CS 4402 Assignment 1

Riley Gavigan

February 2024

1 Problem 1

1.1 Question 1

```
// Pseudocode
void inverse(vector<vector<double>> &A, int B,
    int sRow, int sCol, int eRow, int eCol)
{
    // Base Case: Forward substitution
    if (eRow - sRow <= B) {
        for (int i = sRow; i < eRow; i++) {
            for (int j = sCol; j <= i; j++) {
                if (i == j) {
                    A[i][j] = 1 / A[i][j];
                }
                else {
                    double sum = 0;
                    for (int k = j; k < i; k++) {
                        sum += A[i][k] * A[k][j];
                    }
                    A[i][j] = -sum / A[i][i];
                }
            }
        }
        return;
    }
    int mid = (eRow - sRow) / 2;

    // A1 Inverse Multithreading
    cilk_spawn inverse(A, B, sRow, sCol, mid, mid);

    // A3 Inverse Multithreading
    cilk_spawn inverse(A, B, sRow + mid, sCol + mid, eRow, eCol);
    cilk_sync;
```

```
    // A2 * A1^-1
    multiply(M, M, sRow + mid, sCol, sRow, sCol, mid);

    // A3^-1 Temp
    vector<vector<double>> A3(mid, vector<double>(mid, 0));

    // Temp for A3^-1
    for (int i = 0; i < mid; i++) {
        for (int j = 0; j < mid; j++) {
            A3[i][j] = A[sRow + mid + i][sCol + mid + j];
        }
    }

    // -A3^-1 * (A_2 * A_1^-1)
    multiply(A3, A, 0, 0, sRow + mid, sCol, mid);
    for (int i = 0; i < mid; i++) {
        for (int j = 0; j < mid; j++) {
            A[sRow + mid + i][sCol + j] = -M3[i][j];
        }
    }
}
```

1.2 Question 2

Work:

- Multiplication - Work is $\theta(n^3)$. Utilized Naive iterative matrix multiplication, this is directly from the slides.
- Base Case Forward Substitution - Work is $\theta(n^2)$. Comes from the double for loops.
- Processing - Work is $\theta(2n^2)$. From double for-loops.
- Recursion - Work is $\theta(n^2)$ from recursive calls.
- Total - $\theta(n^5)$

Critical Path:

- Multiplication - Span is $\theta(n)$. Utilized Naive iterative matrix multiplication, this is directly from the slides.
- Recursion - Span is $\theta(logn)$. Fork-join model introduces branches and we traverse down the worst case.
- Total - $\theta(nlogn)$

1.3 Question 3

Performance Test Results:

Table 1: Parallel vs Serial Performance

n	B	Parallel Time	Serial Time	Speedup
4	32	0.00013	3.9e-07	0.003
4	64	1.4e-05	1.4e-07	0.01
4	128	1.1e-06	1.2e-07	0.11
8	32	4.4e-05	4e-07	0.0091
8	64	2.9e-05	2.9e-07	0.01
8	128	2.2e-05	3.1e-07	0.014
16	32	6.1e-05	1.4e-06	0.022
16	64	3.8e-05	1.2e-06	0.032
16	128	2.5e-05	6.3e-07	0.025
32	32	0.00015	5.9e-06	0.04
32	64	4e-05	4.7e-06	0.12
32	128	1.7e-05	5.8e-06	0.34
64	32	0.00012	6.5e-05	0.55
64	64	0.00023	4.3e-05	0.19
64	128	4.7e-05	4.5e-05	0.94
128	32	0.00075	0.00049	0.65
128	64	0.00063	0.00023	0.36
128	128	0.00019	0.00013	0.67
256	32	0.0031	0.0018	0.58
256	64	0.0023	0.0018	0.75
256	128	0.0012	0.0025	2
512	32	0.0052	0.022	4.1
512	64	0.0041	0.021	5
512	128	0.0036	0.021	5.7
1024	32	0.02	0.19	9.1
1024	64	0.019	0.19	9.8
1024	128	0.021	0.19	8.7
2048	32	0.11	1.9	16
2048	64	0.11	1.9	17
2048	128	0.12	1.9	17

1.4 Question 4

Serial Performance Based on B:

The best value for B in the serial elision is irrelevant. Results are very similar among all values.

Parallel Performance Based on B:

The best value for B in the parallel example is 32. Results are very similar among all values, however. By looking at n = 4096, it was more conclusive that

the best value is 32: B = 32 = 0.9s, B = 64 = 0.95s, B = 128 = 0.93s.

1.5 Question 5

Processor Information:

- Name: Intel Core i5-13600K
- Cores: 14 (6 P-Cores, 8 E-Cores
- Threads: 20
- Hyperthreading: Enabled
- Speed: 3.5GHz

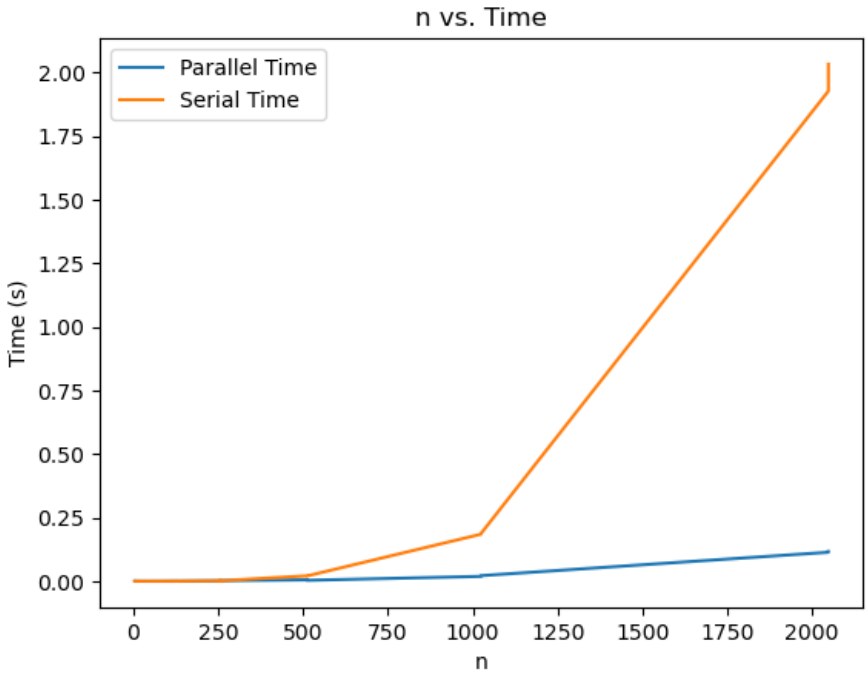

Figure 1: Parallel vs Serial Performance

1.6 Question 6

Serial elision cache complexity according to the ideal cache model, using float values instead of double values for simplicity.
Basic Information:

- Cache Information: 32KB, Direct Mapped, 64 Byte Lines, Fits $2^9 * 2^4 = 2^{13}$ floats. Access = 1 cycle, Memory Access = 100 cycles.
- Number of Values In Cache: $32768/4 = 8192$ float values can be in the cache at once.
- Number of Lines in Cache: $8192/n$ lines of a matrix M can be in the cache at any given time.
- I am using n = 64 and B = 4 for formula construction, to demonstrate examples as I go.

Multiplication Algorithm:

- Temp: Allocation is $n^2 = 64^2 = 4096$ float values. $n^2/64$ cold misses.
- Sum: 1 float value, cold miss.
- M1: $n^2 = 64^2 = 4096$ float value array. Accessed in row-major order.
- M2: $n^2 = 64^2 = 4096$ float value array. Accessed in column-major order.
- Innermost Loop (k): $n = 64$ iterations. M1 only has 1 cold miss per invocation. M2 has 64 cold misses per first invocation. Formula: $100n + n + 99$.
- Secondmost Loop (j): $n = 64$ iterations. Sum has 1 cold miss and $n-1 = 63$ hits. M1 has 1 cold miss and $n(n-1) = 64*63$ hits. M2 has n^2 misses. Formula: $100n^2 + n^2 + 1$.
- Outermost Loop (i): $n = 64$ iterations. Formula: $100n^3 + n^3 + n$.
- Copying Loop: $n = 64$ iterations, double-nested. Access to temp and M1 in row-major order. $8192/2n$ lines of both can be in cache at any given time. Formula: $100n + n - 1$.
- Total: $100n^3 + n^3 + n^2/64 + 102n = n^3$. This is expected, the algorithm I used is naive MM which is pretty oblivious to blocking.

Forward Substitution Base Case:

- If Case: A miss if $i \ mod \ 16 = 0$, hit otherwise. In other words, hits 15/16 times and misses 1/16 times. There are n(n-1) iterations of the loops. The if branch is traversed n times in these. Formula: $100n/16 + 15n/16$.
- Else Case: Traversed $n(n-1) - n = n^2$ times. Sum is 1 cold miss each time. Inner loop accesses M in both row-major and column-major order simultaneously. Formula: $n^2(100 + Innermost\ Loop + Finalize)$.
- Innermost Loop (k): M row-major access has $n(n-1)(n-2)$ total accesses, with 1 miss every $8192/2n$ iterations. M column-major access has $n(n-1)(n-2)$ total accesses, with all misses.

- Finalize: M has row-major access with $n(n-1)$ total accesses, 1 miss every $8192/2n$ iterations. M has diagonal access with 1 miss every outer-loop iteration, total n misses.
- Total: $(n(n-1)(n-2))(8192/2n+100(1-8192/2n))+n(n-1)(8192/2n+100(1-8192/2n))+100n/16+15n/16 = n^3$.

Additional Inverse Matrix Cache Use:

- M3 initialized, and M and M3 accessed in row-major order together. Same thing is done again, making the multiplicand 4 instead of 2. Formula: $4(15n/16+100n/16)$.

Total Cache Complexity:

- Multiplication: $2n^3$ total per recursive call.
- Forward Substitution: n^3 total.
- Additional Cache Use: $4(15n/16+100n/16)$ per recursive call.
- Total: $n^3+n(2n^3+4(15n/16+100n/16)) = n^4$.
- With a better multiplication algorithm that is cache-aware, this could go down to $n^4/(L\sqrt{Z})$

2 Problem 2

2.1 Question 1

```
// Pseudocode
struct Point {
    double x, y;
};

// Distance between 2 points
double distance(Point pi, Point pj) {
    return sqrt(pow(pj.x - pi.x, 2) + pow(pj.y - pi.y, 2));
}

// Sorting by y-coordinate
vector<Point> sortY(vector<Point> points) {
    sort(points.begin(), points.end(), [](Point a, Point b) {
        return a.y < b.y;
    });
    return points;
}
```

```
vector<Point> closestPair(vector<Point> points, int start, int end) {
    // Base Case: If there are only two points, return them
    vector<Point> pair;
    if (end - start == 2) {
        pair.push_back(points[start]);
        pair.push_back(points[end]);
        return pair;
    }

    // Find value x to divide in half
    int half = (start + end) / 2;
    double x = (points[half - 1].x + points[half].x) / 2;

    // Recursively find closest pair in L/R
    vector<Point> L = closestPair(points, start, half);
    vector<Point> R = closestPair(points, half, end);
    double dL = distance(L[0], L[1]);
    double dR = distance(R[0], R[1]);

    // Check if L points or R points are closer
    double d;
    if (dL < dR) {
        pair = L;
        d = dL;
    } else {
        pair = R;
        d = dR;
    }

    // Discard all points with xi < x - d or xi > x + d
    vector<Point> S;
    for (int i = 0; i < points.size(); i++) {
        if (points[i].x >= x - d && points[i].x <= x + d) {
            S.push_back(points[i]);
        }
    }

    // Sort S by y-coordinate
    S = sortY(S);

    // Go through S and for each point, compare to next 6
    for (int i = 0; i < S.size(); i++) {
        for (int j = i + 1; j < i + 7 && j < S.size(); j++) {
            if (distance(S[i], S[j]) < d) {
                d = distance(S[i], S[j]);
                pair[0] = S[i];
```

```
                pair[1] = S[j];
            }
        }
    }
    return pair;
}
```

Work: $W(n) = 2W(n/2) + O(nlog(n))$

- The two recursive calls to closestPair() explain the $2W(n/2)$ portion of the recurrence, as we are branching with two recursive calls each time.
- The base case is $O(1)$, as it simply returns the pair.
- The distance calculations are $O(1)$, as calculating distance between 2 points is a constant operation between 2 points.
- Sorting the remainder of the points in S after discarding points is $O(nlogn)$, the complexity of comparison-based sorting algorithms like this one, utilizing std::sort from the C++ STL.
- Going through S is $O(6n)$.
- Total: The recurrence $W(n) = 2W(n/2) + O(nlog(n))$ properly describes this algorithm because of the sorting portion making up the complexity of the added O term.

Recurrence Master Theorem:
$T(n) = 2T(n/2) + O(nlogn) \in O(nlog^2n)$, where $a = 2, b = 2, k = 1, i = 1$

2.2 Question 2

```
...
// Recursively find closest pair in L/R
vector<Point> L = cilk_spawn closestPair(points, start, half);
vector<Point> R = cilk_spawn closestPair(points, half, end);
cilk_sync;
...
```

$T_1(n) = 2T(n/2) + O(nlogn) = O(nlog^2n)$
$T_\infty = T_\infty(n/2) + O(nlogn) = O(nlogn)$
Therefore, parallelism is $T_1(n)/T_\infty(n) = O(logn)$. This is puny parallelism and a new solution is needed to improve parallelism in question 3.

2.3 Question 3

Utilizing P_MergeSort to sort the points can improve parallelism to $O(n/log(n))$.

```
// Pseudocode
int binarysearch(int *data, int low, int high, int key)
...
void parallel_submerge(int *c, int *data, int lowx,
    int highx, int lowy, int highy, int sp)
...
int parallel_merge(int* c,int* a, int lowx,
    int highx, int lowy, int highy, int sp)
...
void serial_mergesort(int* data, int istart, int iend)
...
void parallel_mergesort(int* c,int* data, int istart,
    int iend, int BASE)
...
vector<Point> parallel(vector<Point> points, int start, int end) {
    ...
    // Sort by y-coordinate using parallel mergesort
    int* data = (int *)malloc(sizeof(int)*S.size());
    for (int i = 0; i < S.size(); i++) {
        data[i] = S[i].y;
    }
    int* c = (int *)malloc(sizeof(int)*S.size());
    int BASE = 32;
    parallel_mergesort(c,data, 0, S.size()-1, BASE);
    for (int i = 0; i < S.size(); i++) {
        S[i].y = data[i];
    }
    free(data);
    free(c);
    ...
}
```

2.4 Question 4

Below is a table and corresponding graph showcasing the runtime of the serial implementation, first parallel implementation, and improved parallel implementation. System specs are the same as in problem 1.

n	Serial Time (s)	Parallel Time (s)	Improved Parallel Time (s)	Speedup
16	0.000004	0.000167	0.000022	0
32	0.000007	0.000017	0.000021	0
64	0.000021	0.000039	0.000013	1
128	0.000060	0.000088	0.000012	4
256	0.000198	0.000177	0.000013	15
512	0.000778	0.002156	0.000054	14
1024	0.003067	0.002252	0.000030	103
2048	0.013281	0.001324	0.000039	343
4096	0.059069	0.005613	0.000077	767
8192	0.255083	0.022323	0.000237	1076
16384	1.099937	0.078809	0.000214	5143
32768	4.645010	0.333423	0.000564	8230
65536	20.101736	1.458940	0.001096	18333

Table 2: Performance Testing for Closest Pair Problem

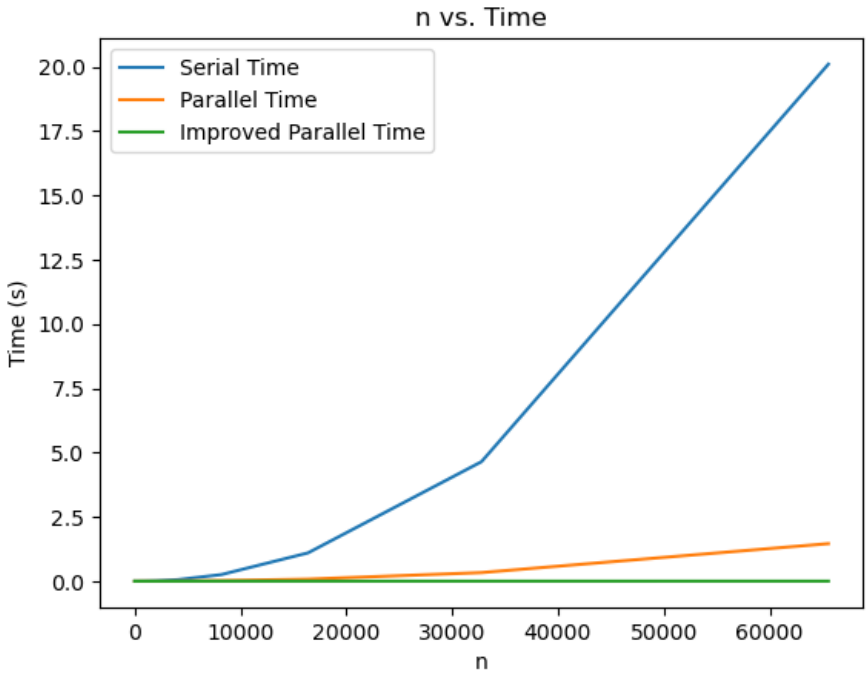

Figure 2: Serial vs Parallel vs Improved Parallel Performance