Final Project Proposal

My inspiration derives from Homework4. In homework4, we explore the OpenMPI of message sending and receiving as well as the matrix multiplication comparison between cpu parallel and gpu parallel. So, in final project I would like to extend the 2 libraries to different languages, which is c++, Python, Golang, Julia, Rust

PART I

OpenMPI comparison between c++, Python, Golang, Julia, Rust
OpenMPI for Python
https://mpi4py.readthedocs.io/en/stable/
OpenMPI for Golang
https://pkg.go.dev/github.com/cpmech/gosl/mpi
OpenMPI for Julia
https://juliaparallel.org/MPI.jl/stable/
OpenMPI for Rust
https://rsmpi.github.io/rsmpi/mpi/index.html

I will use the basic message sending and histogramming program, as in homework4, to test the running time of openMPI between the 5 languages above.

PART II

CUDA comparison between c++, Python, Golang, Julia, Rust
CUDA for Python
https://nvidia.github.io/cuda-python/overview.html
CUDA for Golang
https://pkg.go.dev/gocv.io/x/gocv/cuda
CUDA for Julia
https://cuda.juliagpu.org/stable/
CUDA for Rust
https://docs.rs/rustacuda/latest/rustacuda/index.html

I will use the vector add and matrix multiply to test the running time of CUDA between the 5 languages above.

PART III

Multi-threading comparison between c++, Python, Golang, Julia, Rust
Pthreads in c++
Threading in Python
https://docs.python.org/3/library/threading.html
Goroutine in Golang
https://go.dev/tour/concurrency/1
Multi-Threading in Julia
https://docs.julialang.org/en/v1/manual/multi-threading/
Threads in Rust
https://doc.rust-lang.org/book/ch16-01-threads.html

We could use the same matrix multiply in Part II to test for the running time between different languages. At the same time we could also compare the running time on cpu and gpu of the same language.

As for the conclusion of the final project, I will use charts to directly show the difference between each language's running time. Moreover, I will analyze the possible reason for any differences.

This project will be a good opportunity for me to learn Julia and Rust because I have never worked with those two languages before. Meanwhile, I could have a deeper understanding of openMPI and CUDA GPU computing, which are two important HPC libraries. However, for libraries of other languages, they may be just a simple conversion or translation of c/c++ based library, so the running time may be significantly slower than c++ version. We have to note that here.

If the professor thinks my final project proposal is somehow meaningless or has some overlap with the example project by previous students, please let me know and I will change the direction at once. Thanks!

REFERENCE:

1. https://mpi4py.readthedocs.io/en/stable/
2. https://pkg.go.dev/github.com/cpmech/gosl/mpi
3. https://juliaparallel.org/MPI.jl/stable/
4. https://rsmpi.github.io/rsmpi/mpi/index.html
5. https://nvidia.github.io/cuda-python/overview.html
6. https://pkg.go.dev/gocv.io/x/gocv/cuda

7. https://cuda.juliagpu.org/stable/
8. https://docs.rs/rustacuda/latest/rustacuda/index.html
9. https://docs.python.org/3/library/threading.html
10. https://go.dev/tour/concurrency/1
11. https://docs.julialang.org/en/v1/manual/multi-threading/
12. https://doc.rust-lang.org/book/ch16-01-threads.html

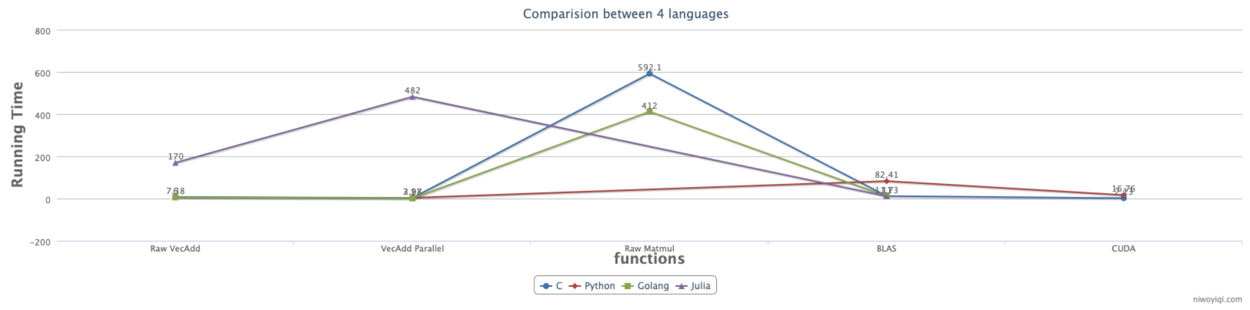
Final Project Report

Target and milestones

1. Implementing vector addition and matrix multiplication on all 4 languages
2. Compare the performance with multithreading
3. Using popular matrix multiplication package BLAS and GEMM function
4. Try to compare the GPU performance of multiplication(C && Python only)
5. All the benchmarks are done by 1,000,000 items vector, 10 threads and 512 * 512 matrix

Final result in chart

Functions	C	Python	Golang	Julia
Raw VecAdd	7.38	5460.58	6	170
VecAdd Parallel	2.98	3.27	1	482
Raw Matmul	592.1	972034.18	412	22297
BLAS	11.73	82.41	11	11
CUDA	2.13	16.76		

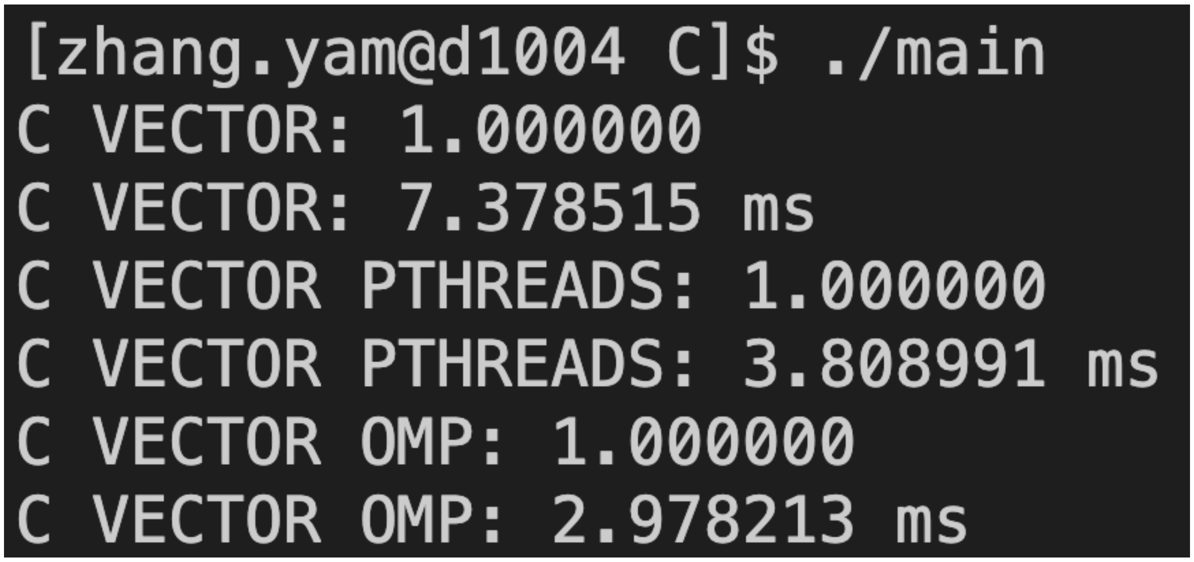


C

The Vector Add

Brute Force	7.378515 ms

Pthread	3.808991 ms
OpenMP	2.978213 ms

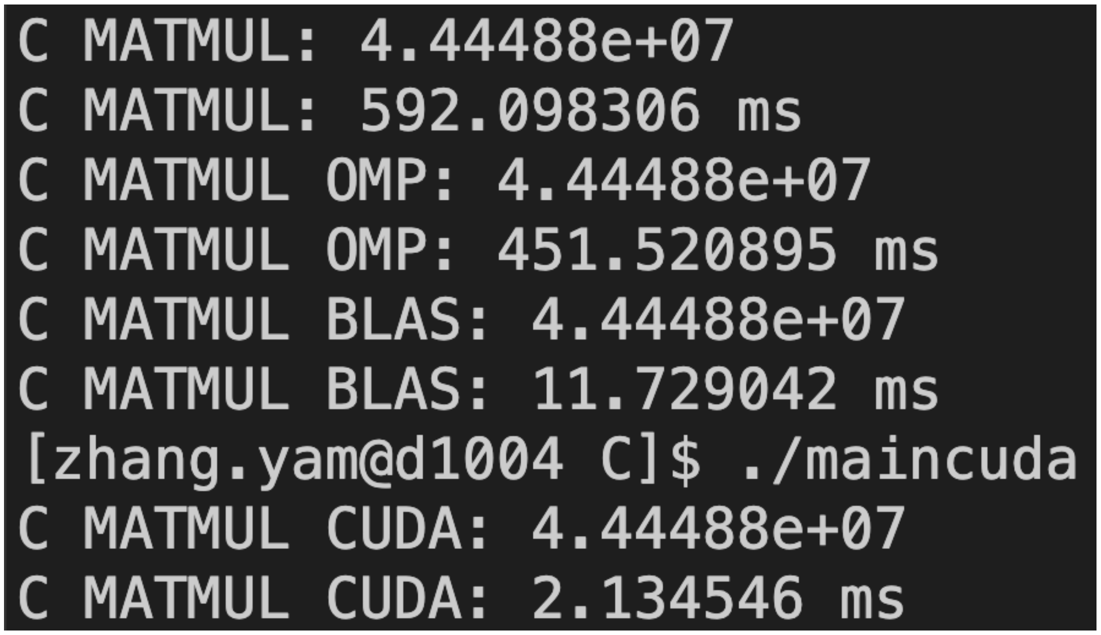


```
[zhang.yam@d1004 C]$ ./main
C VECTOR: 1.000000
C VECTOR: 7.378515 ms
C VECTOR PTHREADS: 1.000000
C VECTOR PTHREADS: 3.808991 ms
C VECTOR OMP: 1.000000
C VECTOR OMP: 2.978213 ms
```

We could find that the running time of Pthreads and Openmp is significantly improved. In the comparison of Pthreads and Openmp, Openmp is better not only because of its running time but also of the simpleness of the code writing. We only need a clause to finish the parallel part of the vector addition.

The Matrix Multiplication

Brute Force	592.098306 ms
OpenMP + block	451.520895 ms
BLAS.GEMM	11.729042 ms
CUDA	2.134546 ms



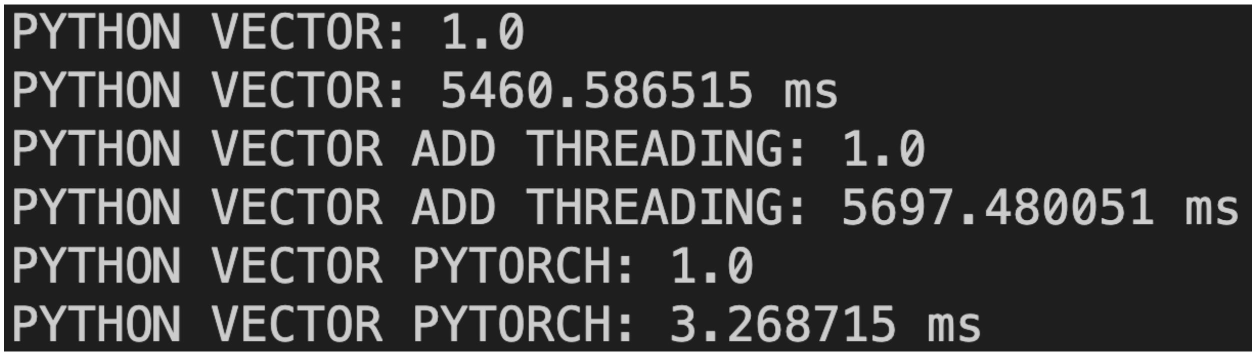
```
C MATMUL: 4.44488e+07
C MATMUL: 592.098306 ms
C MATMUL OMP: 4.44488e+07
C MATMUL OMP: 451.520895 ms
C MATMUL BLAS: 4.44488e+07
C MATMUL BLAS: 11.729042 ms
[zhang.yam@d1004 C]$ ./maincuda
C MATMUL CUDA: 4.44488e+07
C MATMUL CUDA: 2.134546 ms
```

We could tell that from the brute force to CUDA, the running time is improved step by step. Cuda is surely the fastest, which is under our expectation. We will analyze the performance of BLAS in the following passage.

Python

Vector Add

Brute Force	5460.586515 ms
MultiThreading	5697.480051 ms
Torch	3.268715 ms



```
PYTHON VECTOR: 1.0
PYTHON VECTOR: 5460.586515 ms
PYTHON VECTOR ADD THREADING: 1.0
PYTHON VECTOR ADD THREADING: 5697.480051 ms
PYTHON VECTOR PYTORCH: 1.0
PYTHON VECTOR PYTORCH: 3.268715 ms
```

The surprising result of Python vector add is that the program runs even slower with the multithreading.

Why is multithreading in Python slower?

In fact, a Python process cannot run threads in parallel but it can run them concurrently through context switching during I/O bound operations.

This limitation is actually enforced by GIL. The Python Global Interpreter Lock (GIL) prevents threads within the same process to be executed at the same time.

The GIL is necessary because Python's interpreter is **not thread-safe**. This global lock is enforced every time we attempt to access Python objects within threads. At any given time, only one thread can acquire the lock for a specific object. Therefore, CPU-bound code will have no performance gain with Python multithreading.

How about multiprocessing in Python?

In Python, multi-processing can be implemented using the multiprocessing module (or concurrent.futures.ProcessPoolExecutor) that can be used in order to spawn multiple OS processes. Therefore, multi-processing in Python side-steps the GIL and the limitations that arise from it since every process will now have its own interpreter and thus own GIL.

However, the issue here of multiprocessing is that when you use multiprocessing to open a second process, an entirely new instance of Python, with its own global state, is created. That global state is not shared, so changes made by child processes to global variables will be invisible to the parent process.

So, the multiprocessing here is more likely a distributed processing like OpenMPI we have learned in the HPC class. We may compare the multiprocessing with OpenMPI in some sense. However, we do not do deeper search and benchmark here.

PyTorch

PyTorch is an optimized tensor library for deep learning using GPUs and CPUs.

Why does PyTorch or Numpy matrix multiplication so fast?

According to the document of Pytorch, some of the matrix processing in Pytorch uses BLAS and LAPACK Operations, like addmm, which is to perform a matrix multiplication of the matrices mat1 and mat2. So the running time of torch matrix processing is way faster than brute force. We will analyze the performance of BLAS in the following passage.

Golang

What are Goroutines?

Goroutines are functions or methods that run concurrently with other functions or methods. Goroutines can be thought of as lightweight threads. The cost of creating a Goroutine is tiny when compared to a thread. Hence it's common for Go applications to have thousands of Goroutines running concurrently.

Advantages of Goroutines over threads

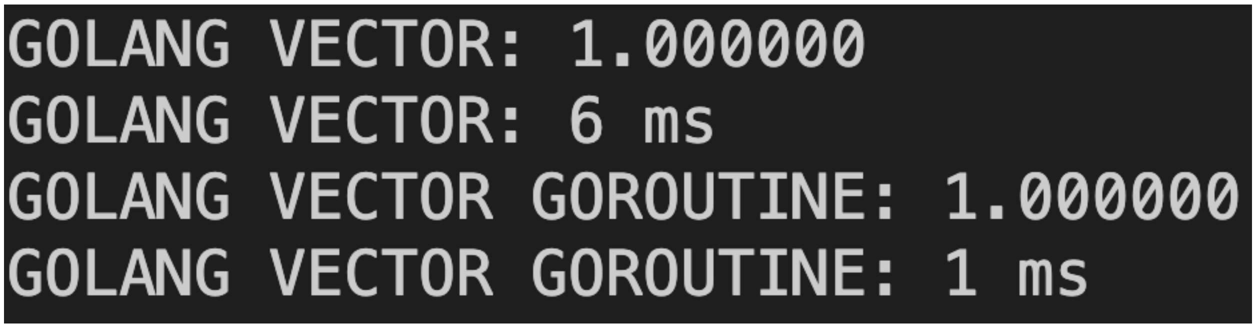
Goroutines are extremely cheap when compared to threads. They are only a few kb in stack size and the stack can grow and shrink according to the needs of the application whereas in the case of threads the stack size has to be specified and is fixed.

The Goroutines are multiplexed to a fewer number of OS threads. There might be only one thread in a program with thousands of Goroutines. If any Goroutine in that thread blocks, waiting for user input, then another OS thread is created and the remaining Goroutines are moved to the new OS thread. All these are taken care of by the runtime and we as programmers are abstracted from these intricate details and are given a clean API to work with concurrency.

Goroutines communicate using channels. Channels by design prevent race conditions from happening when accessing shared memory using Goroutines. Channels can be thought of as a pipe using which Goroutines communicate. We will discuss channels in detail in the next tutorial.

Vector Add

Brute Force	6 ms
Goroutine	1 ms

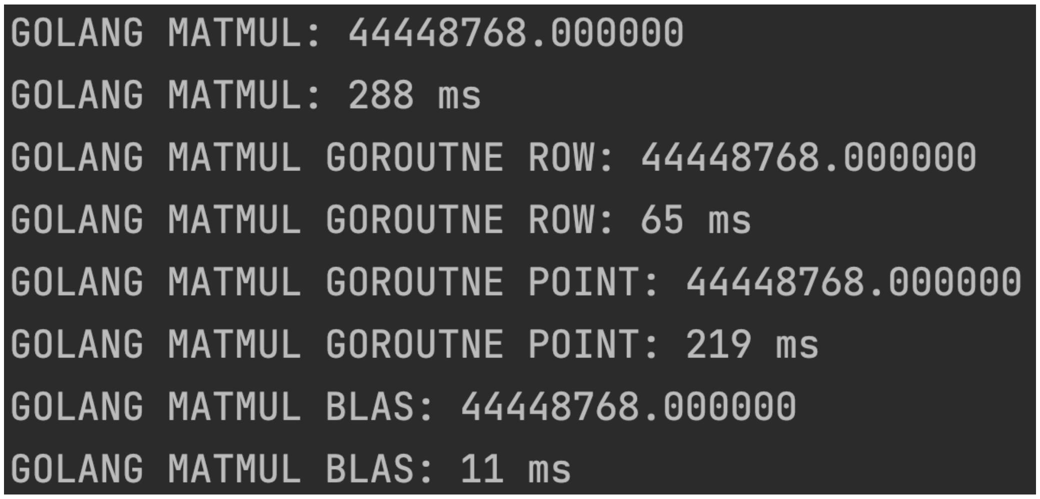


```
GOLANG VECTOR: 1.000000
GOLANG VECTOR: 6 ms
GOLANG VECTOR GOROUTINE: 1.000000
GOLANG VECTOR GOROUTINE: 1 ms
```

We could see that with goroutines, the lightweight “threads”, the running time of parallel vector addition is significantly improved.

Matrix Multiplication

Brute Force	288 ms
Goroutine Row	65 ms
Goroutine Point	219 ms
BLAS	11 ms



```
GOLANG MATMUL: 44448768.000000
GOLANG MATMUL: 288 ms
GOLANG MATMUL GOROUTNE ROW: 44448768.000000
GOLANG MATMUL GOROUTNE ROW: 65 ms
GOLANG MATMUL GOROUTNE POINT: 44448768.000000
GOLANG MATMUL GOROUTNE POINT: 219 ms
GOLANG MATMUL BLAS: 44448768.000000
GOLANG MATMUL BLAS: 11 ms
```

The running time of multithreading matrix multiplication is improved a little by the brute force one. The fastest one is still the BLAS GEMM operation. So, here we are going to analyze why BLAS operations are so fast.

Why is BLAS GEMM so fast?

BLAS is divided into three levels:

Level 1 defines a set of linear algebra functions that operate on vectors only. These functions benefit from vectorization (e.g. from using SSE).

Level 2 functions are matrix-vector operations, e.g. some matrix-vector products. These functions could be implemented in terms of Level1 functions. However, you can boost the performance of these functions if you can provide a dedicated implementation that makes use of some multiprocessor architecture with shared memory.

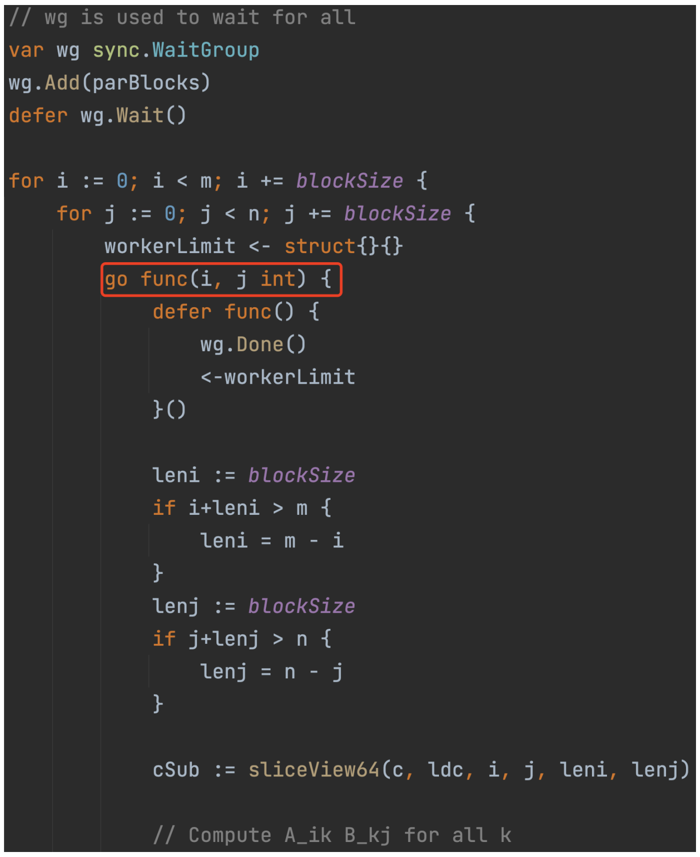
Level 3 functions are operations like the matrix-matrix product. Again you could implement them in terms of Level2 functions. But Level3 functions perform O(N^3) operations on O(N^2) data. So if your platform has a cache hierarchy then you can boost performance if you provide a dedicated implementation that is cache optimized/cache friendly. This is nicely described in the book. The main boost of Level3 functions comes from cache optimization. This boost significantly exceeds the second boost from parallelism and other hardware optimizations.

Take a glance at the BLAS.GEMM function code of Golang

```
        // dgemmParallel computes a parallel matrix multiplication by partitioning
        // a and b into sub-blocks, and updating c with the multiplication of the
sub-block
        // In all cases,
        // A = [        A_11  A_12 ...       A_1j
        //              A_21  A_22 ...       A_2j
        //                         ...
        //               A_i1  A_i2 ...      A_ij]
        //
        // and same for B. All of the submatrix sizes are blockSize×blockSize
except
        // at the edges.
        //
        // In all cases, there is one dimension for each matrix along which
        // C must be updated sequentially.
        // Cij = \sum_k Aik Bki,    (A * B)
        // Cij = \sum_k Aki Bkj,    (A^T * B)
        // Cij = \sum_k Aik Bjk,    (A * B^T)
        // Cij = \sum_k Aki Bjk,    (A^T * B^T)
        //
        // This code computes one {i, j} block sequentially along the k dimension,
        // and computes all of the {i, j} blocks concurrently. This
        // partitioning allows Cij to be updated in-place without race-conditions.
        // Instead of launching a goroutine for each possible concurrent
computation,
        // a number of worker goroutines are created and channels are used to
pass
        // available and completed cases.
```

Here are the comments on the function GemmParallel, which explains how matrix multiplication is processed and paralleled in BLAS. The GemmParallel function used blocked algorithms and goroutines parallel for each block in matrix c, which is the result of the matrix multiplication. Note that Cij is updated in-place without race-conditions here.

What is the Golang code like?


```go
// wg is used to wait for all
var wg sync.WaitGroup
wg.Add(parBlocks)
defer wg.Wait()

for i := 0; i < m; i += blockSize {
    for j := 0; j < n; j += blockSize {
        workerLimit <- struct{}{}
        go func(i, j int) {
            defer func() {
                wg.Done()
                <-workerLimit
            }()

            leni := blockSize
            if i+leni > m {
                leni = m - i
            }
            lenj := blockSize
            if j+lenj > n {
                lenj = n - j
            }

            cSub := sliceView64(c, ldc, i, j, leni, lenj)

            // Compute A_ik B_kj for all k
```

Here are the code pieces of the function. The line in the read box is where Golang starts the parallelism with goroutines.

We took a glance at how BLAS code is generated to have extreme performance even without the help of GPUs. The BLAS matrix multiplication function is the fastest one that I have tested on CPU. Lots of languages are using BLAS to accelerate the linear algebra operations like Python's Numpy, PyTorch, Golang's Gonum, and Julia's Linear Algebra package, which will be discussed in the following paragraph.

Blocked Algorithm

To see the full benefits of blocking we need to consider an algorithm operating on matrices, of which matrix multiplication is the most important example. Suppose we wish to compute the product C = AB of n\times n matrices A and B. The natural computation is, from the definition of matrix multiplication, the “point algorithm”

```
C = 0
for i=1
  for j=1
    for k=1
      c_{ij} = c_{ij} + a_{ik}b_{kj}
    end
  end
end
```

Let A = (A_{ij}), B = (B_{ij}), and C = (C_{ij}) be partitioned into blocks of size b, where r = n/b is assumed to be an integer. The blocked computation is

```
C = 0
for i=1
  for j=1
    for k=1
      C_{ij} = C_{ij} + A_{ik}B_{kj}
    end
  end
End
```

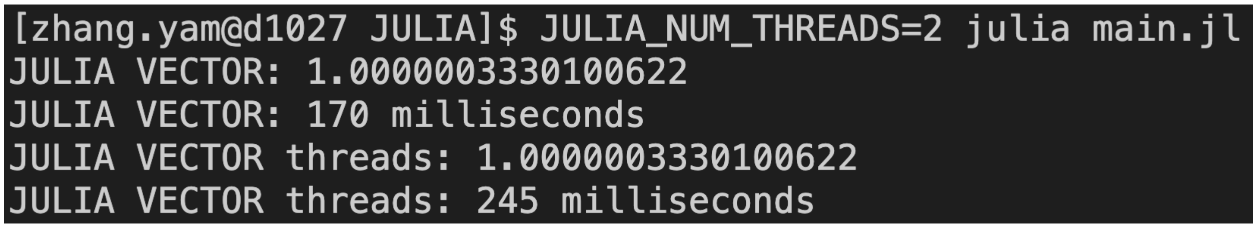
On a computer with a hierarchical memory the blocked form can be much more efficient than the point form if the blocks fit into the high speed memory, as much less data transfer is required. Indeed line 5 of the blocked algorithm performs O(b^3) flops on about O(n^2) data, whereas the point algorithm performs O(1) flops on O(1) data on line 5, or O(n) flops on O(n) data if we combine lines 4–6 into a vector inner product. It is the O(b) flops-to-data ratio that gives the blocked algorithm its advantage, because it masks the memory access costs.

The LAPACK (first released in 1992) was the first program library to systematically use blocked algorithms for a wide range of linear algebra computations.

JULIA

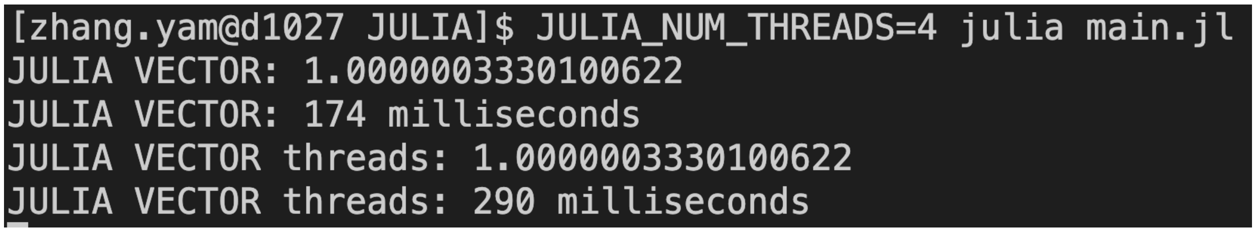
Vector Add

2 threads



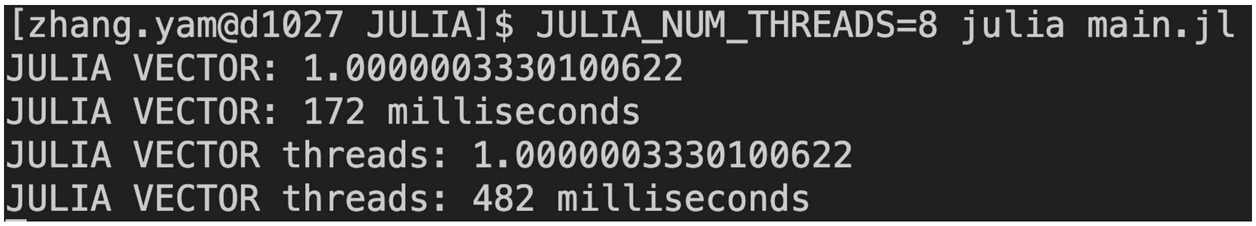
```
[zhang.yam@d1027 JULIA]$ JULIA_NUM_THREADS=2 julia main.jl
JULIA VECTOR: 1.0000003330100622
JULIA VECTOR: 170 milliseconds
JULIA VECTOR threads: 1.0000003330100622
JULIA VECTOR threads: 245 milliseconds
```

4 threads



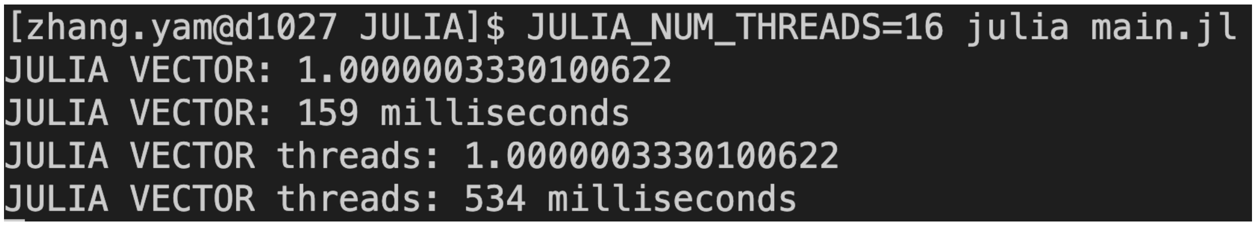
```
[zhang.yam@d1027 JULIA]$ JULIA_NUM_THREADS=4 julia main.jl
JULIA VECTOR: 1.0000003330100622
JULIA VECTOR: 174 milliseconds
JULIA VECTOR threads: 1.0000003330100622
JULIA VECTOR threads: 290 milliseconds
```

8 threads



```
[zhang.yam@d1027 JULIA]$ JULIA_NUM_THREADS=8 julia main.jl
JULIA VECTOR: 1.0000003330100622
JULIA VECTOR: 172 milliseconds
JULIA VECTOR threads: 1.0000003330100622
JULIA VECTOR threads: 482 milliseconds
```

16 threads

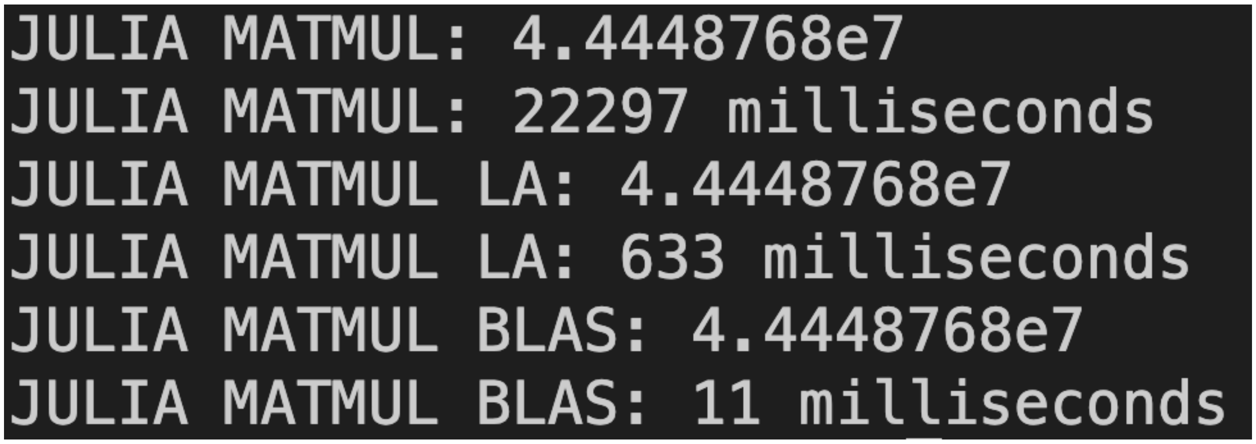


```
[zhang.yam@d1027 JULIA]$ JULIA_NUM_THREADS=16 julia main.jl
JULIA VECTOR: 1.0000003330100622
JULIA VECTOR: 159 milliseconds
JULIA VECTOR threads: 1.0000003330100622
JULIA VECTOR threads: 534 milliseconds
```

Here, we could see that the multithreading does not help the vector addition. Oppositely, with the increment of the number of the threads, the running time is slower and slower. Until now, I have not found the issue of the julia multithreading. I will supply the missing conclusion after exploring deeper.

Matrix Multiplication

Brute Force	22297 ms
LA pkg	633 ms
BLAS	11 ms



```
JULIA MATMUL: 4.4448768e7
JULIA MATMUL: 22297 milliseconds
JULIA MATMUL LA: 4.4448768e7
JULIA MATMUL LA: 633 milliseconds
JULIA MATMUL BLAS: 4.4448768e7
JULIA MATMUL BLAS: 11 milliseconds
```

We could tell that the running time of BLAS is still the fastest.

Other functions in Julia

There are still many other functions in Julia that could help us explore more on High Performance Computing, like MPI.jl, CUDA.jl and so on. I will spare more time on the exploration of Julia.

Conclusion

C and Golang bothe have extreme performance on high performance computing and normal running time. And Golang even runs faster than C thanks to the lightweight goroutines of Golang. That is maybe the reason why Golang has been so popular these years.

Python has the worst performance among all of the 4 languages. Even with CUDA and GPU, the running time of matrix multiplication is slower than CUDA in C. But thanks to Python's package Pytorch, we could use GPUs easily on the cluster and do difficult tensor processing of machine learning and deep learning.

The running time of using BLAS library is similar among C, Golang and Julia. That's mainly because the implementation of BLAS is similar among all the languages with blocked algorithms and parallelism to increase cache locality.

Julia's performance is not so impressive on basic vector addition and matrix multiplication but Julia does have a variety of packages that support HPC, like BLAS, LAPACK, MPI.jl, CUDA.jl and so on. There's more than we could imagine to explore more into Julia.

REFERENCE:

1. https://towardsdatascience.com/multithreading-multiprocessing-python-180d0975ab29
2. https://stackoverflow.com/questions/11055303/multiprocessing-global-variable-updates-not-returned-to-parent
3. https://pytorch.org/docs/stable/torch.html
4. https://nhigham.com/2021/10/28/what-is-a-blocked-algorithm/
5. https://docs.julialang.org/en/v1/stdlib/LinearAlgebra/#stdlib-blas-trans
6. https://golangbot.com/goroutines/
7. https://stackoverflow.com/questions/1303182/how-does-blas-get-such-extreme-performance