Name: Hrishikesh Deshpande hdeshpa@purdue.edu

PUID: 0028994843

Parallel Computing Assignment 1:

Note: For questions 1-5 machine mc12.cs.purdue.edu used for experimentation.

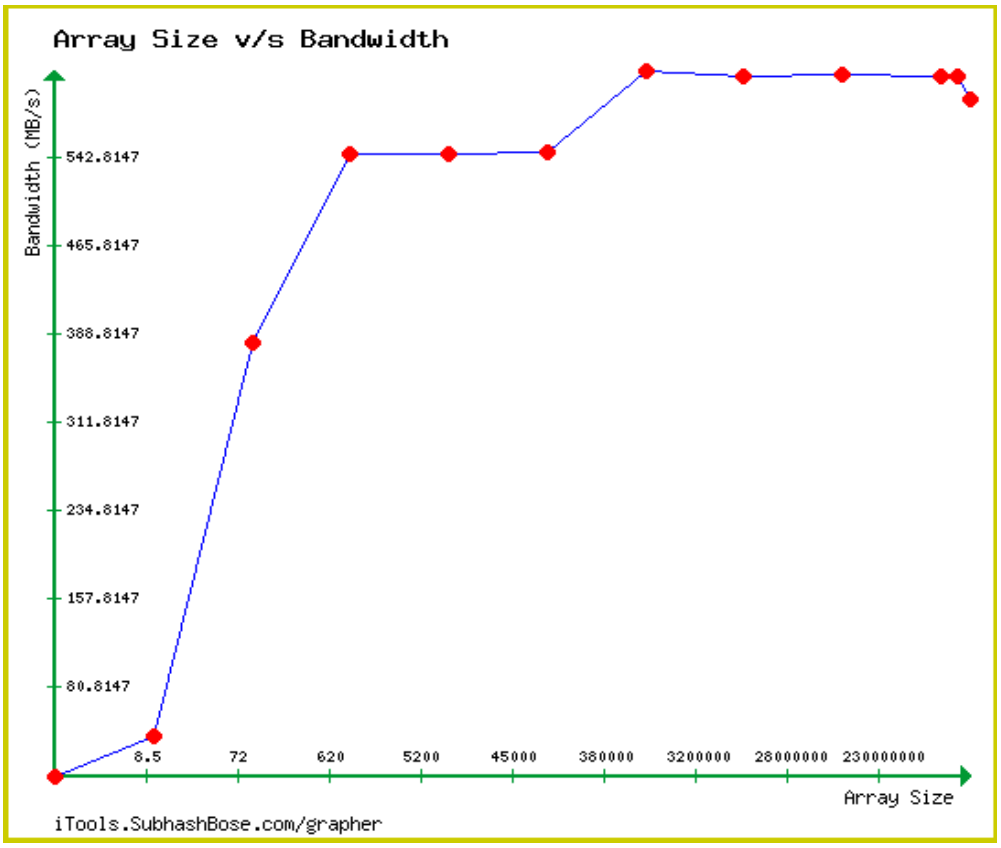
Q1. For smaller values of vector size, we find that the overall execution time is small suggesting extra fast memory accesses. However, this is because a smaller vector will fit almost entirely in the cache and thus data access time is minimal. Thus, with small vector sizes, bandwidth computations are impacted by caching. Therefore, a small vector size does not give us an accurate measure of memory bandwidth since what we get is really a fast cache access.

Sl. No	Array Size	Bandwidth (MB/s)
1.	1	3.8147
2.	10	38.147
3.	100	381.47
4.	1000	544.957
5.	10000	544.989
6.	100000	546.518
7.	1000000	618.666
8.	10000000	613.256
9.	100000000	614.588
10.	1000000000	613.135
11.	1500000000	614.036
12.	2000000000	593.678

As we increase the vector size, memory bandwidth increases exponentially. For large values of vector sizes say 10 million – 2 billion we find that the calculated memory bandwidth averages out. As shown in the graph below, we find that the bandwidth curve flattens out for large values of c.

The graph has logarithm of the array size on the X-axis and the bandwidth in MB/s on the X-axis. For larger values of the input vector (say 1-million-2 billion), a 4-MB cache cannot hold the entire vector and the processor needs to issue repeated read requests to the memory. This gives us a fair measure of the memory bandwidth.

The average value of the memory bandwidth (for large vector sizes 10 thousand- 2billion) is approximately 594.858 MB/s.



Q2.

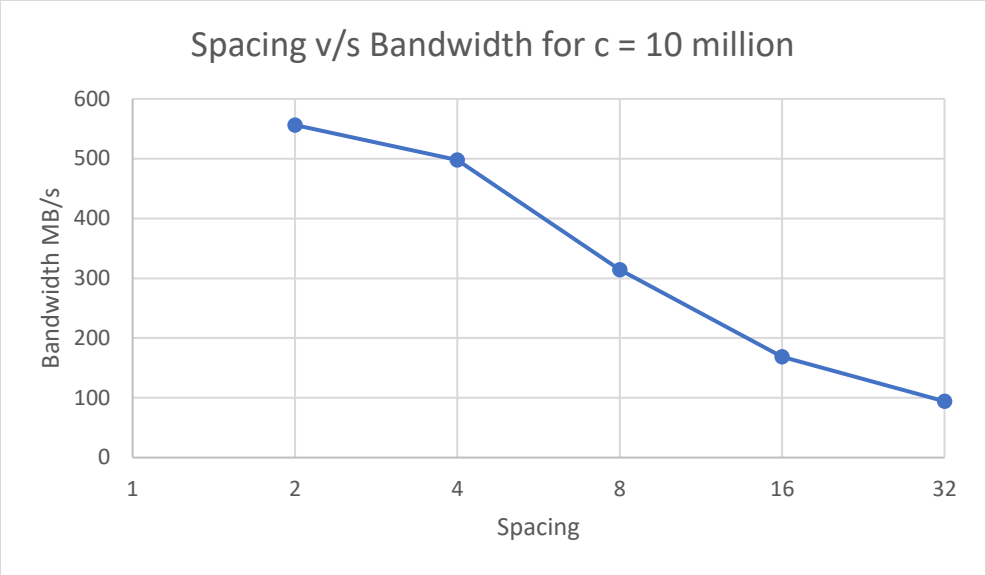
In question two, we keep the no of array accesses fixed at c = 10 million. However, the access is strided in that we access array entries with spacing = 2, 4,8,16 and 32.

With strided access, it is hard to achieve spatial locality since we no longer access data sequentially. Thus, a strided access results in poorer memory performance.

Further, with large array size, not all array data can be retained in the cache resulting in poor temporal locality.

Sl. No	Spacing	Bandwidth (MB/s)
1	2	556.305
2	4	498.041
3	8	314.368
4	16	168.425
5	32	94.1296

As shown in the graph below, memory bandwidth decreases with an increase in spacing since we now have strides, which are farther away from each other. Strided access achieves poor latency amortization and thus decreases the effective memory bandwidth.



Let us compare the value of memory bandwidth for c = 10 million in questions 1 and 2.

Comparing that with question 1, we can reason that a strided access achieves poor spatial and temporal locality in comparison to sequential access and has much lower memory bandwidth. The memory bandwidth for strided access ranges from 556.305 MB/s for a spacing of 2 to as low as 94.1296 MB/s indicating that bigger strides correspond to multiple cache misses and thus resulting in repeated fetch operations from memory. In comparison to that, in question 1, a sequential access for c = 10 million achieves a bandwidth of 613.256 MB/s (c = 10 million).

Question 3. Question 3 requires two different implementation of multiplying a matrix with a vector.

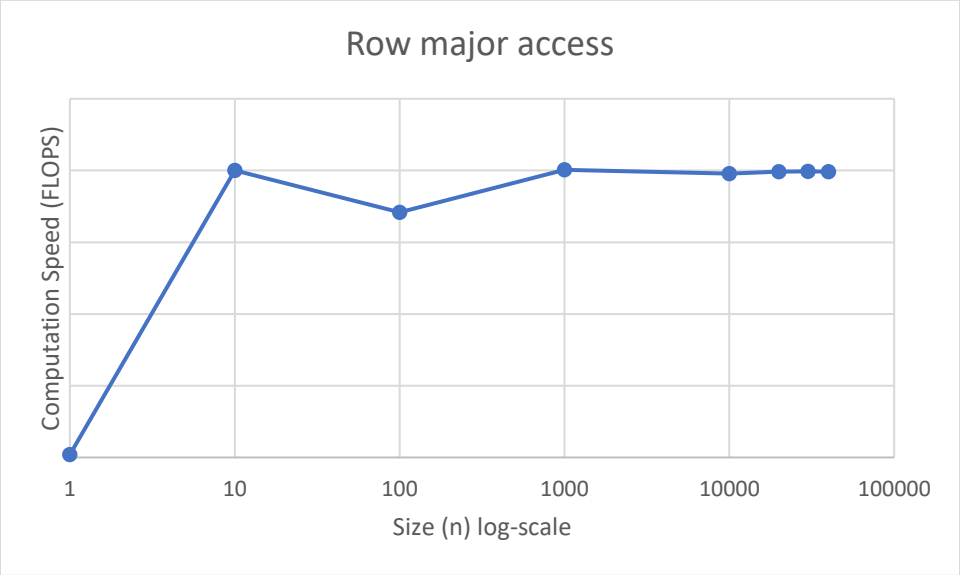
In the first case, we multiply a matrix with a vector with the elements of the matrix accessed row wise. In the second test, we multiply the same matrix with the vector but with the matrix elements accessed column wise.

Sl. No	Size (n)	Row-major FLOPS	Col-Major FLOPS
1	1	2000000.000000	2000000.000000
2	10	200000000.000000	200000000.000000
3	100	170940170.940171	169491525.423729
4	1000	200481154.771451	126135216.952573
5	10000	197872083.612828	98416381.997282
6	20000	199276774.765183	29302843.236565
7	30000	199483691.875716	13878776.122950
8	40000	199214571.649451	17361427.972242

Elements of a matrix (which is a 2-d array) are stored row wise in memory, i.e. the system stores the first row in memory followed by the second row and so on. Thus, a row major access corresponds to a sequential access in memory and has high spatial locality of reference. Consequently, matrix-vector multiplication with row wise matrix access gives us a higher speed of computation.

The graph below shows increasing computation speeds of computions starting from matrix sizes of 10 and onwards and for higher values of siz, it more or less averages out. This is because, beyond a certain limit, the impact of caching is reduced. Further, a sequential access achieves high locality of reference.

The maximum speed for row-major access is observed to be 200481154.771451 FLOPS (approx. 200 MFLOPS).



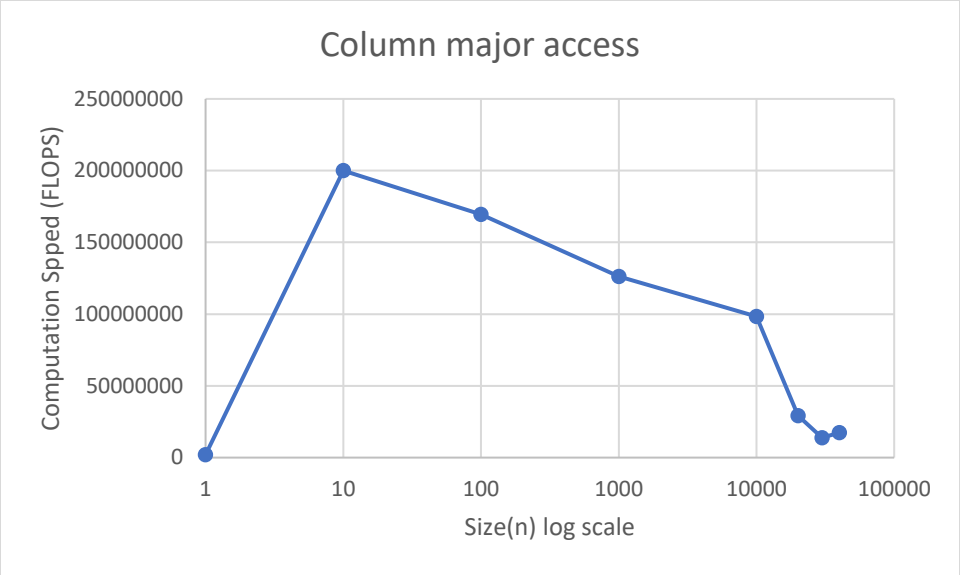
By interchanging the loops, the program gives us the same matrix-vector product but in this case, matrix elements are accessed column-wise. Since matrix elements are stored row-wise in memory, a column-major access of matrix elements corresponds to a strided access and thus results in poor speed of computation as indicated by the plot below. For smaller matrix sizes, the entire matrix might possibly fit in the cache so we see results comparable to row-based access.

For larger matrix sizes however, the impacts of strided access for column-major based multiplication are clearly noticeable.

A side-by-side comparison in the table above, we see a significant difference in computation speeds for matrix sizes of 1000 and above.

For matrix sizes beyond 10000, column based access speed tank exponentially. The maximum speed achieved 126135216.95 FLOPS for matrix sizes of 1000 and above. For a size of 10, we obviously get a much higher speed of 200000000 since the entire matrix might fit in the cache.

The graph below plots column major matrix product times as a function of matrix sizes.



\

Q4.

Question 4 looks at the impact of caches on the speed of computations. More specifically, we perform a matrix multiplication with two n X n matrices where n is the matrix dimension.

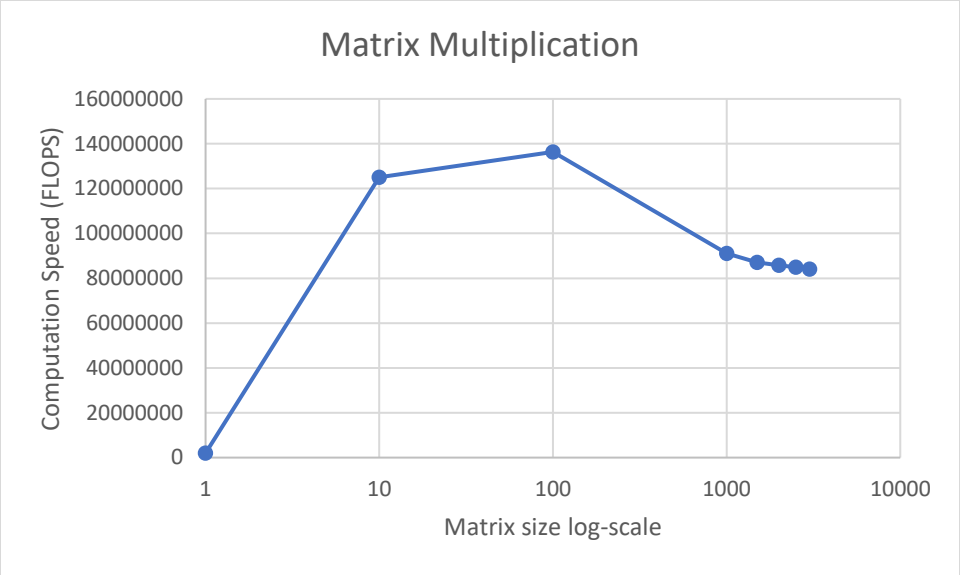
For smaller matrix sizes, both the input matrices and the product matrix can fit almost entirely in the cache and thus data access times are minimal. Thus, we can observe higher than expected speeds of computations w.r.t matrix sizes in the range approximately 10-100.

However, as we increase the matrix size, computation speeds drop down drastically since bigger matrices do not fit in the cache and we thus need to issue repeated fetches to bring in the required matrix data into the cache before being able to process it.

Thus, we can observe an exponential decrease in computation speeds with increase in matrix sizes. However, beyond matrix sizes of 1000, average computation speeds averages out or decreases linearly since the overall cache limit is always exceeded resulting in cache misses and subsequent memory fetch requests. Therefore, the average computation speed flattens out/reduces linearly for higher matrix sizes.

The graph and the table below shows the variation in computation speeds as a function of matrix sizes.

Sl. No	Matrix Size (n)	Computation Speed (FLOPS)
1.	1	2000000.000000
2.	10	125000000.000000
3.	100	136351240.796291
4.	1000	91105655.502003
5.	1500	87000291.431643
6.	2000	85782348.339931
7.	2500	84977670.750569
8.	3000	84126643.426443
9.	3500	82906667.102596
10.	4000	87021624.496367



Q5. In question 5 we have 2 threaded programs threadedpi1 and threadedpi2 which are used to estimate the value of the constant pi using the Monte Carlo method. We run both programs for 10 million darts and plot their efficiency as a function of no of threads.

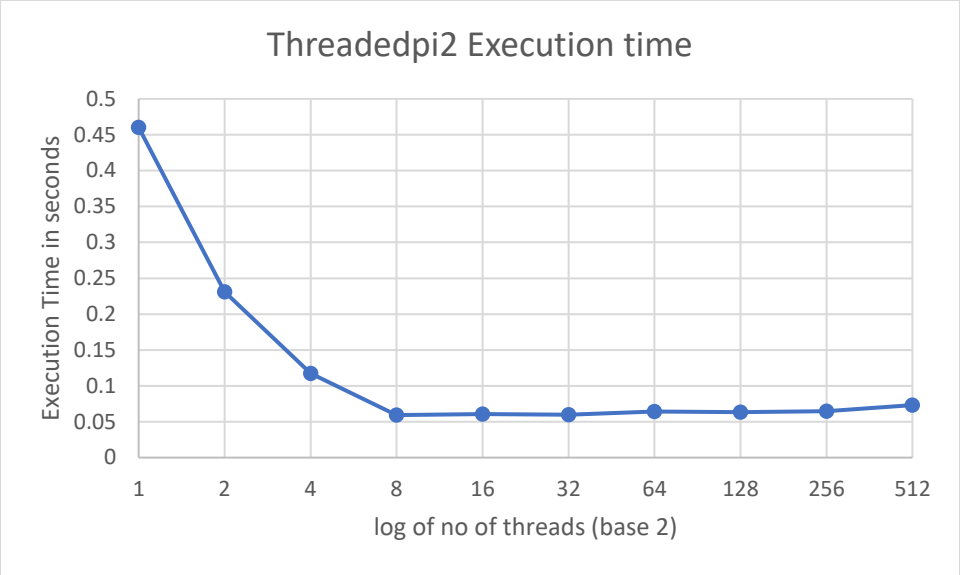
The results are given below:

Sl. No	No of threads	threaded_pi1.c time	threaded_pi2.c time
1.	1	0.464700	0.460236
2.	2	0.511353	0.231162
3.	4	0.585376	0.117107
4.	8	0.879483	0.059232
5.	16	0.891796	0.060434
6.	32	0.318847	0.059933
7.	64	0.188038	0.064151
8.	128	0.139879	0.063240
9.	256	0.117851	0.064888
10.	512	0.172233	0.072937

The only major difference in the two programs is in how they increment the no of hits inside the for loop. While program 1 directly increments the hit pointer as *(hit_pointer)++ and the second program increments a local variable hits and assigns the final incremented value to the hit pointer outside the loop. As expected, the second threaded program achieves high efficiency and thus a lower runtime.

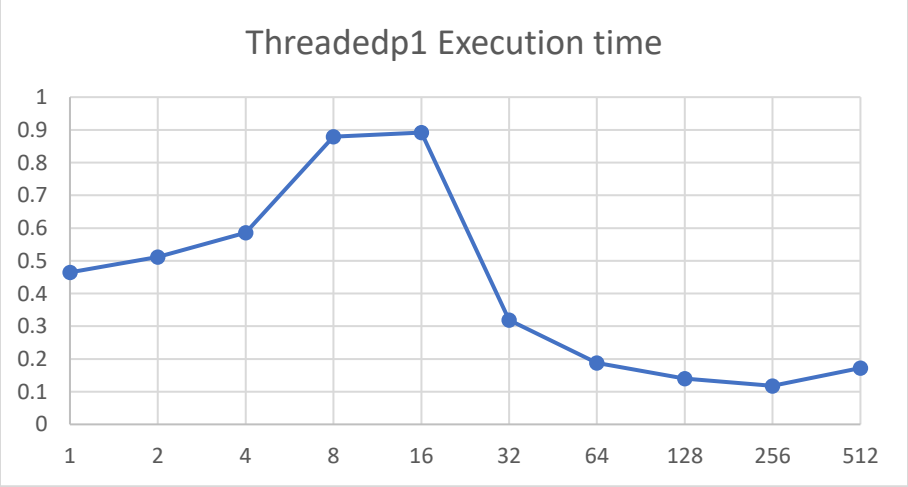
Since each thread has its own copy of stack variables, increment of a local variable “hits” by one thread does not affect the other thread. Thus, we see very small runtimes for the second program.

This is shown in the first graph below.



The first program illustrates the overhead of “false sharing”. important performance overhead called. In the first program, instead of incrementing a local variable, hits and assigning it to the array entry outside the for loop, we now directly increment the corresponding entry in the hits array as *(hit_pointer)++.

In the first program, two adjoining data items (which likely reside on the same cache line) are being continually written to by threads that might be scheduled on different processors. We know that a write to a shared cache line results in an invalidate and a subsequent read must fetch the cache line from the most recent write location. With that, we can clearly see that the cache lines corresponding to the hits array generate many invalidates and reads because of repeated increment operations. This situation, in which two threads 'falsely' share data because it happens to be on the same cache line, is called false sharing.
Due to the false sharing overhead, the running time of the threaded pi1 program is significantly higher in comparison to the second program. The graph below illustrates the impact of false sharing on program execution.



6. In question 6, we execute the program mpi_message_pingpong.c on two processors with ping_pong_limt = 1000 and varying message sizes. The table below gives us the per message time after dividing the overall time by 1000 (ping pong limit) and then by 2 (to account for round trip). The experiment was run for message sizes starting from 1 until 262144 increasing by powers of 2.

Sl. No	Message Length	Overall Time (Seconds)	Per message time after dividing by ping_pong_limit and 2
1.	1	0.000645	3.225E-07
2.	2	0.000649	3.245E-07
3.	4	0.000660	0.00000033
4.	8	0.000708	0.000000354
5.	16	0.000732	0.000000366
6.	32	0.000748	0.000000374
7.	64	0.000833	4.165E-07
8.	128	0.001184	0.000000592
9.	256	0.001385	6.925E-07
10.	512	0.001913	9.565E-07
11.	1024	0.003826	0.000001913
12.	2048	0.005302	0.000002651
13.	4096	0.009077	4.5385E-06
14.	8192	0.014364	0.000007182
15.	16384	0.021892	0.000010946
16.	32768	0.036346	0.000018173
17.	65536	0.064277	3.21385E-05
18.	131072	0.120777	6.03885E-05
19.	262144	0.235728	0.000117864

The same experiment was performed for a shorter range of message sizes with size ranging from 8192 to 100000. The results obtained have been tabulated as below:

Sl. No	Message Length (Floats)	Overall Time (Seconds)	Per-message time after dividing by ping_pong_limit and 2
1.	8192	0.014392	0.000007196
2.	10000	0.017368	0.000008684
3.	16384	0.021865	0.0000109325
4.	20000	0.025641	0.0000128205
5.	30000	0.034348	0.000017174
6.	32768	0.036343	0.0000181715
7.	40000	0.042522	0.000021261
8.	50000	0.051277	0.0000256385
9.	60000	0.060461	0.0000302305
10.	65536	0.064444	0.000032222
11.	70000	0.068627	0.0000343135

12.	80000	0.077231	0.0000386155
13.	90000	0.085444	0.000042722
14.	100000	0.094960	0.00004748

We can estimate the startup time (t_s) and per-word transfer time (t_w) from the total message communication time as:

$t_{comm} = t_s + m*t_w$ where m is the no of words in the message.

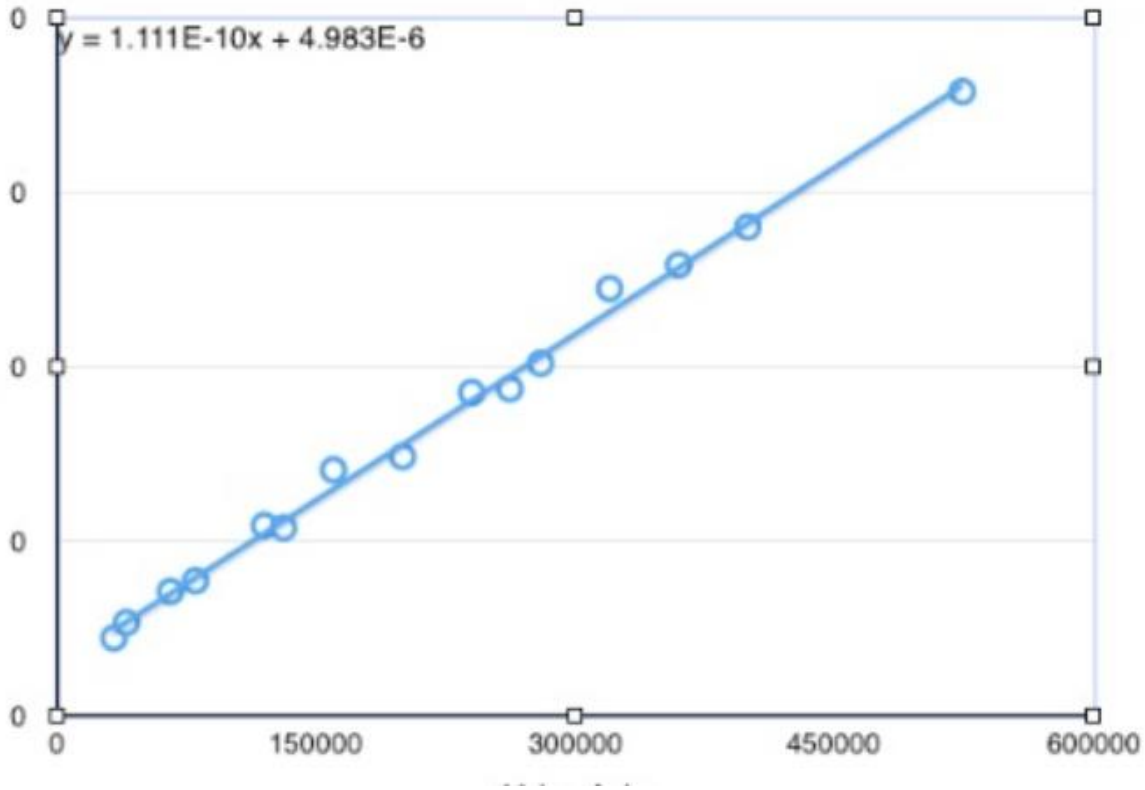
In the graph, we fit a straight line to the resulting plot and estimate the startup and per word transfer time by solving the equation:

y = mx + c --- 1

Where the slope is per-word transfer time and the y-intecept is the startup time.

The graph below depicts the computation:

Message size (in bytes) v/s computation speed at Pingpong limit = 1000



From the above graph, we can estimate, per word transfer time = 1.111E-10 second

Startup time = 4.983 microseconds.

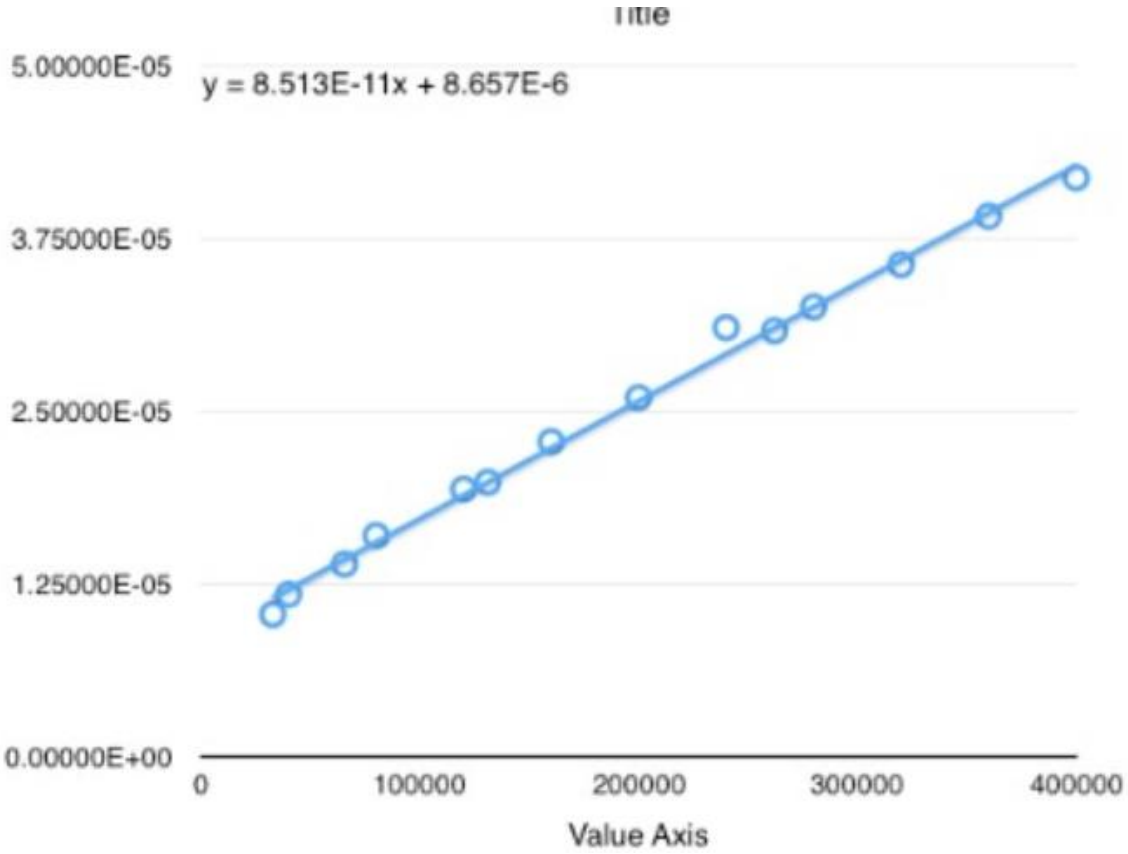
NOTE: the table has message size in FLOPS while in the graph we use bytes for computation as suggested in the question.

7. In question 7 we run the program mpi_message_pingpong_multiple.c with number of processors = 2,4, 8 and 16 and ping_pong_limit = 1000, message_size = variable.

The results obtained can be summarized as below:

a) With 2 processors

Sl. No	Message Length (bytes)	Overall Time (Seconds)	Per-message time after dividing by ping_pong_limit and 2
1.	1	0.000654	0.000000327
2.	2	0.000650	0.000000325
3.	4	0.000639	0.0000003195
4.	8	0.000688	0.000000344
5.	16	0.000703	0.0000003515
6.	32	0.000764	0.000000382
7.	64	0.000880	0.00000044
8.	128	0.001197	0.0000005985
9.	256	0.001378	0.000000689
10.	512	0.001917	0.0000009585
11.	1024	0.003869	0.0000019345
12.	2048	0.005258	0.000002629
13.	4096	0.008937	0.0000044685
14.	8192	0.014377	0.0000071885
15.	10000	0.017157	0.0000085785
16.	16384	0.022030	0.000011015
17.	20000	0.025613	0.0000128065
18.	30000	0.034515	0.0000172575
19.	32768	0.036411	0.0000182055
20.	40000	0.042767	0.0000213835
21.	50000	0.051167	0.0000255835
22.	60000	0.060650	0.000030325
23.	65536	0.064701	0.0000323505
24.	70000	0.068641	0.0000010474
25.	80000	0.076993	0.0000384965
26.	90000	0.085457	0.0000427285
27.	100000	0.094710	0.000047355
28.	131072	1.082582	0.000541291

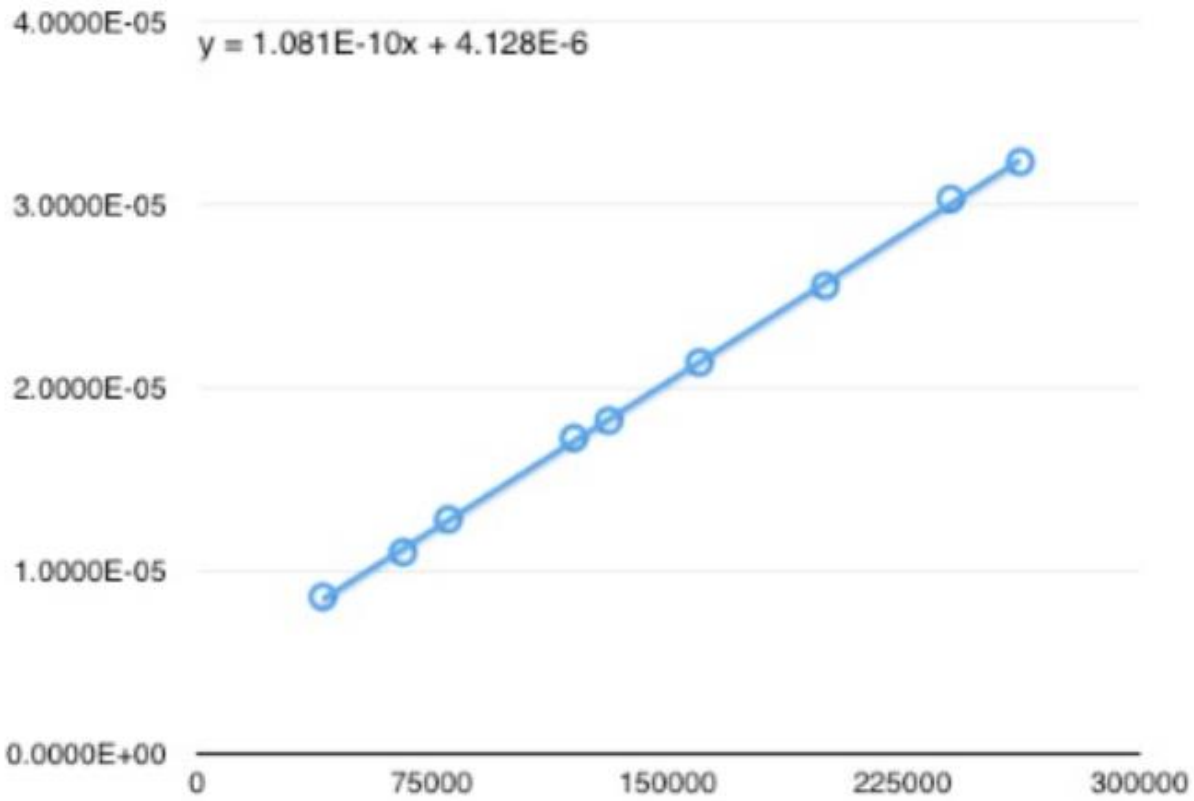


Per word transfer time: 8.5E-11 seconds, startup time: 8.65 microseconds

b) With 4 processors

Sl. No	Message Length (bytes)	Overall Time (Seconds)	Per-message time after dividing by ping_pong_limit and 2
1.	1	0.000721	0.0000003605
2.	2	0.000670	0.000000335
3.	4	0.000782	0.000000391
4.	8	0.000831	0.0000004155
5.	16	0.000824	0.000000412
6.	32	0.000789	0.0000003945
7.	64	0.001015	0.0000005075
8.	128	0.001278	0.000000639
9.	256	0.001432	0.000000716
10.	512	0.002210	0.000002158
11.	1024	0.003969	0.0000019845
12.	2048	0.005867	0.0000029335
13.	4096	0.009796	0.000004898
14.	8192	0.015600	0.0000078
15.	10000	0.018680	0.00000934
16.	16384	0.024922	0.000012461
17.	20000	0.027072	0.000013536
18.	30000	0.038181	0.0000190905

19.	32768	0.037689	0.0000188445
20.	40000	0.049311	0.0000246555
21.	50000	0.051983	0.0000259915
22.	60000	0.064872	0.000032436
23.	65536	0.065580	0.00003279
24.	70000	0.070683	0.0000353415
25.	80000	0.085699	0.0000428495
26.	90000	0.090434	0.000045217
27.	100000	0.098021	0.0000490105
28.	131072	0.125194	0.000062597

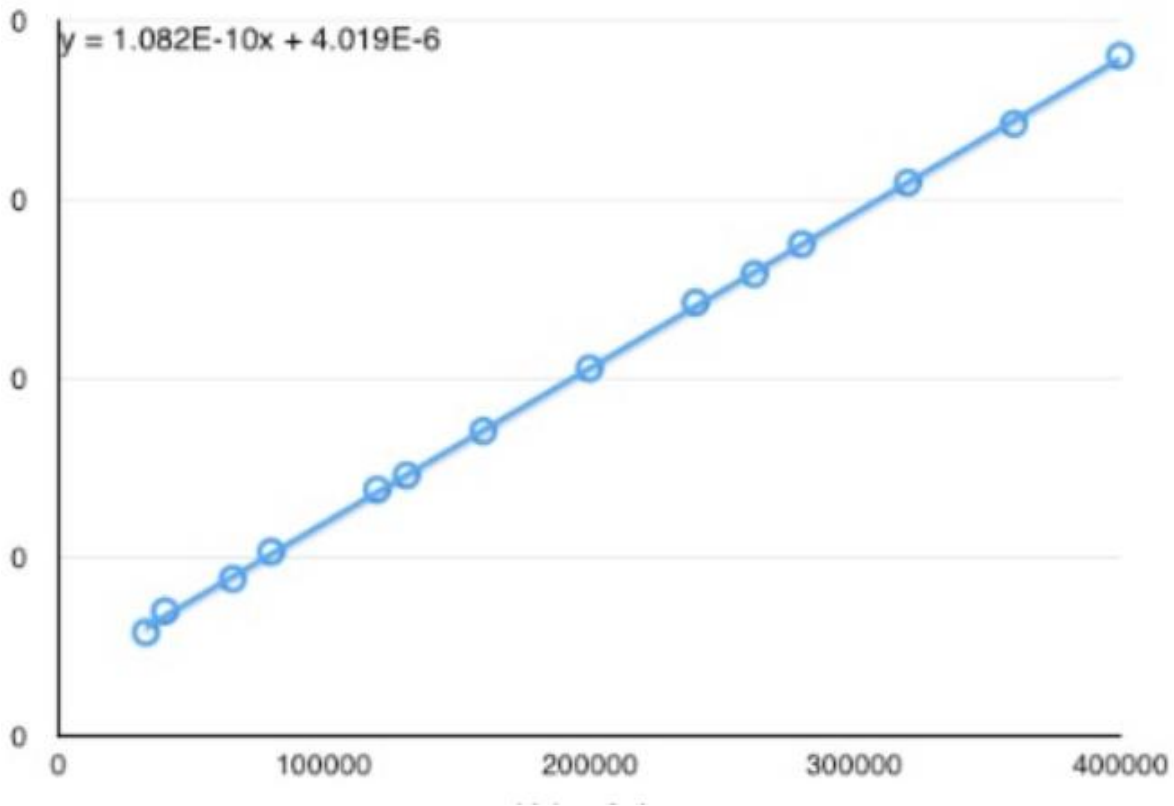


Per word transfer time: 1.081E-10 seconds, Startup time: 4.128 microseconds.

c) With 8 processors

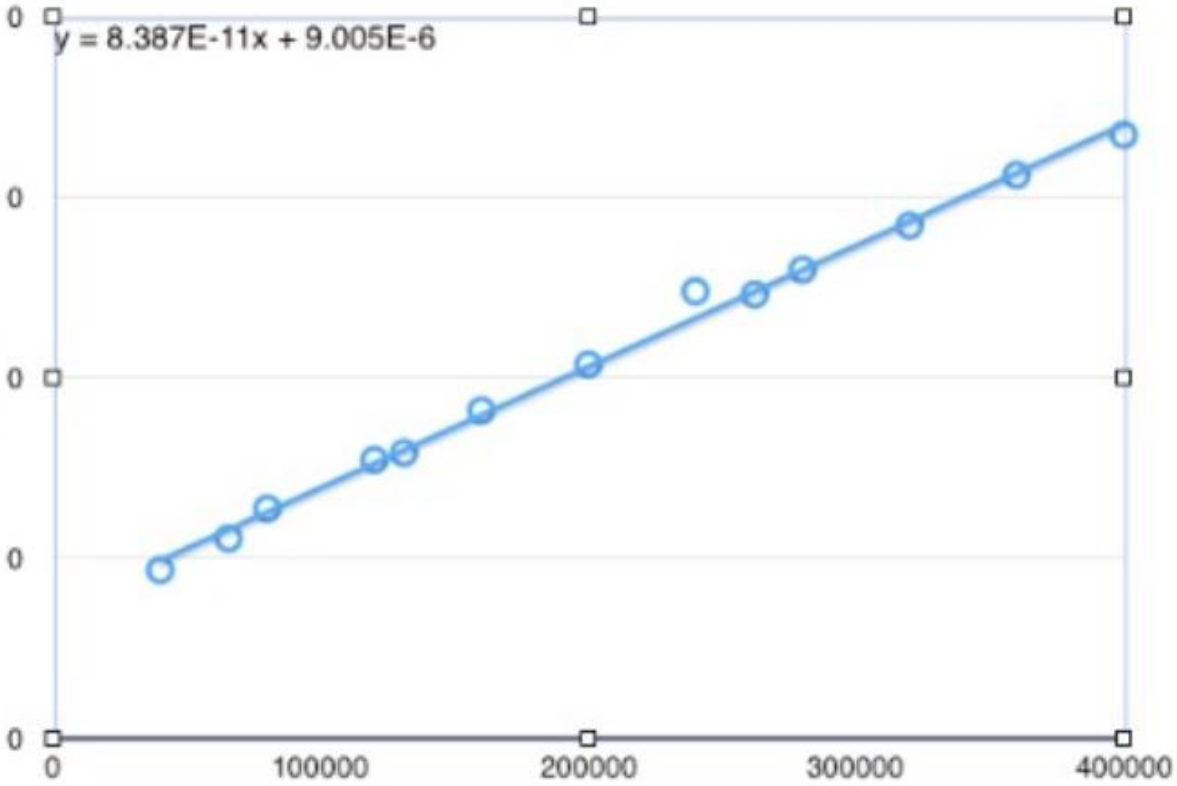
Sl. No	Message Length (bytes)	Overall Time (Seconds)	Per-message time after dividing by ping_pong_limit and 2
1.	1	0.000724	0.000000362
2.	2	0.000731	0.0000003655
3.	4	0.000683	0.0000003415
4.	8	0.000975	0.0000004875
5.	16	0.000888	0.000000444
6.	32	0.001046	0.000000523

7.	64	0.001208	0.000000604
8.	128	0.001428	0.000000714
9.	256	0.001921	0.0000009605
10.	512	0.003121	0.0000030478
11.	1024	0.005915	0.0000057763
12.	2048	0.007520	0.00000376
13.	4096	0.014223	0.0000071115
14.	8192	0.020513	0.0000102565
15.	10000	0.023392	0.000011696
16.	16384	0.027777	0.0000138885
17.	20000	0.031909	0.0000159545
18.	30000	0.038646	0.000019323
19.	32768	0.039617	0.0000198085
20.	40000	0.045494	0.000022747
21.	50000	0.051859	0.0000259295
22.	60000	0.061993	0.0000309965
23.	65536	0.061555	0.0000307775
24.	70000	0.064989	0.0000324945
25.	80000	0.071116	0.000035558
26.	90000	0.078101	0.0000390505
27.	100000	0.083670	0.000041835
28.	131072	0.109169	0.0000545845



Startup time: 1.082E-10 seconds, Per word transfer time: 4.019 microseconds

d) With 16 processors



Startup time: 9 microseconds Per word transfer time: 8.38E-11 seconds

We can estimate the startup time (t_s) and per-word transfer time (t_w) from the total message communication time as:

$t_{comm} = t_s + m*t_w$ where m is the no of words in the message.

In the graph, we fit a straight line to the resulting plot and estimate the startup and per word transfer time by solving the equation:

y = mx + c --- 1

Where the slope is per-word transfer time and the y-intecept is the startup time.

From the above graphs, it is clear that as we increase the no of processors the message startup time increases. The per-word transfer time is variable.

Startup time:

There is an increased pressure on the communication channel which will result in an increase in per-word transfer time. This clearly demonstrates the overheads involved in communication as we increase the no of processors. More processors mean that multiple messages are on the communication channel between different processors which increases communication overheads. Since the no of processors increases, we observe an increased startup time.

Q8. Theoretical Questions

2.6

We are given an SMP with a distributed shared address space.

Given: a) time to access local cache: 10ns

b) time to access local memory: 100ns

c) time to access remote memory: 400ns

We run a parallel program on this machine. 80% of the accesses go to the local cache, 10% to local memory and the rest 10% to remote memory.

So the effective memory access time of this system is:

0.8 * 10 + 0.1 * 100 + 0.1 * 400

= 8 + 10 + 40

= 58ns

Peak computation rate = 1 / 58ns = 17.24 MFLOPS

Now consider the same computation running on one processor. Here the processor hits the cache 70% and the remaining 30% of the accesses are to the local memory.

Effective memory access time for this system is:

0.7 * 10 + 0.3 * 100 = 7 + 30 = 37ns

Peak computation rate = 1 / 37 = 27.027 MFLOPS

Fractional computation rate of serial processor w.r.t parallel processor configuration = 27.027/17.24 = 1.57

2.12 A cycle in a graph is defined as a path originating and terminating at the same node. The length of the cycle is the number of edges in the cycle. Show that there are no odd-length cycles in the d-dimensional hypercube

Solution:

Basic Observations:

As per convention, the nodes in a d-dimensional hypercube can be represented by d-bit vectors. Two nodes are said to be neighbors if they differ by exactly one bit.

For example: a 3-D hypercube will have 8 nodes given by labels (0,0,0), (0,0,1), ….., (1,1,1).

So, when we traverse from a given node in the hypercube to its neighbor we carry out exactly 1 bit flip.

Proof:

We shall prove the result by the method of contradiction. Let us assume that we have an odd-length cycle in the graph A1, A2,…. Ak.

Then starting from a given node A1 we need to traverse an odd no of edges and reach back at node A1 i.e A1 = Ak. This means that starting at A1 we need to carry out an odd-no of bit flips and recover the original bit pattern for A1.

Ex: if A1(0,0,0) then after odd bit flips we need to get back (0,0,0).

This is not possible since we need to flip a bit an even no of times to restore it back to its original value.

So, every cycle in the hypercube (with same starting and ending nodes), must carry out an even no of bit flips i.e. has a path length which is even.

This is clearly a contradiction. Thus, we can prove that there are no odd length cycles in the d-dimensional hypercube.

2.13 The labels in a d-dimensional hypercube use d-bits. Fixing any k of these bits, show that the nodes whose labels differ in the remaining (d-k) bits form a (d-k) dimensional subcube composed of $2^{(d-k)}$ nodes.

Solution:

Key Observations:

In a d-dimensional hypercube, every node can be represented using a d-bit vector. Two nodes are neighbors if they differ by exactly one bit.

Proof:

Initially, we have a d-dimensional hypercube which means that it has 2^d nodes and each node has unique d-bit vectors with neighbors varying by exactly one bit.

Now if we fix k of these bits for all nodes, then we need to prove that the remaining (d-k) bits that correspond to $2^{(d-k)}$ processors form a d-k dimensional hypercube.

Ever node in a p-processor hypercube has log(p) no of neighbors. Since k of these bits are fixed, the remaining (d-k) bits form a (d-k) dimensional hypercube if each processor in this group has log ($2^{(d-k)}$) = (d-k) communication links to other processors in the same group.

Since k bits are fixed, no communication links can exist between processors corresponding to these bit positions in the group (since these bits are same and neighbors must vary by 1 bit). Further, after fixing k bits, we can have $2^{(d-k)}$ unique bit combinations. So, all $2^{(d-k)}$ processors that differ along any of these (d-k) bit positions belong to the same group.

Also, each processor must have a neighbor that belongs to the same group (since k of the bits are fixed, all processors in this group will have identical k bits and differ only in the remaining (d-k) bits).

Thus,

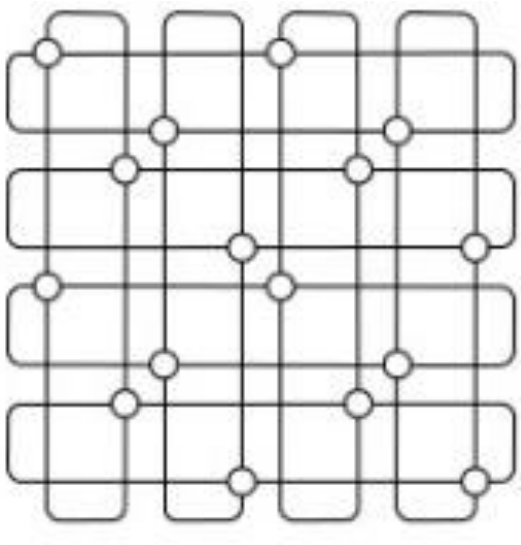
1. There are $2^{(d-k)}$ processors in the group.

2. Each processor has links to processors to processors of the same group (since these nodes which are neighbors must differ by only 1 bit within the given d-k bits).
3. Each processor in the group can be uniquely represented by one the $2^{(d-k)}$ bit patterns.

From 1 ,2 and 3 we can infer that each processor has (d-k) neighbors and two processors which are neighbors belong to the same group.

This proves that the given substructure is a (d-k) dimensional hypercube with $2^{(d-k)}$ nodes each represented by a unique bit pattern.

2.20 Illustrate a 4 x 4 wraparound mesh with equal wire lengths



By looking at the figure above, we can conclude that the mesh has exactly 4 x 4 = 16 nodes. The links are wraparound. Also, since nodes/processors alternate on the links, all links end up having approximately equal lengths. Therefore, this is a 4 x 4 wraparound mesh with equal lengths.