CSE 491-Section 2
Parallel Computing

Homework 1 - Solutions

1. (25 points)

a) 3D Torus of dimensions (4x4x4), Total Cost = 4x4x4 x 5000 + 4 x 4 x 4 x 3 x 1000 = 512,000

b) 6-D hypercube, Total cost = 2^6 x 5000 + 6 x 2^5 x 1000 = 512,000
7-D hypercube, Total cose = 2^7 x 5000 + 7 x 2^6 x 1000 = 1088000

c)

Let us consider possible configurations using $1.1 million

- Ring: 183 compute nodes, 183 links. Cost = 183 * ($5000 + $1000) = $1,098,000.

- Bus: 183 compute nodes, 182 links. Cost = 183 * $5000 + 182 * $1000 = $1,097,000.

- 2D Torus: 156 compute nodes in a 12x13 grid configuration, 312 links (2 per node). Cost = 156 * ($5000 + $2000) = $1,092,000.

- 3D Torus: 128 compute nodes in a 4x4x8 grid configuration, 384 links (3 per node). Cost = 125 * ($5000 + $3000) = $1,024,000.

- 4D Torus: 108 compute nodes in a 3x3x3x4 grid configuration, 432 links (4 per node). Cost = 108 * ($5000 + $4000) = $972,000.

- Hypercube: We can buy a 7 dimensional hypercube that has 128 (=2^7) compute nodes and 448 (=$7*2^6$) edges. Cost = 1088000

Utilities of topologies above for a rating of 5 per node and 1 per link of bisection bandwidth (note that this is NOT the same as the total number of links!):

- Ring: 183 * 5 + 2 * 1 = 917
- Bus : 183 * 5 + 1*1 = 916
- 2D Torus: 156 * 5 + 2*12 * 1 = 804
- 3D Torus: 128 * 5 + 2*4*4 * 1 = 672
- 4D Torus: 108 * 5 + 2*3*3*3 * 1= 594
- Hypercube (6D): 64 * 5 + 32 * 1= 352
- Hypercube (7D): 128 * 5 + 64 * 1 = 704

For a rating of 5 per node and 3 per link of bisection bandwidth:

- Ring: 183 * 5 + 2 * 3 = 921
- Bus : 183 * 5 + 1*3 = 918
- 2D Torus: 156 * 5 + 2*12 * 3 = 852

- 3D Torus: 128 * 5 + 2*4*4 * 3 = 736

- 4D Torus: 108 * 5 + 2*3*3*3 * 3 = 702

- Hypercube (6D): 64 * 5 + 32 * 3 = 416

- Hypercube (7D): 128 * 5 + 64* 3 = 832

  Hence the university will choose RING topology in both the cases.

2. (20 points)

a) Time needed for each processor for execution = 10^9 / 10^6 = 10^3 = 1000. Time needed for passing 10^9(p-1) messages = 10^9(1000 -1) * 10^ (-9) = 1000 + 999 = 1999 sec.

b) Time needed for each processor for execution = 10^9 / 10^6 = 10^3 = 1000. Time needed for passing 10^9(p-1) messages = 10^9(1000 -1) * 10^ (-3) = 1000 + 999000000 = 999001000 sec.

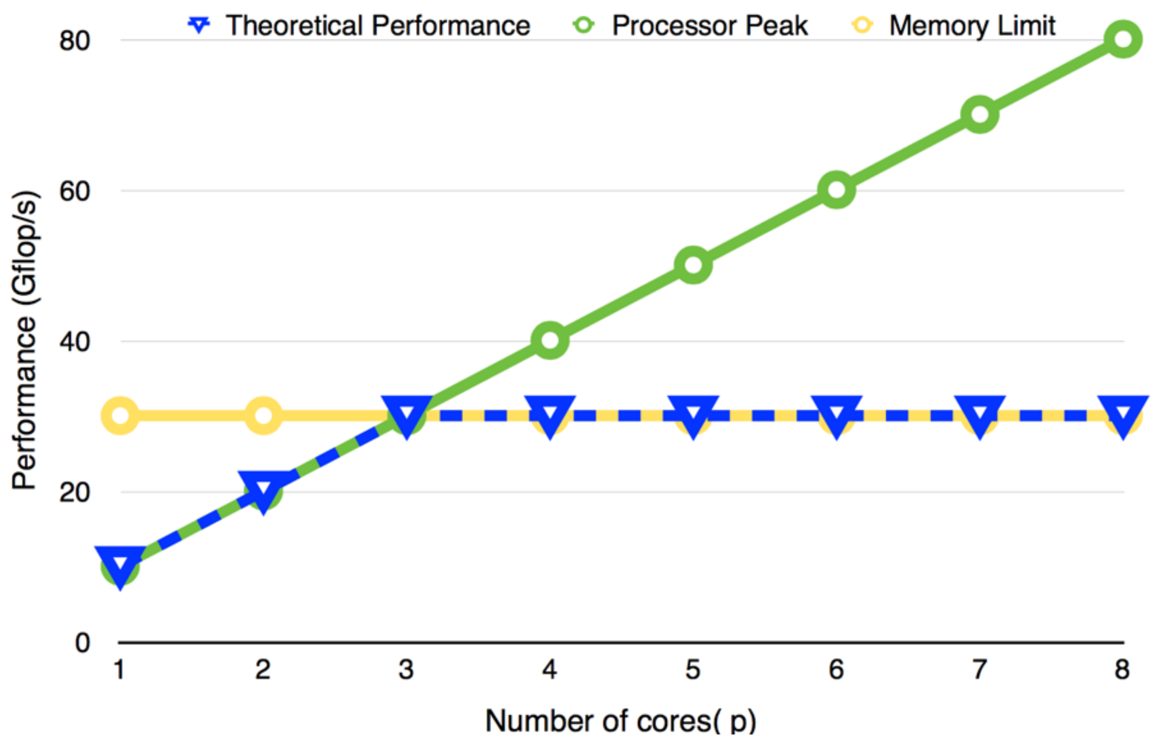
3. (20 points)

A: At each iteration, there are 6 floating point operations (4 multiplications, 2 additions), 2 memory reads (y[i] and z[i]) and 1 memory write (x[i]). In single precision, each read/write corresponds to 4 bytes of data transfer. Consequently, q = f/m = 6/(3*4) = 0.5 flop/byte.

B: We assume a computation over large N. Since all 3 arrays are accessed in a “streaming memory access” pattern, memory latency can safely be ignored. As a result, the the memory bandwidth will be a determining factor in the peak performance.

Memory limit = 60 Gbyte/s * 0.5 flop/byte = 30 Gflop/s.

Assuming we can saturate the memory bandwidth using as few as a single core, the computation will be CPU-bound for 1 <= p <= 3, and memory-bound for p > 3.



4. (35 points)

- **A:** Note that we are overwriting the value in A[i][j]. Therefore, we do not need to load it into cache, the result of x[i] * y[i] can be stored directly in the memory address of A[i][j]. Hence the outer product requires caching x and y vectors only. Both vectors are 1000 * 8 = 8000 bytes . So once they are loaded into cache (occupies 16000 Bytes out of 64 KB), they can continue to reside there until the end of the computation.

The only cache misses will be incurred during loading. Note that a cache line is 64 byes long and a double variable is 8 bytes, meaning that there will be a single cache miss for every 8 elements.

Cache misses = 1000/8 + 1000/8 = 250.

B: In this case, each vector is 10,000 * 8 = 80000 bytes in size. Therefore they will not fit into cache at the same time. With the given implementation, while each element of x needs to be loaded only once into cache, the entire y vector would need to be loaded for every element of x!

Cache misses = 10000/8 (for loading x) + 10000 (for each x[i]) * 10000/8 (for loading y) = 1250 + 10250 * 1250

= 12501250

C: (12 pts) We can use blocking to reduce the number of cache misses. With blocking, the elements of x would still be loaded once, but the number of loads for each element of y will be equally to the number of blocks. Therefore, block sizes should be as large as possible. So given a cache size of 64 KBs, one would choose the block size to be 4000 (4000*8*2 = 64,000 < 65,536 bytes to leave some space for other variables like loop variables, temporary storages, etc.).
In code:

```
B = 4000;
for (bi = 0; bi < N; bi += B)
  for (bj = 0; bj < M; bj += B)
    for (i = bi; i < min(bi+B, N); i++)
      for (j = bj; j < min(bj+B, M); j++)
        A[i][j] = x[i] * y[j];
```

D:

Data Size	Block Size	Basic Code Cache Misses	Blocked Code Cache Misses	Basic Code Runtime	Blocked Code Runtime
5000	1000	1,597,069	1,735,942	0.599199	0.506154
5000	2000		1,453,354		0.483861
5000	4000		1,560,854		0.489032
5000	8000		1,475,150		0.469402
10000	1000	8,348,873	7,532,047	2.7119	2.29141
10000	2000		7,131,237		2.21006
10000	4000		7,693,613		2.33585
10000	8000		8,322,604		2.52782
20000	1000	31,003,289	28,431,197	10.2388	8.92021
20000	2000		26,804,676		8.569
20000	4000		23,558,669		7.85623
20000	8000		27,249,273		8.36611
30000	1000	680,479,97	575,497,71	57.0905	70.4231
30000	2000		671,218,18		59.2589
30000	4000		664,153,40		57.4256
30000	8000		599,905,95		56.5947

E: We observe that the blocked code can reduce the number of cache misses significantly when the block size is tuned for the problem and architecture. Overall, the reductions in the number of cache misses are closely correlated with the execution times (one exception has been the largest problem size where N= 30,000). So in general, blocked code outperforms the basic version with proper tuning.