Assignment 4

Jonathan Gillett

Question 1 – Part A

i

The following pragma is applicable for parallelization because each value of k is independent and block can execute for different values of k in any order.

```
# pragma omp parallel for
for (k=0; k < (int) sqrt(x); k++) {
  a[k] = 2.3 * k;
  if (k<10) b[k] = a[k];
}
```

ii

The following may have issues if the blocks cannot execute at different orders (e.g. data is already partially sorted) and needs to be calculated up until $a[k] < b[k]$ and stop when no longer sorted. Otherwise there is no issue as flag is only being set to 1, there is no risk or a race condition.

```
flag = 0;
# pragma omp parallel for
for (k=0; (k<n) & (!flag); k++) {
  a[k] = 2.3 * k;
  if (a[k]<b[k]) flag = 1;
}
```

iii

The value of k is independent and block can execute for different values of k in any order.

```
# pragma omp parallel for
for (k=0; k<n; k++)
  a[k] = foo(k);
```

iv

The value of k is independent and the block can execute for different values of k in any order.

```
#pragma omp parallel for
for (k=0; k<n; k++) {
  a[k] = foo(k);
  if (a[k]<b[k]) a[k] = b[k];
}
```

v

The value of k is independent and block can execute for different values of k in any order.

```
#pragma omp parallel for
for (k=0; k<n; k++) {
  a[k] = foo(k);
  if (a[k]<b[k]) break;
}
```

vi

The OpenMP reduction operation is used for the *dotp* variable in order to properly perform the reduction summation in parallel.

```
dotp = 0.0;
#pragma omp parallel for reduction(+: dotp)
for (k=0; k<n; k++)
  dotp += a[k] * b[k];
```

vii

The following is not possible to parallelize due to the recurrence. It is not suitable for parallelization as the calculation of each $a[i]$ element is dependent on the results of the previous calculation for $a[i-k]$.

```
for (i=k; i<2*k; i++)
  a[i] = a[i] + a[i-k];
```

viii

The following is not possible to parallelize due to the recurrence. It is not suitable for parallelization as the calculation of each $a[i]$ element is dependent on the results of the previous calculation for $a[i-k]$.

```
for (i=k; i<n; i++)
  a[i] = b * a[i-k];
```

Question 1 – Part B

The code below could have parallelism for the nested loop as well, but it is *highly* impractical as the computation for the first for loop only involves initializing the $rowterm[i]$ and $colterm[i]$ entries. The following code below expresses as much parallelism for the loops below with as little unnecessary overhead as possible, for each nested loop the j index is made private to prevent any race condition errors from the index being overwritten by separate threads.

```
#pragma omp parallel for private(j)
for (i = 0; i < m; i++) {
  rowterm[i] = 0.0;
  for (j = 0; j < p; j++) {
    rowterm[i] += a[i][2*j] * a[i][2*j+1];
  }
}

#pragma omp parallel for private(j)
for (i = 0; i < q; i++) {
  colterm[i] = 0.0;
  for (j = 0; j < p; j++) {
    colterm[i] += b[2*j][i] * b[2*j+1][i];
  }
}
```

Question 3

In order to experimentally determine the values of the equation $t = \lambda + \frac{n}{\beta}$ where n is the number of bytes, λ the *latency* in seconds, and β the *bandwidth* in bytes/second an implementation that sent out a sequence of "ping-pong" timing experiments was implemented in Python and C.

Due to the limitations of access to SHARCNET and issues with the MPI environment the experiments were conducted on an i7 Thinkpad. In order to minimize sources of execution timing errors all processes were terminated except for the ping-pong application being executed. For the implementations in Python and C the results are shown in table 1, with the time in seconds for each implementation. For each size n in bytes tested the "ping-pong" trials were conducted *100* times, between each new trial a random delay was added to minimize the effects of CPU scheduling on execution time.

Table 1: Python & C Timings Results

n (bytes)	Python Timings	C Timings
1000	0.00000651	0.00000246
10000	0.00001191	0.00000524
100000	0.00002132	0.00001525
1000000	0.00011373	0.00010500
10000000	0.00168127	0.00164241
100000000	0.01633679	0.01645497

Given the experimental results calculated in table 1 it was possible to calculate the values of λ and β, however due to the noise of the smaller values of n, the most accurate calculation of λ and β were determined using the two largest values of n, $(10000000, 100000000)$, and representing the problem as a system of linear equations as shown in the following equation.

$$\begin{aligned} t &= \lambda + 0.0000001\beta \\ t &= \lambda + 0.00000001\beta \end{aligned}$$

Using this methodology as a basis, with the larger values of n, the values of λ and β were calculated as shown in table 2. It is interesting to note that unlike in table 1 where there is a significant difference in the timings for Python and C for smaller values of n, this difference diminishes as the size of n increases. Furthermore, it is interesting to note that the distinction between the bandwidth, β, between the Python and C implementation as shown in table 2 is marginal, with the only subtle difference being a reduction in the value of λ for the C implementation.

Table 2: Python & C λ and β Values

Implementation	λ	β
Python	0.0000528789	6.14103×10^9
C	-3.43×10^{-6}	6.07592×10^9