Performance analysis of Julia for a set of parallel benchmarks

CSE 6230

Kapil Agarwal
Kanu Sahai

Contents

1. Introduction

Julia is a high-level, high-performance dynamic programming language for technical computing which facilitates a distributed parallel execution. Julia does not impose any particular style of parallelism on the user. Instead, it provides a number of key building blocks for distributed computation, making it flexible enough to support a number of styles of parallelism, and allowing users to add more. It provides a multiprocessing environment based on message passing to allow programs to run on multiple processes in separate memory domains at once. Julia's implementation of message passing is different from other environments such as MPI. Communication in Julia is generally one-sided, meaning that the programmer needs to explicitly manage only one process in a two-process operation.

Julia has been benchmarked for a set of serially implemented algorithms against other dynamic languages like R and Python, but not much analysis has been done on it's performance in case of parallel benchmarks. Through this project and in regard of performance analysis, we aim to develop parallel implementations of some benchmarks in Julia.

We draw inspiration from HPC Challenge Awards Competition at SC.

2. Background survey

Julia is a high level dynamic programming language which is interpreted but compiled using LLVM, so it is very fast approaching the performance of languages like C. It is useful for technical and high performance computing. It also has an interactive session shell in which Julia code can be run just like a Python shell. Multiple dispatch, dynamic type system, macros and metaprogramming facilities are a few of the main features of Julia. The current stable release is v0.3.3 which we will be using, although Julia development is progressing very rapidly and v0.4 may be released soon.

Benchmarking of Julia

To evaluate Julia's serial performance, its speed has been compared to that of six other languages: C++, Python, MATLAB, Octave, R, and JavaScript. Figure 1 shows timings for five scalar microbenchmarks, and two simple array benchmarks.

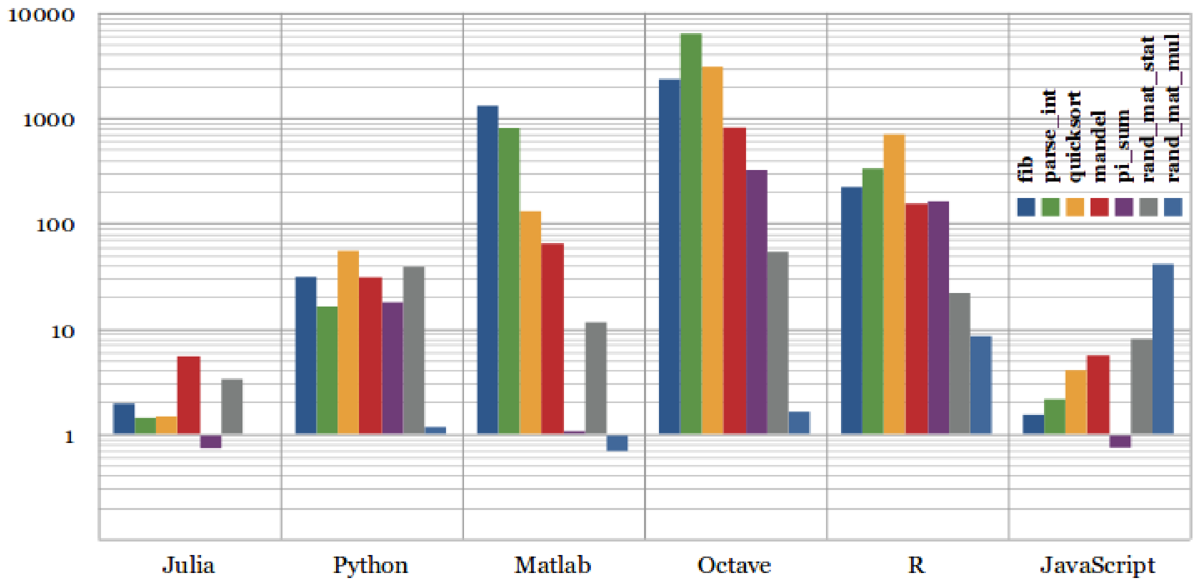


Figure-1 : *Microbenchmark results (times relative to C++, log-scale). These measurements were carried out on a MacBook Pro with a 2.53GHz Intel Core 2 Duo CPU and 8GB of 1066MHz DDR3 RAM. The following versions were used: Python 2.7.1, MATLAB R2011a, Octave 3.4, R 2.14.2, V8 3.6.6.11. The C++ baseline was compiled by GCC 4.2.1, taking best timing from all optimization levels. Native implementations of array operations, matrix multiplication, sorting, are used where available.*

The rand_mat_mul code demonstrates a case where time spent in BLAS dominates. MATLAB gets its edge from using a multi-threaded BLAS (threading is available in the BLAS Julia uses, but it was disabled when these numbers were taken). Julia is not yet able to cache generated native code, and so incurs a startup time of about two seconds to compile basic library functions.

Parallel programming paradigm and constructs used in Julia

Julia supports parallel computing by providing a message passing based multiprocessor environment so that programs can run on multiple processes in separate memory domains. Unlike MPI, communication in Julia is “one-sided” i.e. the programmer needs to explicitly manage only one process in a two-process operation. Instead of operations like “message send” and “message receive”, high level calls to user functions are provided. There are two primitives : “remote references” and “remote calls”. A remote reference is an object that acts as a reference to an object stored on another process. A remote call is a call to a function on another/same process and returns a remote references.

Julia can be started in parallel mode with either -p or --machinefile options. We will be using the -p option for programs that run on a single node and are currently exploring cluster managers for running on multiple nodes as --machinefile requires a passwordless ssh login. Unlike MPI, where the master process also participates in the parallel computation, instructions are started only on the worker processes in Julia and the master process is used for coordinating the processes and for getting the results. Julia's metaprogramming facilities allow the user to develop their own macros very easily and many of the Julia's APIs are macros written in Julia itself. "fetch" is an operation that is used to explicitly get receive data. @fetch and @spawn are constructs which can be used for implicit data transfer. Parallel reductions can be done using the @parallel construct and providing a reducer function. Operations can be started parallel on other processes using "pmap" function as well.

Julia provides distributed arrays which is a data structure that is divided among all or a set of worker processes such that the entire data cannot fit in one machine. A DArray has an element type and dimensions and the data is distributed by dividing the index space into blocks in each dimension. DArrays do not have much functionality as of yet, but their most important use is for communication to be done via array indexing. Like DArray, Julia also provides Shared Arrays which are currently only an experimental feature. Shared Arrays use system shared memory to map the same array across many processes. The difference between shared array and distributed array is that in DArray, each process has access to only its local chunk of the array while in SharedArray, each process has access to the entire array.

The above discussion was pertaining to multiple processes in a single machine. Julia provides Cluster Managers which is an interface to spawn processes on different machines. Currently, Julia supports only Sun Grid Engine, Scyld and HTCondor. The Jinx cluster is a PBS managed cluster for which Julia support does not currently exist. After browsing through julia-users mailing list archives, we found a third party implementation which was able to run Julia on a PBS based cluster, but it is not an official implementation.

Current status of Julia

Julia, being in its early developmental stage, has huge scope of contribution to the language. Many projects have been chalked out by the Julia community for the enhancement of the language. Julia had been accepted as a mentoring organization for GSoC'14. [3] lists a bunch of notable projects. Some work of interest which has been proposed in the area of parallel computing include :

- Building Julia wrappers for high performance GPU programming - Since Julia currently does not support GPU programming, libraries like cuBlas and cuFFT cannot be used for parallel programming in Julia.
- Building wrappers for parallel random number generation

- Providing support for dynamic distributed execution for data parallel tasks - Julia hasn't been explored much in a cluster computing environment. Hence, it is desired to weave together native Julia parallelism constructs such as the ClusterManager for massively parallel execution, and deal with data dependencies using native Julia RemoteRefs.

Existing parallel algorithm implementations in Julia

Currently only a handful of parallel algorithm implementations exist in Julia. A parallel FFT has been implemented by a group of students at MIT. Also, some native Julia implementations of massively parallel dense linear algebra routines have been written. The implementation involves two parallel tiled linear algebra algorithms in Julia, Cholesky and QR decomposition.

3. HPCC Benchmarks

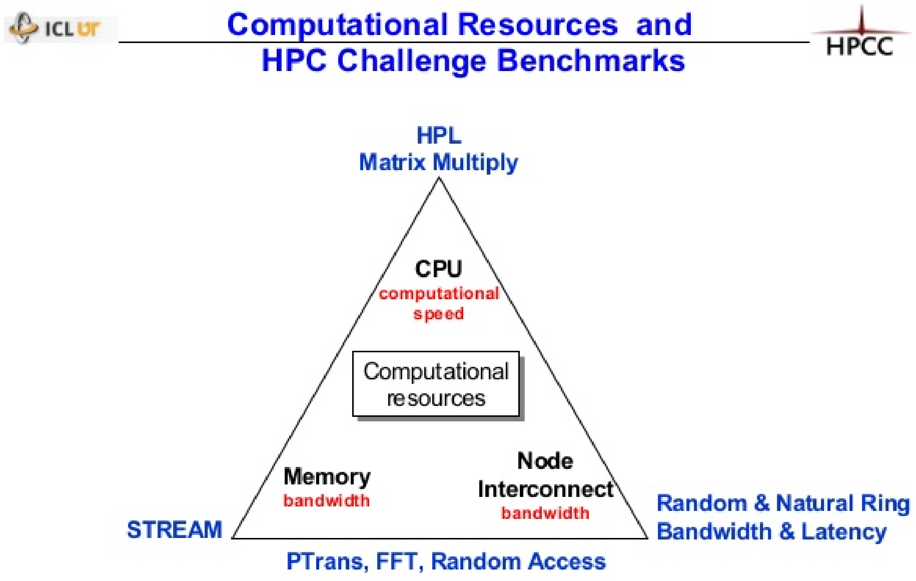


Figure-2 :

A set of benchmarks have been specified in HPC Challenge to measure the performance of different computational resources of a system. We have implemented four of these benchmarks which

1. Exercise the communications where pairs of processors communicate with each other simultaneously. It is a useful test of the total communications capacity of the network.
2. Measure sustainable memory bandwidth (in GB/s) and the corresponding computation rate for simple vector kernel.
3. Measure the rate of integer random updates of memory (GUPS).
4. Measure the floating point rate of execution of double precision complex one-dimensional Discrete Fourier Transform (DFT).

These benchmarks are discussed in the following sections.

4. STREAM

STREAM is a simple synthetic benchmark program that measures sustainable memory bandwidth (in GB/s) and the corresponding computation rate for simple vector kernel. The STREAM benchmark is designed to work with datasets much larger than the available cache on any given system, so that the results are more indicative of the performance of very large, vector style applications. STREAM is intended to measure the bandwidth from main memory. Each array must be at least 4 times the size of the sum of all the last-level caches used in the run or 1 Million elements, whichever is larger. The test consists of multiple repetitions of four of the kernels, and the best results of 10 trials are chosen. We use double precision floating point data elements with each element having a size of 8 bytes. FLOPS refers to the floating point operations per second. The kernels are the following:

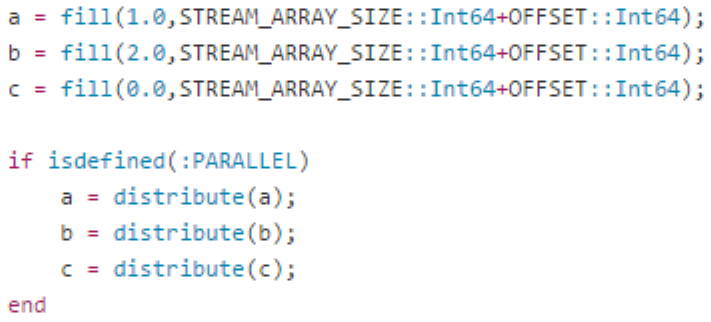
name	kernel	bytes/iter	FLOPS/iter
COPY:	a(i) = b(i)	16	0
SCALE:	a(i) = q*b(i)	16	1
SUM:	a(i) = b(i) + c(i)	24	1
TRIAD:	a(i) = b(i) + q*c(i)	24	2

Jinx has a L3 cache size of 8192 KB. We are running a parallelized implementation of STREAM benchmark on a maximum of 16 cores, so, we have considered arrays containing 64,000,000 elements each. We have implemented both serial and embarrassingly parallel versions of the benchmark. The serial implementation being straightforward updates the values of each element of the arrays according the kernels mentioned above. For the parallel implementation, we launch multiple worker processes among which the arrays are distributed. The same kernel is launched on each worker process which then performs the computation on its local part of the array. After the kernel is finished, the master process collects the result from the worker processes.

Julia Implementation

We use 3 arrays a, b and c each containing 64,000,000 elements of type Float64 (double precision). We then perform the operations Copy, Scale, Add and Triad on these arrays and measure the time to completion. In case of multiple processes, we use distribute() function which divides the arrays into smaller segments and sends it to each process. Now, each process will operate only on its local portion of the array and return it to the master process.

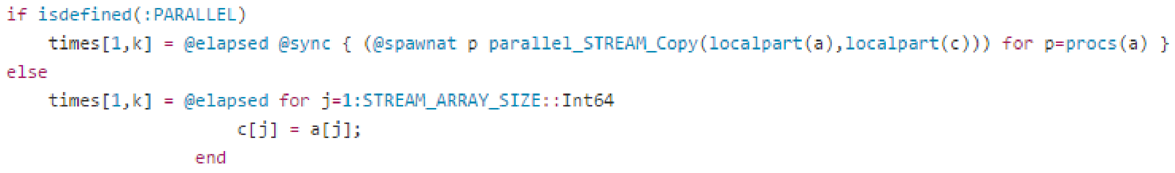
For example,



```
a = fill(1.0,STREAM_ARRAY_SIZE::Int64+OFFSET::Int64);
b = fill(2.0,STREAM_ARRAY_SIZE::Int64+OFFSET::Int64);
c = fill(0.0,STREAM_ARRAY_SIZE::Int64+OFFSET::Int64);

if isdefined(:PARALLEL)
    a = distribute(a);
    b = distribute(b);
    c = distribute(c);
end
```

The above code creates an array 'a' containing STREAM_ARRAY_SIZE number of elements with each element with a value of 1.0. Array 'a' is then distributed across all processes using distribute().



```
if isdefined(:PARALLEL)
    times[1,k] = @elapsed @sync { (@spawnat p parallel_STREAM_Copy(localpart(a),localpart(c))) for p=procs(a) }
else
    times[1,k] = @elapsed for j=1:STREAM_ARRAY_SIZE::Int64
                        c[j] = a[j];
                    end
```

The above code calls the function parallel_STREAM_Copy on each processor. The arguments passed to this function are the local parts of the arrays 'a' and 'c' that are resident on processor 'p'. @spawnat is a macro in Julia which call the function asynchronously on processor 'p' and returns a remote reference. @sync is a macro which acts as a barrier until all @spawnat calls have returned. @elapsed is a macro which returns the time taken to execute the given expression or function. We store the time in an array and find the average, minimum and maximum time and calculate the memory bandwidth achieved. It is important to note that during the first iteration, Julia compiles the program and so we have considered times taken from the 2nd iteration onwards. Similarly, Scale, Add and Triad have been implemented.

5. PTRANS

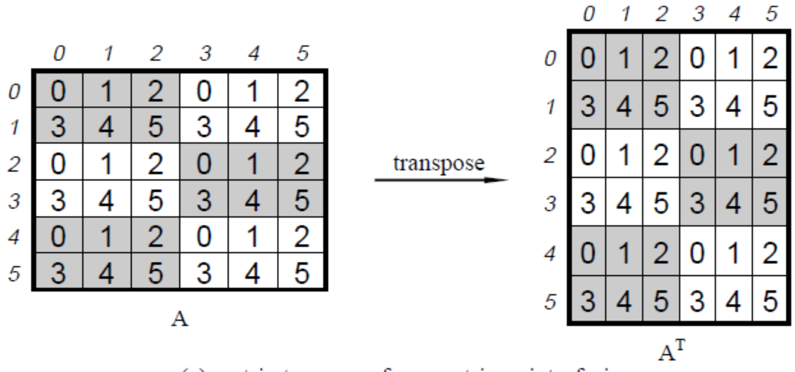
Matrix transpose is an important benchmark because it exercises the communications of computer heavily on a realistic problem where pairs of processors communicate with each other simultaneously. It is a useful test of the total communications capacity of the network. PTRANS (parallel matrix transpose) measures the rate of transfer for large arrays of data from multiprocessor's memory and exercises the communications where pairs of processors exchange large messages simultaneously. The performed operation sets a random *n* by *n* matrix to its transpose:

$$A \leftarrow A^T \text{ where: } A \in \mathrm{R}^{n \times n}$$

The data transfer rate (in MB/s) is calculated by dividing the size of n^2 matrix entries by the time it took to perform the transpose.

Algorithm

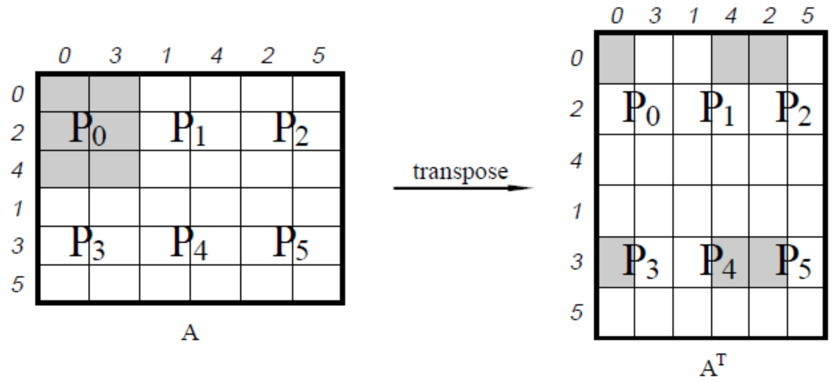
The matrix is distributed over a two dimensional P X Q processor template with a block scattered data distribution. The algorithm uses non-blocking, point-to-point communication between processors which allows a processor to overlap the messages it sends to different processors.



(a) matrix transpose from matrix point-of-view

Figure-3

In Figure 3, we consider a 30 X 30 matrix with 5 X 5 size blocks and distribute these blocks on a 2 X 3 processor mesh. As it can be seen that each processor has several blocks of the matrix. After transposing the matrix, A^T is distributed over the same 2 X 3 template. The elements of each block remain in the same block and each block is itself transposed. But, these blocks can now be in a different processor.



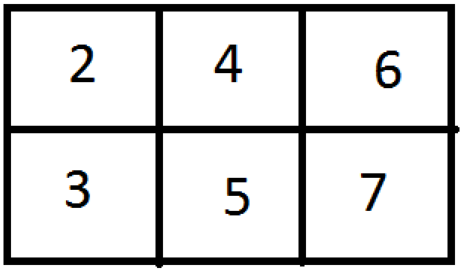
(b) matrix transpose from processor point-of-view

Figure-4

In the case of a 2 X 3 processor mesh, P and Q are relatively prime. As shown in Figure 4, the blocks are scattered to all processors after being locally transposed. This case involves every processor to communicate with every other processor. In the case where P and Q are not relatively prime, different communication patterns are observed which are determined by the greatest common divisor (GCD) of P and Q. This is because each block is not distributed to every processor but at intervals of size GCD.

Julia Implementation

We create a matrix A of dimensions 480 X 480 with blocks of size 5 X 5 and fill it with random values of type Float64 (double precision). In Julia, distribute() function can scatter only contiguous regions of the matrix among different processes. For example, in Figure 3, blocks (0,0), (0,3), (2,0), (2,3), (4,0) and (4,3) belong to process 0. We first rearrange the matrix into the processor point-of-view as shown in Figure 4 (left matrix). Now, each process computes the transpose of each of its blocks locally and sends it to the destination process. As shown in Figure 4, process 0 will compute the transpose of its blocks and send them to every other process. In Julia, distribute() function distributes the matrix into the processes in a vertical fashion as shown in Figure 5 (as opposed to that shown in Figure 3). Also, the master process is indexed 1 and the worker processes are indexed from 2. Here, 2 to 7 are the process ids.



2	4	6
3	5	7

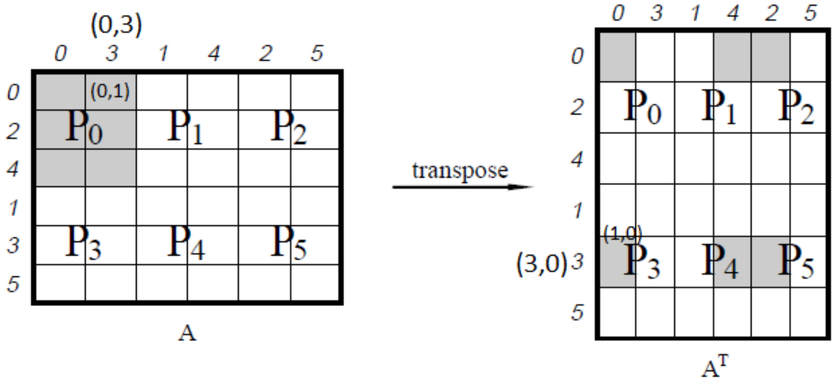
Figure-5

Consider a matrix A which has been block distributed on a processor template of dimensions P X Q. Consider the index (i,j) in the matrix. After transposing the matrix, the data at (i,j) will be moved to index (j,i) in the matrix. In the PTRANS implementation, we transpose a block locally and then send exchange the blocks at locations (x,y) and (y,x) to obtain the transpose of the entire matrix. For example, as can be seen from Figure 6, the block at location (0,3) which belongs to process 0 after transpose is to be sent to location (3,0) which belongs to process 3. We can calculate this as follows-

Here P=2 and Q=3 which are the dimensions of the processor template used in this case.

Initial block = (0,3)
Initial process id = (mod(0,P), mod(3,Q)) = (0,0)
Initial local block location = (0/P, 3/Q) = (0,1)

New block = (3,0)
New process id = (mod(3,P), mod(0,Q)) = (1,0) = 1*3+0 = 3
New local block location = (3/P, 0/Q) = (1,0)



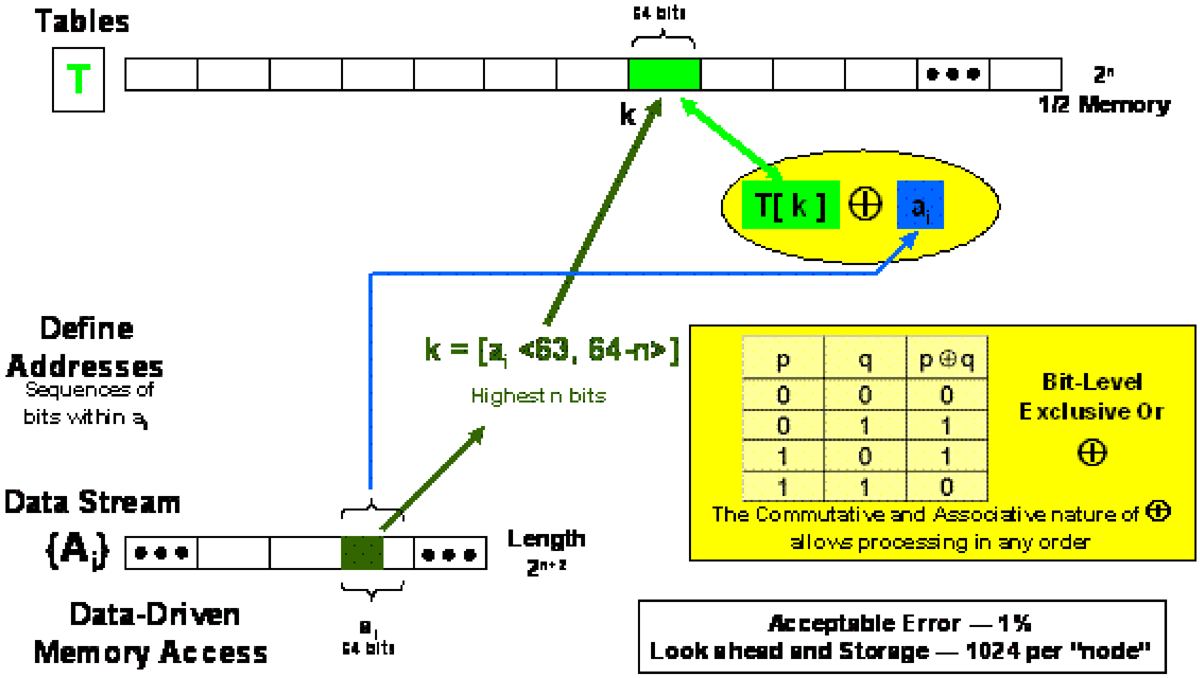
(b) matrix transpose from processor point-of-view

Figure-6

We measure the time it takes to transpose the matrix obtained after converting it into the processor point-of-view. After this, we rearrange the matrix back to the matrix point-of-view to obtain the transpose.

6. Random Access

GUPS (Giga UPdates per Second) is a measurement that profiles the memory architecture of a system and is a measure of performance similar to MFLOPS. GUPS is calculated by identifying the number of memory locations that can be randomly updated in one second, divided by 1 billion (1e9). The term "randomly" means that there is little relationship between one address to be updated and the next. An update is a read-modify-write operation on a table of 64-bit words. An address is generated, the value at that address read from memory, modified by an integer operation (add, and, or, xor) with a literal value, and that new value is written back to memory.



p	q	p⊕q
0	0	0
0	1	1
1	0	1
1	1	0

Figure-7

A table T[ ] is generated having 2^n entries of 64 bit unsigned integers. The parameter n is defined such that n is the largest power of 2 that is less than or equal to half of main memory. Then, a data-stream {A} is generated having 4 times the table size (2^{n+2}) entries of 64 bit unsigned integers. From each of this a_i in A, highest n bits are used to determine a location in the main table (denoted by k) and T[k] is then xor-ed with a_i.

Two additional variables have been defined which are - Acceptable error and Storage. Since updates are applied to table locations in parallel, there is a possibility of losing some updates due to simultaneous read-modify-write at some locations. Hence, an error rate of 1% is defined acceptable. Also, any process can bucket upto 1024 updates before applying them to table. This decreases some overheads and leads to improved performance.

Random Access can be run in three variants :

1. Single process - T[ ] is sized to half of process' memory
2. All processes perform identical to single process with no interactions (i.e.,

embarrassingly parallel)

3. All processes cooperate to solve a larger problem
   a. The table T[ ] is distributed over all processes and uses half (rounded down) of all system memory
   b. Each process calculates a portion of the address stream {A_i} starting at equal spacing. A process may not calculate values outside of its range

For this project and as required by HPC Challenge Awards competition rules, we have implemented the third variant of Random Access (Global variant).

7. Parallel FFT

Like MATLAB and other similar languages focussed on scientific computing, Julia provides an in-built function “fft()” to compute FFT of a given array of coefficients. Parallel implementation of FFT is adopted to compute FFT of large input efficiently. A freely available package FFTW is a C subroutine library for computing the discrete Fourier transform (DFT) in one or more dimensions, of arbitrary input size, and of both real and complex data. It also provides parallelized code for platforms with SMP machines with some flavor of threads (e.g. POSIX) or OpenMP. An MPI version for distributed-memory transforms is also available in FFTW 3.3. However, no such package exists for Julia yet. Hence, in order to implement a parallel version of FFT in Julia, a parallel FFT algorithm has to be implemented from ground up.

Algorithm

The discrete Fourier transform (DFT) is defined by the formula:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N} nk},$$

where k is an integer ranging from 0 to N-1.

An easy way to think about the aforementioned concept and a simple application of FFT is evaluation of a polynomial (given it’s coefficients) of order N at N points. So, if

$$A(x) = a_0 + a_1 x + a_2 x^2 ....a_n x^{n-1}$$

Then, input = $[a_0,\ a_1, ...a_{n-1}]$ to FFT would give us value of the polynomial at Nth roots of unity.

Parallel FFT is based on Cooley-Tukey algorithm which breaks the DFT into smaller DFTs. A radix-2 decimation-in-time (DIT) FFT is the simplest and most common form of the

Cooley–Tukey algorithm. Radix-2 DIT first computes the DFTs of the even-indexed inputs $(x_{2m} = x_0, x_2, \ldots, x_{N-2})$ and of the odd-indexed inputs $(x_{2m+1} = x_1, x_3, \ldots, x_{N-1})$, and then combines those two results to produce the DFT of the whole sequence. This idea can then be performed recursively to reduce the overall runtime to O(N log N).

So, if we again consider the evaluation of polynomial, Cooley-Tukey algorithm makes use of the fact that A(x) can be represented as :

$$A(x) = A_e(x^2) + x.A_o(x^2)$$

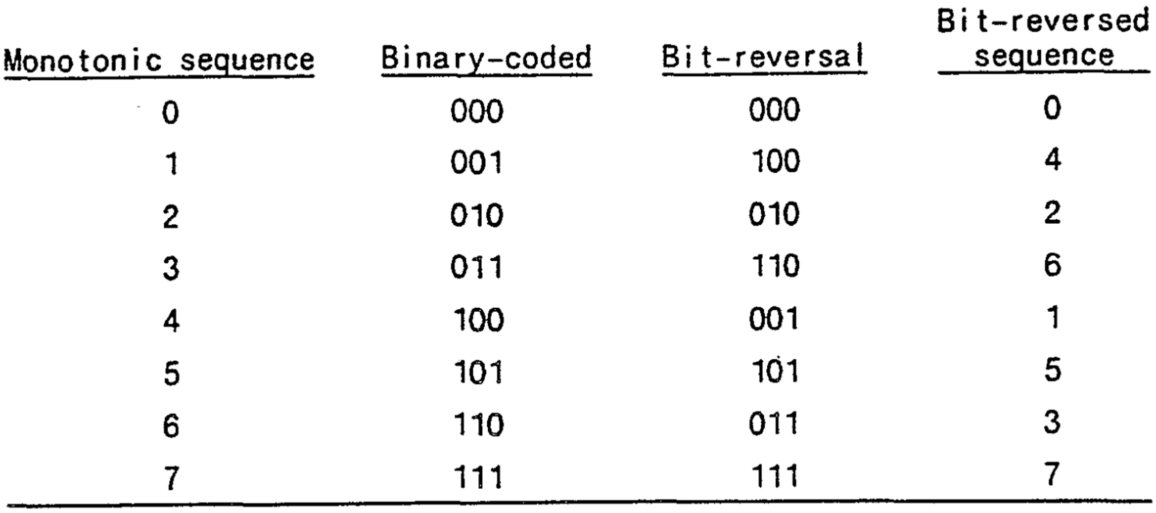
where, $A_e(x) = a_0 + a_2x + a_4x^2\ldots$ and $A_o(x) = a_1 + a_3x + a_5x^2\ldots..$

Owing to the periodicity of roots of unity, to evaluate a polynomial A(x)at 8th roots of unity, one would need to evaluate $A_o(x)$ and $A_e(x)$ at 4th roots of unity. This process can be applied recursively.

Julia Implementation

Following are some concise points describing parallel FFT implementation in Julia :

- Input is represented as a DArray (distributed among different worker processes)
- A monotonic sequence input coefficients is firstly represented as a bit-reversed sequence before distributing it



Monotonic sequence	Binary-coded	Bit-reversal	Bit-reversed sequence
0	000	000	0
1	001	100	4
2	010	010	2
3	011	110	6
4	100	001	1
5	101	101	5
6	110	011	3
7	111	111	7

Figure-8

This helps in the Cooley-Tukey butterfly algorithm.

- Each processor node performs sequential fft (using in-built fft() call of Julia) on its local chunk of input
- For later stages, communication between nodes is required

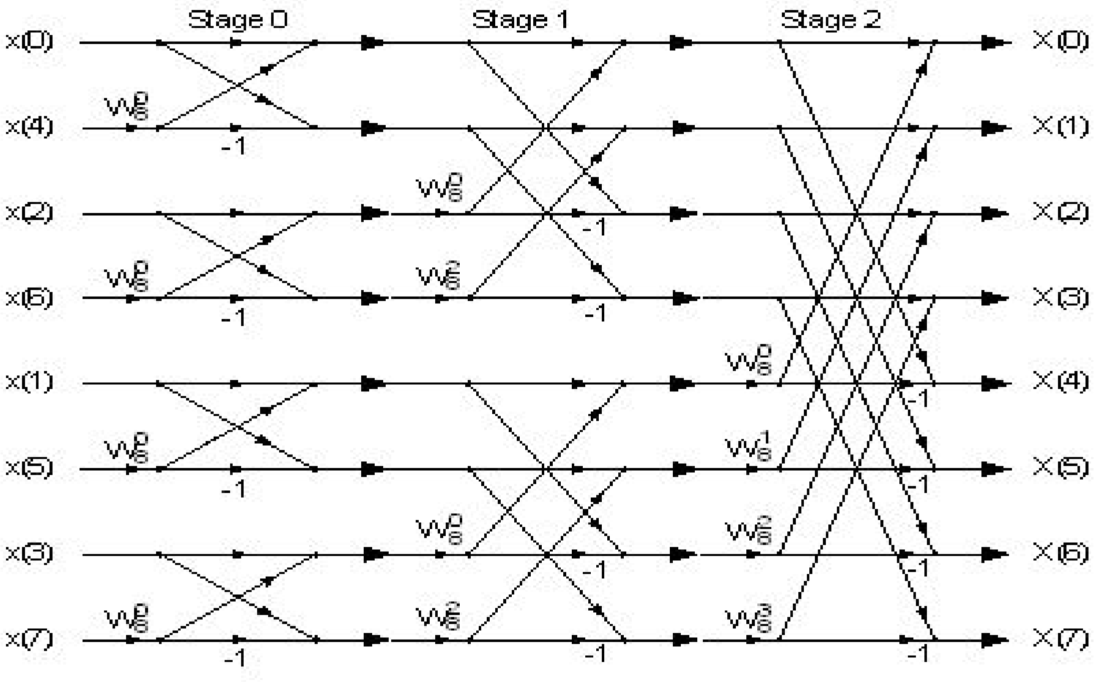


Figure-9

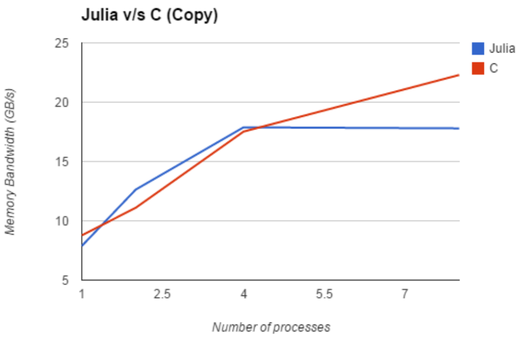
Complexity Analysis

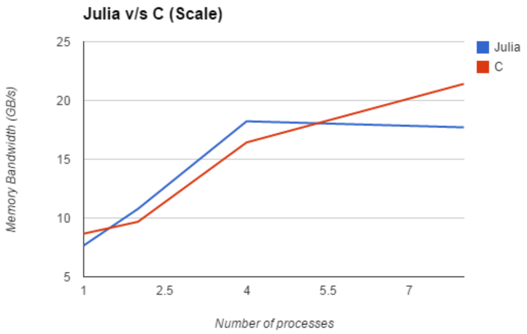
For an input size of N and number of processes = P,
# of stages = log2(N)
# of stages computed locally = log2(N/P)
# of stages requiring inter-node communication = log2(P)

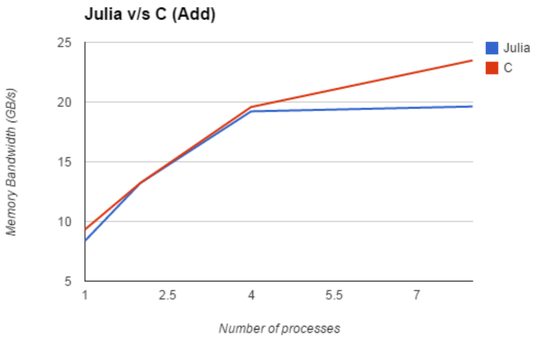
8. Results and Analysis

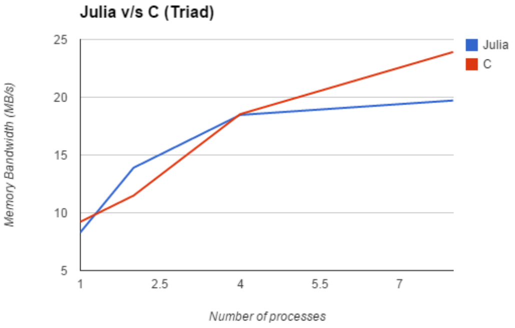
STREAM

We run several experiments with the STREAM benchmark and report the performance (or memory bandwidth) in GB/s.

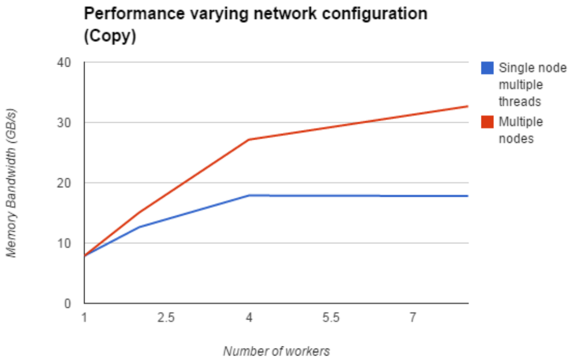

Julia v/s C (Copy)

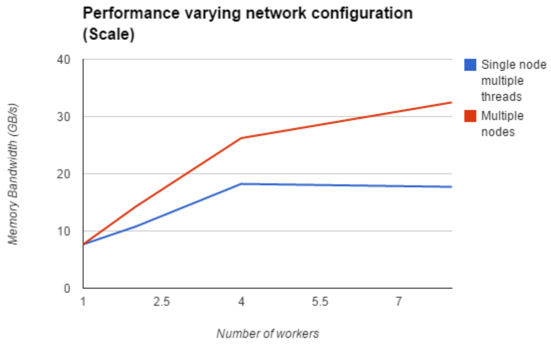

Julia v/s C (Scale)


Julia v/s C (Add)

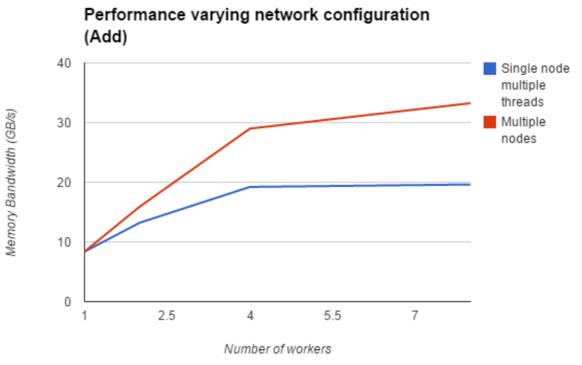

Julia v/s C (Triad)

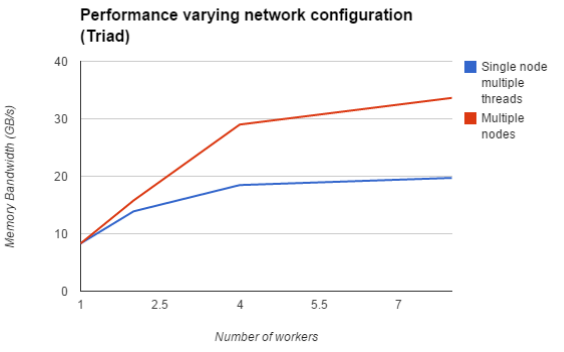
In the first experiment, we ran the STREAM benchmark in both Julia and C programming languages. Since the MPI version of STREAM was not available from the benchmark website, we ran the codes on a single machine and launched multiple processes to achieve parallelism. In Julia, we added processes using the addprocs() function which adds the number of worker threads as provided in the parameter. In C, we used OpenMP programming paradigm and set the value of OMP_NUM_THREADS environment variable to launch the required number of threads. The above graphs show that the performance of Julia matches very well with C.


Performance varying network configuration (Copy)

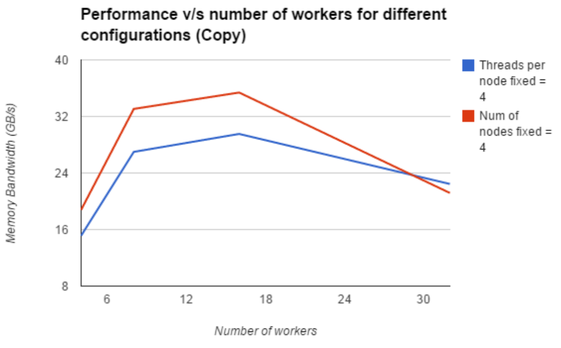

Performance varying network configuration (Scale)

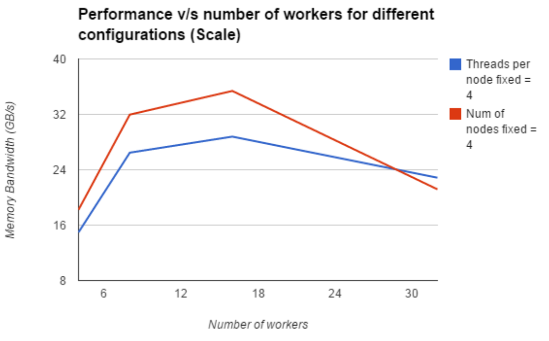
.

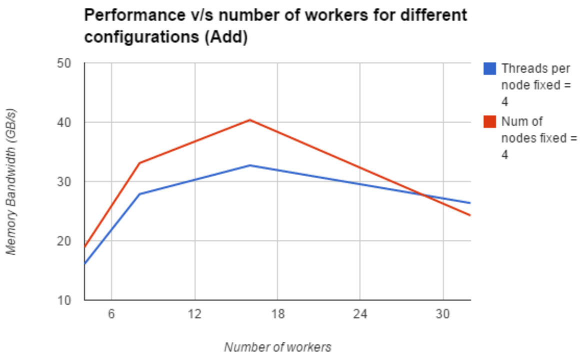

Performance varying network configuration (Add)

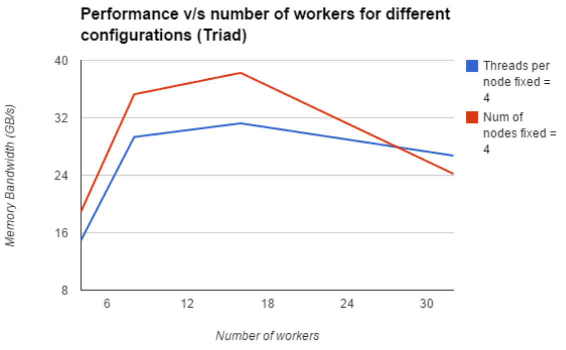

Performance varying network configuration (Triad)

In the second experiment, we compare the performance of STREAM benchmark in Julia using two system configurations. First, we keep the number of nodes fixed to 1 and keep increasing the number of threads. Second, we increase the number of nodes and launch only a single thread on each node. As it can be seen from the above graphs, the performance of running the benchmark on multiple nodes is more than running multiple threads on a single node. This happens because the L3 cache is shared among all the cores in a machine and as we increase the number of threads, there is a lot contention and page faults are quite frequent leading to lots of disk operations. In case of multiple nodes, each machine has its own L3 cache due to which there is less contention and hence the performance is better.


Performance v/s number of workers for different configurations (Copy)


Performance v/s number of workers for different configurations (Scale)

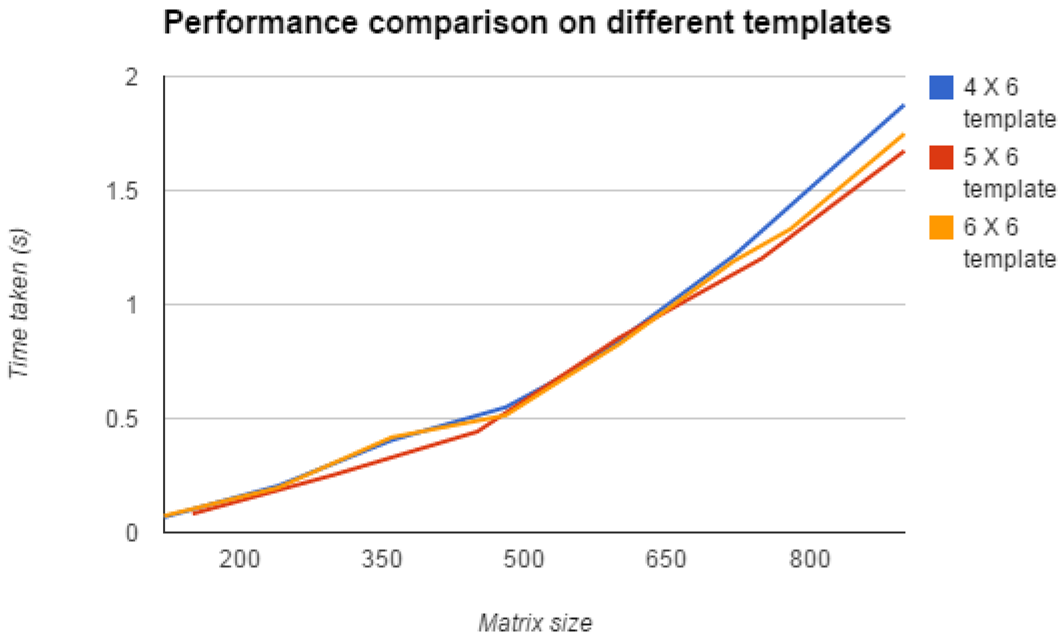

Performance v/s number of workers for different configurations (Add)


Performance v/s number of workers for different configurations (Triad)

In the third experiment, we compare the performance of running the benchmark on the jinx cluster by varying the number of processes. We use two different scenarios- one, where we launch an equal number of threads equal to 4 on each node and vary the number of nodes and second, where we keep the number of nodes fixed equal to 4 and vary the number of threads on each node. We see similar trends in both cases. However, what is surprising is the drop in performance after certain threshold number of workers. Initially, we had few worker threads and more work to be done and so the performance increased with the number of workers as more resources were being able to be utilized and hence more parallelism. But after a certain limit, the number of workers were much more than the work and adding more workers led to an increase in the memory latency and hence the drop in performance.

PTRANS

We ran the PTRANS benchmark for varying matrix sizes and processor templates. As there are only 16 nodes on the jinx cluster, we could not make every worker work on a different node. So, for a P X Q mesh, we used P nodes and on every node, we launch Q threads. The below graph shows the time taken to compute the transpose for 3 templates as the matrix size increases. Intuitively, it should take a longer time to transpose a larger matrix and the trend observed is as expected.



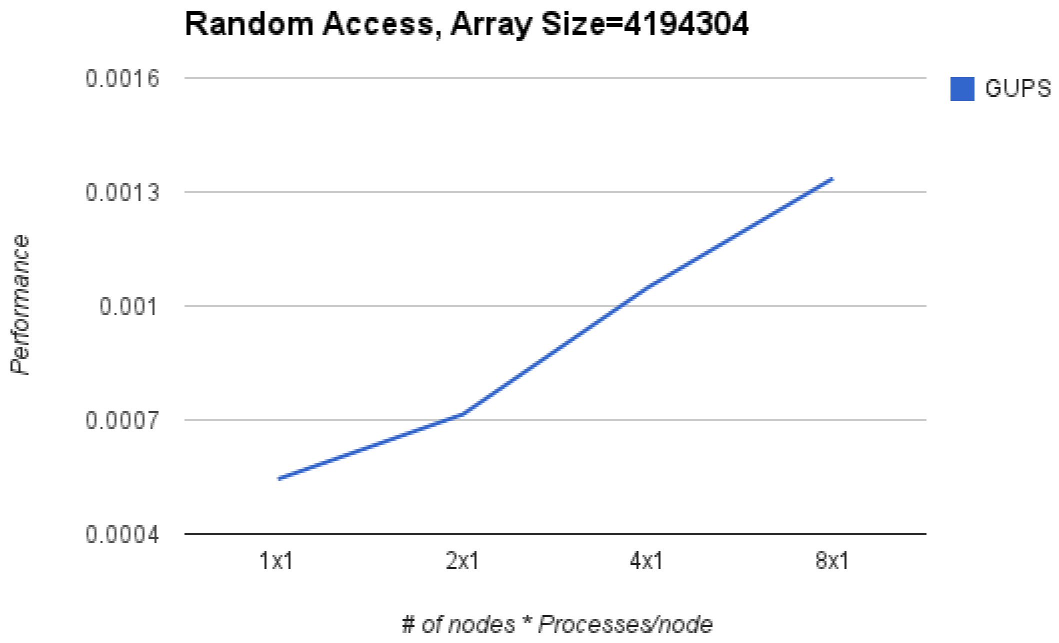
We also varied the number of worker threads keeping the matrix size fixed to 480 X 480. The below table shows the time taken to transpose the matrix for different mesh sizes.

72 workers		64 workers		48 workers	
6 X 16	0.943196 s	4 X 16	0.793438 s	4 X 12	0.68907 s
8 X 12	0.788329 s	8 X 8	0.739207 s	6 X 8	0.596981 s
12 X 8	0.817427 s			8 X 6	0.569312 s
				12 X 4	0.588479 s

As we can see that the time to transpose decreases as Q decreases. This means that we can get higher performance by distributing the matrixes more across rows than columns.

Random Access

As the number of processes increase in this benchmark, due to increasing parallelism, the performance also rises. However, as mentioned previously, due to simultaneous updates of locations in the array some updates may be lost and hence an acceptable failure rate of 0.01% had been mentioned for the benchmark. In Julia, there is one interesting observation. It compromises on the performance of random access and always ends up giving a failure rate of zero. One reason which can be associated with this is that Julia doesn’t support asynchronous sends and receives. It just spawns a task on the required processor. All tasks which are to be performed on a processor must take place sequentially. So, when a task of updating an address is spawned on the destination processor, it gets queued and executed eventually in order.



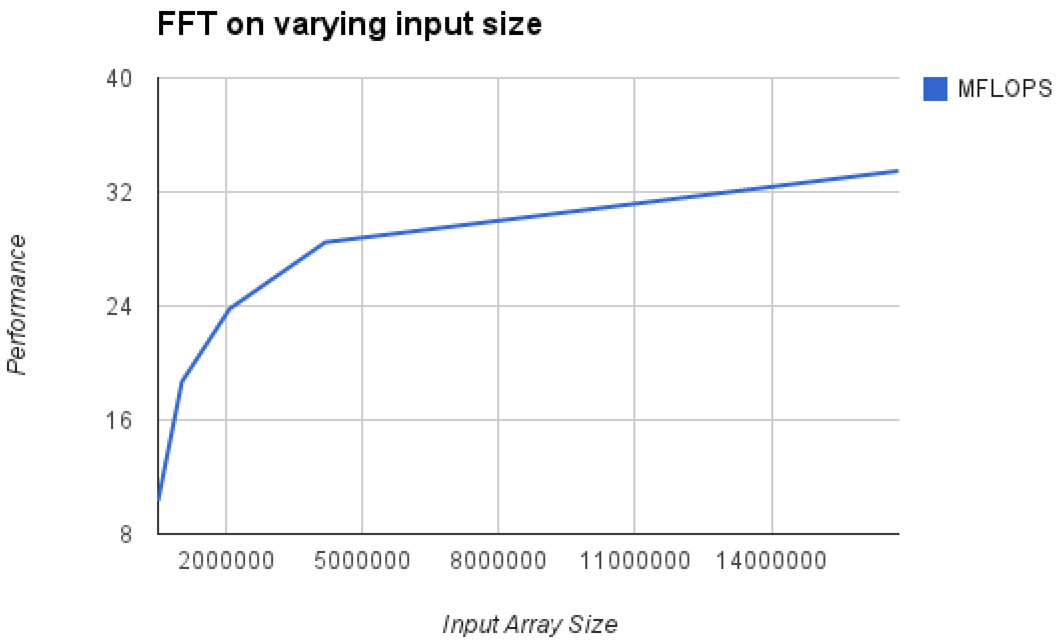
Parallel FFT

To report FFT performance, we plot the "MFLOPS" of each FFT, which is a scaled version of the speed, defined by :

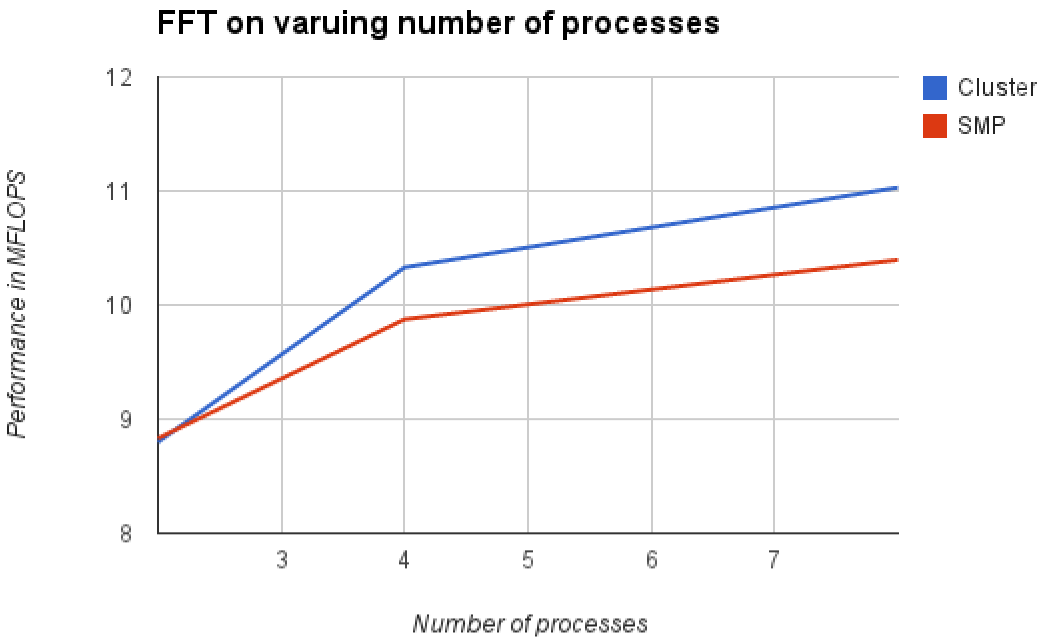
MFLOPS = 5 N $\log_2(N)$ / (time for one FFT in microseconds) for complex transforms
MFLOPS = 2.5 N $\log_2(N)$ / (time for one FFT in microseconds) for real transforms

where N is number of data points (the product of the FFT dimensions). This is not an actual flop count; it is simply a convenient scaling, based on the fact that the radix-2 Cooley-Tukey algorithm asymptotically requires 5 N $\log_2(N)$ floating-point operations. It allows us to compare the performance for many different sizes on the same graph, get a sense of the cache effects, and provide a rough measure of efficiency relative to the clock speed.

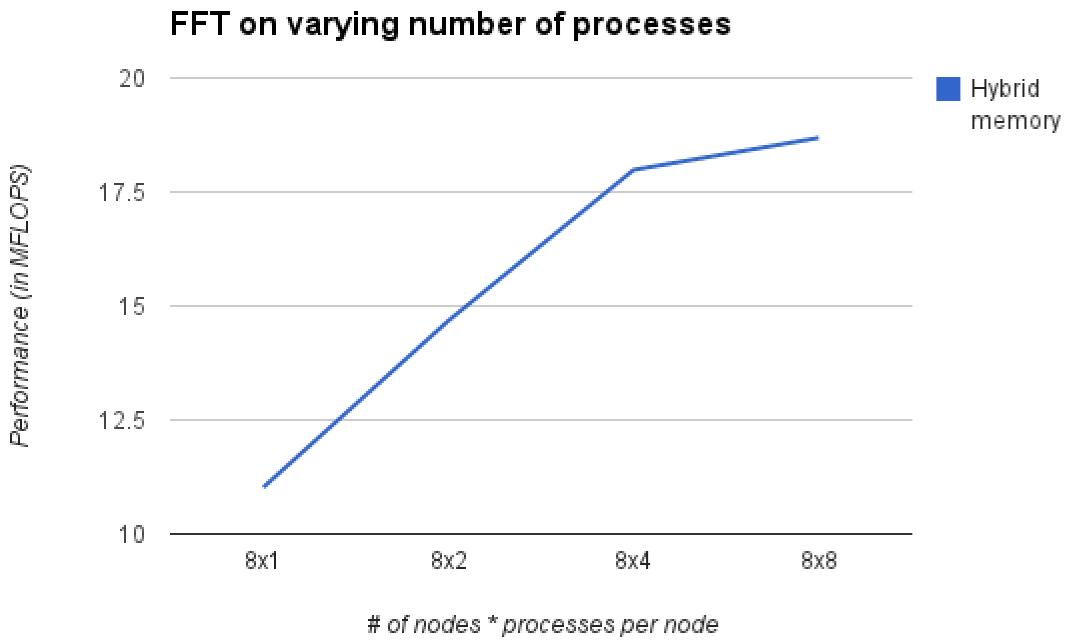
Running FFT on a single process gives the maximum performance of around 44.45 MFLOPS as the entire computation is local and no network latency is encountered. However, as number of processes increase, there is a tradeoff between the network latency faced and parallelism introduced.



FFT on varying input size



FFT on varuing number of processes



9. Ending note

While experimenting with PTRANS benchmark, we discovered that Julia was unable to launch the worker threads and was giving Memory errors. We asked about this on the julia-users mailing list and got to know that this is a bug in Julia. We have created an issue for the same.

We have implemented four HPC Challenge benchmarks and analysed the performance of Julia. Julia is still in a nascent stage of development and shows a lot of promise. It has a simple to understand programming model and provides good performance.

https://github.com/JuliaLang/julia/issues/9149
https://groups.google.com/forum/#!searchin/julia-users/kapil/julia-users/0XTidbuInbQ