Integrating Productivity-Oriented Programming Languages with High-Performance Data Structures

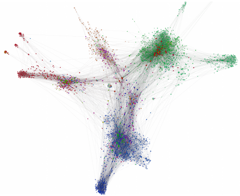
James Fairbanks
Rohit Varkey Thankachan, Eric Hein, Brian Swenson

Georgia Tech Research Institute

September 13 2017

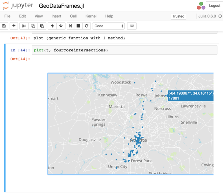
Graph Analysis

- Applications: Cybersecurity, Social Media, Fraud Detection...



(a) Big Graphs

(b) HPC



(c) Productivity

Georgia Tech

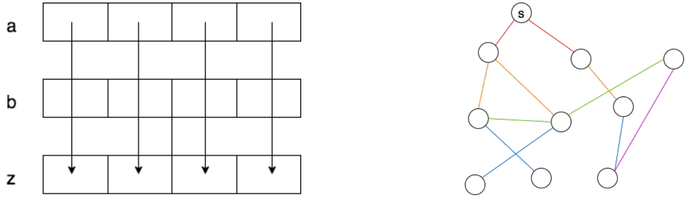
Types of Graph Analysis Libraries

- Purely High productivity Language with simple data structures
- Low level language core with high productivity language interface.

Name	High Level Interface	Low Level Core	Parallelism
SNAP	Python	C++	OpenMP
igraph	Python, R	C	-
graph-tool	Python	C++ (BGL)	OpenMP
NetworKit	Python	C++	OpenMP
Stinger	Julia (new)	C	OpenMP/Julia

Table 1: Libraries using the hybrid model

Why is graph analysis is harder than scientific computing?



(a) $z = exp(a + b^2)$ (b) BFS from s

Figure 2: Computations access patterns in scientific computing and graph analysis

- Less regular computation
- Diverse user defined functions beyond arithmetic
- Temporary allocations kill performance

High Productivity Languages

Feature	Python	R	Ruby	Julia
REPL	✓	✓	✓	✓
Dynamic Typing	✓	✓	✓	✓
Compilation	×	×	×	✓
Multithreading	Limited	×	Limited	✓

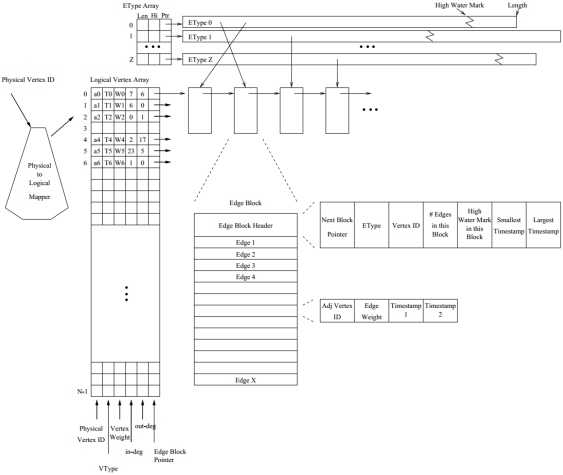
Table 2: Comparison of features of High Productivity Languages

The Julia Programming Language

- Since 2012 - pretty new!
- Multiple dispatch
- Dynamic Type system
- JIT Compiler
- Metaprogramming
- Single machine and Distributed Parallelism
- Open Source (MIT License)

STINGER

- A complex data structure for graphs in C
- Parallel primitives for graph algorithms



Addressing the 2 language problem using Julia

- Two languages incurs development complexity
- All algorithms in Julia
- Reuse only the complex STINGER data structure from C
- Parallel constructs in Julia, NOT low level languages

Integrating Julia with STINGER

- All algorithms in Julia
- Reuse only the complex STINGER data structure from C
- Parallel constructs in Julia, not low level languages
- Productivity + Performance!

Graph 500 benchmark

- Standard benchmark for large graphs
- BFS on a RMAT graph
  - 2^{scale} vertices
  - $2^{scale} * 16$ edges
- Comparing BFS on graphs from scale 10 to 27 in C and using StingerGraphs.jl
- A multithreaded version of the BFS with up to 64 threads was also run using both libraries

Results Preview

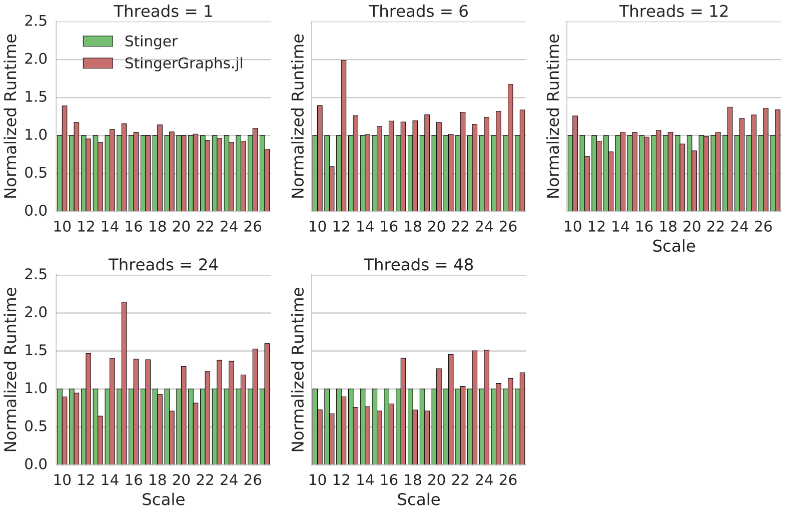


Figure 3: Graph500 Benchmark Results (Normalized to STINGER – C)

Legacy data structures require synchronizing memory spaces

Two approaches lead to different performance characteristics

Operation	Eager	Lazy
getfields	Already cached	Load pointer
setfields	Store pointer	Store pointer
ccalls	Load for every `ccall`	No op

Table 3: Methods for synchronizing C heap with Julia memory Lazy vs Eager

Moving data kills performance

Bulk transfer of memory between memory spaces is more expensive than direct iteration

Scale	Exp (I)	Exp (G)	BFS (I)	BFS (G)
10	1.03	2.43	252.17	1833.70
11	2.21	4.92	504.37	3623.40
12	4.64	10.33	1034.36	7239.56
13	9.70	21.04	2142.28	14461.98
14	20.79	44.18	4328.72	28767.98
15	58.11	107.91	12583.00	67962.16
16	127.92	225.55	27036.85	128637.68

Table 4: Iterators (I) vs Gathering successors (G) – all times in ms

Moving data kills performance

Bulk transfer of memory between memory spaces is more expensive than direct iteration

Scale	Exp (I)	Exp (G)	BFS (I)	BFS (G)
10	1.03	2.43	252.17	1833.70
11	2.21	4.92	504.37	3623.40
12	4.64	10.33	1034.36	7239.56
13	9.70	21.04	2142.28	14461.98
14	20.79	44.18	4328.72	28767.98
15	58.11	107.91	12583.00	67962.16
16	127.92	225.55	27036.85	128637.68

Table 4: Iterators (I) vs Gathering successors (G) – all times in ms

Surprise!

Parallelism options in Julia

- **MPI** style remote processes
- **Cilk** style Tasks that are lightweight “green” threads
- **OpenMP** style native multithreading support - `@threads`

We use the `@threads` primitives to avoid communication costs

Julia Atomics

- Atomic type on which atomic ops are dispatched
- `Atomic{T}` contains a reference to a Julia variable of type `T`
- Extra level of indirection for a vector of atomics

```
@eval unsafe_atomic_cas!(x::Ptr{$typ}, cmp::$typ, new::$typ) =
    llvmcall($"""
            %rv = cmpxchg $lt* %0, $lt %1, $lt %2 acq_rel
            ret $lt %rv
            """, $typ, Tuple{Ptr{$typ},$typ,$typ},
            x, cmp, new)
```

Figure 4: Julia provides easy access to LLVM/Clang intrinsics

Unsafe Atomics

Standard atomic types give poor performance, UnsafeAtomics.jl package reduces overhead.

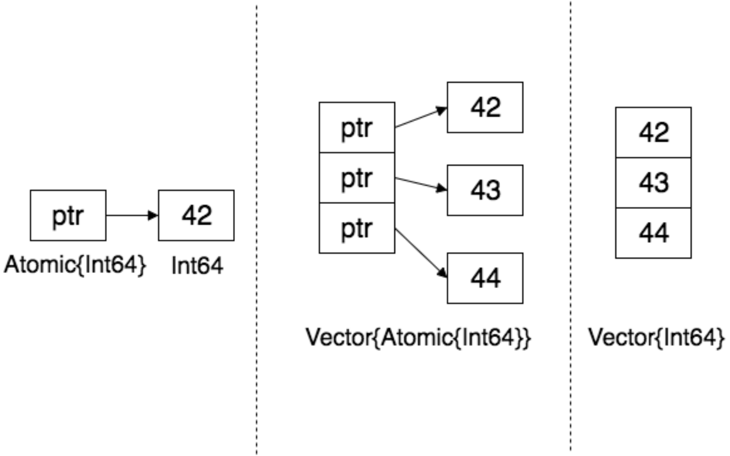


Figure 5: Atomic data structures in Julia

Unsafe Atomics Performance

Scale	Exp (N)	Exp (U)	Exp(N)/ Exp(U)	BFS (N)	BFS (U)	BFS(N)/ BFS(U)
10	0.13	0.1	1.3	47.23	43.27	1.10
11	0.27	0.23	1.17	98.99	91.32	1.08
12	0.62	0.47	1.32	217.44	190.74	1.14
13	1.31	0.97	1.35	505.59	420.84	1.20
14	2.7	2.17	1.24	1158.3	977.1	1.185
15	5.74	3.93	1.46	2576.18	2154.5	1.20
16	11.6	8.77	1.32	5565.87	4559.16	1.22

Table 5: Atomics: Native (N) VS Unsafe (U) (Times in ms)

Runtimes

Threads	STINGER	Stinger.jl	Slowdown
1	276.46	250.18	0.90x
6	169.93	237.21	1.40x
12	140.53	185.74	1.32x
24	97.73	145.83	1.49x
48	86.41	103.08	1.19x

Table 6: Total time to run Graph500 BFS benchmark for all graphs scale 10-27, in minutes

Results: Parallel Scaling is competitive with OpenMP

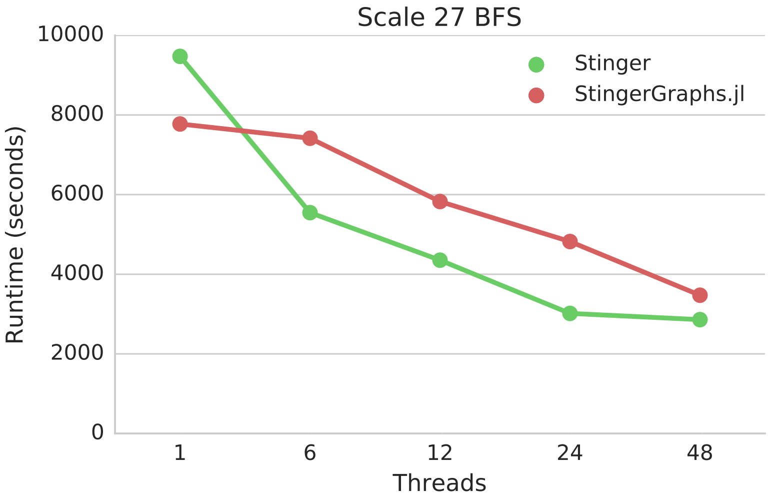


Figure 6: Performance scaling with threads

Conclusions

- Tight integration between high productivity and high performance languages is possible
- Julia is ready for HPC graph workloads
- Julia parallelism can compete with OpenMP parallelism
- We can expand HPC in High Level Languages beyond traditional scientific applications