Evaluating Functional Memory-Managed Parallel Languages for HPC using the NAS Parallel Benchmarks

Michael Wilkins*, Garrett Weil*, Luke Arnold, Nikos Hardavellas†, Peter Dinda†
Northwestern University, Evanston, IL USA
*{wilkins, gsw}@u.northwestern.edu, †{nikos, pdinda}@northwestern.edu

***Abstract*—Functional, memory-managed parallel languages (FMPLs) are a recent innovative approach to shared-memory parallel programming. Despite their rising prevalence in other areas, FMPLs have yet to gain traction in HPC. In this work, we explore the utility of FMPLs for HPC by re-implementing the NAS Parallel Benchmarks in an FMPL.**

For this study, we ported the benchmarks into the Parallel ML language. We discuss the advantages and disadvantages of using Parallel ML for HPC applications based on our development experience. We compare the performance of our Parallel ML implementation to the existing C/OpenMP version. The FMPL implementations are 1.02×-5.76× slower compared to OpenMP. Our positive development experience combined with some competitive performance results suggest that FMPLs have *the potential* to become a viable choice for HPC applications. We conclude by describing our future work to automatically manage *distributed* memory within an FMPL, creating a compelling new programming model for HPC.

I. Introduction

Functional Memory-managed parallel languages (FMPLs) are, as the name implies, memory-managed, functional programming languages that support nested parallelism. Example FMPLs include NESL [9], parallel Haskell [17], and Parallel ML [16], [30]. FMPLs make parallel programming easier by automatically managing memory. In addition, their functional semantics strictly control data mutation, helping users avoid the painful race conditions that are notorious in parallel programming [5]. FMPLs support parallelism through high-level parallelism constructs (e.g., fork-join, parallel-for loops, parallel recursion, and nested data parallelism). These constructs are already popularized in high performance computing (HPC) by tools like OpenMP [13]. All together, FMPLs enable high-level, "fearless" parallel programming.

FMPLs are growing in popularity in other communities, but not HPC. In academia, FMPLs are actively being developed and improved by programming language researchers [30], [6]. The FMPL we utilize in this work, Parallel ML, is a member of the ML language family, which has been integral to programming language research for multiple decades [22]. Industry has also adopted these languages. The most prominent example is the trading company Jane Street, who contribute research and use FMPLs in production code [27], [28]. Lastly, FMPLs are becoming part of the required cirricula of top computer science programs. For example, Carnegie Mellon University teaches algorithms to over 500 undergraduates per year using Parallel ML.

On the other hand, the HPC community has so-far largely ignored FMPLs. However, HPC developers have shown a willingness to consider new parallel programming models and languages both in the past (e.g., High Performance Fortran [21], Coarray Fortran [23]) and present (e.g., SYCL, Chapel [12], modern Fortan [24], etc.). Meanwhile, there is a nascent trend in the HPC community pushing towards modern high productivity languages such as Python [33] to avoid the growing complexity of modern supercomputers.

On paper, FMPLs have the potential to be the best of both worlds. Their functional semantics and strict data/memory control enable powerful optimizations that aim to match the performance of traditional, lower-level languages [30], [29], [31]. Additionally, FMPLs, like other high-level languages, allow expression of the parallelism available in the algorithm without direct reference to underlying threads/hardware, increasing productivity and portability. In general, as supercomputers become more complex, so too will the challenge of programming them. FMPLs may be a natural fit to assist developers with these emerging challenges.

In this paper, we set out to study the intersection between HPC and FMPLs and better understand the benefits and drawbacks of FMPLs for HPC applications. We implement the most well-known and widely-studied benchmark suite for HPC, the NAS Parallel Benchmarks (NPB), in a state-of-the-art FMPL, Parallel ML. Specifically, we implement the five kernels from NPB shown in Table I: EP, FT, CG, IS, and MG. Current FMPLs are designed for shared-memory parallel programming, so we compare our FMPL benchmarks with an existing C+OpenMP NPB implementation [2], since OpenMP is the predominant programming model for shared-memory parallelism in HPC today. We describe our development experience and the major pros and cons we encountered.

We analyze the relative performance in detail, varying the number of processors used, the input size, and the parallelism granularity. We found that FMPLs implementations were 1.02×-5.76× slower than the OpenMP versions, depending on the benchmark. A minimum slowdown of 1.02× (from the EP benchmark) indicates that for massively parallel HPC applications, FMPLs can be performance-competitive with lower-level implementations. On the other hand, a worst-

case slowdown of 5.76× (from the MG benchmark) shows the performance issues of FMPLs, particularly for applications with computationally-intense sequential sections. Overall, these performance numbers as a whole are a promising sign for the future of FMPLs in HPC, as further research will narrow the performance gap.

Another main outcome of this work are the FMPL NPB implementations themselves, which we provide as open source. These implementations serve as a reference for HPC developers curious what HPC applications look like when written in an FMPL like Parallel ML. In addition, the FMPL research community can utilize them for evaluating their future innovations from an HPC perspective.

Our findings are promising for shared-memory parallelism, but there is an obvious lack of support for distributed memory, which is necessary for production HPC applications. We conclude the paper by describing our future work, which aims to add distributed memory parallelism to FMPLs without altering the easy-to-use programming model.

We summarize our contributions as follows.

- An anatomy of the development experience for HPC developers leveraging high-level parallel languages for shared-memory multiprocessing;
- Performance analysis showing that FMPLs can be competitive (as little as 1.02× slowdown) with C/OpenMP;
- Complete NPB implementations in Parallel ML for HPC developers to reference and language developers to use for performance evaluation from an HPC perspective.

II. Background

Here we describe the applications we work with (the NAS Parallel Benchmarks) and the FMPL we target (Parallel ML).

A. NAS Parallel Benchmarks

Benchmark History/Relevance: Stemming from NASA's Numerical Analysis Simulation (NAS) Program, the NAS Parallel Benchmarks (NPB) were developed to test supercomputers. The NAS Parallel Benchmarks were designed to mimic the core kernels of computational fluid dynamics programs. They are formally described as algorithms, meaning any implementation that properly executes the implementation-independent steps is considered valid. This design enables cross-language comparison; two implementations that both adhere to the algorithm can be directly compared.

The original NPB 1.0 specification has been cited nearly 4000 times, demonstrating that NPB are a widely-used tool for assessing HPC performance [7]. Since NPB 1.0's release, several versions have been developed. We specifically study the five kernels shown in Table I from the third-party NPB 3.0 C+OpenMP implementation [4], [2].

Overall, the NAS Parallel Benchmarks and their implementations are a reliable tool to determine the effectiveness of different HPC programming techniques. We therefore proceed with NPB as our target applications for evaluation.

Benchmark Descriptions: Next, we describe each kernel, their primary data structures, and opportunities for parallelism.

TABLE I: NPB kernels that we re-implement in this work

Abbreviation	Descriptive Name
IS	Integer Sort
EP	Embarrassingly Parallel (Random Number Generation)
CG	Conjugate Gradient
MG	Multi-Grid Solver
FT	3-D Fast Fourier Transform

The Embarrassingly Parallel (EP) benchmark generates pairs of Gaussian Deviates. The main data structure is a one-dimensional array containing floating point values. EP's main parallelism comes from generating the pairs in parallel, with a parallel reduction to collect results. As its name suggests, nearly all of EP's calculations are parallelizable, meaning it has the most fine-grain parallelism in the suite.

The Fourier Transform (FT) benchmark solves a partial differential equation using fast fourier transforms. FT works with a three dimensional arrays which contain complex numbers. FT's parallelism comes from parallel loops used to iterate over the working arrays. FT's arrays are constant size, so the benchmark exposes a large amount of parallelism.

The Conjugate Gradient (CG) benchmark utilizes the inverse power method to estimate the largest eigenvalue of a sparse matrix. The main data structures are 1-D floating-point arrays. The majority of CG's parallelism comes from parallel iteration over the arrays and parallel reductions.

The Integer Sort (IS) benchmark sorts a pseudo-random set of integers. The main data structures are 1-D arrays of integers. The primary parallelism opportunity is iterating over the integer set in parallel during the sorting algorithm. IS contains a decent amount of parallelism, but its execution pattern is irregular, as the parallel sections are interrupted by sequential, "master" thread-only sections.

The Multi-Grid (MG) benchmark performs several iterations to approximate a solution to a Poisson problem on a 3-D array. The main data structures are 3-D floating-point arrays of various sizes. MG's parallelism comes from parallel loops which iterate through the dimensions and values of the arrays. Due to its complex operations on large arrays, MG contains the least/coarsest-grained parallelism in the benchmark suite.

B. Parallel ML

Next, we describe our target FMPL: Parallel ML. We discuss why we chose Parallel ML over other FMPLs and detail how to write programs using its the unique characteristics.

Why Parallel ML: We initially considered 3 modern FMPLs: Data Parallel Haskell (DPH), Multicore OCaml (MOC), and Parallel ML (PML). We eliminated DPH because it is a pure-functional language, meaning that all data is immutable. As described in Section II-A, the NPB kernels rely on arrays. In a pure functional setting (no mutations), array manipulations involve copies. This approach causes significant slowdowns because updating an array field is not as simple as updating (i.e., mutating) the proper memory location.

MOC and PML support mutable arrays and are considered generally performant [27], [30], [6]. For this work, both languages are acceptable choices. However, as we describe in Section V, we plan to make significant changes to the language

runtime in our future work. We chose PML over MOC because its design and underlying runtime are more research-focused, whereas MOC's underpinnings are unnecessarily complex for our purposes. Note that both languages are ML derivatives, meaning it is straightforward to translate our findings to MOC; in this spirit, we refer to FMPLs generally when possible to indicate our results are not language-specific.

We specifically employ an important dialect of Parallel ML [30], [6]. Parallel ML is a memory-managed, functional language with support for data mutation (i.e., side effects) and nested parallelism. Parallelism in PML is exposed through a variety of high level parallelism constructs. We compile our programs with MPL, which is a parallel extension of the MLton compiler [1] for the sequential Standard ML language.

Programming in Parallel ML: Here we describe Parallel ML features that are unique compared to traditional HPC languages like Fortran and C/++. Throughout this section, we refer to line numbers from the example in Figure 1.

Immutability by Default: By default, nearly all data in Parallel ML is immutable. Immutable data are values that cannot be changed/updated. In Parallel ML, we declare immutable values as shown starting at line 1. Primitive types like *int* and *real* (i.e., a double in C/C++) are immutable by default. Simple tasks in other languages, such as incrementing an *int*, cannot be performed in Parallel ML. Instead, the runtime will store the result of the summation in a new location, even if the programmer uses the same name.

On the other hand, mutable values can be updated. Starting on line 9, we showcase the two forms of mutable data in Parallel ML: *refs* and *arrays*. On line 22, we update the value at the first index in an array. Note that the return value is stored in "_", meaning the function does not have a useful return value. Instead, the main result is the function's "side-effect": updating the array value.

Functions: As the name "*functional* language" suggests, functions play an out-sized role in Parallel ML compared to lower-level languages. Functions are treated as first-class citizens alongside other types like int and $real$. In practice, this idea means that functions can be stored in variables and used as input arguments and return values. A simple function that increments an array element is shown on line 28.

incrElem takes advantage of the common "let-in-end" structure. This structures enables scoped variables, meaning values like *new* are only valid between "in" and "end".

To fully understand the power of functions in Parallel ML, consider the function shown on line 36. *forLoop* implements a for-loop. By default, Parallel ML does not include an implementation of a sequential for-loop. Instead, programmers use recursion, as we show in our for-loop example.

The *forLoop* function takes another function, *f*, as an input argument. The closest analogue to this behavior in C is passing a *void**. Because it includes another function, *forLoop* is considered a "higher order function". We proceed to use *forLoop* to increment all of *array1* on line 44.

Parallelism: To perform array increments in parallel, we simply swap out our for-loop for the builtin parallel for-loop,

```
1  (*** Immutable Data Types ***)
2
3  (* int *)
4  val i = 4
5
6  (* real (double) *)
7  val a = 0.0
8
9  (*** Mutable Data Types ***)
10
11 (* ref (pointer) to int *)
12 val pointer = ref 0
13 val _ = pointer := 1 (* updated ref
       *)
14 val contents = !pointer (*
       dereferenced ref *)
15
16 (* 4-element array init to 0.0 *)
17 val array1 = Array.array(i, a)
18
19 (* Reading first element in array *)
20 val first = Array.sub(array1, 0)
21
22 (* Updating first element in array *)
23 val _ = Array.update(array1, 0, 1.0)
24
25
26 (*** Functions ***)
27
28 (* Basic func: increment array elem
       *)
29 fun incrElem(array, index) =
30   let
31     val new = Array.sub(array, index)
           + 1
32   in
33     Array.update(array, index, new)
34   end
35
36 (* Higher order func: for-loop *)
37 fun forLoop((i, j), f : int -> unit)
       =
38   if i >=j then ()
39   else (
40     f(i);
41     forLoop((i+1, j), f))
42
43 (* Using HOF: incrementing array *)
44 val len = Array.length(array1)
45 val _ = forLoop((0, len), fn i =>
46   incrElem(array1, i))
47
48 (* Parallelizing array increment *)
49 val G = 1
50 val _ = ForkJoin.parfor G (0, len) (
       fn i => incrElem(array1, i))
```

Fig. 1: Basic Parallel ML examples

ForkJoin.parfor, as shown on line 49. *ForkJoin.parfor* dynamically executes the loop in parallel using work stealing. It only takes one additional parameter: grain size, for which we use *G*. Grain size allows the user to easily control the granularity of parallelism by changing the size of computational units in the application. Grain size has a significant impact on performance, which we show in Section IV.

Parallelization in FMPLs is simpler than OpenMP because of the dynamic parallelization and memory management scheme performed by the language runtime. In FMPLs, there is no user-level concept of threads. Instead, the runtime uses work stealing to distribute parallel computation. During this process, the runtime is responsible for managing memory across the parallel threads. Therefore, important manual optimizations in OpenMP, such as declaring variables as thread-private or shared, are automated by the FMPL runtime.

Parallel ML also recently added support for select high-level parallel operations on data collections [29]. We describe how we specifically utilize one of these operations in Section III-A.

III. Development Experience

In this section, we describe the main advantages and disadvantages of writing HPC programs in FMPLs based on our experience porting the NPB Benchmarks to Parallel ML.

A. Pros

Functional Programming: The first and most apparent feature of FMPLs is functional programming. The primary advantage of this approach is high-level legibility and abstraction. Every expression within an FMPL is a function with inputs and outputs, which means that the flow and purpose of a new piece of code can be easily read and understood. High-level functional languages also include rich semantics for user-defined datatypes. For example, in FT, we created custom datatypes and functions to elegantly manipulate 3-D arrays of complex numbers.

Automatic Memory Management: Another major advantage of high-level functional languages from our experience is automatic memory management. All allocations and frees are handled by the language runtime. Automatic memory management is particularly attractive for HPC applications, as HPC memory systems are becoming increasingly complex (e.g., NUMA architectures, heterogeneous accelerators, etc).

High-Level Parallel Constructs: Parallel ML's support for nested parallelism includes simpler constructs such as *parallel for* loops popularized by OpenMP and more complex parallel data operations such as reduce, map, and filter [30], [29]. In our NPB implementations, we used these operations to expose the same parallelism opportunities as the C code.

The main advantage of these constructs in our eyes is their obviation of critical manual optimizations in OpenMP like shared vs. private variables, which can be difficult for developers to implement in complex scenarios. We also had a positive experience with some of the more complex constructs. Particularly, we frequently used the data-parallel reduction to efficiently parallelize regions of EP, CG, and MG.

Foreign Function Interface: Foreign Function Interface (FFI) support was critical to our development process. The FFI in PML allows full bidirectional interoperability, meaning PML programs can directly call C functions and vice versa. In our implementations, we used the FFI to port non-performance-critical code (e.g., random number generation and data initialization). The FFI allowed us to focus on the other big advantages of using an FMPL while avoiding some pitfalls (as we discuss in Section III-B). Additionally, we used the FFI to port the benchmarks incrementally, maintaining functionality while iteratively expanding the scope of the PML section. We were able to routinely test our code instead of rewriting the entire benchmark from scratch. We note that using the FFI creates the opportunity for mixed language programs, which introduces concerns about maintainability.

File I/O: File I/O is a prevalent and important component of modern HPC applications, During an early iteration of our benchmarks, we used File I/O to import input data generated by the C benchmarks. We found that loading and parsing data from many-gigabyte files into PML was surprisingly feasible. The built-in file processing tools are fairly intuitive and robust, and the functional paradigm allowed us to parse all the data in a handful of lines of readable code. Overall, we found FMPLs elegantly handle file I/O.

Memory Protection/Exceptions: A convenient feature of FMPLs is their robust memory protection scheme. NPB, like most HPC applications, involves complex iterations over large memory regions. When developing these applications, it is easy to make mistakes. In C, errors like out-of-bounds array accesses may go completely undiagnosed at runtime, where they may induce strange behavior or segmentation faults.

Parallel ML solves this issue by protecting data structures and creating detailed exceptions. For example, an out-of-bounds array access halts the program and produces an "Array: Subscript" error. These exceptions point developers to their mistakes, which greatly improved our debugging experience.

While memory protection is beneficial during the development process, it comes at a performance cost. During execution, the language runtime must perform a bounds check for every array access. This extra step can greatly impact performance, which we we later see in our experiments.

B. Cons

Functional Programming: While the functional programming paradigm is useful for HPC applications in many ways, it can be unintuitive for programmers, particularly HPC developers who have years of experience with low-level, imperative programming. Given our HPC background, we also experienced this learning curve. In this vein, certain familiar concepts are more difficult to use. For example, *if* statements are treated as functional expressions, meaning they must resolve to a single value (e.g., the return value of a function call). To illustrate the complexity induced by functional *if* statements, consider the *if* statement in each version of the MG Benchmark shown in Figure 2. Using this *if* statement, the program decides between two sets of

```
(*** C Version of MG Benchmark ***)
if (Class == 'A' || Class == 'S' ||
    Class =='W') {
  c[0] =  -3.0/8.0;
  c[1] =  1.0/32.0; ...}
else {
  c[0] =  -3.0/17.0;
  c[1] =  1.0/33.0; ...}

(*** PML Version of MG Benchmark ***)
fun smallClass(class: string) =
  class = "A" orelse class = "S"
     orelse class = "W";
if smallClass(CLASS) then
  let
    val _ = Array.update(c, 0,
        ~3.0/8.0)
    val _ = Array.update(c, 1,
        1.0/32.0) ...
  in() end
else
  let
    val _ = Array.update(c, 0,
        ~3.0/17.0)
    val _ = Array.update(c, 1,
        1.0/33.0) ...
  in() end;
```

Fig. 2: Porting an *if* statement from C to PML in the MG benchmark. PML's functional semantics result in more verbose *if* statements.

initial conditions according to class size. Each portion of the *if* statement performs array updates. Bringing this behavior into PML, we cannot write an *if* statement that directly performs the array updates because each update has its own (null) return value. Instead, we must use a "let-in-end" structure to assign the return value of each operation to a dummy value, then perform an empty function call for the *if* statement's return value. Behavior like this example is an obvious headache for HPC developers more familiar with imperative languages.

Another issue caused by the functional paradigm is verbosity. For example, C has special syntax to elegantly index arrays, Parallel ML uses function calls.

High-Level Parallel Constructs: While Parallel ML's parallel constructs simplified some programming tasks, the high-level abstractions made some complex parallel control flows difficult to implement. We ran into this challenge with IS.

The C implementation of the IS benchmark is essentially one large parallel section operating on a few thread-private arrays. Figure 3 shows an example parallel computation from our eventual implementation of IS in PML. In C, thread private copies prevent a race condition when *prev_buf*'s elements are incremented at various locations according to *key_buff2*.

In PML, no notion of threads is exposed to the programmer, so no version of thread-private copies can be declared. To circumvent this issue, we implemented the kernel using a

```
fun for() =
let val l = lock_init()
in ForkJoin.parfor G (0, n) (fn i =>
  let
    val ind = Array.sub(key_buff2, i)
    val cur = Array.sub(prv_buff1,
        ind)
  in(
    lock(l);
    Array.update(prv_buff1, ind, cur
        +1);
    unlock(l))
  end)
end
```

Fig. 3: A parallel kernel from IS. In the C implementation, *prv_buff1* is thread-private. In PML, thread-private values persist across iterations, so we must use data locks for a single shared array.

single shared array. To prevent race conditions on the shared array, we had to use manually developed concurrency control. Currently, PML only provides basic atomic operations for concurrency control, so we had to create our own *lock* type and *lock/unlock* methods. Developing our own locks was ultimately straightforward, but we consider this task to be far lower-level than the promised programming model of FMPLs.

Compilation Time: Another pain point with FMPLs is the compilation time. NPB are small applications (<1000 lines of code), meaning the C implementations compile in the blink of an eye with any commonplace compiler (GCC, Clang, etc.). PML benchmarks, on the other hand, take more than a minute to compile smaller files. This issue is a known challenge for MPL and MLton, which perform full-program analyses that quickly grow in complexity relative to the program size. These full-program analyses greatly decrease the execution time for the compiled binaries, so they are very important to the compilation process.

During the debugging process, compile time became a significant overhead as we repeatedly tweaked, re-compiled, and tested the applications. For larger-scale HPC applications, compile time when using FMPLs will also scale, becoming an even bigger headache.

Floating-Point Correctness: During the development process, we encountered issues generating precise floating point values. Initially, we implemented a replica of the C random number generator, which is used to create pseudorandom input values for multiple benchmarks. We found that the FMPL version of the function produced slightly erroneous values. For example, the C implementation would produce a value of .8081127688**77** while PML would generate .8081127688**80** for the exact same input. These discrepancies caused our otherwise-correct FMPL implementations to produce invalid outputs. In the end, we avoided this issue by using the FFI to directly call the C random number generator.

We believe the root cause is that the C implementation applies low-level syntax in a way that we cannot directly

translate to FMPLs. Specifically, the C version casts 64-bit double precision values directly to 32-bit integers, which incurs rounding. x86 platforms provide multiple rounding modes that can be set using rounding control bits. We suspect the C and PML implementations are using different rounding modes, leading to different results. Low-level computational minutiae translates poorly to FMPLs, so HPC developers must pay special attention to maintain correctness in their programs. It is important to note that beyond random number generation, we encountered no other issues with floating point correctness.

Debugging Support: Even more so than lower-level parallel languages, there is a lack of source-level debugging tools for FMPLs. Given their cutting-edge nature, none of the FMPLs we considered for this work have debuggers.

Instead, we embedded source-level debug information into our benchmarks through print statements, which we toggle on and off as needed. This approach to debugging is very similar to the one used by the C implementation, and this issue is a symptom of the much broader need for debugging support for parallel programmers.

C. Overall Thoughts

Overall, we found that FMPLs' advantages grew to outweigh their disadvantages, especially with respect to their programmability. Although functional semantics are sometimes verbose compared to lower-level syntax, they quickly become familiar due to the strict and consistent rule set. Once we overcame the initial hurdles, we were able to use higher-order functions to quickly scale our implementations once we laid the groundwork for each benchmark. Additionally, by using the FFI, we could complete our implementations incrementally and more quickly find the source of bugs.

Perhaps the most interesting way to highlight FMPL's advantages from our experience is what we did *not* have to do. Thanks to the language abstractions, we did not have to concern ourselves with threads or memory management, which made for a straightforward programming experience.

The performance trade-off for these features is significant, as we detail in the next section, but we believe these issues can be addressed through continued improvement to these nascent languages.

IV. Evaluation

A. Experimental Methodology

For our experiments, we utilized a single-node, four socket machine with Intel Xeon Gold 6238L CPUs with 384GB of DDR4 memory. Each processor contains 22 physical cores, each with 2 hyperthreads, resulting in 176 logical cores. We collected each measurement 5 times. Overall, we saw minimal run-to-run performance variation.

To compile the C+OpenMP NPB implementations, we used GCC version 9.4.0 with *O3* and *mavx512* optimization flags. Unfortunately, MPL only supports MLton's source-to-source compilation to C. We therefore compiled our Parallel ML to C and then to binaries using MPL and GCC with the exact same flags as the C version. However, we believe full-program compilation to LLVM IR is a better approach for generating performant applications in the future and further improve performance. When compiling to C, MPL already generates pseudo-SSA code in chunks that look like LLVM basic blocks. Switching to LLVM would allow MPL to provide guarantees like SSA, and more easily communicate other properties that fall out of the MPL language by construction, directly to the rest of the compiler toolchain, enabling stronger optimizations like better alias analysis.

In Figure 4, we present graphs for each NPB kernel showing how performance changes over the three configurable performance factors: input size (i.e., class or class size), thread count, and grain size. For each benchmark (IS, EP, CG, MG, and FT), we performed a sweep over the three factors and measured execution time for both the PML and C implementations.

Each graph varies one parameter along the x-axis, while showing the relative performance difference between PML and C on the y-axis. We represent the performance difference as the relative slowdown of PML compared to C (i.e., how many times slower the PML implementation is when compared to C). We consider input sizes between S-C, thread counts up to 176, and grain sizes up to 64 or 20000 if the benchmark performance improved with greater grain sizes. Regarding grain size, we vary the grain size in PML, but we leave the equivalent OpenMP parameter (chunk size) as its default value because the C implementation of NPB does not vary chunk size. When manipulating other variables, we select the best-performing number of threads/grain size and class size B.

B. Analysis

Next, we analyze the results from each benchmark, scanning Figure 4 top-to-bottom and left-to-right. The figures are ordered by relative performance, best to worst. In general, we found a correlation between the amount of parallelism available in the benchmark and the performance of the PML implementation.

EP: We begin with EP, which is the best-performing benchmark in PML. Across input sizes, thread counts, and grain sizes, we measured relative slowdowns at most 25-30% more than the C implementation. Typically, the performance difference is negligible, and the PML version is actually faster for small input sizes.

EP, as indicated by its full name "Embarrassingly Parallel", contains the most parallelism of any benchmark. The results we present here show that PML can reach performance parity with C code when supplied with significant amounts of fine-grained parallelism.

FT: Next is FT, where the PML implementation maintains relative slowdowns less than $2\times$. FT's many array operations expose plenty of parallelism opportunities, but they are less frequent than in EP because only the outermost loop of three when mutating the 3-D arrays can be parallelized.

CG: Third is CG, with relative slowdowns between 2-3$\times$ for large input sizes. This benchmark contains many fine-grained parallelism opportunities, but they are broken into several sections, with parallel reductions interspersed among

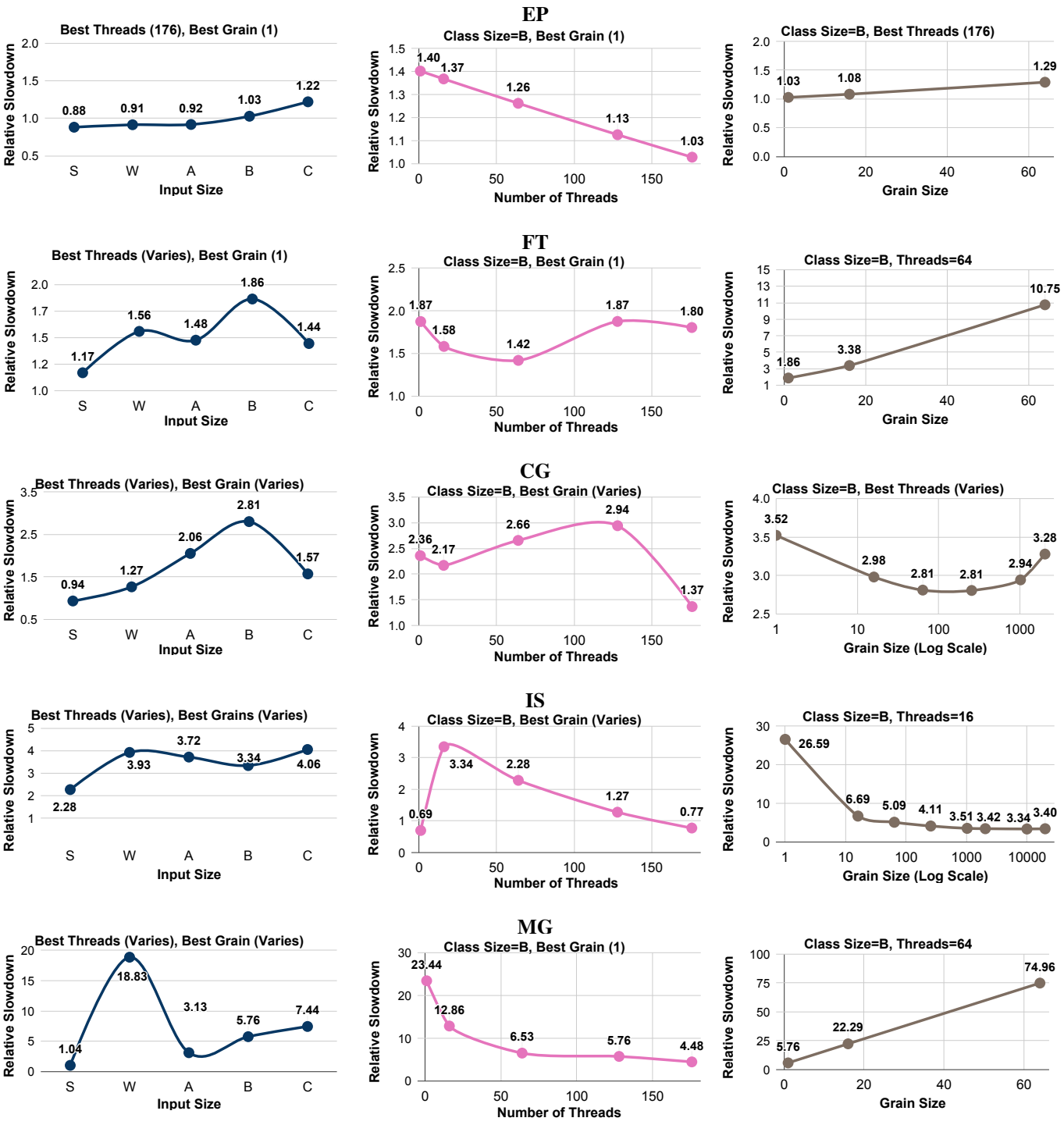

Fig. 4: Relative slowdowns for PML vs. C implementations for varying input sizes, numbers of threads, and grain sizes. To choose the fixed parameters for each graph, we selected the largest class size that we fully evaluated (B) and the best-performing number of threads/grain size. These decisions are made per-benchmark, so the graphs for different benchmarks will feature different grain sizes and thread counts. Details regarding the problem sizes and their associated parameter values can be found online [3].

some parallel loops. The fine-granularity is highlighted by the varying grain size plot (brown line), where for the first time, the PML benchmark performs better with larger grain sizes. PML CG's performance is likely held back by the frequent switches between parallel calls. The program must finish each section before moving on to the next, causing a bottleneck where the workstealing queues empty as the last few parallel iterations of a call complete.

IS: Second-to-last is IS, which generally hovers between 3.5×-4× slower than the C implementation. IS again contains bountiful parallelism opportunities, but as highlighted in Section III-B, it has a complex parallel workflow. In IS, the parallel sections are separated by sequential critical sections only executed by the "master" thread. These sequential sections have a similar effect to the smaller parallel sections in CG, slowing down the PML implementation.

MG: Last is MG, the worst performing PML benchmark. PML MG tends to be 4-8× slower than the same C configuration, with far more variance than the other benchmarks. MG's poor performance is due to the operations requiring multiple array accesses (e.g., stencil-like calculations) in its parallel region. Frequently, new array values are calculated using four or more older values from multiple arrays. Each of these individual array accesses incur a performance penalty compared to C to support higher-level language features, (e.g., bounds checks to enable protection from Section III-A). These slow calculations decrease the parallelism of the benchmark, because like FT, only the outermost loop is parallelizable. As such, PML MG struggles to keep up with C.

General Trends: Performance-wise, using an FMPL to develop HPC applications can incur a significant performance loss. However, the magnitude of this slowdown shifted drastically from one application to another depending the amount and structure of parallelism in each benchmark.

In Figure 5, we selected the best-performing thread count and grain sizes for the PML and C implementations of each benchmark for class B. We show these values in Table II. Figure 5 shows a massive difference in relative slowdown between the best-performing PML implementation (EP=1.02×) and the worst (MG=5.76×). The average relative slowdown across all benchmarks is 2.96×.

FMPLs' Suitability for HPC: In their current state, FMPLs are not truly useful to existing HPC developers for a few reasons. Performance is an obvious concern, but moving to LLVM IR instead of source-to-source compilation which could significantly improve PML's performance. Similarly, existing HPC libraries could be utilized in FMPLs either through FFI or a thin wrapper library that interfaces with the existing code at the compiler level.

More fundamentally, FMPLs lack a groundbreaking feature to out-compete existing shared memory programming models. Interfaces like OpenMP already provide a programmer-friendly, directive-based approach to shared memory parallelism. Seasoned HPC developers who are familiar with OpenMP or an equivalent are unlikely to see benefit from switching to an FMPL.

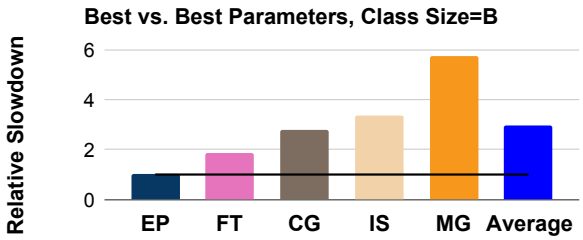

Fig. 5: Relative slowdown for each benchmark where both the C and PML implementations use their most performant parameters (e.g., thread count) for Class Size B. The average across all benchmarks is 2.96×.

TABLE II: Optimal thread count and grain size for Class=B

Benchmark	Language	Thread Count	Grain Size
EP	C	176	N/A (Default)
EP	PML	176	1
FT	C	128	N/A (Default)
FT	PML	128	1
CG	C	128	N/A (Default)
CG	PML	128	256
IS	C	16	N/A (Default)
IS	PML	16	10000
MG	C	128	N/A (Default)
MG	PML	128	1

That said, the future of FMPLs in HPC is not all doom and gloom. For example, FMPLs could help attract more users to HPC. Data science and deep learning are currently hot topics in HPC. In these fields, developers use high-level languages like Python. Efforts are already underway to provide HPC-enabled programming models better suited for these communities [33], [8]. FMPLs could provide a familiar programming model and substantially better parallel performance for these applications.

Lastly, it is important to note that these languages are still nascent and under active research, which may create new features that appeal to HPC. In the next section, we detail our planned effort to revolutionize the Parallel ML runtime for HPC by adding automatic distributed memory parallelism.

V. Future Work

To expand FMPL's potential for developing HPC applications, we seek to apply them to distributed memory systems without complicating the existing programming model.

A. Background

The MPL compiler's runtime for Parallel ML manages memory in a strict manner that makes it especially suitable for

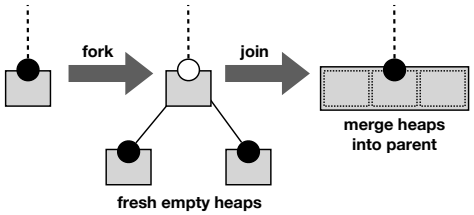

Fig. 6: Heap hierarchy at forks and joins. Black circles are active child threads, and white circles are suspended parent threads. Gray rectangles represent heaps [30].

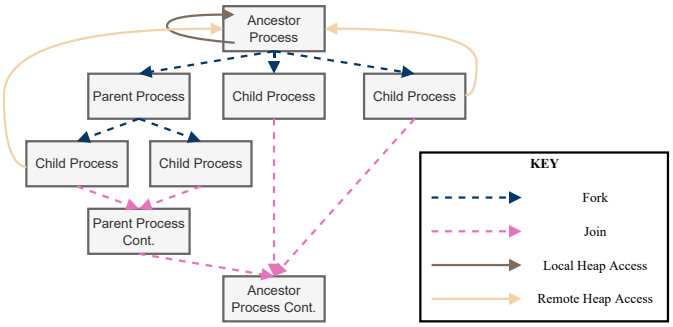

Fig. 7: Example execution path of a disentangled parallel program. Memory accesses may occur in the local heap or specific remote heaps depending on forks and joins.

high-performance, distributed computation. Specifically, MPL enforces the "disentanglement" memory property [30]. From a systems perspective, disentanglement ensures that concurrent threads remain oblivious to each other's allocations.

MPL automatically ensures disentanglement by enforcing a strict memory heap hierarchy. When a thread forks and creates parallel child threads (e.g., to begin executing a parallel-for loop), each child thread allocates a new, separate heap of memory. Child threads can access their parent's heap, which they can use freely for communication. Because parent threads are suspended while their children execute, they are not concurrent threads, and accesses to the parent heap by the child threads do not violate disentanglement. On the other hand, child threads remain unaware of each other's personal heaps. This policy ensures that no child threads can access their sibling's heap concurrently, maintaining disentanglement. The heap management surrounding forks and joins is illustrated in Figure 6, which we borrow from Westrick et al. [30].

B. Distributed FMPL Execution

Next, we describe our plan to automatically execute disentangled FMPL programs across distributed memory in HPC systems. Disentangled parallel programs create a specific execution model with two distinct types of memory accesses and four forms of communications overall. An example execution is shown in Figure 7. In this example, assuming each thread is now a process distributed across separate memory domains, we see the four forms of inter-process communication we must support for distributed memory:

- **Forks** (navy): When a process forks, it must spawn new "child" processes and tell them what computation to complete.
- **Joins** (pink): When all of the child processes complete their computation, they must communicate their results (i.e., return values, relevant portions of their local heap) to the "parent" and exit.
- **Local Heap Access** (brown): During computation, any process may access the local heap that they allocated.
- **Remote Heap Access** (tan): During computation, a child process may access a remote ancestor heap.

We observe that each type can be mapped onto existing patterns in lower-level HPC communication libraries:

- **Forks** (navy): Forks can be performed by distributing work to parallel processes using bulk-synchronous collective communication (e.g., broadcast the function to run and scatter the input values).
- **Joins** (pink): Joins can also be performed using bulk-synchronous collective communication (e.g., gather or reduce the results).
- **Local Heap Access** (brown): Local accesses can occur without any communication.
- **Remote Heap Access** (tan): Remote accesses can occur through one-sided, atomic operations. By only allowing ordered accesses on a single copy of the memory location, we effectively maintain the processor consistency model expected by fork/join programs.

Using these ideas, we are in the initial stage of re-implementing the MPL runtime using conventional HPC programming models, in our case, MPI.

A distributed Parallel ML language could be massively beneficial to HPC developers. The new runtime will make it possible to run FMPL programs on distributed HPC systems *without modification*. The language runtime will utilize both the existing shared-memory back-end and the new distributed back-end to automatically load-balance applications.

Considering these major benefits, we hope distributed FMPLs will be compelling new direction for HPC programming model research. We plan to study this system thoroughly and seek novel ways to minimize the performance penalty of distributed FMPLs.

VI. Related Work

Parallel ML is one of many recently-developed FMPLs. From the ML family of languages alone, other relatively-new FMPLs include Multicore OCaml [27], [28] and the Manticore project [15], which has its own dialect of Parallel ML.

Multicore OCaml is an industry-led, shared-memory parallel version of OCaml. In this work, we chose Parallel ML using the MPL compiler because of our future work, which will benefit from MPL's research-oriented design. A natural future step is bringing any MPL enhancements to Multicore OCaml.

The Manticore project defines its own dialect of Parallel ML, which includes implicitly threaded data structures such as parallel arrays. We focus on MPL's PML over Manticore because MPL's high-level parallelism constructs are more similar to parallelism APIs currently used in HPC (e.g., OpenMP).

Beyond the ML family, another recent FMPL is Distributed Parallel Haskell [11]. DPH focuses on pure functional programming, which is far less suitable for our purposes because it not does not support mutable arrays.

The modern FMPLs are heavily inspired by older parallel functional languages such as NESL [9]. NESL is an ML-like language, and it was one of the original functional languages to support nested data parallelism and automatic vectorization.

There is a long trail of new HPC parallel programming languages/models designed to maximize performance and/or im-

prove developer productivity. Highlights include UPC/++ [10], [32], Coarray Fortran [23] and High Performance Fortran (HPF) [21]. UPC integrated the partitioned global address programming model into C and later C++. Coarray Fortran extended F95 with SPMD *coarrays*, which were later adopted into Fortran 2008. HPF extended F90 with innovative features like parallelism directives, but it largely fizzled [20]. Languages such as Chapel [12] attempt to improve on HPF's failings. Other efforts have targeted non-CPU/heterogeneous programming, such as CUDA and SYCL [25], which has been adopted by Intel's "OneAPI" for future HPC programs.

The NAS Parallel Benchmarks have frequently been used to evaluate the effectiveness of emerging languages for HPC applications. Examples include UPC [14], [19] and Chapel [12]. In addition, NPB is commonly used to compare different HPC programming models [26].

VII. Conclusion

In this work, we use an FMPL to implement the NAS parallel benchmarks for shared-memory systems, and we study FMPL's larger potential for HPC. Based on that experience, we believe FMPLs have the *potential* to make HPC programming easier. Emerging domains with smaller existing code footprints may soon benefit from FMPLs, and we are excited to see how future improvements to FMPLs further increase their usefulness for HPC applications. We plan to contribute by bringing MPL to distributed memory.

Lastly, it is important to note that our implementations are a careful but first attempt bring to NPB into an FMPL. We believe these are useful to both the HPC and parallel languages communities. We don't claim our implementations are optimal, and we plan to improve them.

References

[1] "Mlton." [Online]. Available: www.mlton.org

[2] "Nas parallel benchmarks 3.0 - unofficial openmp c version." [Online]. Available: https://github.com/benchmark-subsetting/NPB3.0-omp-C

[3] "Problem sizes and parameters in nas parallel benchmarks." [Online]. Available: https://www.nas.nasa.gov/software/npb_problem_sizes.html

[4] "Nas parallel benchmarks," Jan 2022. [Online]. Available: https://www.nas.nasa.gov/software/npb.html

[5] S. Adve, "Data races are evil with no exceptions: Technical perspective," *Communications of the ACM*, vol. 53, no. 11, pp. 84–84, 2010.

[6] J. Arora, S. Westrick, and U. A. Acar, "Provably space-efficient parallel functional programming," *Proceedings of the ACM on Programming Languages*, vol. 5, no. POPL, pp. 1–33, 2021.

[7] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber *et al.*, "The nas parallel benchmarks summary and preliminary results," in *Supercomputing'91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*. IEEE, 1991, pp. 158–165.

[8] M. Bauer, W. Lee, M. Papadakis, M. Zalewski, and M. Garland, "Supercomputing in python with legate," *Computing in Science & Engineering*, vol. 23, no. 4, pp. 73–79, 2021.

[9] G. E. Blelloch, *NESL: a nested data parallel language*. Carnegie Mellon Univ., 1992.

[10] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks, and K. Warren, "Introduction to upc and language specification," Technical Report CCS-TR-99-157, IDA Center for Computing Sciences, Tech. Rep., 1999.

[11] M. M. Chakravarty, R. Leshchinskiy, S. Peyton Jones, G. Keller, and S. Marlow, "Data parallel haskell: a status report," in *Proceedings of the 2007 workshop on Declarative aspects of multicore programming*, 2007, pp. 10–18.

[12] B. L. Chamberlain, S. Deitz, M. B. Hribar, and W. Wong, "Chapel," *Programming Models for Parallel Computing*, pp. 129–159, 2015.

[13] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *IEEE computational science and engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[14] T. El-Ghazawi and F. Cantonnet, "Upc performance and potential: A npb experimental study," in *SC'02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*. IEEE, 2002, pp. 17–17.

[15] M. Fluet, M. Rainey, J. Reppy, and A. Shaw, "Implicitly threaded parallelism in manticore," *Journal of functional programming*, vol. 20, no. 5-6, pp. 537–576, 2010.

[16] A. Guatto, S. Westrick, R. Raghunathan, U. Acar, and M. Fluet, "Hierarchical memory management for mutable state," *ACM SIGPLAN Notices*, vol. 53, no. 1, pp. 81–93, 2018.

[17] K. Hammond, "Why parallel functional programming matters: Panel statement," in *International Conference on Reliable Software Technologies*. Springer, 2011, pp. 201–205.

[18] H.-Q. Jin, M. Frumkin, and J. Yan, "The openmp implementation of nas parallel benchmarks and its performance," 1999.

[19] H. Jin, R. Hood, and P. Mehrotra, "A practical study of upc using the nas parallel benchmarks," in *Proceedings of the Third Conference on Partitioned Global Address Space Programing Models*, 2009, pp. 1–7.

[20] K. Kennedy, C. Koelbel, and H. Zima, "The rise and fall of high performance fortran: an historical object lesson," in *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, 2007, pp. 7–1.

[21] D. B. Loveman, "High performance fortran," *IEEE Parallel & Distributed Technology: Systems & Applications*, vol. 1, no. 1, pp. 25–42, 1993.

[22] D. MacQueen, R. Harper, and J. Reppy, "The history of standard ml," *Proceedings of the ACM on Programming Languages*, vol. 4, no. HOPL, pp. 1–100, 2020.

[23] R. W. Numrich and J. Reid, "Co-array fortran for parallel programming," in *ACM Sigplan Fortran Forum*, vol. 17, no. 2. ACM New York, NY, USA, 1998, pp. 1–31.

[24] J. Reid, "The new features of fortran 2018," in *ACM SIGPLAN fortran forum*, vol. 37, no. 1. ACM New York, NY, USA, 2018, pp. 5–43.

[25] R. Reyes and V. Lomüller, "Sycl: Single-source c++ accelerator programming," in *Parallel Computing: On the Road to Exascale*. IOS Press, 2016, pp. 673–682.

[26] H. Shan, F. Blagojević, S.-J. Min, P. Hargrove, H. Jin, K. Fuerlinger, A. Koniges, and N. J. Wright, "A programming model performance study using the nas parallel benchmarks," *Scientific Programming*, vol. 18, no. 3-4, pp. 153–167, 2010.

[27] K. Sivaramakrishnan, S. Dolan, L. White, S. Jaffer, T. Kelly, A. Sahoo, S. Parimala, A. Dhiman, and A. Madhavapeddy, "Retrofitting parallelism onto ocaml," *arXiv preprint arXiv:2004.11663*, 2020.

[28] K. Sivaramakrishnan, S. Dolan, L. White, T. Kelly, S. Jaffer, and A. Madhavapeddy, "Retrofitting effect handlers onto ocaml," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2021, pp. 206–221.

[29] S. Westrick, M. Rainey, D. Anderson, and G. E. Blelloch, "Parallel block-delayed sequences," in *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2022, pp. 61–75.

[30] S. Westrick, R. Yadav, M. Fluet, and U. A. Acar, "Disentanglement in nested-parallel programs," *Proceedings of the ACM on Programming Languages*, vol. 4, no. POPL, pp. 1–32, 2019.

[31] M. Wilkins, S. Westrick, V. Kandiah, A. Bernat, B. Suchy, E. A. Deiana, S. Campanoni, U. A. Acar, P. Dinda, and N. Hardavells, "Warden: Specializing cache coherence for high-level parallel languages," in *IEEE/ACM International Symposium on Code Generation and Optimization*, 2023, p. To Appear.

[32] Y. Zheng, A. Kamil, M. B. Driscoll, H. Shan, and K. Yelick, "Upc++: a pgas extension for c++," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE, 2014, pp. 1105–1114.

[33] A. N. Ziogas, T. Schneider, T. Ben-Nun, A. Calotoiu, T. De Matteis, J. de Fine Licht, L. Lavarini, and T. Hoefler, "Productivity, portability, performance: data-centric python," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–13.