Name: Bryce Chudomelka **RedID**: 823380825
Class: COMP605 **Date**: May 7, 2021

Final Project

The current state of High Performance Computing (HPC) is done in ancient languages such as C/C++ or Fortran. These languages are extremely low-level, *i.e.* the computer language is closer to the machine code than a high-level language like Python, which makes them efficient for computation. The drawback is that these languages were designed at a time when only one core processors were in production. Today's computing environments utilizes central processing units (CPUs) with many cores and each core having two threads, thus allowing many operations to be performed concurrently. We can benchmark the potential performance increase on a matrix-matrix multiplication algorithm, see Figure 1.

Concurrency is the most likely the principal design of the computing language Go, aka "Concurrency made easy by Google." Developed in 2009 by Google, Go is a language that borrows from C/C++ but is designed with multi-core CPUs in mind. With regards to HPC, concurrency can be done in C/C++ using a variety of techniques such as `pthreads`, `OpenMP`, or `MPI`, which are not native to the language itself, and impose restrictions in development by requiring complex code. Many of these concerns have been addressed in the development of Go and are native to the language itself without syntactic sugar.

Concurrency made easy

Go's syntax for performing concurrency is as simple as typing `go `. In just three characters, `g`, `o`, and a space, a light-weight operation, known as a `goroutine`, is spawned. A `goroutine` is similar to a thread that is created by `pthreads`, but is handled by the `Go` runtime without additional code complexity. The `Go` runtime identifies the number of CPUs available and sets a variable known as `GOMAXPROCS` to optimize the number of `goroutines` that can be concurrently spawned.

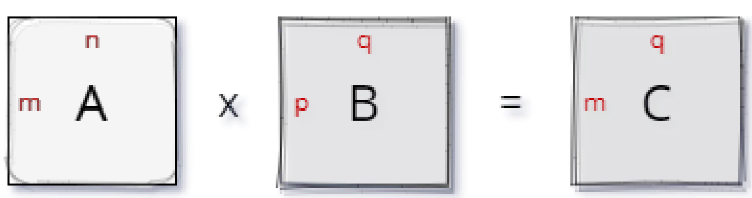

Figure 1: Matrix-matrix multiplication of two non-square matrices $A_{m\times n} \times B_{p\times q}$ yields a matrix $C_{m\times q}$; assuming $n = p$. This problem has $\mathcal{O}(mnp)$ time complexity naively, which can be reduced in a myriad of ways..

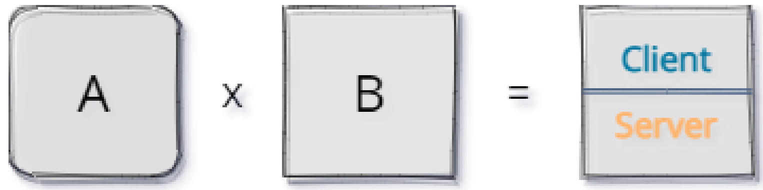

Figure 2: Half of the computation is split between the client and the server. Both machines receive an equal amount of work.

In HPC we utilize threads to perform tasks concurrently to optimize computation. We have already observed in class that the time complexity of a given program can be drastically reduced by performing concurrent operations. Also, we observed that using more threads does not always directly lead to performance increases. This is easily observed when trying to spawn more threads than 2 times the number of CPUs, roughly speaking. This introduces a possible bottleneck for computation though with current hardware restrictions; current CPUs have 8-32 cores, at most.

Distributed computing

We can access more threads if we add more CPUs, but this problem requires us to add another machine. `MPI` is a message passing interface implemented in `C` that allows the use of many machines for distributed computation. Thus, giving the user access to more CPUs, *i.e.*, threads, to improve computation and reduce time complexity. As shown in a previous homework, I found an MPI-like package in `Go` and implemented it successfully to test the network bandwidth, *e.g.*, determining the latency of sending bytes across ports on the same machine. While that exercise was useful, the more challenging problem is communicating across machines in the network.

We can simulate this behavior using a TCP connection between two machines to pass information; see Figure 2. Bytes of information can be sent across the network for a relatively low cost, *e.g.*, latency as seen in a recent homework, but memory cannot be shared across a TCP connection. This adds an additional operation of combining the distributed computation into its final form. Some might call it, "the cost of doing business," because adding *unneeded* operations to the task should increase run time. This will be addressed in the analysis section.

Methodology

In this project, a proof-of-concept has been created to distribute computational resources across two machines. I will connect two nodes on the server `tuckoo.sdsu.edu` via a transmission control protocol (TCP) connection, where the instructions will be sent in a client/server paradigm to distribute computation between two machines. The client will be responsible

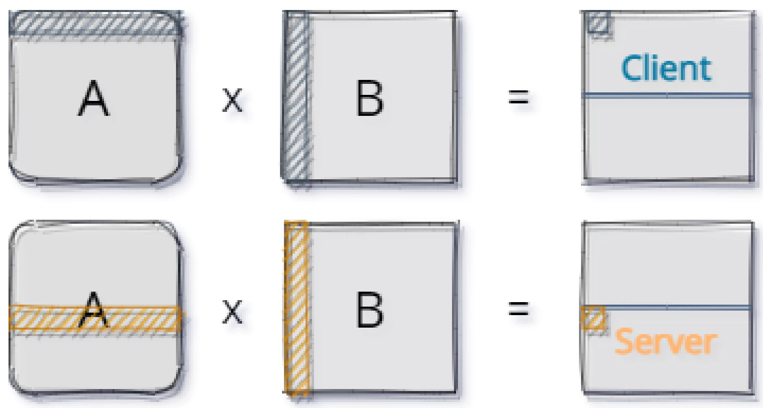

Figure 3: This visualization shows how either the client or the server perform their piece of the computation. This allows us to easily combine the resultant matrix after computation.

for initializing and populating the matrices, as, providing the server with instructions, combining the results, as well as doing half the work in matrix-matrix multiplication. The server will listen and wait until it receives a message from the client then perform the matrix-matrix multiplication as indicated and return the results via TCP.

Matrix-matrix computation is costly when performed in serial, and we will utilize the `goroutine` version created in a previous homework assignment to improve performance when compared to the traditional row-wise multiplication algorithm. Only a slight modification needs to be made to indicate starting row and ending row for computation. Each machine will perform their portion of matrix-matrix computation using `goroutines`, thus effectively turning a big problem for one machine into a smaller problem for two machines; see Figure 3. We note that for small matrices this is sub-optimal due to network latency and the added operation of combining two matrices.

First, the client will initialize three slices of floats according to the input parameters. Next, the client will determine which rows of the matrix C that the both the client and server has to compute, then pass the required information via a `struct` to the server. At this point in the client/server paradigm, each machine begins to perform their individual tasks according to the `goRowMultMat` function, which performs row-wise matrix multiplication on a flattened matrices using `goroutines`. After the server finishes its work, the server then dials up the client via TCP and sends over its half of the resultant matrix along with the number of `goroutines` used. Finally, the client combines the two results and exits the program. Even though the algorithm is designed to multiply non-square matrices, the benchmarks performed in this experiment will only consider square matrices, *i.e.* $A_{n\times n} \times B_{n\times n} = C_{n\times n}$, to simplify analysis.

n	Row	Goroutine	Distributed
250	0.049	0.007	0.022
500	0.395	0.034	0.072
750	1.354	0.105	0.164
1000	2.920	0.217	0.300
1250	6.118	0.435	0.501
1500	10.103	0.723	0.786
1750	34.168	2.177	1.439
2000	41.578	2.874	1.902
2250	77.667	4.555	3.137
2500	93.060	5.463	3.694
5000	820.133	43.913	25.819
7500	3573.429	175.860	95.659
10000	6558.381	353.265	203.442
15000	*	1318.689	702.739
20000	*	9400.968	3656.001

Table 1: The results of each trial run are compiled here with each timing being measured in seconds. Here Row refers to row-wise, Goroutines to single-machine, and Distributed to matrix-matrix multiplication; respectively.

Results

We perform matrix-matrix multiplication on `tuckoo.sdsu.edu`, a network of CPUs with various properties. We will multiply matrices where n will take on various values. The results of the experiments are tabulated in Table 1. The computation time will be measured, in seconds, for each run as well.

Analysis

The time complexity associated with row-wise matrix-matrix multiplication is observed to be significantly worse than the Goroutine algorithm. It is important to distinguish between the Goroutine algorithm and the distributed method. It was measured that the Goroutine method used 16 threads and that the distributed method between `node10` and `node12` was 32. There is a cost associated with sending the data across the network and combining the received results.

As n increases, the cost associated with distributing the computation diminishes to the point where distributed computation out performs goroutines. This is reflective of the number of threads/goroutines and how there is a performance bottleneck. A simple threaded algorithm on one machine is sufficient to perform matrix-matrix multiplication until the size of the matrices surpasses a particular amount, which we observe to be roughly $n = 2000$. Distributed computation is more performant than single machine threaded, or serial, com-

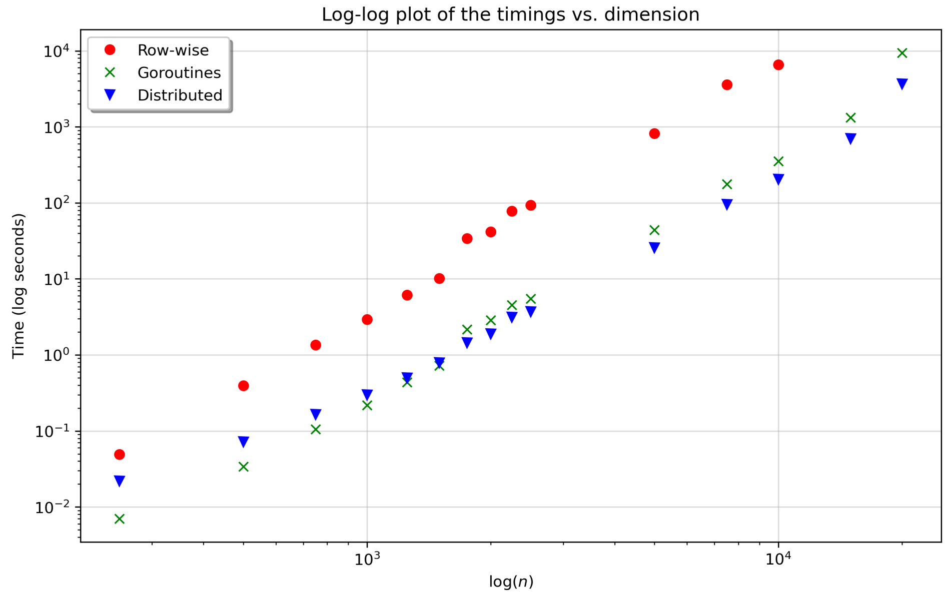

Figure 4: This visualization shows how either the client or the server perform their piece of the computation. This allows us to easily combine the resultant matrix after computation.

putation whenever the dimension of the matrices are too large.

Drawbacks

Sometimes we view the world through rose-colored glasses and it can simply be put as, "Not all problems are alike!" This is correct because depending on the parameters of your problem, *i.e.*, dimension size, one should be careful about the algorithm/methodology used to solve their problem. If you are trying to solve a 3×3 matrix, then a trivial row-wise matrix-matrix multiplication would "make-sense". The problem is that when we try to scale up our input dimensions that our algorithms become slow, less performant, and inefficient. Thus, we must introduce distributed computing, in order to accommodate for increased input dimensions once our matrix size is sufficiently large. Scientific computing at scale *requires* distributed computing. This experiment loosely demonstrates that as a proof-of-concept. A final note that more care should be taken when computing timings to consider the average with some tolerance for a true benchmark.

Future Work

This work can now easily be extended to include more machines in a network. If the number of nodes being used is odd then divvy up the rows in the output matrix C accordingly, but if the number of nodes is even, and greater than 2, then split on the columns, as seen in Figure 5. This pattern can alternate rows and columns as needed. Care should be taken on

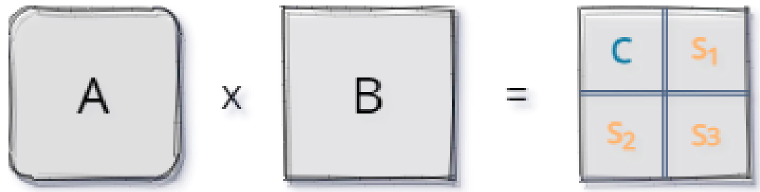

Figure 5: This visualization shows how either the client or server perform their piece of the computation; an example using 4 nodes labeled C, S_1, S_2, and S_3 is depicted here where C is the client, and S_1, S_2, S_3 are the server nodes. This allows us to easily combine the resultant matrix after computation.

behalf of the client side as to receive all results before combining, but that is a problem for another day.

Code

Row

```go
package main

import (
	"fmt"
	"math/rand"
	"os"
	"runtime"
	"strconv"
	"sync"
	"time"
)

func main() {
	L := len(os.Args)
	m, n, p, q, err := mapVars(L, os.Args)
	if err != 0 {
		return
	}

	fmt.Println("The product array has dimensions...")
	fmt.Printf("\tC is %dx%d\n", m, q)

	fmt.Println("\nPopulating matrix A.")
	A, _ := createMat(m, n)
	if m < 5 && n < 5 {
		fmt.Println("Matrix A.")
		printMat(m, A)
	}

	fmt.Println("Populating matrix B.")
	B, _ := createMat(p, q)
	if p <= 5 && q <= 5 {
		fmt.Println("Matrix B.")
		printMat(p, B)
	}

	fmt.Println("\nPerforming row-wise matrix-matrix multiplication AB.")
	C, _ := initMat(m, q)
	startRow := time.Now()
	rowMultMat(m, n, q, A, B, C)
	dtRow := time.Since(startRow)
	fmt.Printf("Time elapsed: %v\n", dtRow)

	fmt.Printf("\nPerforming row-wise matrix-matrix multiplication AB using %d
	 goroutines.\n", runtime.GOMAXPROCS(0))
	E, _ := initMat(m, q)

```

```go
    startGo := time.Now()
    goRowMultMat(m, n, q, A, B, E)
    dtGo := time.Since(startGo)

    fmt.Printf("Time elapsed: %v\n", dtGo)
}

func mapVars(l int, args []string) (m int, n int, p int, q int, err int) {
    if l == 2 {
        m, _ := strconv.Atoi(args[1])
        n, _ := strconv.Atoi(args[1])
        p, _ := strconv.Atoi(args[1])
        q, _ := strconv.Atoi(args[1])
        fmt.Printf("Creating two arrays, A, B, with square dimensions.\n")
        fmt.Printf("\tA is %dx%d\n\tB is %dx%d\n", m, n, p, q)
        return m, n, p, q, 0
    } else if l == 5 || n != p {
        m, _ := strconv.Atoi(args[1])
        n, _ := strconv.Atoi(args[2])
        p, _ := strconv.Atoi(args[3])
        q, _ := strconv.Atoi(args[4])
        fmt.Println("Creating two arrays, A, B, with dimensions.")
        fmt.Printf("\tA is %dx%d\n\tB is %dx%d\n", m, n, p, q)
        return m, n, p, q, 0
    } else {
        fmt.Println("
````````````````````````````````````````````````````````````````````````````````\
n\n")
        fmt.Println("\tALERT: Incorrect number of input arguments.\n\t
Exiting.\n\n")
        fmt.Println("
````````````````````````````````````````````````````````````````````````````````\
n\n")
        fmt.Println("\tUsage:\n")
        fmt.Println("\t$ args rowsA columnsA rowsB columnsB\n")
        fmt.Println("\trowsA: The number of rows in Matrix A.\n")
        fmt.Println("\tcolumnsA: The number of columns in Matrix A.\n")
        fmt.Println("\trowsB: The number of rows in Matrix B.\n")
        fmt.Println("\tcolumnsB: The number of columns in Matrix B.\n")
        return 0, 0, 0, 0, 1
    }
}

func initMat(m int, n int) (M [][]float64, rows []float64) {
    M = make([][]float64, m)
    rows = make([]float64, n*m)
    for i := 0; i < m; i++ {
        M[i] = rows[i*n : (i+1)*n]
    }
    return M, rows
}
```

```go

func createMat(m int, n int) (M [][]float64, rows []float64) {
    M = make([][]float64, m)
    rows = make([]float64, n*m)
    for i := 0; i < m; i++ {
        for j := 0; j < n; j++ {
            rows[i*n+j] = rand.Float64()
        }
        M[i] = rows[i*n : (i+1)*n]
    }
    return M, rows
}

func printMat(row int, M [][]float64) {
    for i := 0; i < row; i++ {
        fmt.Printf("%v\n", M[i])
    }
}

func rowMultMat(m int, n int, q int, A [][]float64, B [][]float64, C [][]
    float64) {
    for i := 0; i < m; i++ {
        for j := 0; j < q; j++ {
            C[i][j] = 0
            for k := 0; k < n; k++ {
                C[i][j] = C[i][j] + A[i][k]*(B[k][j])
            }
        }
    }
}

func colMultMat(m int, n int, q int, A [][]float64, B [][]float64, C [][]
    float64) {
    for j := 0; j < q; j++ {
        for i := 0; i < m; i++ {
            C[i][j] = 0
        }
        for k := 0; k < n; k++ {
            for i := 0; i < m; i++ {
                C[i][j] += A[i][k] * (B[k][j])
            }
        }
    }
}

func goRowMultMat(m int, n int, q int, A [][]float64, B [][]float64, C [][]
    float64) {
    var wg sync.WaitGroup
    for i := 0; i < m; i++ {
        wg.Add(1)
        go func(i int) {
```

```
			for j := 0; j < q; j++ {
				C[i][j] = 0
				for k := 0; k < n; k++ {
					C[i][j] = C[i][j] + A[i][k]*(B[k][j])
				}
			}
			wg.Done()
		}(i)
	}
	wg.Wait()
}
```

### Client

```
package main

import (
	"encoding/gob"
	"fmt"
	"log"
	"math/rand"
	"net"
	"os"
	"runtime"
	"strconv"
	"sync"
	"time"
)

type Data struct {
	M, N, Q            int
	RowStart, RowEnd int
	A, B, C            []float64
}

type Result struct {
	Goroutines int
	C          []float64
}

func main() {
	// Get vars
	m, n, p, q, ipv4, errInt := mapVars(os.Args)
	if errInt != 0 {
		return
	}
	fmt.Println("Connecting to: ", ipv4)
	fmt.Println("Starting client....\nNumber of goroutines: ", runtime.GOMAXPROCS(0))
	// Establish connection
```

```go
    conn, err := net.Dial("tcp", ipv4+":8080")
    if err != nil {
        log.Fatal("Dial:", err)
    }

    // Init params
    fmt.Println("\nPopulating matrix A.")
    A := createMat(m, n)

    fmt.Println("Populating matrix B.")
    B := createMat(p, q)

    C := initMat(m, q)

    // Compute message info
    message := &Data{}
    message.M = m
    message.N = n
    message.Q = q
    rowStart := m / 2
    message.RowStart = rowStart
    message.RowEnd = m
    message.A = A
    message.B = B
    message.C = C
    // Encode data struct and send instructions
    encoder := gob.NewEncoder(conn)
    encoder.Encode(message)
    conn.Close()
    fmt.Println("\nMessage sent. Starting matrix-matrix multiplication.\n")

    startRow := time.Now()
    // Do some stuff in a goroutine
    var wg sync.WaitGroup
    wg.Add(1)
    go goRowMultMat(m, n, q, 0, rowStart, A, B, C, &wg)

    // Wait for other process to signal done
    fmt.Println("Waiting for results...")
    ln, err := net.Listen("tcp", ":8081")
    if err != nil {
        log.Fatal("Could not listen:", err)
    }
    conn, err = ln.Accept() // this blocks until connection or error
    if err != nil {
        log.Fatal("Could not accept:", err)
    }
    dec := gob.NewDecoder(conn)
    result := &Result{}
    dec.Decode(result)
    conn.Close()
```

```go
    wg.Wait()

    _ = combineMatrix(m, q, 0, rowStart, C, result.C)
    dtRow := time.Since(startRow)
    if m <= 5 && q <= 5 {
        fmt.Println("Matrix C.")
        printMat(m, q, C)
    }

    fmt.Printf("Time elapsed: %v \n", dtRow)
    fmt.Println("Total number of goroutines used:", runtime.GOMAXPROCS(0)+result.Goroutines)
}

func mapVars(args []string) (m int, n int, p int, q int, ipv4 string, err int) {
    l := len(args)
    if l == 3 {
        m, _ := strconv.Atoi(args[1])
        n, _ := strconv.Atoi(args[1])
        p, _ := strconv.Atoi(args[1])
        q, _ := strconv.Atoi(args[1])
        ipv4 := args[2]
        fmt.Printf("Creating two arrays, A, B, with square dimensions.\n")
        fmt.Printf("\tA is %dx%d\n\tB is %dx%d\n", m, n, p, q)
        return m, n, p, q, ipv4, 0
    } else if l == 6 || args[2] != args[3] {
        m, _ := strconv.Atoi(args[1])
        n, _ := strconv.Atoi(args[2])
        p, _ := strconv.Atoi(args[3])
        q, _ := strconv.Atoi(args[4])
        ipv4 := args[5]
        fmt.Println("Creating two arrays, A, B, with dimensions.")
        fmt.Printf("\tA is %dx%d\n\tB is %dx%d\n", m, n, p, q)
        return m, n, p, q, ipv4, 0
    } else {
        fmt.Println("````````````````````````````````````````````````````````````````````````````````````````\n\n")
        fmt.Println("\tALERT:Incorrect number of input arguments.\n\tExiting.\n\n")
        fmt.Println("````````````````````````````````````````````````````````````````````````````````````````\n\n")
        fmt.Println("\tUsage:\n")
        fmt.Println("\t$ args rowsA columnsA rowsB columnsB IPv4\n")
        fmt.Println("\trowsA: The number of rows in Matrix A.\n")
        fmt.Println("\tcolumnsA: The number of columns in Matrix A.\n")
        fmt.Println("\trowsB: The number of rows in Matrix B.\n")
        fmt.Println("\tcolumnsB: The number of columns in Matrix B.\n")
        fmt.Println("\tIPv4: The IPv4 address of the current machine.\n")
```

```
        return 0, 0, 0, 0, "", 1
    }
}

func initMat(m int, n int) []float64 {
    M := make([]float64, m*n)
    return M
}

func createMat(m int, n int) []float64 {
    M := make([]float64, m*n)

    for i := 0; i < m; i++ {
        for j := 0; j < n; j++ {
            M[i*n+j] = rand.Float64()
        }
    }
    return M
}

func printMat(m, n int, M []float64) {
    for i := 0; i < m; i++ {
        for j := 0; j < n; j++ {
            fmt.Printf("%v ", M[i*n+j])
        }
        fmt.Print("\n")
    }
}

func goRowMultMat(m, n, q, start, end int, A []float64, B []float64, C []
    float64, wg *sync.WaitGroup) {
    defer wg.Done()
    var wg_internal sync.WaitGroup
    for i := start; i < end; i++ {
        wg_internal.Add(1)
        go func(i int) {
            for j := 0; j < q; j++ {
                C[i*q+j] = 0
                for k := 0; k < n; k++ {
                    C[i*q+j] = C[i*q+j] + A[i*n+k]*B[k*q+j]
                }
            }
            wg_internal.Done()
        }(i)
    }
    wg_internal.Wait()
}

func combineMatrix(m, q, start, stop int, A, B []float64) []float64 {
    // performs matrix addition: A + B => move B elements into A
    for i := start; i < stop; i++ {
```

```
        for j := 0; j < q; j++ {
            B[i*q+j] = A[i*q+j]
        }
    }
    A = B
    return A
}
```

### Server

```
package main

import (
    "encoding/gob"
    "fmt"
    "log"
    "net"
    "os"
    "runtime"
    "sync"
)

type Data struct {
    M, N, Q            int
    RowStart, RowEnd int
    A, B, C            []float64
}

type Result struct {
    Goroutines int
    C          []float64
}

func main() {
    ipv4 := os.Args[1]
    fmt.Println("Waiting for connection from:", ipv4)
    // fmt.Println("Listening on port 8080...\nNumber of goroutines: ",
    runtime.GOMAXPROCS(0))
    ln, err := net.Listen("tcp", ":8080")
    if err != nil {
        log.Fatal("Could not listen:", err)
    }
    conn, err := ln.Accept() // this blocks until connection or error
    if err != nil {
        log.Fatal("Could not accept:", err)
    }
    dec := gob.NewDecoder(conn)
    var data Data
    dec.Decode(&data)
    conn.Close()
```

```go

    fmt.Println("Received messaage")
    // fmt.Printf("Received : %+v \n", data)
    A := data.A
    B := data.B
    C := data.C
    start := data.RowStart
    end := data.RowEnd
    m := data.M
    n := data.N
    q := data.Q
    goRowMultMat(m, n, q, start, end, A, B, C)

    // fmt.Println("Matrix C.")
    // printMat(m, q, C)

    fmt.Println("Sending result....")
    // Establish connection
    conn, err = net.Dial("tcp", ipv4+":8081")
    if err != nil {
        log.Fatal("Dial:", err)
    }
    message := &Result{}
    message.C = C
    message.Goroutines = runtime.GOMAXPROCS(0)
    encoder := gob.NewEncoder(conn)
    encoder.Encode(message)
    conn.Close()

}

func goRowMultMat(m, n, q, start, end int, A []float64, B []float64, C []
    float64) {
    var wg sync.WaitGroup
    for i := start; i < end; i++ {
        wg.Add(1)
        go func(i int) {
            for j := 0; j < q; j++ {
                C[i*q+j] = 0
                for k := 0; k < n; k++ {
                    C[i*q+j] = C[i*q+j] + A[i*n+k]*B[k*q+j]
                }
            }
            wg.Done()
        }(i)
    }
    wg.Wait()
}

func printMat(m, n int, M []float64) {
    for i := 0; i < m; i++ {
```

```
        for j := 0; j < n; j++ {
            fmt.Printf("%v ", M[i*n+j])
        }
        fmt.Print("\n")
    }
}
```

## Results

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy import optimize as opt

text = "\
250&     0.049 &    0.007 &   0.022&\
500&     0.395 &    0.034 &   0.072&\
750&     1.354 &    0.105 &   0.164&\
1000&     2.920 &    0.217 &   0.300&\
1250&     6.118 &    0.435 &   0.501&\
1500&    10.103 &    0.723 &    0.786&\
1750&    34.168 &    2.177 &    1.439&\
2000&    41.578 &    2.874 &    1.902&\
2250&    77.667 &    4.555 &    3.137&\
2500&    93.060 &    5.463 &    3.694&\
5000&   820.133 &   43.913 &   25.819&\
7500&  3573.429 & 175.860 &   95.659&\
10000&  6558.381 & 353.265 & 203.442&\
15000&       *     &1318.689 & 702.739&\
20000&       *     &9400.968 & 3656.001\
"

parsed = text.split('&')
print(parsed)
results = []
for i, t in enumerate(parsed):
    if '*' in t:
        results.append(0)
    else:
        results.append(float(t))

print(results)
n, row, go, distributed = [], [], [], []
for i, _ in enumerate(results):
    if  i % 4 == 0:
        n.append(_)
    elif i % 4 == 1:
        row.append(_)
    elif i % 4 == 2:
        go.append(_)
```

```python
    elif i % 4 == 3:
        distributed.append(_)

def objective(x, a, b):
    return a*x + b

def exponential(x, a, b):
    return a*x**b

def fit(x_values, y_values):
    popt, _ = opt.curve_fit(exponential, x_values, y_values)
    a, b = popt
    x_new = np.linspace(min(x_values), max(x_values), 10)
    y_new = exponential(x_new, a, b)
    return x_new, y_new


plt.figure(figsize=(10,6))
plt.loglog(n[:-2], row[:-2], 'ro', label='Row-wise')
# n_new, row_hat = fit(n, row)
# plt.plot(n_new, row_hat, 'r--', label='Row-wise Fit')
plt.loglog(n, go, 'gx', label='Goroutines')
# _, go_hat = fit(n, go)
# plt.plot(n_new, go_hat, 'g--', label='Goroutines Fit')
plt.loglog(n, distributed, 'bv', label='Distributed')
# _, distributed_hat = fit(n, distributed)
# plt.plot(n_new, distributed_hat, 'b--', label='Distributed Fit')

plt.legend(shadow=True)
plt.grid(alpha=0.4618)
plt.title("Log-log plot of the timings vs. dimension")
plt.xlabel('$\\log(n)$')
plt.ylabel('Time ($\\log$ seconds)')
plt.savefig('./imgs/results.png', bbox_inches="tight", dpi=300)
plt.show()
```