksort	Slicing	157.4	147.1	81.9	66.4	59.9	59.2
	Threads	158.6	145.1	82.8	68.0	61.5	59.8
Matrix multiplication	Slicing	184.8	95.0	51.2	24.0	14.1	7.3
	Threads	178.0	91.7	46.6	23.6	12.6	7.3

Table 3. Mean running times (in seconds)

4.1 Quicksort

Listing 5 shows the constructor of the Quicksort example implemented using slices instead of regular arrays (compare Section 2). The main difference is the usage of `slice_head` and `slice_tail` instead of storing the bounds in variables. The implementation of `sort` can stay the same, although there is no need for storing the bounds and extra features for swapping and retrieving since data is no longer separate.

```
data: SLICE[T]

make (a_data: SLICE[T]; n: INTEGER)
  do
    if n > 0 then
      create data.slice_head (a_data, n)
    else
      create data.slice_tail (a_data, -n)
    end
  end
```

Listing 5. Quicksort algorithm using slices

For the performance measurement, the Quicksort benchmark sorts an array of size 10^8, which is first filled using a random number generator with a fixed seed. The benchmarked code is similar to listing 5, but also adds a limit on the number of processors used. As evident from Figure 1, the performance characteristics of the slicing technique and threading is almost identical.

4.2 Matrix multiplication

Listing 6 shows a class facilitating parallel multiplication of matrices, using a two dimensional version of slices and views (`SLICE2` and `SLICE_VIEW2`, implemented in a very similar fashion to the one-dimensional version discussed in Section 3.1). The worker is created using `make` which slices off the first `n` rows into `product`. The `multiply` command actually fills the slice with the result of the multiplication of the left and right matrices. Afterwards, the views are decoupled using `free`. Dividing the work between multiple workers and merging the result is left to the client of the worker.

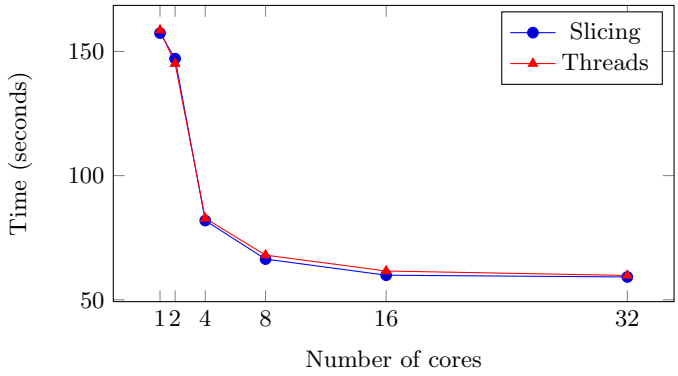

Fig. 1. Quicksort: scalability

```
left, right: SLICE_VIEW2[T]
product: SLICE2[T]

make (l, r, p: separate SLICE2[T]; n: INTEGER)
  do
    create left.make(l); create right.make(r)
    create product.slice_top (n, p)
  end

multiply
  local k, i, j: INTEGER
  do
    from i := product.first_row until i > product.last_row loop
      from j := product.first_column until j > product.last_column loop
        from k := left.first_column until k > left.last_column loop
          product[i, j] := product[i, j] + left[i, k] * right [k, j]
          k := k + 1
        end
        j := j + 1
      end
      i := i + 1
    end
    left.free; right.free
  end
```

Listing 6. Matrix multiplication worker using slices and views

For the performance measurement, the matrix multiplication test multiplies a 2000 to 800 matrix with an 800 to 2000 matrix. Figure 2 shows again similar performance characteristics between slicing and threads.

5 Related work

We are not aware of any programming model supporting slicing while avoiding race conditions. However, similar means to create an alias to a subset of an array's content are common in most programming languages or their standard library. For example, the standard library of Eiffel as provided by Eiffel

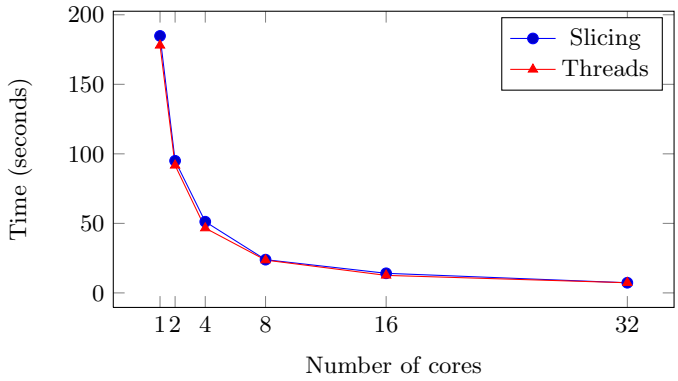

Fig. 2. Matrix multiplication: scalability

Software [7] can create subarrays. Perl [9] has language integrated support for slicing arrays. Slices and slicing are a central feature of the Go programming language [10]. However, these slicing solutions were not created with the intention of guaranteeing safe access: the portion of memory aliased by the new array/slice remains accessible through the original array, which can lead to race conditions if two threads access them at the same time.

Enabling many processors to access different parts of a single array is a cornerstone of data parallel programming models. OpenMP [5] is the de-facto standard for shared-memory multiprocessing. Its API offers various data parallel directives for handling the access to arrays, e.g. in conjunction with parallel-for loops. Threading Building Blocks [11] is a C++ library which offers a wide variety of algorithmic skeletons for parallel programming patterns with array manipulations. Chapel [6] is a parallel programming language for high-performance computation offering concise abstractions for parallel programming patterns. Data Parallel Haskell [12] implements the model of nested data parallelism (inspired by NESL [13]), extending parallel access also to user-defined data structures. In difference to our work, these approaches focus on efficient computation but not on safety guarantees for concurrent access, which is our starting point.

The concept of views is an application of readers-writers locks first introduced by Courtois, Heymans and Parnas [14], tailored to the concept of slices.

6 Conclusion

While programming models for concurrency and parallelism have different goals, they can benefit from each other: concurrency models provide safety mechanisms that can be advantageous for parallelism as well; parallelism models provide performance optimizations that can also be profitable in concurrent programming. In this paper, we have taken a step in this direction by extending a concurrency model, SCOOP, with a technique for efficient parallel access of arrays, without compromising the original data-race freedom guarantees of the model. An im-

portant insight from this work is that safety and performance do not necessarily have to be trade-offs: results on two typical benchmark problems show that our approach has the same performance characteristics as ordinary threading.

In future work, it would be interesting to explore the relation between models for concurrency and parallelism further, with the final goal of defining a safe parallel programming approach. In particular, programming patterns such as parallel-for, parallel-reduce, or parallel-scan could be expressed in a safe manner. In order to ascertain how this API is used by programmers, empirical studies are needed.

Acknowledgments. The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013) / ERC Grant agreement no. 291389, the Hasler Foundation, and ETH (ETHIIRA).

References

1. Agha, G.: Actors: a model of concurrent computation in distributed systems. MIT Press (1986)
2. Armstrong, J., Virding, R., Wikström, C., Williams, M.: Concurrent programming in Erlang. Prentice-Hall, 2nd edn. (1996)
3. Blelloch, G.E.: NESL: A nested data-parallel language. Tech. Rep. CMU-CS-95-170, Carnegie Mellon University (1995)
4. Chakravarty, M.M.T., Leshchinskiy, R., Peyton Jones, S., Keller, G., Marlow, S.: Data parallel Haskell: a status report. In: Proceedings of the 2007 Workshop on Declarative Aspects of Multicore Programming. pp. 10–18. ACM (2007)
5. Chamberlain, B., Callahan, D., Zima, H.: Parallel programmability and the Chapel language. International Journal of High Performance Computing Applications 21(3), 291–312 (2007)
6. Courtois, P.J., Heymans, F., Parnas, D.L.: Concurrent control with readers and writers. Communications of the ACM 14(10), 667–668 (Oct 1971)
7. Dagum, L., Menon, R.: OpenMP: An industry-standard API for shared-memory programming. IEEE Computer Science & Engineering 5(1), 46–55 (1998)
8. Eiffel Software: `http://www.eiffel.com/` (2013)
9. Go programming language: `http://golang.org/` (2013)
10. Meyer, B.: Object-Oriented Software Construction. Prentice-Hall, 2nd edn. (1997)
11. Nienaltowski, P.: Practical framework for contract-based concurrent object-oriented programming. Ph.D. thesis, ETH Zurich (2007)
12. Okur, S., Dig, D.: How do developers use parallel libraries? In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering. pp. 54:1–54:11. FSE'12, ACM (2012)
13. Perl programming language: `http://www.perl.org/` (2013)
14. Reinders, J.: Intel threading building blocks – outfitting C++ for multi-core processor parallelism. O'Reilly (2007)