# CS 6230: Homework 2

Christopher Mertin

Due Date: February 7, 2016

Question 1: Vector-Vector Outer Product .......................................................... **[20]**

Given two vectors $\vec{x}, \vec{y} \in \mathbb{R}^{n\times 1}$, we want to compute their *outer product* $\mathbf{A} = \vec{x} \bigotimes \vec{y} \in \mathbb{R}^{n\times n}$, defined by $A_{i,j} = x_i y_j$. Give the work-depth pseudocode for this problem. Derive the work, depth, and parallelism as a function of $n$.

**Solution:** The Pseudocode for calculating the outer product between two vectors can be seen in Algorithm 1. From the work/depth model, the work would be defined as $\mathcal{O}\left(n^2\right)$ as it's performing $n^2$ computations to build $\mathbf{A}$. However, the depth would be $\mathcal{O}(1)$ as there is no dependencies on any of the computations. In other words, with at least $n^2$ CPUs, it would be possible to finish it in a single clock cycle.

**Algorithm 1** Vector-Vector Outer Product($\mathbf{A}, \vec{x}, \vec{y}$)

```
Input: $x, y \in \mathbb{R}^{n\times 1}$,  $\mathbf{A} \in \mathbb{R}^{n\times n}$
Output: $\mathbf{A} \in \mathbb{R}^{n\times n}$
1: for $i = 0$ to $n-1$ do
2:     for $j = 0$ to $n-1$ do
3:         $A[i \times n + j] = x[i] \times y[j]$
4:     end for
5: end for
6: return $\mathbf{A}$
```

The parallelism $P(n)$ is defined as the work over depth, which is $\mathcal{O}\left(n^2\right)$ since the work is $\mathcal{O}\left(n^2\right)$ and the depth is $\mathcal{O}(1)$.

Question 2: Matrix Forward Substitution ........................................................... **[20]**

Let $\mathbf{A}$ be a $n \times n$ lower triangular matrix (*i.e.* $A_{i,j} = 0$ if $j > i$) such that $A_{i,i} \neq 0$ for $1 \leq i \leq n$, and let $\vec{b}$ be a $n$-dimensional vector. Consider the forward-substitution algorithm for solving $\mathbf{A}\vec{x} = \vec{b}$ for $\vec{x}$:

$$x_1 = \frac{1}{A_{1,1}} b_1$$

$$x_i = \frac{1}{A_{i,i}} \left( b_i - \sum_{j=1}^{i-1} A_{i,j} x_j \right) \qquad i = 2, \ldots, n$$

Determine the work and depth of the algorithm. State a PRAM version of this algorithm and derive its time and work complexity as a function of the input size $n$ and the number of processors $p$. *Optional*: Suggest ways to improve the complexity of this algorithm.

**Solution:** In the sequential case, the work/depth model would have work $\mathcal{O}\left(n^2\right)$ and depth being $\mathcal{O}\left(n\right)$ since (a) there are $\mathcal{O}\left(n^2\right)$ computations that need to be made, and (b) there are $\mathcal{O}\left(n\right)$ dependent computations.

Parallelizing this algorithm is difficult due to the multiple dependencies at each row of the matrix, though it can be seen in Algorithm 2.

**Algorithm 2** Parallel Forward Substitution($\mathbf{A}, \vec{x}, \vec{b}$)

```
Input: $\mathbf{A} \in \mathbb{R}^{n \times n}$ and $\vec{x}, \vec{b} \in \mathbb{R}^{n \times 1}$
1: for $i = 1$ to $n - 1$ do
2:     #pragma omp parallel for shared($\mathbf{A}$)
3:     for $j = 0$ to $i - 1$ do
4:         $x_j = b_j - A_{j,i-1} \times x_{i-1}$
5:     end for
6: end for
```

Using this algorithm gives the parallel version of the code which speeds up the computation time. The work is still $\mathcal{O}\left(n^2\right)$ but the depth is $\mathcal{O}\left(n\right)$ now.

The time complexity can be found by figuring out the total number of computations. The for loop in line 3 will have $\mathcal{O}\left(\log\left(\frac{n}{p}\right)\right)$ calculations by making it a reduction, and it will be called $n$ times, making the time complexity $\mathcal{O}\left(n \log\left(\frac{n}{p}\right)\right)$.

Question 3: Matrix-Vector Multiplication .......................................................... **[20]**

Consider the *matrix-vector* multiplication problem $\vec{y} = \mathbf{A}\vec{x}$ on a PRAM machine. For PRAM, we discussed an algorithm that uses row-wise partitioning of $\mathbf{A}$ and $\vec{y}$. This time of partitioning is called *one-dimensional*, because partitions run across one dimension of the matrix. State a PRAM algorithm that uses a two-dimensional partitioning of the matrix. Derive the complexity ($T(n,p)$ and $W(n,p)$) and the speedup of the algorithm. Is your algorithm work efficient, assuming $W_{\text{sequential}}(n) = \mathcal{O}\left(n^2\right)$?

**Solution:** The two-dimensional partitioning is equivalent to giving one element of $\mathbf{A}$ to each processor. The algorithm can be seen in Algorithm 3

**Algorithm 3** Matrix Vector Multiplication($\mathbf{A}$, $\vec{x}$, $\vec{y}$)

```
Input: A ∈ $\mathbb{R}^{n\times n}$ and $\vec{x}, \vec{y} \in \mathbb{R}^{n\times 1}$
1: #pragma omp parallel for num_threads($n$)
2: for $i = 0$ to $n-1$ do
3:    #pragma omp parallel for num_threads($n$)
4:    for $j = 0$ to $n-1$ do
5:       $y_i = y_i + A_{i,j} \times x_j$
6:    end for
7: end for
```

In using Algorithm 3, the inner most loop can take $\mathcal{O}\left(\log\left(\frac{n}{p}\right)\right)$ computations if you make it into a reduction. The outer loop is completely independent on the values in the other rows of $\mathbf{A}$, so it's time complexity can be $\mathcal{O}(1)$ for $n^2$ processors. This makes the overall time complexity as being $\mathcal{O}\left(\log\left(\frac{n}{p}\right)\right)$. The work is $\mathcal{O}\left(n^2\right)$ as you have to perform $n^2$ computations to multiply the matrix and vector.

An algorithm is defined as being *work efficient* if the work complexity is the same for both the serial and parallel case. Therefore, this algorithm is work efficient as you don't need to do more work to parallelize this algorithm.

Question 4: Generic Scan . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **[20]**

Implement an in-place generic scan operation in `C++` using OpenMP (no MPI). The signature of your function should be similar to

```
void genericScan(void *X, size_t n, size_t l, void (*oper)(void *x1, void *x2));
```

You may change the interface to improve performance if you wish. Give an example in which each element in the input array `X` is a three-dimensional double precision vector and the binary operator is the vector addition. Your code should compile using GNU or Intel compilers on x86 platforms. Please report wallclock times for summing (a) 1D and (b) 3D double-precision vectors. The input of the array should be 300M keys long. Write a driver routine called scan that takes one argument, the size of the array to be scanned (initialized randomly). In the write-up, give pseudocode for your algorithm, and report wall clock time/core/$n$ results for $n$ = 1M, 10M, 100M, and 1B elements, using up to one node on Tangent (no MPI). Report weak and strong scaling results.

**Solution:** Table 1 shows the results for running `GenericScan` on 16 cores and $n = 300,000,000$. The test was performed for 10 different instances for both a 1D and 3D vector for each element of the array, and then the average was taken from each of the setups to get the average runtime for each instance. Each value of `X` was created with the use of `std::rand()` function, where the seed was changed each run to a value based on the system time. The raw output for the 300 million case can be found in `output_scan_300M.dat`.

Table 2 shows the results for sequentially running scan on a 3D vector sequentially for each case from 1 million to 1 billion for both sequential and parallel. The raw output for these cases can be found in `output_scan.dat` which has all the results for the cases from 1 million to 1 billion.

From these tables, the weak and strong scaling can be calculated. *Strong scaling* is defined for the case where the problem size stays fixed but the number of processing elements are increased and can be calculated by

$$\text{Strong Scaling} = \frac{t_1}{N \times t_N} \times 100\% \qquad (1)$$

where $t_1$ is the time it takes the process to complete with one processor, and $t_N$ is the time with $N$ processors.

Weak scaling on the other hand looks at the direct scalability of the code by just dividing the sequential time by the parallel time, as seen by

$$\text{Weak Scaling} = \frac{t_1}{t_N} \times 100\% \qquad (2)$$

The results for strong and weak scaling can be found in their respective tables.

**Algorithm 4** GenericScan($A$)

```
Input: $A \in \mathbb{R}^{n\times 1}$
for $i = 0$ to $\log_2(A.size()) - 2$ do
    #pragma omp parallel for
    for $j = 0$ to $A.size() - 1$ by $2^{i+1}$ do
        $A\left[j + 2^{i+1} - 1\right] = A\left[j + 2^i - 1\right] + A\left[j + 2^{i+1} - 1\right]$
    end for
end for{Upsweep}
for $i = \log_2(A.size()) - 1$ to 1 do
    #pragma omp parallel for
    for $j = 0$ to $A.size()$ by $2^i$ do
        $A\left[j + 2^i + i - 1\right] = A\left[j + 2^i - 1\right] + A\left[j + 2^{i+1} - 1\right]$
    end for
end for{Downsweep}
```

Table 1: 1D and 3D Vectors `GenericScan` runtime ($n = 300,000,000$)

| Trial | 1D Sequential | 1D Parallel | 3D Sequential | 3D Parallel |
|---|---|---|---|---|
| 1 | 9.507 | 3.508 | 21.746 | 8.803 |
| 2 | 9.514 | 3.518 | 21.899 | 8.635 |
| 3 | 9.530 | 3.546 | 21.684 | 8.741 |
| 4 | 9.471 | 3.536 | 21.806 | 8.600 |
| 5 | 9.634 | 3.509 | 17.402 | 8.812 |
| 6 | 9.571 | 3.493 | 21.793 | 8.758 |
| 7 | 10.312 | 3.589 | 19.660 | 8.697 |
| 8 | 10.304 | 3.559 | 19.901 | 8.754 |
| 9 | 9.507 | 3.527 | 21.675 | 8.754 |
| 10 | 9.494 | 3.555 | 17.342 | 8.763 |
| Average: | 9.684 | 3.534 | 20.491 | 8.733 |
| Strong Scaling: | 17.127 | | 14.665 | |
| Weak Scaling: | 274.0 | | 234.6 | |

† Given values are in seconds

Table 2: `GenericScan` for Sequential and Parallel

| | Sequential | | | | Parallel | | | |
|---|---|---|---|---|---|---|---|---|
| Trial | $10^6$ | $10^7$ | $10^8$ | $10^9$ | $10^6$ | $10^7$ | $10^8$ | $10^9$ |
| 1 | 0.071 | 0.723 | 4.263 | 81.715 | 0.071 | 0.724 | 2.825 | 29.019 |
| 2 | 0.071 | 0.725 | 7.259 | 82.030 | 0.071 | 0.725 | 2.932 | 29.114 |
| 3 | 0.071 | 0.725 | 7.271 | 82.197 | 0.071 | 0.724 | 2.911 | 29.199 |
| 4 | 0.071 | 0.724 | 4.249 | 81.976 | 0.071 | 0.724 | 2.888 | 28.878 |
| 5 | 0.075 | 0.729 | 7.283 | 81.963 | 0.071 | 0.728 | 2.899 | 29.162 |
| 6 | 0.071 | 0.721 | 7.219 | 81.894 | 0.071 | 0.721 | 2.917 | 29.268 |
| 7 | 0.071 | 0.425 | 7.232 | 81.892 | 0.071 | 0.423 | 2.918 | 29.174 |
| 8 | 0.072 | 0.725 | 4.303 | 81.892 | 0.072 | 0.725 | 2.885 | 29.138 |
| 9 | 0.071 | 0.423 | 7.231 | 56.201 | 0.071 | 0.422 | 2.936 | 27.624 |
| 10 | 0.071 | 0.723 | 7.252 | 82.097 | 0.071 | 0.726 | 2.876 | 28.816 |
| Average: | 0.072 | 0.664 | 6.356 | 79.399 | 0.071 | 0.664 | 2.899 | 28.939 |
| Strong Scaling: | 6.286 | 6.250 | 13.704 | 17.148 | | | | |
| Weak Scaling: | 100.6 | 100.0 | 219.3 | 274.4 | | | | |

† Given values are in seconds

Question 5: Parallel Quicksort .......................................................................... **[20]**

Parallelize your quicksort implementation from *Assignment 1* using OpenMP. In the write-up, give pseudocode for your algorithm, and report wall clock time/core/$n$ results for $n$ = 1M, 10M, 100M, and 1B elements, using up to one node on Tangent (no MPI). Report weak and strong scaling results.

**Solution:** I was unable to parallelize the algorithm that I had implemented in Assinment 1, so I changed it to a different quicksort algorithm [1] which was easier to parallelize. The function call is the same, though it's the sorting part that was changed. This made it easier to parallelize. The pseudocode for this algorithm can be found in Algorithm 5.

In testing the algorithms, I opted to run 10 different trials for each of the algorithms, and also run each array on `std::sort()` as well to see how my sort funciton comapres to the `std::sort()` in `C++`. Table 4 shows the results for the sequential implementation of quicksort, Table 5 is the parallel version of quicksort, and Table 6 details the results from `std::sort()`. The sort operation was performed with a list of integers for each given value of $n$, and the array was formed with the use of `std::rand()` to generate the values. The code was ran on Tangent with a single node and used 24 cores. The raw output from these runs can be found in `output_quicksort.dat` which has all this information. Table 3 reports the scalability of the algorithm for the different values of $n$.

Table 3: Quicksort Scalability

| | $10^6$ | $10^7$ | $10^8$ | $10^9$ |
|---|---|---|---|---|
| Strong Scalability: | 4.999 | 5.090 | 5.398 | 5.733 |
| Weak Scalability: | 119.978 | 122.165 | 129.561 | 137.591 |

**Algorithm 5** Quicksort($A$, $low$, $high$)

```
Input: $A \in \mathbb{R}^{n\times 1}$, $low$, $high$
$lowTemp = low$
$highTemp = high$
$midPoint = A\left[\frac{low+high}{2}\right]$
while $lowTemp \leq highTemp$ do
    while $A[lowTemp] < midPoint$ do
        $lowTemp = lowTemp + 1$
    end while
    while $A[highTemp] > midPoint$ do
        $highTemp = highTemp - 1$
    end while
    if $lowTemp \leq highTemp$ then
        swap($A[lowTemp], A[highTemp]$)
        $lowTemp = lowTemp + 1$
        $highTemp = highTemp - 1$
    end if
end while
#pragma omp parallel sections
#pragma omp section
if $low < highTemp$ then
    Quicksort($A$, $low$, $highTemp$)
end if
#pragma omp section
if $lowTemp < high$ then
    Quicksort($A$, $lowTemp$, $high$)
end if
```

Table 4: Sequential Quicksort

| Trial | $10^6$ | $10^7$ | $10^8$ | $10^9$ |
|---|---|---|---|---|
| 1 | 1.834 | 10.303 | 96.025 | 1074.892 |
| 2 | 0.998 | 9.893 | 96.486 | 991.330 |
| 3 | 1.045 | 10.014 | 86.391 | 1004.014 |
| 4 | 1.025 | 9.697 | 95.041 | 1010.462 |
| 5 | 1.005 | 9.711 | 96.526 | 980.120 |
| 6 | 0.999 | 9.441 | 95.888 | 983.255 |
| 7 | 1.045 | 9.579 | 97.899 | 1025.144 |
| 8 | 1.032 | 9.673 | 103.794 | 976.797 |
| 9 | 1.102 | 9.902 | 97.106 | 1000.652 |
| 10 | 1.024 | 9.511 | 98.356 | 968.181 |
| Average: | 1.111 | 9.772 | 96.351 | 1001.485 |

† Given values are in seconds

Table 5: Parallel Quicksort

| Trial | $10^6$ | $10^7$ | $10^8$ | $10^9$ |
|---|---|---|---|---|
| 1 | 1.396 | 9.412 | 82.749 | 627.391 |
| 2 | 0.780 | 10.210 | 88.215 | 932.276 |
| 3 | 0.741 | 10.119 | 86.078 | 823.983 |
| 4 | 0.812 | 8.434 | 85.709 | 617.227 |
| 5 | 1.001 | 5.841 | 69.367 | 673.040 |
| 6 | 0.866 | 8.230 | 54.480 | 573.038 |
| 7 | 1.049 | 5.428 | 80.597 | 821.517 |
| 8 | 0.909 | 9.820 | 65.599 | 897.548 |
| 9 | 0.712 | 5.649 | 53.588 | 716.353 |
| 10 | 0.997 | 6.846 | 77.292 | 587.314 |
| Average: | 0.926 | 7.999 | 74.367 | 727.869 |

† Given values are in seconds

Table 6: `std::sort()`

| Trial | $10^6$ | $10^7$ | $10^8$ | $10^9$ |
|---|---|---|---|---|
| 1 | 0.043 | 0.461 | 5.892 | 53.614 |
| 2 | 0.051 | 0.817 | 5.043 | 52.540 |
| 3 | 0.043 | 0.488 | 5.798 | 55.786 |
| 4 | 0.051 | 0.526 | 5.825 | 52.779 |
| 5 | 0.051 | 0.517 | 5.025 | 57.002 |
| 6 | 0.051 | 0.719 | 5.876 | 53.381 |
| 7 | 0.049 | 0.464 | 6.019 | 54.771 |
| 8 | 0.051 | 0.562 | 6.035 | 53.615 |
| 9 | 0.051 | 0.474 | 5.806 | 52.790 |
| 10 | 0.051 | 0.478 | 5.640 | 52.570 |
| Average: | 0.049 | 0.551 | 5.696 | 53.885 |

† Given values are in seconds

## References

[1] S. Qin, Florida Institute of Technology, `http://cs.fit.edu/~pkc/classes/writing/hw15/song.pdf`