PROBLEM SET 3

Zhiqiang Xie 77892769

Problem 1:

Isomorphic:

- For $q > p/2$, a q-shift is isomorphic to a $(p-q)$-shift on a p-node hypercube. Which can be shown by just flipping all bits of all nodes to get the isomorphic cube.

Induction:

- As for $p = 2$, the base is a 2-processor hypercube, which means $q = 1$ and it's a line actually. It's no doubt to be true.
- From the property above, what we want to prove could be for any $q < p$, the paths in a q-shift in a $2p$-node hypercube are congestion free.
- In the $2p$-node hypercube, all the $p-q$ data paths from processor i to a processor j, $(i < j < p)$ are the same as in a p-node hypercube. Therefore, by the induction hypothesis, do not conflict with each other (since it's satisfied in the p node sub-hypercube).
- The remaining q data paths $(j < i < p)$, which means there are unique single links for every processor i to processor $j + p$, because the circular q-shift on the $2p$-node hypercube is based on q-shift on the p-node hypercube and one more single link (one more bit in the $2p$-node hypercube). Besides, this link is not shared by any others.
- Therefore, the q-shifts are congestion-free in a $2p-$node hypercube by induction. And any q-shift can be performed in $t_s + t_m * m$ time when all messages have size m.

Problem 2

1.

```
#include <stdlib.h>
#include <omp.h>
#define N 100000
#define NUM_THREADS 10
int main(int argc, char *argv[])
{
    int x[N], y[N]; // The public array to read and write the result
    int i, j, tid, rank;
    for (i=0; i<N; i++)
        x[i] = N - i;
    omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel for schedule(dynamic) shared(x,y) private(j,i,tid,rank)
    // Here the private variables are for iterating and comparing
    {
        tid=omp_get_thread_num();
        for (i=tid*N/NUM_THREADS; i<tid*N/NUM_THREADS+N/NUM_THREADS; i++)

        {
```

```
            rank = 0
                for (j=0; j<N; j++)
                    if ( x[i] > x[j] )
                        rank++;
            y[rank] = x[i];
        }
    }
}
```

2.

```
#include <stdlib.h>
#include <omp.h>
#define N 100000
#define NUM_THREADS 10
int main(int argc, char *argv[])
{
    int x[N], y[N]; // The public array to read and write the result
    int count[N] = {0};
    omp_lock_t lock[N];
    int i, j, tid, rank;
    for (i=0; i<N; i++)
        omp_init_lock(&(lock[i]));
        x[i] = N - i;
    omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel for schedule(dynamic) shared(x,y,count,lock) private(j,i,tid,rank)
    // Here the private variables are for iterating and comparing
    {
        tid=omp_get_thread_num();
        for (i=tid*N/NUM_THREADS; i<tid*N/NUM_THREADS+N/NUM_THREADS; i++)
        {
            rank = 0
                for (j=0; j<N; j++)
                    if ( x[i] > x[j] )
                        rank++;
            omp_set_lock(&(lock[rank]));
            count[rank] += 1;
            omp_unset_lock(&(lock[lock]));
            y[rank+count[rank]] = x[i];
        }
    }
}
```

Problem 3

1. The load balance of threads will be poor, since quite different amount of works are allocated to different threads.
2. Better than the previous one, here different parts of matrix are sampled to a thread. It's more load-balanced. The performance will depends on the chunk-size, because it's a triangular case (though the size of 1 is specified).

3. It should be close to the previous one, but somewhat slower due to the overhead of dynamic scheduling. Because here the input is triangular which means it's highly structured, a dynamic scheduling can't take the its advantages.

Problem 4

```
/* Back Substitution */
for (i = 0; i < n; i++) {
    x[i] = b[i]/a[i][i];
    #pragma omp parallel for schedule(static)
    for (j = i+1; j < n; j++) {
        b[j] = b[j] - a[j][i]*x[i];
        a[j,i] = 0;
    }
}
```

Since the back substitution solver has a sequential part that can't be parallelized (solve x_1 depends on solving x_0), our strategy is to assign the iterations of inner loop to different threads to accelerate.

A static scheduling is suitable, since the workloads of iterations are approximately the same.