ILLINOIS INSTITUTE OF TECHNOLOGY


Department of Computer Science

**IIT HExSA Lab Technical Report**
**Number: IIT-CS-OS-19-01**

October, 2019

# Evaluating Julia as a Vehicle for High-Performance Parallel Runtime Construction

**Amal Rizvi, Kyle C Hale**

## Abstract

Practitioners of high-performance parallel computing have long sought better programming models and languages to ease the task of writing programs for large-scale systems. However, there is an unde- niable tension that exists between extreme performance and developer friendliness. While the steadily increasing performance of high-level lan- guages shows promise, and the ubiquity of concurrent programming is on the rise, these advancements have not yet made a significant impact in the HPC community. The Julia language is of particular interest as it boasts single-threaded performance on par with C and Fortran while retaining familiar echoes of MATLAB, Python, and Ruby-like syntax in the lan- guage. In this paper, we present a comprehensive set of experiments with the goal of evaluating Julia as a vehicle for building high-performance parallel runtime systems. We conclude that Julia's performance on a se- ries of microbenchmarks indicates that it is a reasonable candidate for the bedrock of an HPC runtime.

## Keywords

**Julia. HPC, Benchmark**

## 1 Introduction

Programmers today have a wide array of software frameworks at their disposal which allow them to deploy their applications to large-scale infrastructure. Programming frameworks like TensorFlow [1], Naiad [21], Spark [29], and MapReduce [12] allow the programmer to focus on application logic while often leaving concerns specific to the computational environment, such as task mapping, fault tolerance, scheduling, and synchronization up to the runtime system. As cloud providers continue to reduce the granularity at which they offer computational infrastructure (both temporally and spatially) we can expect the trends of increasing runtime system sophistication and programming model innovation to continue. In the HPC community, there is a similar need for developer-friendly parallel programming frameworks and runtime systems, but progress is slower. The main contributor to this stagnation is the relentless pursuit of raw performance. Programming frameworks and the runtime systems that support them, often written in high-level languages (HLLs), sacrifice node-level performance for programmer productivity and the natural expression of parallelism. In an HPC environment, this sacrifice is unacceptable. When an application can run for weeks or even months at a time, even the smallest modicum of performance improvement can mean hours or days of execution time saved.

The *de facto* standard for programming large-scale machines involves some combination of MPI paired with C/C++/Fortran for inter-node parallelism, and runtime support such as OpenMP [11], pthreads [6], Cilk [5], or Qthreads [27] for handling intra-node parallelism. More recently, applications leverage heterogeneous hardware by using accelerator frameworks like OpenCL [26], OpenACC [14] or CUDA [22]. Ideally, HPC programmers would also be able to describe their application logic and e xpress parallelism succinctly in a high-level programming language, and while recent languages like Regent [25] and more established languages like Chapel [8], HPF [20], UPC [7], X10 [9], and Erlang [17] all make significant strides to bring productive programming to the HPC world, they have yet to see widespread adoption. As lower layers of the hardware/software stack change rapidly, from increasing hardware heterogeneity to lightweight OSes [**?**, 15, 16, 18, 19] containers the need for new runtime systems which bridge the gap between the HLL and the system software and hardware architecture is increasingly pressing. However, most of the runtime systems for the above language are written in a low-level language like C/C++. For example, the Legion runtime system [2] (which supports Regent) consists of tens of thousands lines of C++. In this paper, we aim to evaluate the Julia programming language[1] on each of these points. We present detailed experiments which evaluate a set of Julia primitives which would likely be used to construct a parallel runtime for HPC. We make the following contributions:

– We present the first comprehensive set of experiments evaluating Julia's primitives as a vehicle for runtime construction.

[1] `https://julialang.org`

– We discuss the current performance limitations of Julia, and possible reasons for performance lag.

The remainder of this paper is organized as follows: Section 2 gives an overview of parallel runtimes and the Julia language. Section 3 presents the evaluation of Julia as a language for building such runtimes. In Section 4, we discuss our results and give potential avenues towards building high-performance parallel runtimes in Julia. Section 6 presents our conclusions and future work.

## 2 Background

### 2.1 Parallel Runtime Systems

A parallel runtime system comprises the machinery which allows an application developer to express their application logic in a way that allows the system to exploit parallelism by mapping the application across available hardware. The degree to which the developer must be involved in mapping code to hardware resources, and the effort involved in expressing parallelism varies widely depending on the programming model and the sophistication of the runtime system. For example, pthreads, the standard threading library for POSIX threads, allows the developer to decompose his or her application logic such that it can be operated on by threads executing on different cores of a machine. Developer effort in this case is high since the runtime system is essentially just the underlying OS kernel, and the programming model involves explicitly creating, managing, and coordinating threads using a thin interface to the system's kernel threads. On the other end of the spectrum, some parallel languages allow a developer to write a parallel program at a very high level, and the compiler and runtime extract the implicit parallelism inherent in the high-level operation.

Some parallel runtimes offer the ability to distribute an application among many nodes in a large-scale system, and may have the capability to handle hardware heterogeneity. If the programmer supplies multiple platform-specific implementations of core application kernels, or if the kernels are written at a high enough level such that the compiler can generate code for different platforms, the runtime may make intelligent decisions to map application threads to heterogeneous hardware (e.g. GPGPUs or special-purpose accelerators) at execution time. Figure 1 shows an example structure of a runtime for a distributed, parallel machine. A parallel application conforms to the programming model dictated by the runtime and runs atop several nodes. Node 0 is a dual-core system with two NUMA nodes (depicted by memory DIMMs) and node 1 is a uniprocessor system with an attached accelerator. Here, for example, application functions are split into a number of asynchronous tasks (1) which the runtime system is in charge of scheduling onto the machines. The runtime shown executes tasks according to a dataflow-like firing rule, where tasks are scheduled once the tasks that they depend on have completed (depicted by the edges between the task nodes). Here, task `t3` depends on both `t1` and `t2`, and `t4` depends on `t3`. The dependencies between the tasks may be explicitly indicated by the programmer, or the runtime may infer them indirectly, e.g. by coordinating access to shared data. In this

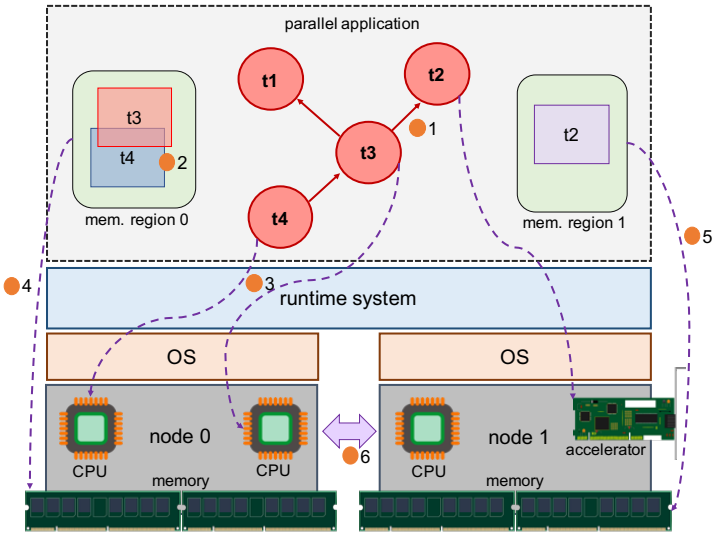


**Fig. 1.** Example structure of a parallel runtime system.

application there are two logical memory regions (for example, large matrices) which are accessed by the application. Memory region 1 (2) is accessed by both task `t3` and `t4`, as indicated by the overlapping boxes. Because these tasks both access the same shared data, the runtime might make an intelligent decision to map them onto the same node (3), and to map the memory region onto a physical memory region with on the same node (4) with optimal locality. Task `t2` may have an alternate implementation which can run on an accelerator (e.g. a CUDA or OpenCL implementation for a GPU), so the runtime will map that task and the memory region it accesses to node 1 (5). The runtime will also manage communication between nodes (6) to facilitate any necessary coordination between tasks. To build an efficient version of a runtime such as the one shown in Figure 1, one needs access to several low-level features such as: task creation/scheduling (e.g. to support (1)), synchronization and atomics (e.g. to support (2)), control over mapping of tasks/threads/execution contexts (e.g. to support (3), (4), and (5)), communication between contexts, and efficient data placement facilities. In low-level languages such as C and C++, these are readily available via established interfaces like pthreads (for creating/managing execution contexts), `numactl` (for efficient data placement), and `sysconf` (for detecting hardware features like cache line size and page size for proper data alignment). However, while these low-level languages expose the necessary interfaces and mechanisms for extracting performance from the system, they are cumbersome to use and increase the likelihood of programmer errors (e.g. because of manual memory management). In general, the language will have high-performance primitives or constructs which support:

- Mapping execution contexts to hardware precisely (e.g. NUMA and processor affinity)
- Hardware heterogeneity
- Low-latency communication between execution contexts (threads/tasks/processes)
- Low-latency creation and scheduling of tasks.
- Efficient exploitation of parallelism (e.g. parallel `map()`s or parallel loop constructs)

While this list is certainly not complete, we believe that by meeting the above requirements, the language warrants investigation as a tool for runtime construction.

### 2.2 Julia

Julia [3] is a high-level programming language designed for numerical computing. Our main interest in Julia stems from its promising single-threaded performance (which the Julia developers have shown can approach that of C and Fortran), its native parallel constructs, its extensibility, and its rich set of user-developed packages. Particularly its performance relative to other high-level languages, which is in part due to its usage of LLVM's JIT compilation framework, indicates that it might be a suitable candidate for building parallel runtime systems which support even higher-level (possibly domain-specific) parallel languages. Some other salient features of the Julia language include multiple dispatch, dynamic types, metaprogramming features, light-weight user threads, easy support for native code invocation, and packages for distributed parallel computing.

In this paper we aim to evaluate Julia as a vehicle for runtime construction by measuring the performance of several of its constructs which relate to the list outlined in Section 2.1.

## 3 Evaluation

In this section we present a series of experiments comparing the performance of Julia's runtime-supporting primitives with baselines based on C implementations. All experiments were carried out on a Dell PowerEdge R430 with a 10-core (20 HW threads) Intel Xeon E5-2630v4 (Broadwell) CPU clocked at 2.2GHz and 16GB of RAM. This machine runs Fedora 24 with Linux kernel version 4.11.12.

For the Julia experiments, we use Julia version 0.6.2. Unless otherwise noted, all experiments were conducted using 100 trials, with an additional 10 trials initially thrown out to warm up the cache, JIT compiler, and memory allocations. C code (e.g. for OpenMP and pthreads experiments) was compiled with `-O3` using gcc version 6.3.1. Timing was conducted using the `time_ns()` function in Julia and `clock_gettime()` in C with the `CLOCK_REALTIME` timer. We assume that these clocks are synchronized across cores in the system.

Many runtimes support the notion of a *task*, a schedulable unit of work which may or may not have the ability to modify state. Tasks can essentially be thought of as glorified functions, but they can be moved around the system.

Tasks may be mapped to threads or processes, but ultimately they should be very light-weight. The more overhead that the runtime introduces when creating and managing tasks, the coarser the units of work must be to amortize these costs, and the amount of parallelism available to the runtime is reduced. Julia has support for such tasks (with the `Task` abstraction), which they describe alternatively as light-weight coroutines or *green threads*. These are essentially user-level threads of which the underlying OS is unaware. Our first experiment measures the latency of creating these tasks. As a baseline we compare against the time to create a pthread (using `pthread_create()`). Since pthreads map directly to kernel threads (visible to the OS scheduler), we expect that the Julia tasks should be much cheaper to create. To make the comparison as close as possible, in Julia we measure the time to create a task using the `Task()` function followed by a call to `schedule()` which places the created task on the task scheduler's run queue (which is local to the current Julia process on a single core). Note that unlike most `schedule()` functions, this Julia version does not immediately yield to the scheduler.

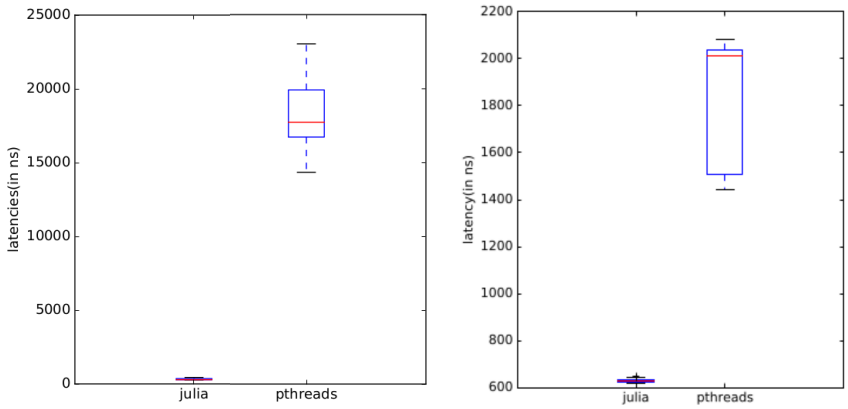


**Fig. 2.** (a) Comparison of task creation latencies for Julia tasks and pthreads. (b) Context switching latencies for Julia tasks and pthreads

Figure 2(a) shows the results. Creation of a pthread costs roughly 20$\mu$s while Julia task creation and the subsequent schedule call costs only about 400ns. This is in line with our expectations. However, tasks on a remote core must be scheduled on a remote worker process, which must be explicitly created with the `addprocs()` routine. While this routine takes about 400ms, it must only be called once during runtime startup.

In task-based runtimes, there are often many more tasks than there are hardware resources available. This allows a high degree of concurrency when tasks perform blocking operations. it is important that tasks be light-weight to max-

imize concurrency. Here we measure the context switching overhead of Julia tasks by creating two tasks in the same worker process, having them cooperatively `yield()` to each other a fixed number of times, and approximating the latency from the number of yields in the recorded time period. For our baseline we use the same strategy with two pthreads pinned to the same core.

Figure 2(b) shows that Julia task context switching is roughly one order of magnitude faster than for pthreads. Again, this is to be expected since the kernel is not involved in a Julia context switch unless a task yields and there is nothing else to run.

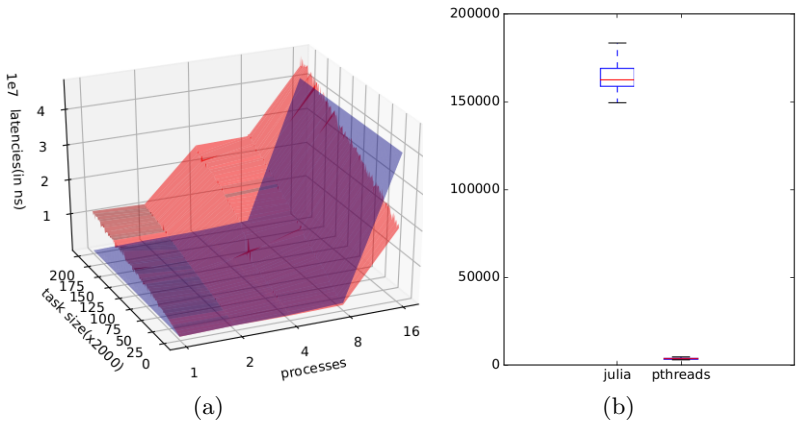


**Fig. 3.** (a)Average latencies (in ns) for Julia's one-sided future-based parallelism (spawn and fetch). Task size is approximated by the input to a simple fibonacci routine (red). The blue surface represents a baseline task size where no work is done.(b)Event notification latency for Julia and pthread condition variables.

For the next experiment, we attempt to get an estimate of the granularity of tasks that Julia can support. Julia's native parallel processing features are based on one-sided task-launch operations (futures). When a future is created (using `spawn()`), its corresponding object is immediately returned to the caller. The work associated with the future may complete immediately, or at some later point in time. The result is retrieved by an explicit call (`fetch()`) to get the value of the future, which will block if the execution has yet to take place. We aim to identify the minimum amount of work that we must provide to the Julia runtime in order to amortize the costs of remote task creation. For this experiment we created two benchmarks: a tunable benchmark which returns the first n Fibonacci numbers sequentially and a baseline benchmark which creates a dummy task which performs no actual work at all. The dummy benchmark is meant to approximate the overheads of remote task creation. To formulate a threshold for which creating a task would be too expensive, we set an arbitrary constant (2x) times the overhead of task creation. Thus, a runtime would decide to launch a

remote task only if it had an estimate that the overheads involved would only comprise a third of the total work. Figure 3(a) shows two surfaces. The blue (darker) surface represents the baseline task, and the red (lighter) surface shows latencies for executing different amounts of work with our tunable function. We plot this along two axes, the first being the work amount (argument to fibonacci) and the second being the number of workers involved (on a single node). For this experiment, we only distribute the work to one other worker, so this should not change the latency, but we see that having more workers actually increases latency. For a single process, the threshold value is 4000, roughly corresponding to a latency of 0.013 ms, for two processes the threshold value reported is 18000 which translates to roughly 0.25 ms. The threshold for 16 processes is right at 3 million, translating to 39.9 ms. This means that the task creation threshold becomes higher as we increase the amount of available parallelism. We suspect this is due to inefficiencies in Julia's task scheduling algorithm, but we intend to investigate further in future work.

Another important feature of runtime systems is low-latency communication. This is needed to support explicit communication between tasks (e.g. using explicit messaging), implicit communication based on access to shared data, and communication between different entities in the runtime (e.g. scheduler threads or processes).

One such type of communication is the notification of an event, which might signify several things such as the creation of a task, the availability of data, an error condition, or the completion of a task. Event notifications are particularly important for data-flow runtime systems, which are driven by asynchronous events. One of the most common software event notification mechanisms is a *condition variable*. A condition variable is essentially a queue protected by a mutex, and its interface comprises three functions: `wait()`, `signal()`, and `broadcast()`. A thread/task calls `wait()` to block execution until some *condition* becomes true (e.g., a unit of work is ready to execute, a message delivery has completed). The thread will then be put on the condition variable's queue and go to sleep (i.e. yield execution to another task/thread) until the condition becomes true. Some other thread will then call `signal()` to wake one waiting task and remove it from the queue, or call `broadcast()` to wake all waiting tasks. A common model in runtimes is to have worker threads/tasks spread across cores, and they wait on work to do using condition variables. Unfortunately, Julia's `Condition()` construct only applies to `Tasks` on the same worker process, so there is no way to use them for cross-core notifications. To approximate this behavior, we used Julia's `RemoteChannel` primitive, which is essentially a producer/consumer queue that can be referenced by `Tasks` running on different Julia worker processes. We create an unbuffered (0-element) `RemoteChannel` and issue a `take()` operation to simulate a condition variable `wait()`, and have a `Task` on a worker processes running on remote core issue a `put()` to simulate a condition variable `signal()`. We compare against a baseline implementation written in pthreads which creates a remote thread whose job is to signal a condition that the local thread is waiting on (using `pthread_cond_signal()`). The remote thread takes

a time stamp after signaling, and the local thread records a time stamp after waking up. We report the latency as the difference between these two timestamps. Figure 3 shows the results. We see a median latency of roughly 16$\mu$s for the Julia case, and less than a microsecond for notification using the pthreads condition variable. While the performance gap is quite large, one should realize that the Julia notification is occurring across worker processes (OS processes) in separate address spaces, while in the pthreads case this is happening between kernel threads sharing the same address space. While the notification latency for Julia is already fairly small, we expect it will get smaller once proper threading support (with the attendant condition variable implementation) is added.

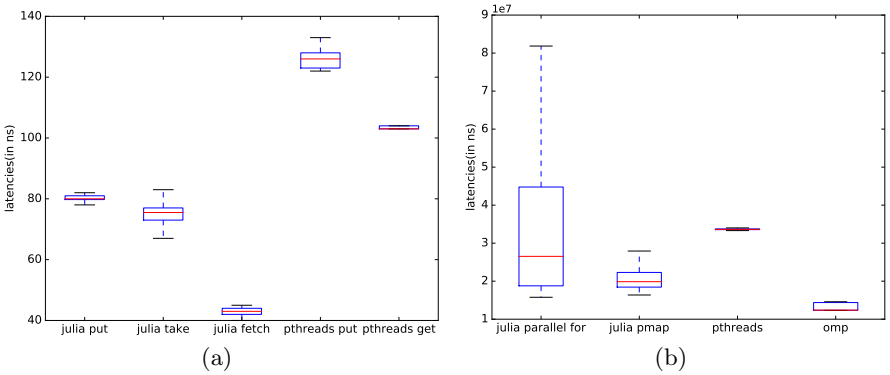


**Fig. 4.** (a) Latency of channel operations (put/get/fetch) using Julia's channel construct and a simple producer/consumer queue written in C with pthreads synchronization primitives. (b) Box plot showing the latency to initialize a 50MB array across execution contexts using Julia's parallel for, pmap, OpenMP threads, and pthreads with a manual decomposition. synchronization primitives.

Yet another important feature of an ideal HPC runtime system is and efficient exploitation of parallelism. We measured the latency of Julia's parallel map constructs with pthreads and OMP. Julia includes two such parallel constructs called `pmap()` and `@parallel for` The difference between these two constructs can be explained by examining the sites at which they prove to be most useful eg. `pmap()` is used for an instance where each function call has to a large amount of work whereas `@parallel for` is used for instances where each iteration is tiny. For both these functions and their baseline comparisons we initialized a 50MB array across execution contexts and `pmap` performed better than `@parallel for`. `pmap` needed us to divide the array into segments with respect to the number of processes added. Since Julia implements user level threads and pthreads implement kernel level threads the performance of Julia's parallel level constructs significantly look better in Figure 4(b).

In addition to simulating condition variables, channels can also be used as shared queues. The common use case for this in runtimes is work queues, where

one thread (e.g. a master thread) adds work or tasks to the queue and other workers pull from it to complete the work. Julia exposes three operations on channels: `put()` to place an object on the queue, `take()` to remove an object from the queue, and `fetch()` to read the front of the queue (without removing the object). Ideally operations on channels will just be a read or write protected by a lock or an atomic operation. We compare Julia's channel against a simple producer/consumer queue written in C that is protected by a mutex from the pthreads library (`pthread_mutex_t`). Figure 4(a) shows the results. We see here that Julia's channel operations are all very light-weight, each taking less than 100ns.

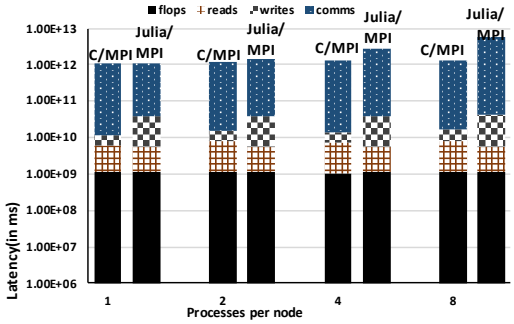


**Fig. 5.** Latencies of Julia with MPI, and C with MPI measured for a synthetic BSP benchmark for 2, 4 8, 16 processes.

For Julia to become a viable candidate for an HPC runtime system, it is important to examine Julia in a multi-node environment. An HPC runtime system written in a HLL requires performance close to that achieved by present HPC techniques. In our first set of experiments, we approximate a typical BSP code (e.g. a stencil computation) by creating a synthetic benchmark which consists of compute (memory reads/writes and floating point ops) and communication (message send/recv) steps. We compare the performance of this benchmark with C and Julia (using native parallel constructs and using MPI bindings) implementations. We ran these experiments on 16 nodes using the Chameleon testbed with varying number of processes on each node (1,2,4 and 8). This program measured the latency of 5,000,000 local read and write operations and 1,000,000 floating point operations for 100 iterations. Communication between processes followed a simple ring topology. We performed 10,000 communication operations and over 100 iterations. The results are shown in Figure 5. The breakdown of compute operations was similar for C/MPI and Julia/MPI experiments showing that floating point operations are least expensive and write operations are most expensive. Results from Figure 5 confirm that communication operations in Julia/MPI incur some overheads as compared to communication operations deployed in C/MPI.

MPI in Julia is connected in an all-to-all (not master-slave) format but in a "lazy" way wherin master process will be able to connect with all workers but a worker process will only establish a connection with another worker process if a remote invocation between them exists. Communciation latencies increase with added number of processes especially for Julia/MPI, because each process makes an attempt to communicate with another off machine process over the network. `MPI.jl` is simply a Julia wrapper for the MPI protocol. It ccalls the C library functions and the latency incorporated in Julia/MPI communications can be attributed to `ccall` on sufficiently big array structures instead of primitive data types like `Int64,Float64` etc. Moreover, for each `MPI.Send()`function call that we use in the experiment, Julia makes at least two function calls before it ccalls `MPI_SEND` operation. These overheads add up for 10000 send and recieve operations that we call on increasing number of processes.

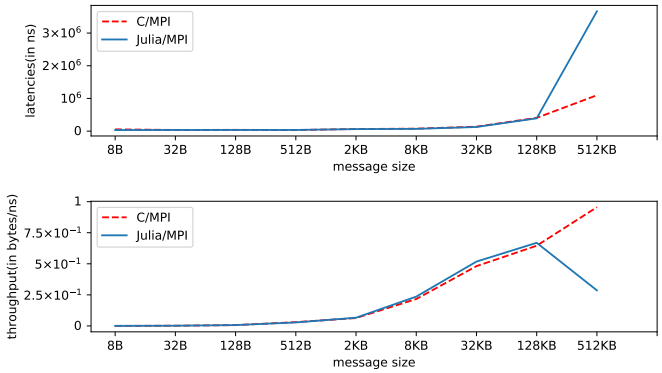


**Fig. 6.** Throughput (above) and Latencies (below) of Julia Native, Julia with MPI, and C with MPI measured for a ping pong benchmark using two processes.

In our second experiment, we wanted to measure the latency of sending messages of various sizes and the message size throughput for each communication for Julia/MPI and C/MPI . By measuring and comparing these, we benchmark the performance of message communication in Julia. For this we wrote a simple ping pong benchmark. This program created two processes and both processes communicated with each other by sending and receiving messages to one another. This experiment was repeated for various message sizes ranging from 8B to 512KB. The results are shown in Figure 6 . We can see from these that the throughput of message transfer in C/MPI is significantly better than Julia/MPI and increases significantly when the message size exceeds 128KB. This validates that Julia's wrapper ccalls to `MPI_SEND` or `MPI_RECV` become more expensive as the messages become heavier and more complex than simple primitive data types.

## 4 Discussion

In order to evaluate Julia as a candidate language for creating an HPC runtime, we performed experiments that test Julia's ability in task creation and switching, communication, event notification, and its native future-based parallel constructs. Julia's user-level threads perform on par with what we would expect. Its parallel constructs also show that Julia's parallel constructs (such as `pmap()`) enable performance comparable to OMP for our microbenchmarks. Julia's channel operations perform better than the equivalent producer/consumer queue implemented with pthreads. We also performed experiments on Julia's parallel constructs like `@spawn` and `fetch` and found that their latency surprisingly increases with an increased number of worker processes a single node. We measured and reported the amount of work that needs to be done by a task to amortize the overheads incurred for respective number of added processes. We then examined the multinode behaviour of Julia by measuring the latencies incurred with native Julia, Julia/MPI and compared them against C with MPI. We inferred that local floating point operations computed in a distributed fashion work better in native Julia and when Julia is optimized with MPI. On the other hand, local memory read and writes are cheaper in C/MPI. Distributed communications using MPI are not as optimized as for C and MPI whereas native Julia communication overheads increase significantly with an increased number of processes. Although Julia suffers from overheads in some cases, its relative ease of use, and performance close to the performance of present compiled languages indicates it is a viable contender for HPC runtime construction.

## 5 Related work

Most recent work on Julia focuses on using it to implement various algorithms for scientific computing [4].

There is, however, work on developing frameworks for parallel computation using Julia. For example, JuliaFEM [23] provides a concise programming framework for solving large-scale problems using finite element methods. Others have explored using high-level frameworks for GPU computing using Julia [28]. Chen et al. investigated using polymorphic code features in Julia applied to parallel and distributed computing [10].

Regent [25] is a high-level parallel programming language designed for implicit data-flow parallelism, which is built on Terra [13] and Legion [2]. While Regent provides a convenient and concise parallel programming model, it is ultimately built on top of a large C/C++ runtime system.

The Celeste [24] package is the first HPC application written entirely in Julia that has reported to achieve petascale performance.

## 6 Conclusions and future work

In the HPC community it is becoming more evident that there is a void for developer friendly parallel programming runtime systems which can also provide

the raw performance of existing lower-level solutions. We need a higher-level runtime system with support for higher-level parallel programming models as well as for existing underlying heterogeneous hardware and software system design. We carried out experiments on Julia, stressing on the performance of its various parallel constructs. We showed through an extensive set of microbenchmarks that the performance of Julia's parallel programming constructs in many cases is on par with C, both on single node and multi-node MPI setups.These results indicate that Julia is a viable candidate for building high-performance parallel runtimes. In future work, we intend to investigate the performance limitations of Julia's task scheduling system, the implications of its garbage collection and memory allocation subsystems on structured codes, and its (currently experimental) threading system. We also hope to expand our multi-node setup to a large-scale system and evaluate its performance on a more representative set of benchmarks and real applications. Finally, we hope to implement a high-performance parallel runtime in Julia using the insights we have gleaned from our investigations.

## References


1. Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D.G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., Zheng, X.: TensorFlow: A system for large-scale machine learning. In: Proceedings of the $12^{th}$ USENIX Symposium on Operating Systems Design and Implementation (OSDI 2016). pp. 265–283 (Nov 2016)
2. Bauer, M., Treichler, S., Slaughter, E., Aiken, A.: Legion: Expressing locality and independence with logical regions. In: Proceedings of Supercomputing (SC 2012) (Nov 2012)
3. Bezanson, J., Edelman, A., Karpinski, S., Shah, V.B.: Julia: A fresh approach to numerical computing. SIAM Review **59**(1) (Mar 2017)
4. Bezanson, J., Karpinski, S., Shah, V.B., Edelman, A.: Julia: A fast dynamic language for technical computing (2012)
5. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: An efficient multithreaded runtime system. Journal of Parallel and Distributed Computing **37**(1), 55–69 (1996)
6. Buttlar, D., Farrell, J.: Pthreads programming: A POSIX standard for better multiprocessing. O'Reilly Media, Inc. (1996)
7. Carlson, W.W., Draper, J.M., Culler, D.E., Yelick, K., Brooks, E., Warren, K.: Introduction to UPC and language specification. Tech. Rep. CCS-TR-99-157, IDA Center for Computing Sciences (May 1999)
8. Chamberlain, B.L., Callahan, D., Zima, H.P.: Parallel programmability and the Chapel language. International Journal of High Performance Computing Applications **21**(3), 291–312 (Aug 2007)
9. Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioğlu, K., von Praun, C., Sarkar, V.: X10: An object-oriented approach to non-uniform cluster computing. In: Proceedings of the $20^{th}$ ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2005). pp. 519–538 (Oct 2005)
10. Chen, J., Edelman, A.: Parallel prefix polymorphism permits parallelization, presentation amp;amp; proof. In: 2014 First Workshop for High Performance Technical Computing in Dynamic Languages (Nov 2014)
11. Dagum, L., Menon, R.: OpenMP: An industry-standard API for shared-memory programming. IEEE Computational Science & Engineering **5**(1), 46–55 (1998)
12. Dean, J., Ghemawat, S.: MapReduce: Simplified data processing on large clusters. In: Proceedings of the $6^{th}$ Symposium on Operating Systems Design and Implementation (OSDI 2004) (Dec 2004)
13. DeVito, Z., Hegarty, J., Aiken, A., Hanrahan, P., Vitek, J.: Terra: A multi-stage language for high-performance computing. In: Proceedings of the $34^{th}$ ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2013). pp. 105–116 (Jun 2013)
14. Group, O.W., et al.: The OpenACC application programming interface, version 1.0. `http://www.openacc.org/sites/default/files/OpenACC.1.0_0.pdf` (Nov 2011)
15. Hale, K.C., Dinda, P.A.: A case for transforming parallel runtimes into operating system kernels. In: Proceedings of the $24^{th}$ ACM Symposium on High-performance Parallel and Distributed Computing (HPDC 2015) (Jun 2015)
16. Hale, K.C., Hetland, C., Dinda, P.A.: Multiverse: Easy conversion of runtime systems into os kernels via automatic hybridization. In: Proceedings of the $14^{th}$ IEEE International Conference on Autonomic Computing (ICAC 2017) (Jul 2017)

17. Johansson, E., Pettersson, M., Sagonas, K.: A high performance Erlang system. In: Proceedings of the $2^{nd}$ ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP 2000). pp. 32–43 (Sep 2000)
18. Kivity, A., Laor, D., Costa, G., Enberg, P., Har'El, N., Marti, D., Zolotarov, V.: OSv—optimizing the operating system for virtual machines. In: Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC 2014) (Jun 2014)
19. Madhavapeddy, A., Mortier, R., Rotsos, C., Scott, D., Singh, B., Gazagnaire, T., Smith, S., Hand, S., Crowcroft, J.: Unikernels: Library operating systems for the cloud. In: Proceedings of the $18^{th}$ International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2013). pp. 461–472 (Mar 2013)
20. Merlin, J., Hey, A.: An introduction to High Performance Fortran. Scientific Programming **4**(2), 87–113 (Apr 1995)
21. Murray, D.G., McSherry, F., Isaacs, R., Isard, M., Barham, P., Abadi, M.: Naiad: A timely dataflow system. In: Proceedings of the $24^{th}$ ACM Symposium on Operating Systems Principles (SOSP 2013). pp. 439–455 (Nov 2013)
22. NVIDIA Corporation: CUDA C Programming Guide—Version 6.0 (Feb 2014)
23. Rapo, M., Aho, J., Frondelius, T.: Natural frequency calculations with juliafem pp. 300–303 (elo 2017)
24. Regier, J., Miller, A.C., McAuliffe, J., Adams, R.P., Hoffman, M.D., Lang, D., Schlegel, D., Prabhat: Celeste: Variational inference for a generative model of astronomical images. In: ICML (2015)
25. Slaughter, E., Lee, W., Treichler, S., Bauer, M., Aiken, A.: Regent: A high-productivity programming language for HPC with logical regions. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 2015) (Nov 2015)
26. Stone, J.E., Gohara, D., Shi, G.: OpenCL: A parallel programming standard for heterogeneous computing systems. Computing in Science Engineering **12**(3), 66–73 (May 2010)
27. Wheeler, K.B., Murphy, R.C., Thain, D.: Qthreads: An API for programming with millions of lightweight threads. In: Proceedings of the $22^{nd}$ International Symposium on Parallel and Distributed Processing (IPDPS 2008) (Apr 2008)
28. Yadav, A.C.: Construction of right gyrogroup structures on a non-abelian group. ArXiv e-prints (Apr 2016)
29. Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I.: Spark: Cluster computing with working sets. In: Proceedings of the $2^{nd}$ USENIX Conference on Hot Topics in Cloud Computing (HotCloud 2010) (Jun 2010)