```
#include <iostream>
#include <vector>
#include <random>
#include <chrono>
#include <limits>
#include <iomanip>
#include <omp.h>
using namespace std;

class ParallelReduction {
private:
    vector<int> data;
    int size;

    vector<int> generateRandomData(int size, int min, int max) {
        vector<int> result(size);
        random_device rd;
        mt19937 gen(rd());
        uniform_int_distribution<> distrib(min, max);

        for (int i = 0; i < size; i++) {
            result[i] = distrib(gen);
        }

        return result;
    }

    template<typename Operation>
    double measureExecutionTime(Operation op, const string& name) {
        auto start = chrono::high_resolution_clock::now();
        auto result = op();
        auto end = chrono::high_resolution_clock::now();

        chrono::duration<double, milli> duration = end - start;
        cout << name << " result: " << result << ", Time: " << fixed <<
        setprecision(3) << duration.count() << " ms" << endl;

        return duration.count();
    }

public:
    ParallelReduction(int dataSize, int minVal, int maxVal) : size(dataSize) {
        data = generateRandomData(size, minVal, maxVal);
    }
```

```
int sequentialMin() {
    int minVal = numeric_limits<int>::max();
    for (int i = 0; i < size; i++) {
        if (data[i] < minVal) {
            minVal = data[i];
        }
    }
    return minVal;
}

int parallelMin() {
    int minVal = numeric_limits<int>::max();
    #pragma omp parallel reduction(min:minVal)
    {
        #pragma omp for
        for (int i = 0; i < size; i++) {
            if (data[i] < minVal) {
                minVal = data[i];
            }
        }
    }
    return minVal;
}

int sequentialMax() {
    int maxVal = numeric_limits<int>::min();
    for (int i = 0; i < size; i++) {
        if (data[i] > maxVal) {
            maxVal = data[i];
        }
    }
    return maxVal;
}

int parallelMax() {
    int maxVal = numeric_limits<int>::min();
    #pragma omp parallel reduction(max:maxVal)
    {
        #pragma omp for
        for (int i = 0; i < size; i++) {
            if (data[i] > maxVal) {
                maxVal = data[i];
            }
        }
    }
```

```
    return maxVal;
}

long long sequentialSum() {
    long long sum = 0;
    for (int i = 0; i < size; i++) {
        sum += data[i];
    }
    return sum;
}

long long parallelSum() {
    long long sum = 0;
    #pragma omp parallel reduction(+:sum)
    {
        #pragma omp for
        for (int i = 0; i < size; i++) {
            sum += data[i];
        }
    }
    return sum;
}

double sequentialAverage() {
    long long sum = sequentialSum();
    return static_cast<double>(sum) / size;
}

double parallelAverage() {
    long long sum = parallelSum();
    return static_cast<double>(sum) / size;
}

void runBenchmark() {
    int numThreads;
    #pragma omp parallel
    {
        #pragma omp master
        numThreads = omp_get_num_threads();
    }

    cout << "Array size: " << size << ", Number of threads: " << numThreads
    << endl;
    cout << "------------------------------------------------------------"
```

```
        << endl;

        // Measure sequential operations
        double seqMinTime = measureExecutionTime([this]() { return this-
        >sequentialMin(); }, "Sequential Min");
        double seqMaxTime = measureExecutionTime([this]() { return this-
        >sequentialMax(); }, "Sequential Max");
        double seqSumTime = measureExecutionTime([this]() { return this-
        >sequentialSum(); }, "Sequential Sum");
        double seqAvgTime = measureExecutionTime([this]() { return this-
        >sequentialAverage(); }, "Sequential Average");

        cout << "-------------------------------------------------------------"
        << endl;

        // Measure parallel operations
        double parMinTime = measureExecutionTime([this]() { return this-
        >parallelMin(); }, "Parallel Min");
        double parMaxTime = measureExecutionTime([this]() { return this-
        >parallelMax(); }, "Parallel Max");
        double parSumTime = measureExecutionTime([this]() { return this-
        >parallelSum(); }, "Parallel Sum");
        double parAvgTime = measureExecutionTime([this]() { return this-
        >parallelAverage(); }, "Parallel Average");

        cout << "-------------------------------------------------------------"
        << endl;

        // Calculate speedups
        cout << "Speedups:" << endl;
        cout << "Min: " << fixed << setprecision(2) << (seqMinTime /
        parMinTime) << "x" << endl;
        cout << "Max: " << fixed << setprecision(2) << (seqMaxTime /
        parMaxTime) << "x" << endl;
        cout << "Sum: " << fixed << setprecision(2) << (seqSumTime /
        parSumTime) << "x" << endl;
        cout << "Average: " << fixed << setprecision(2) << (seqAvgTime /
        parAvgTime) << "x" << endl;
    }
};
```

```
int main() {
    const int dataSize = 1e7; // 100 million elements
    const int minValue = -10000;
    const int maxValue = 10000;

    // Set number of threads (optional, can also be set with environment
       variable)
    // omp_set_num_threads(4);

    ParallelReduction reduction(dataSize, minValue, maxValue);
    reduction.runBenchmark();

    return 0;
}
```

Output:

```
Array size: 10000000, Number of threads: 4
------------------------------------------------------------
Sequential Min result: -10000, Time: 26.820 ms
Sequential Max result: 10000, Time: 25.295 ms
Sequential Sum result: 24786793, Time: 25.013 ms
Sequential Average result: 2.479, Time: 25.156 ms
------------------------------------------------------------
Parallel Min result: -10000, Time: 10.991 ms
Parallel Max result: 10000, Time: 10.891 ms
Parallel Sum result: 24786793, Time: 11.327 ms
Parallel Average result: 2.479, Time: 11.378 ms
------------------------------------------------------------
Speedups:
Min: 2.44x
Max: 2.32x
Sum: 2.21x
Average: 2.21x
```