PROGRAMMING ASSIGNMENT - I

Name: Arun Krishnakumar Rajagopalan
UIN: 623007516
Date: 03/03/2015

1. You should write programs to compute the prefix sums on *eos.tamu.edu*. Actually, you will write two programs, one in shared memory (OpenMP) and one in message passing (MPI).

   a. Your program should be able to use a variable, user-specified, number of processors. For example, the user should be able to run the program using any number of processors between 1 and 16. The number of processors can either be selected using a command line argument or the program can prompt the user.
   b. Your program should be able to sum a variable, user-specified, number of integers. The integers in the input sequence should be randomly generated inside your program. (That is, the user will input *n*, and the program will then randomly generate *n* numbers.)
   c. Your program should print out (either to the standard output or to a file) the prefix sums computed. It should also print out the original input.
   d. Your program should print timing statistics. In particular, you should print out the elapsed time taken to compute the prefix sums, i.e., *do not* include the time taken to generate the input sequence (or at least account for it separately).

Ans -

MPI

Arguments

The program takes three command-line inputs -

1. The number of processors to mpirun
2. The number of elements
3. The number of iterations

Sample run -

```
> mpirun -np 4 ./prefix_sum.x 10 1
```

Compile and Run

```
> cd MPI
> make all;
> qsub prefix_sum.job
```

Output

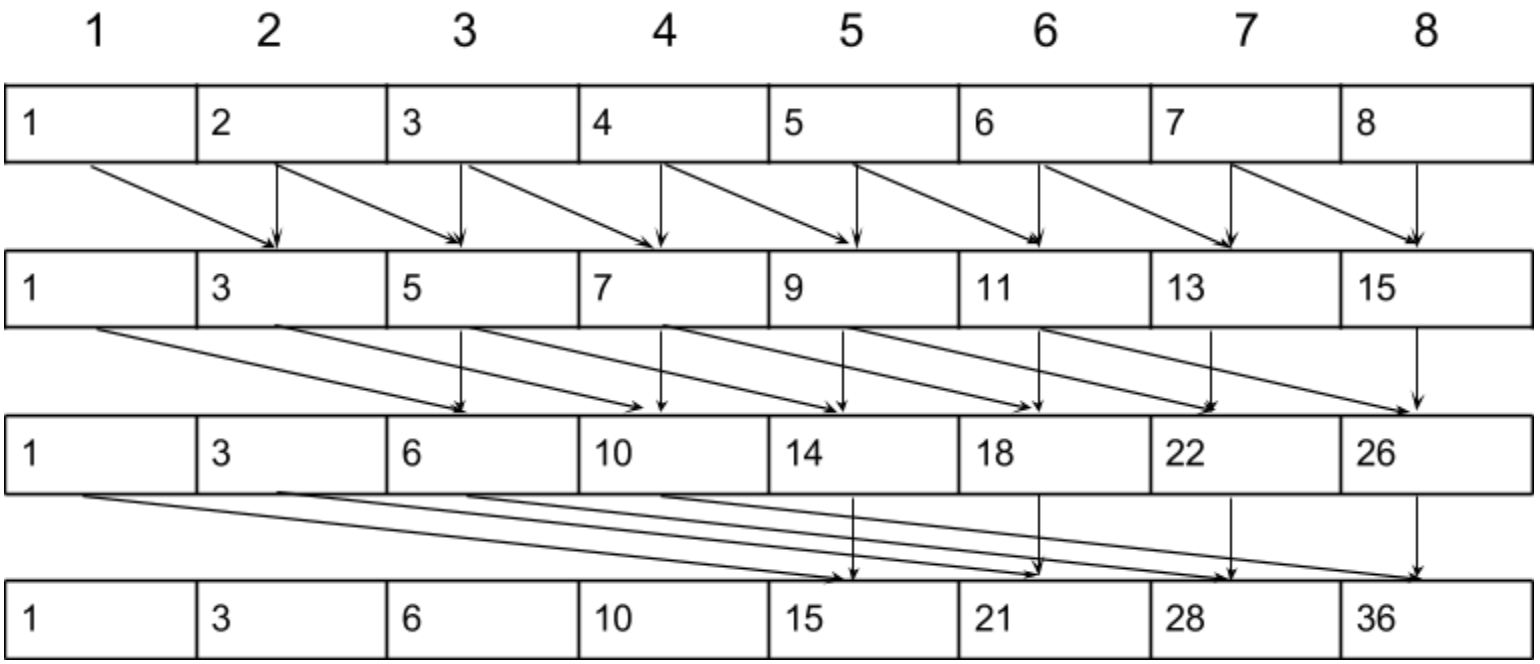
The output from the runs will be in prefix_sum_mpi_32.o#### where #### is the job id. The output will contain the input to the program and the output after running prefix_sum. It will also print out the time taken in generating the input and the output.

```
Executing ./prefix_sum.x: nprocs=4, numints=10, numiterations=1
==============BEGIN INPUT==============================
 102303416 162316515 62340151 93851588 78131811 36040865 182938229 186848643 7494729
63057242
==============END INPUT================================
Input generation time - 3 (usec)

==============BEGIN OUTPUT=============================
 102303416 264619931 326960082 420811670 498943481 534984346 717922575 904771218
912265947 975323189
==============END OUTPUT===============================
Output generation time - 16.7 (usec)
Average output generation time - 16.7 (usec)
```

Algorithm

The MPI solution uses the following prefix sum algorithm. This algorithm (explained in HW 1), runs in **O(log n)** time.

```
for(int i = 0; i < iters - 1 ; i++) {
     //Send only to processors in range
     if(my_id + pow(2, i) < nprocs) {
         MPI_Send(&sum, 1, MPI_INT, my_id + pow(2, i), 111, MPI_COMM_WORLD);
     }

     //Receive only from processors in range
     if(my_id - pow(2,i) >= 0) {
         MPI_Recv(buffer, 1, MPI_INT, my_id - pow(2, i), 111, MPI_COMM_WORLD, &status);
         sum = sum + *buffer;
     }
 }
```



Since the number of elements is larger than the number of processors, we equally divide the elements into each processor. In case it is not exactly divisible,we pad enough 0s to make it divisible. For each of the n/p section, we calculate the prefix-sum sequentially. The last element from each section is then sent to be used in the above algorithm.

```
//Compute the prefix section for each n/p section.
 for (int i = 1; i < numints; ++i) {
     mymemory[i] += mymemory[i-1];
 }

int sum = mymemory[numints - 1]; /* sum of each individual processor */
```

When p is not a power of 2, we calculate the number of iterations to be the rounding up of log p. i.e

```
int iters = pow(2,ceil(log(nprocs)/log(2)));
```

We don't have to worry about non-powers of 2 since the algorithm does not depend on it.

Once the prefix sum for these elements is calculated, we find the difference between the last value and the value returned from the prefix-sum algorithm, and add it sequentially to each element.

```
//Now that we have the prefix sums, find the diff and
//add to the remaining elements to get a true prefix sum
int diff = sum - mymemory[numints - 1];
for(int i = 0; i < numints; i++) {
    mymemory[i] = mymemory[i] + diff;
}
```

Iterations

The program can run multiple times, the iteration count being specified by the user on the commandline. For each iteration, we re-generate the input to avoid cache dependent variations in timing. The average time reported at the end is the average for only the output generation time.

OPENMP

Arguments

The program takes three command-line inputs -

1. The number of threads
2. The number of elements
3. The number of iterations

Sample run -

```
> ./prefix_sum.x 8 16 1
```

Compile and Run

```
> cd OpenMP
> make all;
> qsub prefix_sum.job
```

Output

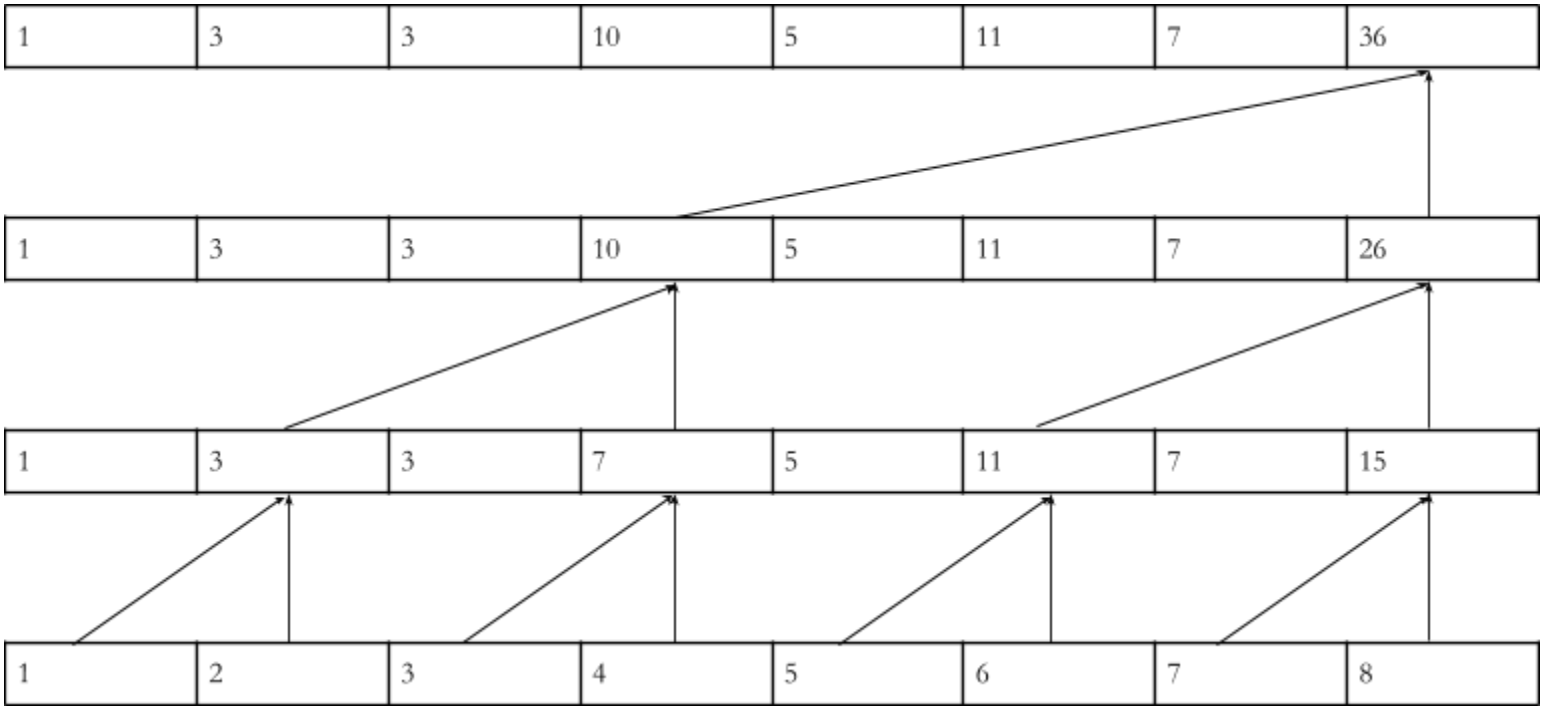
The output from the runs will be in prefix_openmp_8cpu.o#### where #### is the job id. The output will contain the input to the program and the output after running prefix_sum. It will also print out the time taken in generating the input and the output.

```
Executing ./prefix_sum.x: nthreads=8, numints=16, numiterations=1

==============BEGIN INPUT================================
  39491385  98536680  111097412  34468301  60696551  37412135  49457293  30145715  78399406
39225226 93232514 105704659 49655654 115483236 112940909 42674359
==============END INPUT==================================

Input generation time = 36 (usec)

==============BEGIN OUTPUT===============================
  39491385  138028065  249125477  283593778  344290329  381702464  431159757  461305472
539704878 578930104 672162618 777867277 827522931 943006167 1055947076 1098621435
==============END OUTPUT=================================

Output generation time = 65 (usec)
Average output generation time - 65 (usec)
```

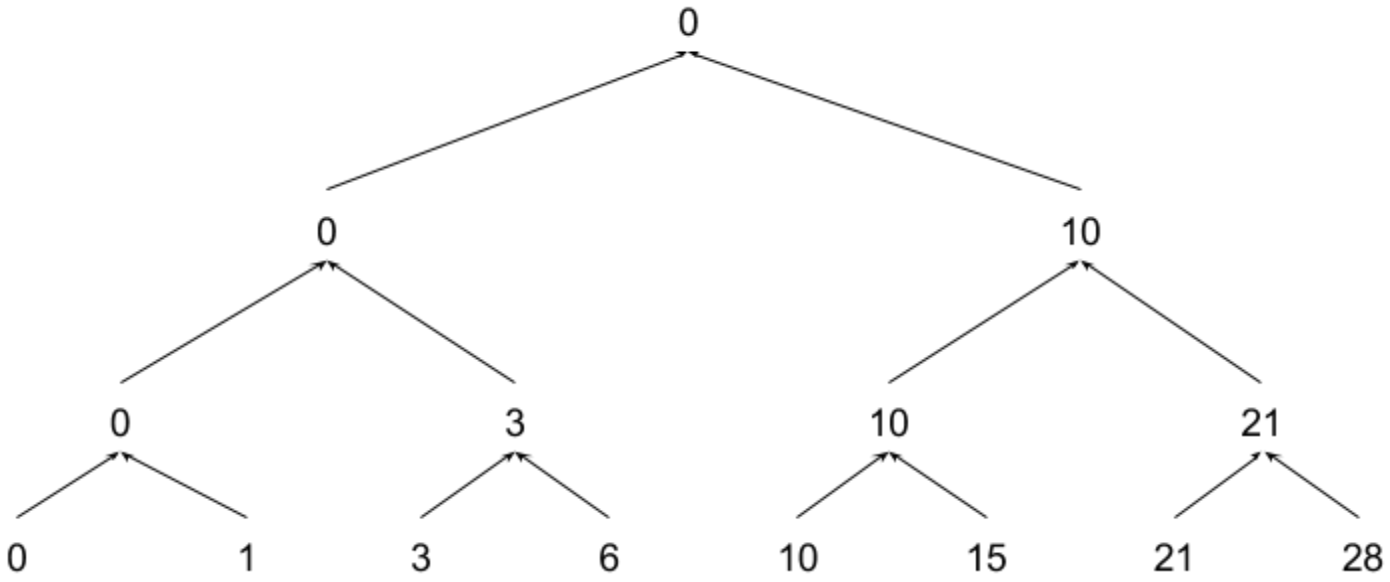
Algorithm

The OpenMP solution uses the following prefix sum algorithm. This algorithm (explained in HW 1), runs in **O(log n)** time.

```
//Starting Sweep-up operation
for(int h = 0; h < floor(log(s_proc)/log(2) + 0.5); h++) {
    #pragma omp parallel for num_threads(s_proc/(int) pow(2, h))
    for(int i = 0; i < (s_proc/(int) pow(2, h+1)); i++) {
        int a           = (((int) pow(2, h+1)) * (i + 1)) -1;
        int b           = a - (int) pow(2,h);
        partial_sums[a] = partial_sums[a] + partial_sums[b];
    }
}
```




```
//Starting Sweep-down operation
 int max = partial_sums[s_proc-1];
 partial_sums[s_proc-1] = 0;

 for(int h = floor(log(s_proc)/log(2) + 0.5) - 1; h > -1; h--) {
     #pragma omp parallel for num_threads(s_proc/(int) pow(2, h))
     for(int i = 0; i < (s_proc/(int) pow(2, h+1)); i++) {
         int a           = (((int) pow(2, h+1)) * (i + 1)) -1;
         int b           = a - (int) pow(2,h);
         int temp        = partial_sums[a];

         partial_sums[a] = partial_sums[a] + partial_sums[b];
         partial_sums[b] = temp;
     }
 }
```



Since the number of elements is larger than the number of processors, we equally divide the elements into each processor. In case it is not exactly divisible,we pad enough 0s to make it divisible. For each of the n/p section, we calculate the prefix-sum sequentially. The last element from each section is then sent to be used in the above algorithm.

```
//Calculate n/p prefix sums
 #pragma omp parallel num_threads(proc)
 {
     int tid               = omp_get_thread_num();
     long long partial_sum = 0;
     int start_id          = tid * numints;
     int end_id            = (tid + 1) * numints;

     for(int i = start_id + 1; i < end_id; ++i) {
         data[i] += data[i-1];
     }

     /* Write the partial result to share memory */
     partial_sums[tid] = data[end_id - 1];
 }
```

When p is not a power of 2, we calculate the number of iterations to be the rounding up of log p. i.e

```
floor(log(s_proc)/log(2)
```

The only modification we make over the standard Balanced tree algorithm for non-powers of two is that we use this rounded up value, and allocate extra memory initialized to 0 (calloc). However, we don't use more than p processors, and leave the remaining elements untouched. This still gives us the right result..

Once the prefix sum for these elements is calculated, we get the value returned from the prefix-sum algorithm, and add it sequentially to each element.

```
 //Now calculate the prefix sum for elements inside each n/p section
 #pragma omp parallel num_threads(proc)
 {
     int tid        = omp_get_thread_num();
     int start_id   = tid * numints;
     int end_id     = (tid + 1) * numints;

     for(int i = start_id; i < end_id; ++i) {
       data[i] += partial_sums[tid];
     }
 }
}
```

Iterations

The program can run multiple times, the iteration count being specified by the user on the commandline. For each iteration, we re-generate the input to avoid cache dependent variations in timing. The average time reported at the end is the average for only the output generation time.

References

1. https://www8.cs.umu.se/kurser/5DV050/VT11/lab1.pdf
2. An Introduction to Data Structures and Algorithms by J.A. Storer