Computer Science 133, Homework 3

Michael Wu
UID: 404751542

February 11th, 2019

Problem 1

We can use the following algorithm to calculate this.

```
n = 1000;
arr[n];
for(i=0; i<n; i++){
    arr[i] = x;
}
for(step=0; step < ceil(log_2(n)); step++) {
    size = 2^step;
    for(i=0; i<size && i+size<n; i++) { //in parallel
        x[i+size] = x[i]*x[size-1];
    }
}
return x;
```

This produces the following operations.

$$\begin{array}{lll} x^2 = x^1x^1 & x^3 = x^1x^2 & x^5 = x^1x^4 \\ & x^4 = x^2x^2 & x^6 = x^2x^4 \quad \dots \\ & & x^7 = x^3x^4 \\ & & x^8 = x^4x^4 \end{array}$$

Assuming that each inner loop can be done in parallel, this would require $O(\log n)$ steps since the outer loop has $\lceil \log(n) \rceil$ iterations. Since each power of x is calculated using one multiplication, this has a total work time equal to

the work time of the sequential algorithm that continuously does $x^{k+1} = x^1 x^k$. This is optimal since no extra operations are needed, and our algorithm would perform just as well on a single processor.

Problem 2

Since the list might not be sorted, we can use the following algorithm to find the maximum in parallel.

```
for(step=0; step < ciel(log_2(n)); step++) {
    size = 2^(step);
    for(i=0; i+size<n; i+=size*2) { //in parallel
        arr[i] = max(arr[i],arr[i+size]);
    }
}
return arr[0];
```

This has the effect of storing the maximum of every pair of two elements in the even indices on the first iteration. Afterwards, it stores the maximum of every set of four elements in indices that are a multiple of four, then the same with eight elements, sixteen, etc. If we have enough processors to complete the inner loop in parallel, this algorithm will take $O(\log n)$ time. The total number of comparisons is equal to $O(n)$, since the first iteration makes at most $\frac{n}{2}$ comparisons, the next makes $\frac{n}{4}$, and so on. This is a geometric series that at most sums to n. Therefore this program is optimal, since the best sequential algorithm would make $O(n)$ comparisons as it traverses through the list.

Problem 3

We can use the following algorithm to find the rank.

```
rank[n];
rank[n-1]=0;
for(step=0; step < ciel(log_2(n)); step++) {
    size = 2^(step);
    for(i=n-1-size; i>=n-2*size && i>=0; i--) { //in parallel
        rank[i] = size+rank[i+size];
```

```
    }
}
return rank;
```

This algorithm first sets the rank of the last element to zero. Then on the first iteration it sets the $(n-1)$th item to have rank 1. It then sets the next two elements from the end to have rank 2 and 3, then the next four to have ranks 4 through 7, etc. In terms of runtime, this will take $O(\log n)$ time if there are enough processors to complete the inner loop in parallel, as there are $O(\log n)$ iterations of the outer loop. In terms of total work, each rank is calculated only once so it performs $O(n)$ assignments. This is optimal since traversing the list sequentially would also produce the same number of assignments.

Problem 4

The following code computes the histogram by splitting up the domain of the input array and then collecting the result into one array.

```
#include <mpi.h>
#include <stdio.h>

int m = 10;
int N = 100;

int main(int argc, char** argv) {
    int outbuf[m], inbuf[m];
    int rank, size;
    int a[N] = {
        6,1,4,6,4,8,8,8,1,2,
        8,3,1,1,2,5,9,3,4,2,
        8,4,9,2,4,8,3,9,4,2,
        7,6,7,9,5,2,0,9,1,5,
        4,4,9,5,1,7,9,5,1,3,
        1,2,1,0,9,0,9,6,3,7,
        9,4,4,0,2,1,6,7,5,9,
        5,5,2,9,6,3,6,1,1,6,
        7,9,3,2,6,1,3,2,3,1,
        5,4,4,3,8,6,4,1,8,5 };
    int h[m] = {};
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    int blockSize = (N + size - 1)/size;
    for (int i=0; i<m; i++) {
        outbuf[i] = 0;
    }
    for (int i=0; i<blockSize && i<N-rank*blockSize; i++) {
        outbuf[a[rank*blockSize+i]]++;
```

```
    }
    if(rank!=0) {
        MPI_Recv(inbuf, m, MPI_INT, rank-1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        for (int i=0; i<m; i++) {
            outbuf[i] += inbuf[i];
        }
    }
    if(rank!=size-1) {
        MPI_Send(outbuf, m, MPI_INT, rank+1, 0, MPI_COMM_WORLD);
    } else {
        for (int i=0; i<m; i++) {
            h[i] = outbuf[i];
            printf("%d\n", h[i]);
        }
    }
    MPI_Finalize();
}
```

Problem 5

The serial execution time of matrix multiplication takes $O(n^3)$ time. For the parallel implementation, we would divide the board up into blocks of size $\frac{n^2}{k^2}$. Thus our analysis is similar to the checkerboard block decomposition for vector multiplication. We would divide the right matrix by $\frac{n}{k}$ rows, distribute it among the columns of the mesh, process each block in $O(\frac{n^3}{k^2})$ time, then reduce the partial result vectors among the rows. Since each chunk of the right matrix that is broadcast is $\frac{n}{k}$ high and n across, our broadcast takes $O(\frac{n^2}{k})$ time. Then the overall parallel complexity is $O(\frac{n^3}{k^2} + \frac{n^2}{k})$. The overhead is only due to the broadcast, which gives us the following isoefficiency relation and scalability.

$$\begin{aligned} n^3 &\geq Cn^2k \\ n &\geq Ck \\ M(Ck)/k^2 &= C^2 \end{aligned}$$

This indicates that this system is highly scalable. This makes sense because matrix multiplication reuses data by passing over it multiple times in the array, so less memory is needed when compared to the scaling of the number of operations.

Problem 6

`find(int y, int list[])`

```
if length(list)==1
    return start index of list
split list into blocks of size sqrt(length(list))
in parallel compare the first element of each block to y
subblock = highest block whose first element is < y
return find(y, subblock)
```

This algorithm divides the list of size n into $\sqrt{n}$ blocks, then in parallel finds the block that contains y by comparing the first element of each block to y. It continues to do this recursively, similar to binary search, until one element remains. Then it returns the desired index value. This has a runtime of $O(\log \log n)$, since each iteration results in the exponent of the length of the list being halved. For example if there was initially 2^8 elements, the first iteration would reduce this to 2^4, then 2^2, etc. This is not an optimal algorithm, since $O(\sqrt{n} \log \log n)$ comparisons are made. This is more than the number of comparisons for binary search, which is $O(\log n)$.