Parallel Computing

Assignment: OpenMP Loops

* REDUCE

Ans: Speedup increases when value of 'n' increases. For static scheduling the speedup seems to increase with increase in number of threads at a higher value of 'n'. but eventually the speedup drops with more number of threads. This seems to be due to creation of multiple threads increases the total work & this seems to be substantial amount of work for such problem. Hence, speedup is less optimal and drops at very high number of threads.

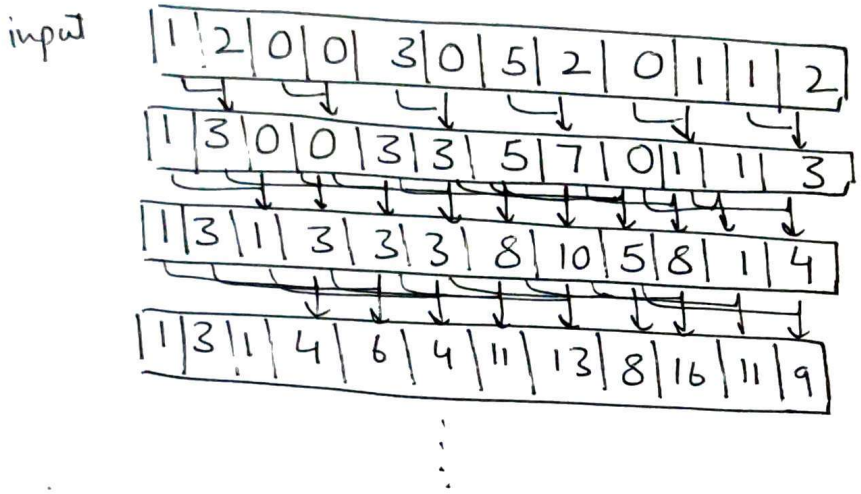
For dynamic scheduling the speedup is insubstantial for low granularity. But as granularity increases the speedup also increases, due to dynamic allocation of large chunk of work to each thread.

* Numerical Integration

Ans: The measurement seems erratic for a low value of 'n', but, seems less erratic as value of 'n' increases. Also, speedup seems erratic for lower 'intensity' and seems to be less erratic as value of 'intensity' increases. As intensity increases the total work being done is overshadowed by the repeated calls made to the function due to the intensities. However, as intensity decreases, the total work being done is more notable.

* Prefix Sum

Ans1 : Prefix Sum can be made parallel by calculating partial results in each iteration and iterating till we get the final result. In each iteration we just ~~copy~~ compute the sum of neighboring cell and store it for re-use.



Here, we assume that in each parallel step all the 'Read' procedures occur before 'Write' procedures.

Algorithm PrefixSum (Pr[0.....n])

```
for a = 1 to log n
    b = 2^n(c-1)
    for i = b to n-1
        S[i] = s[i] + S[i-b]
```

Ans 3: Here, as well at lower values of 'n' the value measurement of speedup seems erratic but becomes less erratic at higher values of 'n'. The speedup increases continuously with increase in value of 'n'. and there is no major drop in speedup overall. This is due to even distribution of the total work amongst each thread. Hence, speedup becomes more optimal.

* Merge Sort

Ans1:

```
MergeSort (A[0 . . . . n-1])
  if n>1
     copy A[0 . . . . ⌊n/2⌋-1] to B[0 . . . . ⌊n/2⌋-1]
     copy A[⌊n/2⌋ . . . . n-1] to C[0 . . . . . ⌈n/2⌉-1]
     # parallel part starts
     MergeSort (B[0 . . . . . ⌊n/2⌋-1])
     MergeSort (C[0 . . . . . ⌈n/2⌉-1])
     Merge (B, C, A)
     # parallel part ends

Merge (B[0 . . . . p-1], C[0 . . . . q-1], A[0 . . . p+q-1])
  i ← 0, j ← 0, k ← 0
  while i<p and j<q do
     if B[i] ≤ C[j]
        A[k] ← B[i] ; i ← i+1
     else A[k] ← C[j] ; j ← j+1
     k ← k+1
  if i=p
     copy C[j . . . . . q-1] to A[k . . . . . p+q-1]
  else copy B[i . . . . p-1] to A[k . . . . . p+q-1]
```

Ans 3: The overall speedup for the code seems to be low. As the value of 'n' increases speedup increases. However, as the number of threads increase, there does not seem to be a vast improvement in speedup. After a particular point there seems to be a plateau in speedup with increase in number of threads.