Bend: Immutability and Native Parallelism as a Solution to Parallel Programming Challenges

Ramon C. Jales de Barros[1], **Samuel Xavier de Souza**[2]

[1]Instituto Metrópole Digital – Universidade Federal do Rio Grande do Norte (UFRN)
Natal – RN – Brazil

[2]Departamento de Engenharia da Computação e
Automação – Universidade Federal do Rio Grande do Norte (UFRN)
Natal – RN – Brazil

ramon.jales.700@ufrn.edu.br, samuel@dca.ufrn.br

Abstract. *With the physical limits of modern processors, parallel programming has become essential to fully harness the potential of current architectures. However, challenges such as race conditions and deadlocks increase the complexity of parallel development. Bend, a functional language with native parallelism, proposes an immutability-based approach to simplify this process. This article analyzes how immutability and native parallelism in Bend eliminate shared memory conflicts, enabling safe and effortless parallelism without explicit synchronization. Through a case study involving tree summation, we compare Bend with a mutable approach in C.*

1. Introduction

Parallel programming is fundamental for exploiting the computational power of modern architectures with multiple processors, such as multi-core CPUs and GPUs. Since the introduction of independent control channels in 1962, which allowed simultaneous execution of programs and I/O operations, concurrency has evolved significantly. In the 1960s, the emergence of multiprocessor computers created new opportunities for developers, while the advent of computer networks in the 1970s and 1980s enabled distributed systems, dividing processing across multiple machines [Andrews 1999].

Despite advances in processor performance, physical limits such as the speed of light impose barriers to further gains. At the same time, the computational demand of modern software continues to grow, often outpacing hardware improvements [Maxwell]. Parallel programming thus arises as a solution to these limitations. However, it introduces significant challenges, such as race conditions and deadlocks [Tanenbaum and Bos 2015]. Traditional solutions, such as locks, semaphores, and atomic operations, help mitigate these issues but at the cost of added code complexity and increased risk of errors [Adelusola 2016].

The Bend language incorporates immutability from the functional programming paradigm, preventing data from being modified after creation [O'Sullivan et al. 2008]. As a result, threads can share memory without interference, eliminating the need for manual synchronization and simplifying parallel program development. Unlike functional languages such as Haskell, where parallel execution depends on specific libraries or extensions (opt-in), or Erlang, where concurrency is explicit and based on processes and

message passing, Bend performs parallelization automatically as an intrinsic part of its runtime. The *High-Order Virtual Machine 2* (HVM2) transparently distributes any non-sequential computation across multiple threads and GPUs, without pragmas or explicit synchronization, enabling a more seamless and scalable development model. This article investigates how immutability and native parallelism in Bend facilitates the design of parallel algorithms. Through a case study, we demonstrate the advantages of Bend compared to mutable approaches like C and discuss its limitations.

2. Theoretical Background

2.1. The Bend Language and HVM2

Bend is a high-level functional programming language designed to combine the simplicity of Python syntax with the scalability of languages like CUDA [Bend Team 2025]. It runs on the HVM2, a compiler that uses interaction nets for parallel evaluation [Taelin 2025]. Bend enables automatic execution of programs on thousands of threads across CPUs and GPUs, without requiring the programmer to explicitly manage parallelism. This feature is particularly beneficial for algorithms that are not "inherently sequential," allowing full exploitation of parallel architectures while abstracting thread management.

This mechanism contrasts with compilers such as *Glasgow Haskell Compiler* (GHC), which require explicit annotations or library support to exploit parallelism. In Bend, parallel execution is a native feature: any expression with independent subcomputations is automatically distributed by HVM2. Moreover, its transparent integration with both GPUs and CPUs in a single execution model distinguishes it from earlier solutions limited to CPU execution or from languages with explicit concurrency models, such as Erlang.

2.2. Immutability in Bend

Immutability is a central principle in Bend, inspired by functional languages such as Haskell. In this paradigm, the values of variables or data structures cannot be modified after creation [O'Sullivan et al. 2008]. To make changes, a new variable or structure is created, leaving the original unchanged. This is illustrated in Figure 1, where the creation of a new variable does not affect the original one.

```
1 let x = 1;
2 let new_x = x + 1; # x remains unchanged
```

Figure 1. Example of immutability in Bend: creating a new variable.

Persistent data structures are essential to support immutability, as they retain previous versions of data by reusing unchanged memory segments, avoiding full duplication[Okasaki 1999]. This approach enhances efficiency, as demonstrated by Dickerson in [Dickerson 2020], who showed that persistent structures enable safe concurrent execution by allowing multiple versions to coexist without requiring explicit synchronization.

In Bend, all variables are immutable by definition, therefore preventing concurrent modifications to shared memory[Dung and Hansen 2011]. Abstractions such as `fold`

and `bend` leverage this property, enabling safe parallel consumption and construction of recursive structures[Bend Team 2025]. Moreover, immutability can boost performance in parallel programs: Dung and Hansen [Dung and Hansen 2011] reported speedups of over 20× by optimizing cache usage in functional structures for Presburger Arithmetic.

Thus, immutability not only simplifies reasoning about concurrency but also enhances the scalability of algorithms.

2.3. The Fold Operator

The `fold` operator is a functional abstraction that reduces a recursive structure—such as a tree—into a single value by processing substructures in parallel. It replaces data constructors with programmer-defined operations, serving as a universal pattern for structure traversal [Bend Team 2025]. The example below demonstrates the sum of values in a binary tree:

```
def sum(tree: Tree(u24)) -> u24:
  fold tree:
    case Tree/Leaf:
      return tree.value
    case Tree/Node:
      return tree.left + tree.right
```

Figure 2. Summing values in a binary tree using `fold`.

Here, each `Tree/Node` is replaced by the sum of its subtrees, and each `Tree/Leaf` by its value. The HVM2 compiler automatically parallelizes subtree evaluations, ensuring safe memory sharing since immutable data cannot be modified by concurrent threads [Taelin 2025].

2.4. The Bend Operator

The `bend` operator, on the other hand, is used to construct recursive structures in parallel, functioning as the opposite of `fold`. Starting from an initial state, it builds layers of a structure until a stopping condition is met [Bend Team 2025]. The following example generates a complete binary tree:

```
def main() -> Tree(u24):
  bend x = 0:
    when x < 3:
      tree = ![fork(x + 1), fork(x + 1)]
    else:
      tree = !7
  return tree
```

Figure 3. Generating a complete binary tree using `bend`.

In this example, `bend` starts with an initial state (`x = 0`) and, while `x < 3`, recursively creates new nodes using parallel calls with `fork`. When `x` reaches 3 or more,

it creates a leaf with the value 7. The result is a binary tree of depth 3. Since all data in Bend is immutable, no state is shared between threads, allowing safe and concurrent structure generation without explicit synchronization.

HVM2 optimizes the execution of both `fold` and `bend`, distributing computations across threads without requiring synchronization [Taelin 2025]. Immutability eliminates race conditions, since no thread can modify data accessed by another, greatly simplifying parallel algorithm development. The key point is that HVM2 automatically detects `bend` and `fold` abstractions and generates parallel tasks for their independent subcomputations, distributing them efficiently across available processors.

3. Case Study: Tree Summation

To evaluate the benefits of immutability in Bend, we examine a tree summation algorithm implemented using the `sum` function shown in Figure 2.

For comparison, we implemented the same algorithm in C using POSIX Threads (pthreads), where shared memory requires synchronization to avoid race conditions. The C implementation is shown in Figure 4.

```
1  void* sum_tree(void* arg) {
2    Tree* tree = (Tree*)arg;
3    if (!tree) return NULL;
4
5    pthread_t left_thread, right_thread;
6    if (tree->left) pthread_create(&left_thread, NULL, sum_tree,
       tree->left);
7    if (tree->right) pthread_create(&right_thread, NULL, sum_tree,
       tree->right);
8
9    if (tree->left) pthread_join(left_thread, NULL);
10   if (tree->right) pthread_join(right_thread, NULL);
11
12   pthread_mutex_lock(&mutex);
13   global_sum += tree->value;
14   pthread_mutex_unlock(&mutex);
15 }
```

Figure 4. Summing values in a binary tree in C with POSIX Threads (code reduced for illustration).

In contrast, the Bend implementation (Figure 2) requires no thread or synchronization management. Thanks to immutability, subtrees are processed independently, and HVM2 **parallelizes execution automatically**. Memory sharing is safe due to immutable data, eliminating the need for mutexes or atomic operations. Bend thus significantly reduces code complexity while maintaining correctness and scalability.

Beyond the tree-sum example, Bend has also demonstrated significant performance in more complex scenarios. In the Bitonic Sort algorithm, as described in the [Bend Team 2025], a speedup of up to 51× was observed when running on a GPU

(NVIDIA RTX 4090) compared to single-threaded execution. Another relevant case is immutable "pixel shading" image rendering, achieving over 40,000 MIPS with optimizations applied to the generated CUDA code, showing that the approach extends to both graphical and scientific workloads. These cases reinforce that the elimination of shared state, combined with HVM2's automatic parallelization, benefits not only didactic examples but also large-scale applications.

4. Discussion

Immutability in Bend eliminates the need for managing shared memory conflicts, allowing parallel algorithms to be expressed concisely and safely. Unlike traditional languages that require explicit thread management, synchronization primitives, or lock mechanisms, Bend provides native support for parallel execution. This model enables developers to express parallelism declaratively, without the overhead and complexity commonly associated with concurrent programming. As a result, algorithms written in Bend can be both scalable and easier to reason about, since the absence of shared mutable state avoids race conditions and deadlocks by design. Persistent data structures further enhance this model by allowing memory to be reused across versions efficiently, minimizing the cost of maintaining immutability [Okasaki 1999].

Despite its advantages, Bend has practical limitations. As a new language, it lacks the mature ecosystem of libraries found in languages like Python or C, which can hinder the development of complex systems. Additionally, the HVM2 compiler is still under development, and the current 4GB memory limit on GPUs constrains data processing size [Bend Team 2025]. These limitations, however, are related to the project's maturity rather than the language design itself.

Compared to languages like C, Bend offers high-level syntax similar to Python with scalability comparable to low-level languages. The `GUIDE.md` documentation demonstrates that algorithms like Bitonic Sort achieve speedups of up to 51x on GPUs, indicating that immutability does not compromise performance [Bend Team 2025]. Thus, Bend is suitable for complex parallel algorithms, including sorting, graphics rendering, and scientific computing, provided data fits within memory constraints.

5. Conclusion

This article has shown that immutability and native parallelism in Bend simplifies parallel programming by eliminating shared memory conflicts such as race conditions and deadlocks. Through a case study on tree summation, we demonstrated that abstractions like `fold` enable concise expression of parallel algorithms, while the HVM2 compiler manages parallel execution automatically. Compared to a mutable C implementation, Bend significantly reduces code complexity and removes the need for explicit synchronization, enhancing scalability. Despite current limitations such as compiler immaturity and memory constraints—Bend stands out as a promising approach for scalable, high-performance parallel algorithms. Distinguishing itself from common functional languages such as Haskell, which do not feature native parallelism.

References

Adelusola, M. (2016). Scalability challenges in fault-tolerant parallel computing algorithms. —, pages —.

Andrews, G. R. (1999). *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison-Wesley, Reading, MA.

Bend Team (2025). Bend programming language: Guide.md. Available at: https://github.com/HigherOrderCO/Bend/blob/main/GUIDE.md. Accessed: 19 June 2025.

Dickerson, T. (2020). *Persistent Data Structures for Efficient Concurrency*. PhD thesis, University of Maryland, College Park, MD. Available at: https://drum.lib.umd.edu/handle/1903/26331.

Dung, P. A. and Hansen, M. R. (2011). A parallel approach to decision procedures for presburger arithmetic. In *Proceedings of the 12th International Conference on Computer Aided Systems Theory (EUROCAST)*, pages 223–230. Springer. Available at: https://backend.orbit.dtu.dk/ws/portalfiles/portal/103220584.

Maxwell (—). Computação paralela. Available at: https://www.maxwell.vrac.puc-rio.br/16409/16409_5.PDF. Accessed: 19 June 2025.

Okasaki, C. (1999). *Purely Functional Data Structures*. Cambridge University Press, Cambridge, UK.

O'Sullivan, B., Goerzen, J., and Stewart, D. B. (2008). *Real World Haskell*. O'Reilly Media, Sebastopol, CA.

Taelin, V. (2025). Hvm2: A parallel evaluator for interaction combinators. Available at: https://github.com/HigherOrderCO/HVM/blob/main/paper/HVM2.pdf. Accessed: 19 June 2025.

Tanenbaum, A. S. and Bos, H. (2015). *Modern Operating Systems*. Pearson, Upper Saddle River, NJ, 4 edition.