CSE 6220, Spring 2015

Programming Assignment 1 - Report

Ankit Srivastava & Cansu Tetik

February 26, 2015

1 Problem Description

Given a set of n l-bit integers ($l \leq 64$) : $\{S_1, S_2 \dots S_n\}$ and a number d ($d \leq l$), we want to find all the integers which are at a Hamming distance at most d from all of the given n integers. Hamming distance between two integers is the number of bits that are different in the bit representation of the two numbers.

2 Algorithm Design Description

In this section, we give a brief description of the algorithms that we implemented for solving the given problem.

2.1 Sequential Algorithm

For solving the problem sequentially, we take one of the input numbers (S_1 in our algorithm) as the reference and start inverting its bits one by one, going from LSB to MSB, up to a maximum of d bits. In each iteration, we flip bits in the reference recursively at every level until we reach the maximum number of toggles allowed ($flipCount$), this number is bounded by d.

The candidate numbers are then checked one by one, using Hamming distance. If a candidate number is at most d Hamming distance from all the numbers in input S, then it is appended to the *result* vector.
Pseudocode for the steps described above is shown in Algorithm 1.

Algorithm 1 Sequential Find Motifs

```
1: result ← φ
2: procedure HAMMING(input1, input2)
3:     hammingDistance ← Hamming distance between input1 and input2
4:     return hammingDistance
5: procedure CHECKSOLUTION(n, d, S, number, result)
6:     while S[i] ∈ S and hamming(S[i], number) ≤ d do
7:         ++i
8:     if i == n then
9:         append number to result
10: procedure EXPLORESOLUTIONS(n, l, d, S, flipCount, number, result,
    numTraversed, numFlipped)
11:    if flipCount > 0 then
12:        for flipPos from numTraversed to l − (flipCount − numFlipped)
    do
13:            flipped ← flip the bit on flipPos in number
14:            if (numFlipped + 1) < flipCount then
15:                EXPLORESOLUTIONS(n, l, d, S, flipCount, flipped, result, flipPos+
    1, numFlipped + 1)
16:            else
17:                CHECKSOLUTION(n, d, S, flipped, result)
18:    else
19:        CHECKSOLUTION(n, d, S, number, AllSolutions)
20: procedure FINDMOTIFS(n, l, d, S)
21:    flipCount ← 0
22:    for flipCount ≤ d do
23:        EXPLORESOLUTIONS(n, l, d, S, flipCount, S_0, result)
```

2.2 Parallel Algorithm

For finding motifs in parallel, we used master-worker paradigm wherein the master solves the problem until depth k and then sends sub-problems to workers. Workers solve the sub-problems completely, i.e., until depth l, and send the solutions back to the master.

2.2.1 Master

The master process solves the problem until depth k, i.e., it flips bits of S_1 until k depth is reached and generates sub-problems, which it then sends to the worker processes for obtaining solutions for the problem. The algorithm for the Master portion of the Parallel Find Motifs is given in Algorithm 2 below.

Algorithm 2 Parallel Find Motifs - Master

```
1: procedure ExploreMaster(l, flipCount, number, workerId, busyCount, result, resultSize, numTraversed, numFlipped)
2:     if flipCount > 0 then
3:         for flipPos from numTraversed to l − (flipCount − numFlipped) do
4:             flipped ← flip the bit on flipPos in number
5:             if (numFlipped + 1) < flipCount then
6:                 ExploreMaster(l, flipCount, flipped, workerId, busyCount, result, resultSize, flipPos+1, numFlipped + 1)
7:             else
8:                 SendPartial(flipped, result, busyCount, workerId)
9:     else
10:        SendPartial(number, result, busyCount, workerId)
11: procedure FindMotifsMaster(n, l, d, S, tillDepth)
12:    result ← φ
13:    vector resultSize(2 ∗ p − 1, 0)
14:    workerId ← 1
15:    busyCount ← 0
16:    maxDepth ← min(tillDepth, d)
17:    for flipCount from 0 to maxDepth do
18:        ExploreMaster(tillDepth, flipCount, S_0, workerId, busyCount, result, resultSize)
19:    if busyCount > 0 then
20:        ReceiveResults from worker with workerId
21: procedure MasterMain(n, l, d, S)
22:    send (n, l, d, S, k) to all workers
23:    FindMotifsMaster(n, l, d, S, k)
24:    send EXIT_TAG to all workers
25:    return result
```

2.2.2 Worker

Workers receive sub-problems from the master and solve them by continuing to traverse the number from kth position and flip the bits similar to the master process. Workers then send the solution (if any) back to the master process.

Algorithm 3 Parallel Find Motifs - Worker

```
1: procedure FindMotifsWorker(n, l, d, S, k, startVal)
2:     result ← φ
3:     hammingDistance ← Hamming(startVal, S_0)
4:     for flipCount from hammingDistance to d do
5:         ExploreSolutions(n, l, d, S, flipCount, result, k, hammingDistance)
6:     return result
7: procedure WorkerMain( )
8:     vector resultSize(2, MAX)
9:     vector start(2)
10:    vector result(2)
11:    receive n, l, d, S, k from master
12:    iter ← 0
13:    receive the first sub-problem with start[iter]
14:    while 1 do
15:        if resultSize[iter] < MAX then
16:            waitall on requestSize
17:        if EXIT_TAG is received then
18:            iter ← (iter + 1)%2
19:            if resultSize[iter] < MAX then
20:                waitall on requestSize
21:            return
22:        iter ← (iter + 1)%2
23:        receive the next sub-problem with start[iter]
24:        iter ← (iter + 1)%2
25:        result[iter] ← FindMotifsWorker(n, l, d, S, k, start[iter])
26:        resultSize[iter] ← result[iter].size()
27:        Send resultSize[iter]
28:        if resultSize[iter] > 0 then
29:            Send result[iter][0] with RESULT_TAG
30:        iter ← (iter + 1)%2
```

2.2.3 Communication and Computation

As discussed above, we used master-worker paradigm for finding motifs. For achieving overlap of communication and computation, we used immediate version of MPI commands for sending and receiving data (*MPI_Isend* and *MPI_Irecv*). In the following paragraphs, we describe how we used these, in the master and workers, for achieving overlap of communication and computation.

Master distributes sub-problems using cyclic decomposition. Since all the workers can handle two sub-problems, solve one while they send results for another, master initially starts off by sending two cycles of data and then goes cycle by cycle. Computation and communication in the master is described in

the steps below:

1. The master starts off by solving a problem until depth k and then sending it off to worker with rank 1. The master then starts an immediate receive request for receiving the data from worker 1.

2. The master repeats the above step for all the workers, twice over (since each worker can hold two sub-problems). At the end of the two cycles, the master has $2 \times (p-1)$ active receive requests.

3. Since all the workers should be operating at their maximum capacity at the end of the two cycles, the master now waits to receive the result size from worker 1.

4. Once the master has received result size and result (if any) from worker 1, it sends another sub-problem to worker 1 and starts another receive request (similar to what was done in step 1).

5. The above two steps are then repeated for each worker rank in a cyclic order, until all the sub-problems have been dispatched.

6. The master keeps waiting for all the receive requests to complete and then receives all the corresponding results.

7. At the end, the master sends a message with a special tag to all the workers, signaling them to exit.

Worker works on two sub-problems for achieving overlap of computation and communication. Therefore, it maintains two separate result buffers, each of which is used in an alternate iteration. Steps in the worker are enumerated below:

1. Worker starts off by waiting for the first sub-problem. Once it has received the first sub-problem, it starts an immediate receive for the second sub-problem and then starts solving the first sub-problem.

2. Once the first sub-problem is solved, the worker starts an immediate send for the result size and results (if any) to the master.

3. The worker then starts waiting on immediate receive in the first step. Again, after it has received the sub-problem, it starts another immediate receive before starting to solve the sub-problem.

4. The worker starts another immediate receive for the result size and results (if any) obtained in the above step.

5. Two immediate sends are now in progress. Worker frees the buffer that has been occupied the longest by waiting on the send requests started in step 2. It then receives another sub-problem and starts solving it using the recently freed buffer.

6. The above step is now repeated until master signals the worker to exit. The worker then waits for all sends to complete and then exits.

3 Plots and Observations

First of all, as suggested in the readme file supplied with the code framework, we had to choose a suitable combination of all the parameters that took 5-10 minutes to solve serially. Therefore, we chose $(n = 10, l = 42, d = 13, k = 6)$ configuration, which took 597.91 seconds to solve serially. We then varied each parameter in the neighborhood of the corresponding benchmark value, while keeping the others constant. In order to verify the plots so obtained, we varied some other parameters as well and obtained three such plots for every parameter. We discuss the obtained plots in the following sections.

We used *jinx4* & *jinx7* nodes, both of which have 4×6 cores, for running our experiments.

3.1 Number of Processors (p)

We varied number of processors by setting it to different values in the set $\{1, 2, 4, 8, 12, 16, 20, 24\}$ and obtained the runtimes for solving the problem for three different parameter combinations: $(n = 10, l = 42, d = 12, k = 6)$, $(n = 10, l = 42, d = 13, k = 6)$, and $(n = 10, l = 43, d = 12, k = 6)$. The corresponding runtime and speedup plots are shown in Figure 1 & Figure 2 respectively.

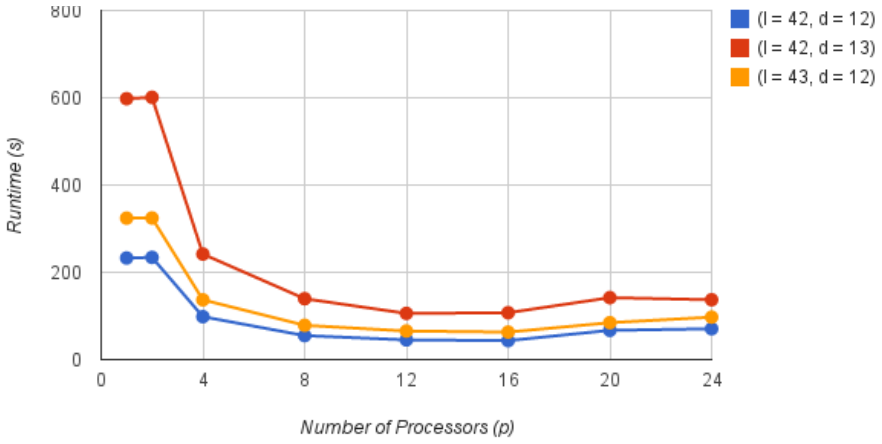

Figure 1: Runtime v/s Number of Processors $(n = 10, k = 6)$

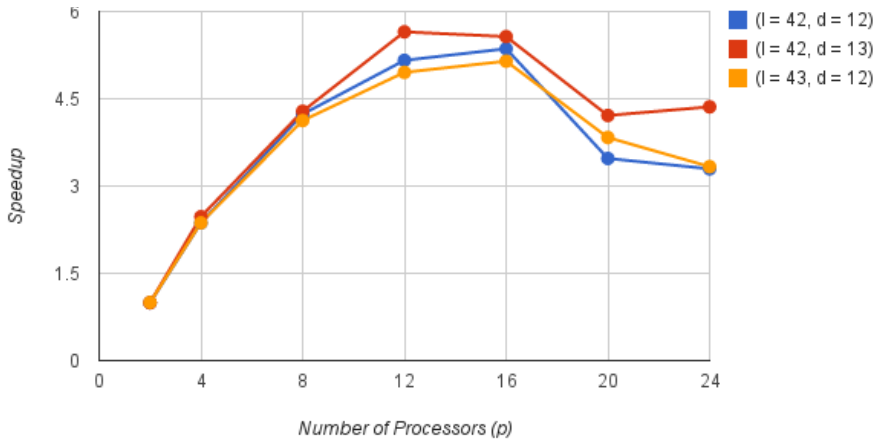

Figure 2: Speedup v/s Number of Processors ($n = 10, k = 6$)

From the runtime plot, it can be observed that the runtime decreases up to $12 - 16$ processors and then becomes more or less constant. From the speedup plot, it can be seen that the speedup is almost linear up to $12 - 16$ processors and then actually drops a bit. We concluded that $12 - 16$ is the optimal number of processors required for solving the problem. Theoretically, we should have seen constant speedup after maxima but practical issues of working with larger number of processor are probably playing a part in decreasing the speedup after the maximum. We also observe that the optimal number of processors increases with increase in d & n. This is expected since the problem size increases on increasing d, or n, or both.

Since we obtained maximum speedup using 12 processors for our benchmark problem ($n = 10, l = 42, d = 13, k = 6$), we used 12 processors for running all the subsequent experiments.

3.2 Number of Inputs (n)

We varied number of inputs in the range $[2, 15]$ and obtained the runtimes for solving the problem for three parameter combinations: ($l = 42, d = 11, k = 6, p = 12$), ($l = 42, d = 12, k = 6, p = 12$), and ($l = 42, d = 13, k = 6, p = 12$). The runtime plot is shown in Figure 3.

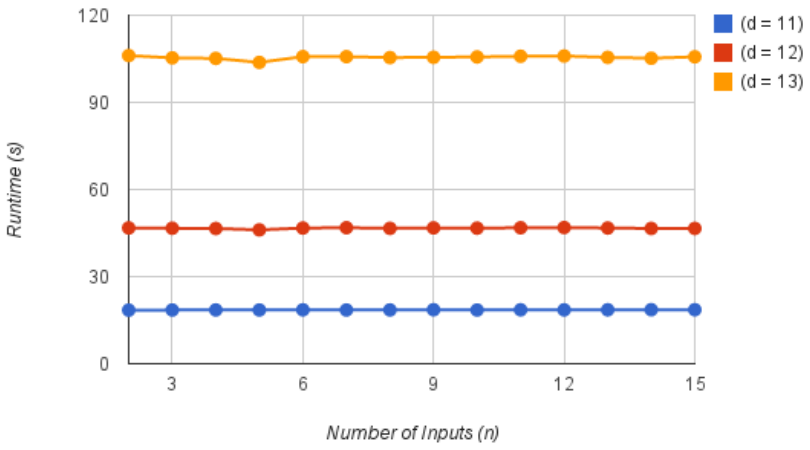

Figure 3: Runtime v/s Number of Inputs ($l = 42, k = 6, p = 12$)

We observed that the runtime didn't vary much on changing the number of inputs, which was consistent with what we expected. As we have explained Section 2.2, we always flip all the possible combinations of bits in S_1 and then compare it with other inputs at the very end. Since the comparison at the end is inexpensive, changing the input size shouldn't have a tangible effect on the runtime.

3.3 Input Size (l)

We varied input size in the range $[34, 44]$ and obtained the runtimes for solving the problem for three different parameter combinations: $(n = 10, d = 12, k = 6, p = 12)$, $(n = 10, d = 13, k = 6, p = 12)$, and $(n = 10, d = 14, k = 6, p = 12)$. The runtime plot is shown in Figure 4.

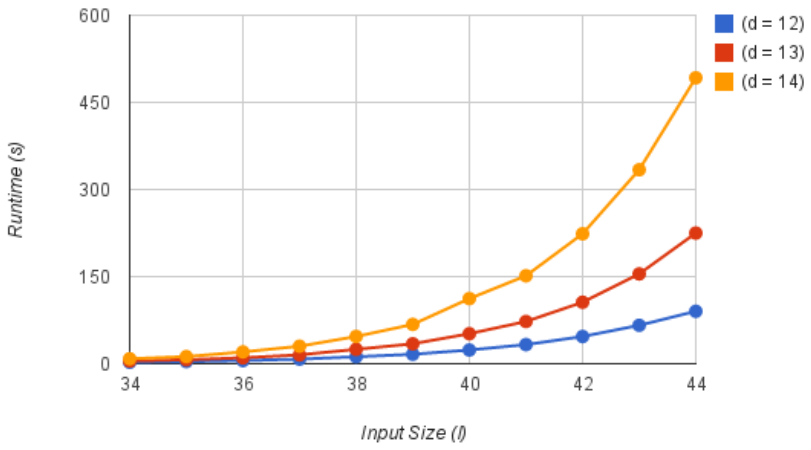

Figure 4: Runtime v/s Input Size ($n = 10, k = 6, p = 12$)

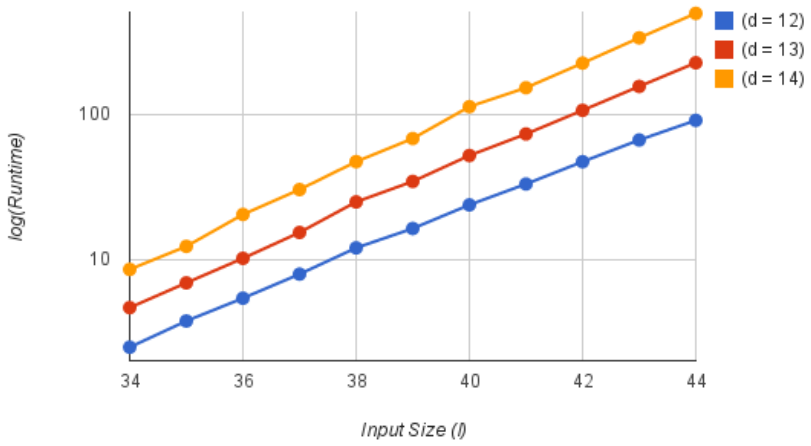

Figure 5: log(Runtime) v/s Input Size ($n = 10, k = 6, p = 12$)

We observed that the growth of runtime with increasing l looks exponential. We confirmed this by plotting log of runtime against input size in Figure 5. The plot showed linear growth, thus verifying our observation. Again, this is in line with what we would have expected since, for obtaining potential solutions, we are essentially doing a depth-first traversal of a binary tree of numbers. Therefore, increasing the number of bits in the input increases the number of levels in the binary tree, resulting in exponential increase in the number of potential solutions and runtime.

3.4 Maximum Distance (d)

We varied maximum distance in the range $[8, 15]$ and obtained the runtimes for solving the problem for three different parameter combinations: $(n = 10, l = 41, k = 6, p = 12)$, $(n = 10, l = 42, k = 6, p = 12)$, and $(n = 10, l = 43, k = 6, p = 12)$. The runtime plot is shown in Figure 6.

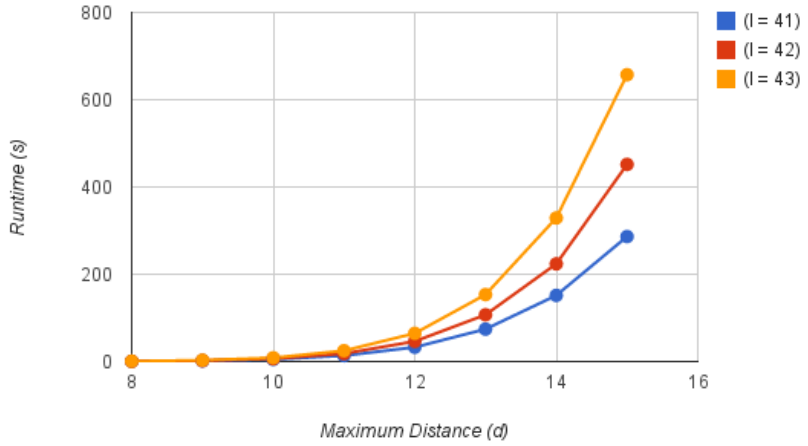

Figure 6: Runtime v/s Maximum Distance ($n = 10, k = 6, p = 12$)

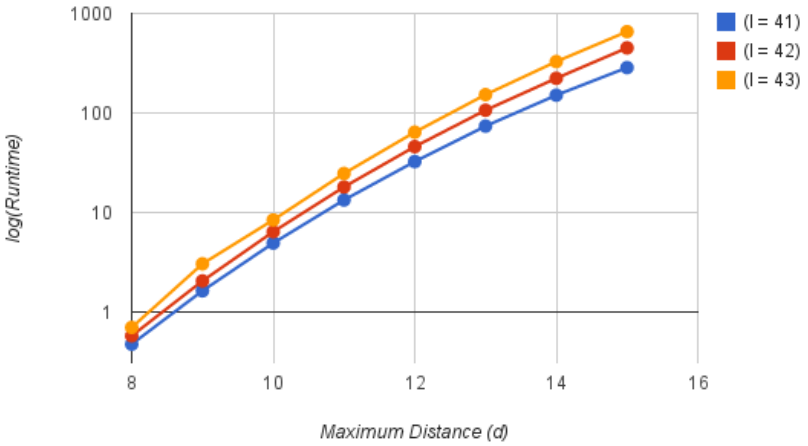

Figure 7: log(Runtime) v/s Maximum Distance ($n = 10, k = 6, p = 12$)

Much like Figure 4, the plot of growth of runtime with increasing d looked exponential. However, when we plotted log of runtime against maximum dis-

tance in Figure 7, we observed sublinear growth, thus implying sub-exponential growth of the original plot. This can be explained by observing that d is just a maximum value and all the values in the range $[0, d]$ are valid. Therefore, on increasing d, we are only adding one more value to the set of allowed values, which can lead to a sub-exponential growth at every step.

3.5 Master Depth (k)

We varied master depth in the range $[1, 33]$ and obtained the runtimes for solving the problem for three different parameter combinations: $(n = 10, l = 42, d = 11, p = 12)$, $(n = 10, l = 42, d = 12, p = 12)$, and $(n = 10, l = 42, d = 13, p = 12)$. The runtime plot is shown in Figure 8.

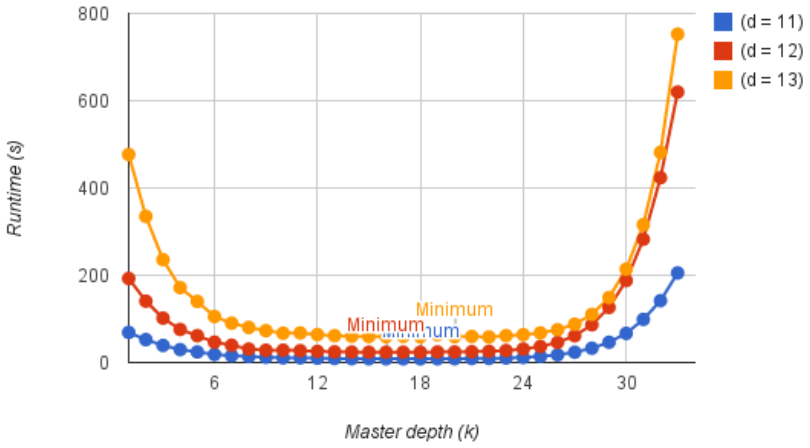

Figure 8: Runtime v/s Master Depth $(n = 10, l = 42, p = 12)$

We observed that, on increasing the master depth, the runtime first decreased, then stabilized around some minimum value, and then increased again. This can be explained by observing that if the master depth is too low or too high then we don't generate enough sub-problems and either one of the workers, when it is too low, or the master, when it is too high, end up doing bulk of the work. The minimum value is therefore obtained when all the nodes have enough work to do at all the times and the master isn't over-solving the problem.

The value of k for which the runtime is minimum, for the three cases mentioned above, is marked in the plot as well as tabulated in Table 1.

Problem parameters	**Optimal k-value**
$(n = 10, l = 42, d = 11, p = 12)$	18
$(n = 10, l = 42, d = 12, p = 12)$	16
$(n = 10, l = 42, d = 13, p = 12)$	20

Table 1