Kyle Salitrik
CMPSC 450
Homework 3 Report

Algorithms Implemented

Three different approaches were used to find the sum of the arrays. The first approach is the binary tree PRAM summation. For this approach, the array is divided amongst the number of threads, where each thread computes a set number of two element sums and then places them into an array of size N/2. The function then recurses, passing down this array of N/2 to be summed. Although we know the theoretical parallel time complexity to be $O(log_2 N)$, looking at the serial complexity, the following can be observed:

$$O(\text{binary\_sum}) = O(N + \frac{N}{2} + \frac{N}{4} + \cdots + \frac{N}{2^i} + \cdots + 2) = O(2N - 1) = O(N)$$

The second approach was to divide the array across the number of processors and sum all elements assigned to that processor, place that partial sum into a scratch array with the size equal to the number of threads, and then sum all of the scratch array elements afterwards. This divide-and-conquer function was named parallel_sum. When the number of threads is equal to 1, the order

$$O(\text{parallel\_sum}) = O(N); p = 1$$
$$O(\text{parallel\_sum}) = O(N/p)$$

The third approach is, in theory, identical to the parallel sum approach. By implementing the parallel for reduction method of OpenMP, the array should be split into equal segments per thread, each section added into a temporary sum, and then the actual sum computed by summing all of these partial sums. The time complexity should be the same as that of the previous approach. In practice, however, we will observe that the extra overhead in manually managing the divide-and-conquer method causes the time to be slightly higher than that of the parallel reduction method.

All three approaches were run using 1, 2, 4, 8, 16, 20, and 24 threads. One preliminary result that was interesting is that, even for the divide-and-conquer methods, as the number of threads diverged from powers of two performance was reduced. Using 16 threads had the best performance for all algorithms.

Algorithm Performance Comparison

For all of the graphs in this section, time is on the Y axis, the number of array elements is on the X axis and the number of threads is indicated by the legend. This first graph plots the performance of the binary tree summation. For all numbers of threads, the amount of time spiked at $N = 8,200,000$. As stated before, each power of two increases the performance (which is to be expected), however using 20 threads has a lower performance than even 8 threads and using 24 threads is only slightly better than 8 threads.

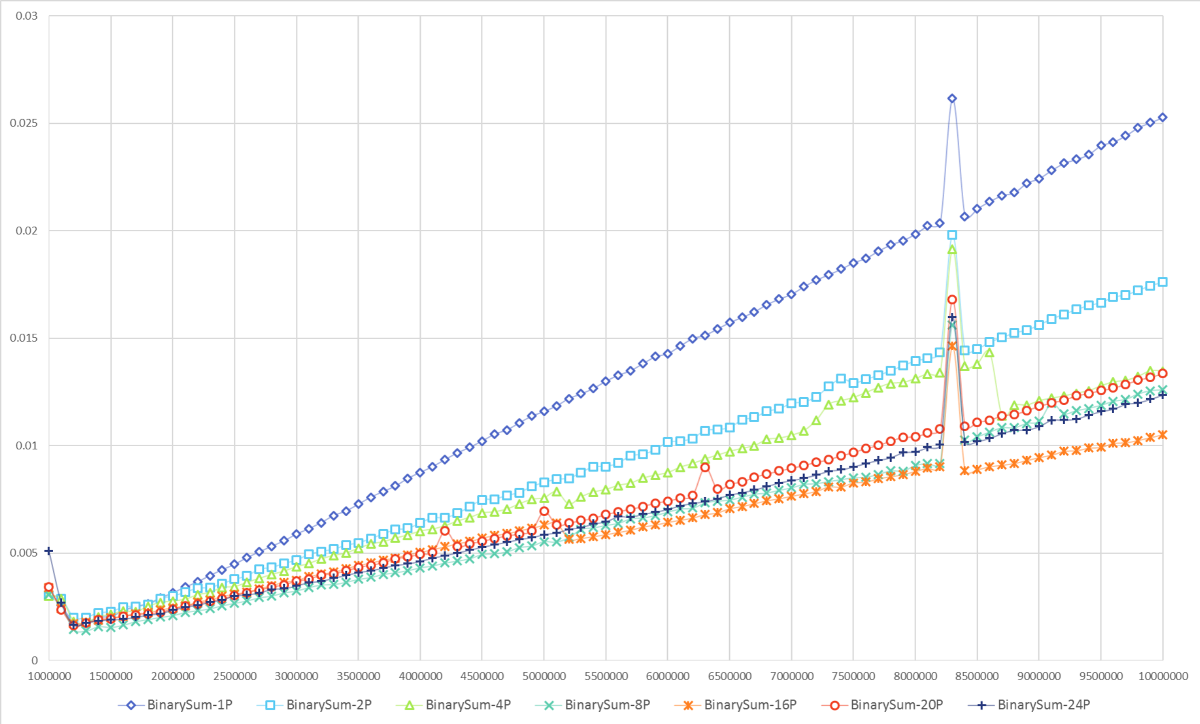

Figure 1: Binary Tree Sum Performance

Second up is the parallel sum method. Even in the worst case (1 thread) it is still more than twice as fast as the binary sum method. This follows our realistic expectations of the asymptotic complexity. The time complexity of the serial binary sum is $O(2N-1)$ while the parallel sum method only requires $O(N)$ time. As the number of threads increases, this difference becomes more and more drastic. The same performance spike around 8.2 million is present here as well. It is also worth noting that there is odd behavior when using 4 threads.

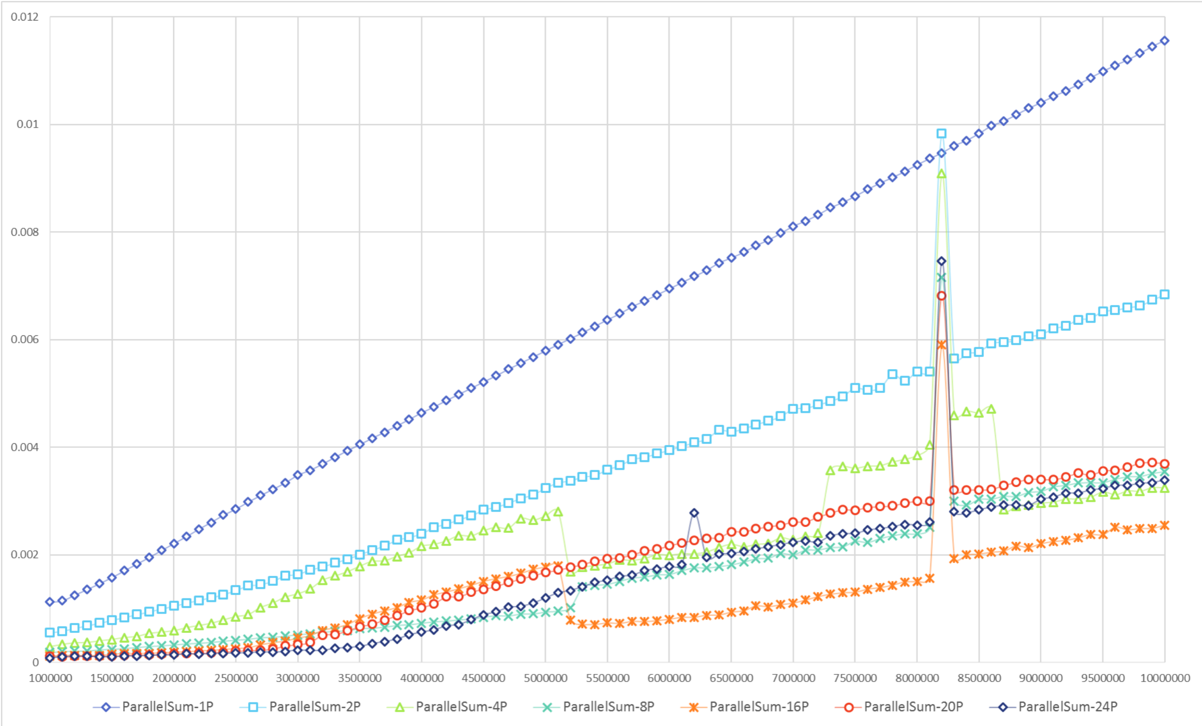

Figure 2: divide-and-conquer Sum Performance

When observing the fast sum method, we see very similar values of time for threads and, as with the previous method, there is odd stepping in performance when using 4 threads. It is worth noting, however, that while in the previous two methods using 20 threads was worse than 8 threads, when using the reduction method, the performance is almost equivalent. The major feature that stands out is that using this method does not cause the peak in performance around 8.2 million elements.

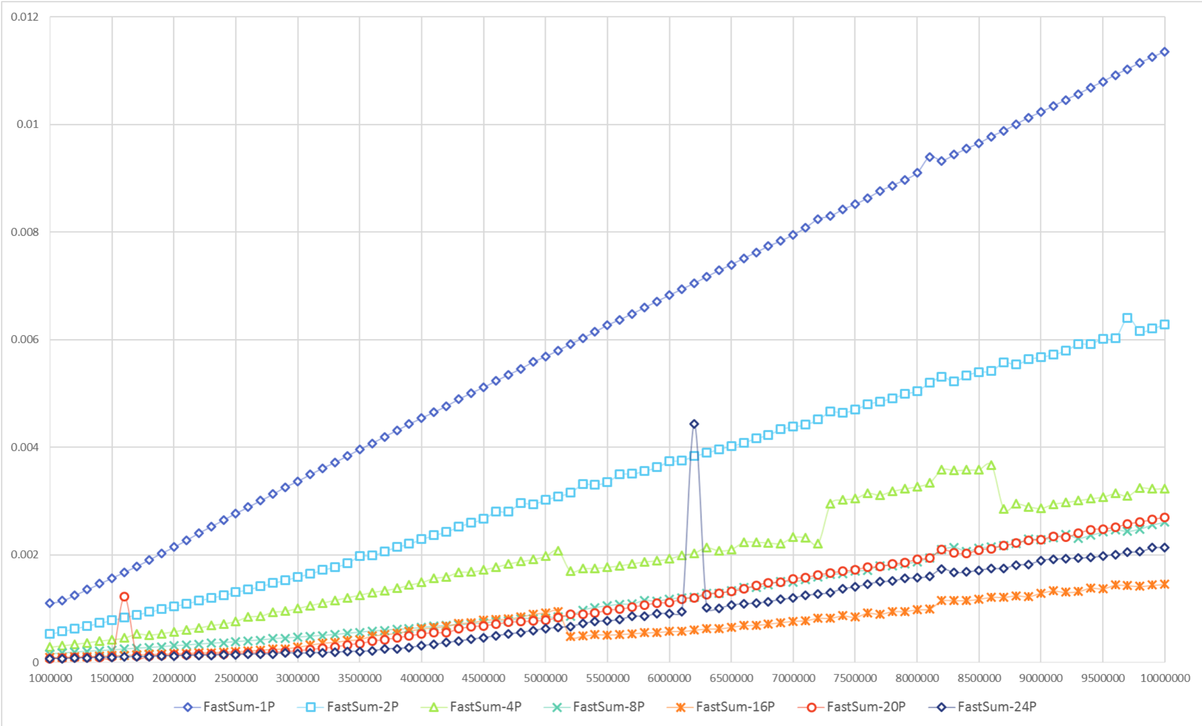

Figure 3: Reduction Sum Performance

As a side-by-side comparison of the two methods, the fast sum and parallel sum performances are plotted below. The fast sum values are marked by the dashed line. As expected, by manually managing the shared variables and scratch space used by the parallel method, the performance is slightly worse on all accounts.

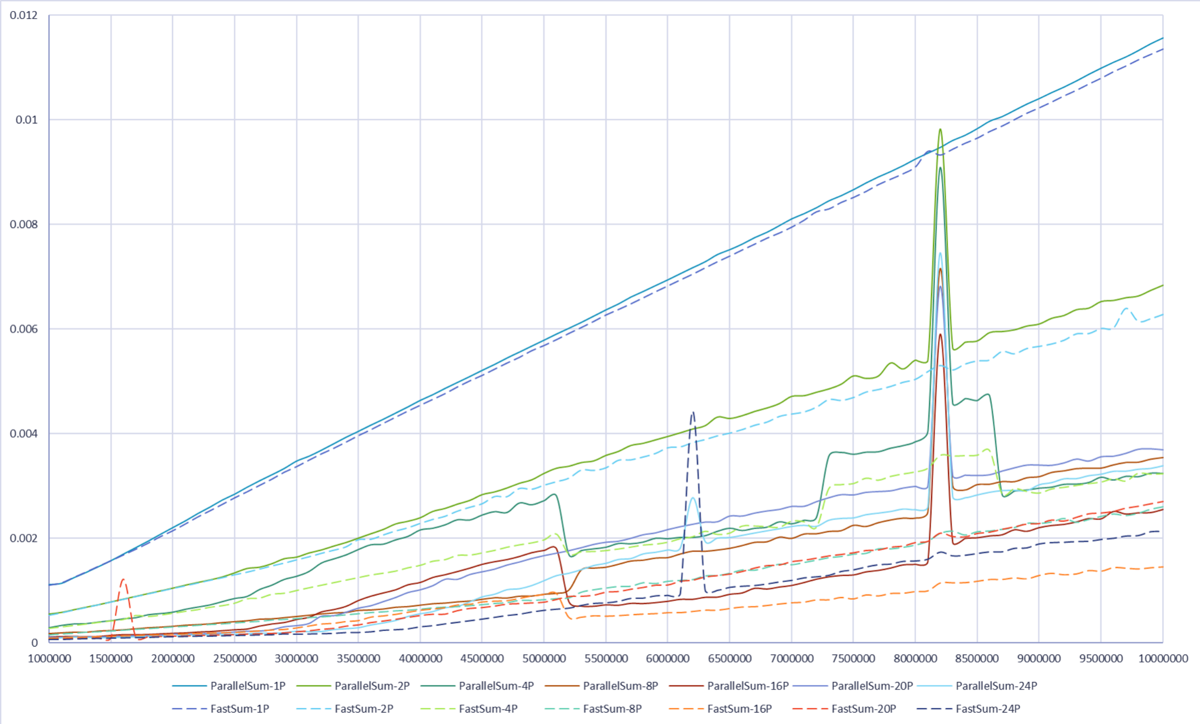

Figure 4: Parallel vs Fast Sum Performance

This graph provides a simple comparison between the serial performance of all three methods. Again, as assumed, the binary sum takes nearly twice the time that the parallel and fast methods do at large N. The management of the scratch space used also takes a toll on $N < 1,300,000$, as can be observed by the significantly higher runtime for these smaller N values in the binary tree sum.

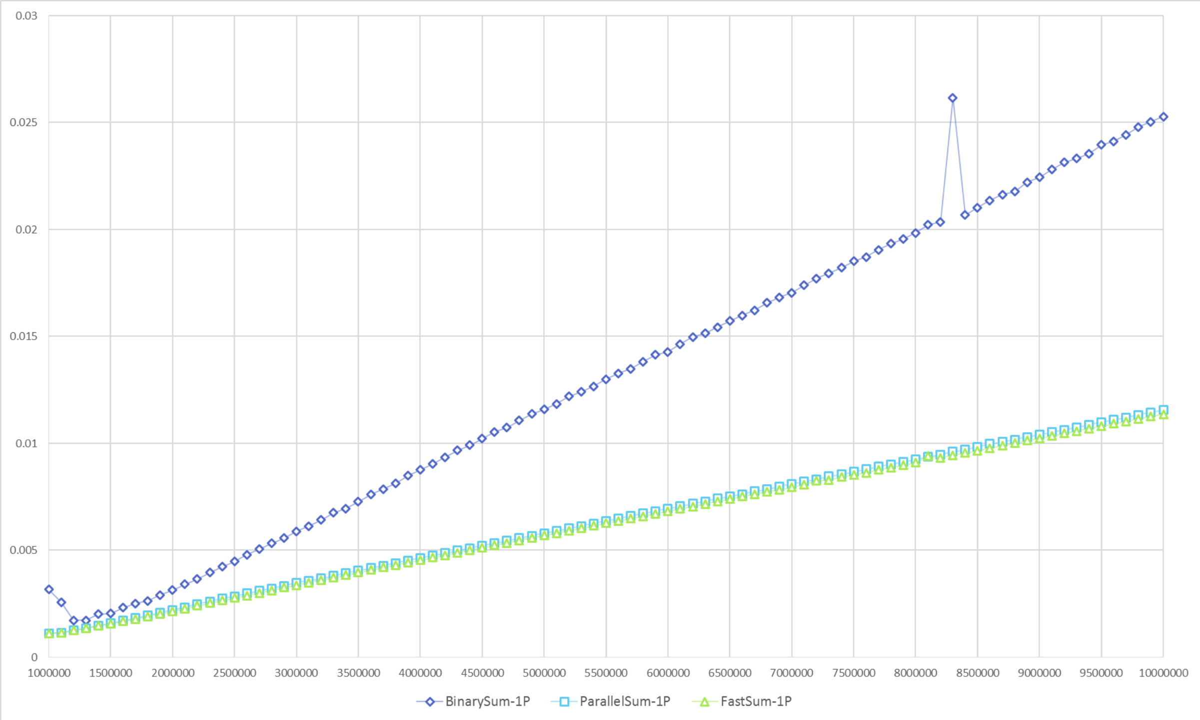

Figure 5: Single Thread Performance

The final chart shows the performance of each method when using 16 threads, as this number of threads produced the best performance. We can see, again that the management required for the binary sum creates a very large disparity between the runtimes when compared to the other two methods. At the largest value of N, the runtime of the binary sum is $\approx 3.64x$ that of the manually managed divide-and-conquer method and $\approx 5.79x$ of the fast sum runtime.

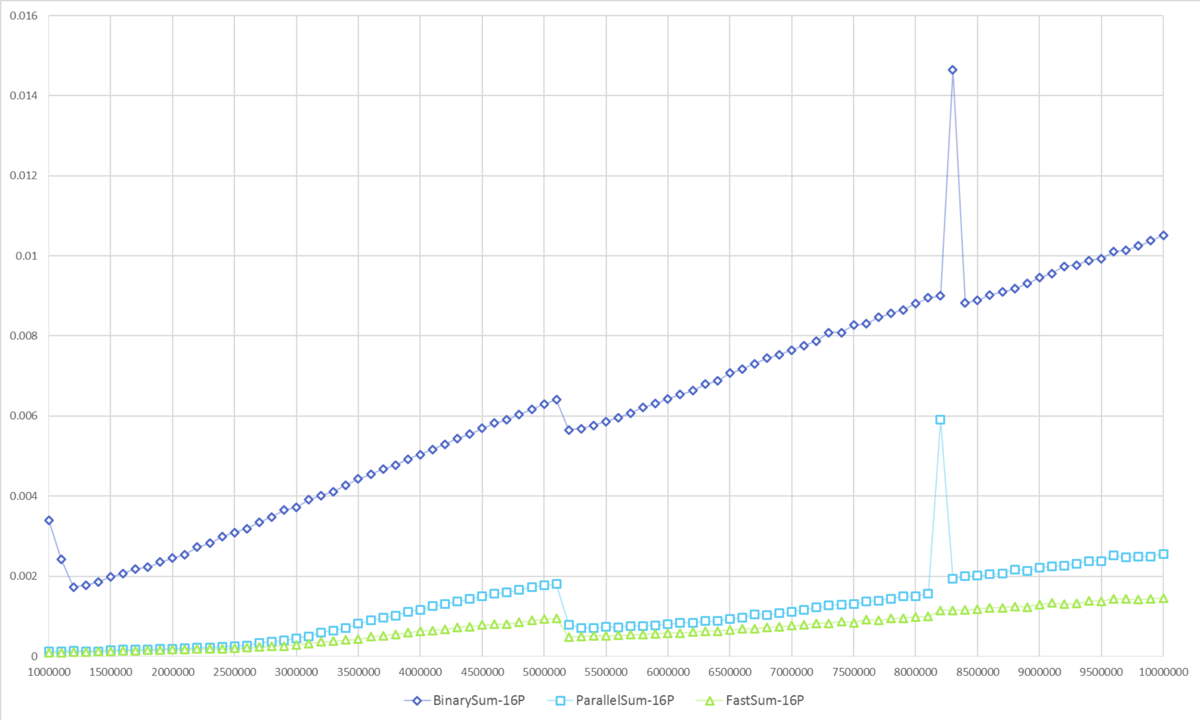

Figure 6: 16 Threads Performance

Conclusion

Although we know the theoretical limit of the binary sum to be $O(log_2 N)$, in practice the actual parallel performance is closer to that of the serial performance. Even at small values of N, which we could realistically have more processors than data points, the standard parallel sum may actually run faster due to the amout of overhead it takes to create the scratch space for the binary tree sum method.

Code Appendix

```
/***************************************************************************
Compile command: g++ -std=c++0x -fopenmp -o assignment3 assignment3.cpp -O3
***************************************************************************/
#include <omp.h>
#include <iostream>
#include <cmath>
#include <random>
#include <sys/time.h>
#include <fstream>

void get_walltime(long double *wcTime)
{
    struct timeval tp;
    gettimeofday(&tp, NULL);
    *wcTime = (long double) (tp.tv_sec + tp.tv_usec / 1000000.0);
}

/** binary_sum
 *
 * Calculates the sum of an array in a recursive binary-tree PRAM fashion.
 *
 * @param *array - the array to be computed
 * @param array_size - necessary for proper assignment
 * @param *time - reference to a variable to store the walltime
 *
 * @return - sum of array as a double
*/
double binary_sum(double *array, int array_size, long double *time)
{
    long double start_time = 0.0, end_time = 0.0, scratch_time = 0.0;
    double answer = 0.0, *results;
    int scratch_size, num_elements;

    // Handle edge cases of 1 or 2 element arrays
    if (array_size == 1)
    {
        return (array[0]);
    }
    if (array_size == 2)
    {
        return (array[0] + array[1]);
    }

    /** Create a scratch array to put temporary results in. If the input array is not divisible by 2, add 1 so that
     * scratch array can be a round integer. */
    if (array_size % 2 == 1)
    {
        scratch_size = (array_size + 1) / 2;
        results = new double[scratch_size]();
    } else
    {
        scratch_size = array_size / 2;
        results = new double[scratch_size]();
    }

    // Gets wall time and start parallel block.
    get_walltime(&start_time);
```

```c
    #pragma omp parallel
    {
        int threads, thread_id, start, end;
        threads = omp_get_num_threads();
        thread_id = omp_get_thread_num();
        #pragma omp single
        {
            /** Sets variables shared between threads and handles the final element in arrays first dependent on
             * whether or not the scratch array size is even or odd. */
            if (array_size % 2 == 1)
            {
                results[scratch_size - 1] = array[array_size - 1]; // Dynamic Allocation
            } else
            {
                results[scratch_size - 1] = array[array_size - 1] + array[array_size - 2]; // Dynamic Allocation
            }
            // divide and conquer b-tree adds based on number of threads
            num_elements = (int) ceil((double) (scratch_size) / (double) threads);
        }
        // Compute start and end values for each thread
        start = num_elements * thread_id;
        end = start + num_elements;

        // Ends at scratch array - 1 because final element is handled above.
        for (start; start < end && start < scratch_size - 1; start++)
        {
            // start*2 is used to access the correct array elements:
            // results[i] = array[2*i] + array[2*i + 1]
            results[start] = array[start * 2] + array[start * 2 + 1];

        }
    }
    // RECURSE! Scratch time is a dummy variable created in place of overloading the function call
    answer = binary_sum(results, scratch_size, &scratch_time);
    get_walltime(&end_time);
    *time = end_time - start_time;
    free(results); // frees dynamically allocated memory
    return answer;
}


/** parallel_sum
 *
 * Calculates the sum of an array by dividing array into subsections and then summing those on different threads.
 *
 * @param *array - the array to be computed
 * @param array_size - necessary for proper assignment
 * @param *time - reference to a variable to store the walltime
 *
 * @return - sum of array as a double
*/
double parallel_sum(double *array, int array_size, long double *time)
{
    double *scratch, sum;
```

```
    long double start_time = 0.0, end_time = 0.0;
    int scratch_size, num_elements;
    sum = 0.0;
    get_walltime(&start_time);
    #pragma omp parallel
    {
        int threads, thread_id, start, end;
        threads = omp_get_num_threads();
        thread_id = omp_get_thread_num();
        #pragma omp single
        {
            /** Allocate variables necessary to share between threads. Scratch space is allocated based on the
             * number of threads obtained by OpenMP (one array element per thread). */
            scratch = new double[threads](); // Dynamic allocation
            scratch_size = threads;

            // Divide array up into a set number of elements per thread.
            num_elements = (int) ceil((double) array_size / (double) threads);
        }

        // Set start and endpoints for each thread
        start = num_elements * thread_id;
        end = start + num_elements;

        // Sum up elements into scratch array
        for (start; start < end && start < array_size; start++)
        {
            scratch[thread_id] += array[start];
        }
    }

    // Go through scratch array and calculate final sum.
    for (int i = 0; i < scratch_size; i++)
    {
        sum += scratch[i];
    }
    free(scratch); // Free dynamically allocated memory
    get_walltime(&end_time);
    *time = end_time - start_time;
    return sum;
}


/** fast_sum
 *
 * Calculates the sum of an array using the OpenMP reduction method.
 *
 * @param *array - the array to be computed
 * @param array_size - necessary for proper assignment
 * @param *time - reference to a variable to store the walltime
 *
 * @return - sum of array as a double
*/
double fast_sum(double *array, int array_size, long double *time)
{
    double sum;
    long double start_time = 0.0, end_time = 0.0;
    sum = 0.0;
```

```cpp
    get_walltime(&start_time);
    #pragma omp parallel for reduction(+:sum)
    // declares sum to be the accumulator variable for all threads
    for (int i = 0; i < array_size; i++)
    {
        sum += array[i];
    }
    get_walltime(&end_time);
    *time = end_time - start_time;
    return sum;
}


int main()
{
    int set_threads = 10; // Threads to be requested by OpenMP
    char filename[11];
    sprintf(filename, "log_%02d.txt", set_threads);
    omp_set_num_threads(set_threads); // Set global variable (shell script did not appear to actually set value)

    std::ofstream log;
    log.open(filename, std::ofstream::out | std::ofstream::app); // Open file
    log << "ArraySize\tBinarySumTime\tBinarySumTotal\tParallelSumTime\tParallelSumTotal\tFastSumTime\t"
           "FastSumTotal\tBinParDiff\tBinFastDiff\tParFastDiff\n";

    for (int array_size = 1000000; array_size <= 10000000; array_size += 100000)
    {
        auto *x = new double[array_size](); // Dynamic allocation
        long double parallel_time = 0.0, fast_time = 0.0, binary_time = 0.0;
        for (int i = 0; i < array_size; i++)
        {
            x[i] = (double) (rand()) / (double) (RAND_MAX) * 5.0;
        }
        double binary_total = binary_sum(x, array_size, &binary_time);
        double parallel_total = parallel_sum(x, array_size, &parallel_time);
        double fast_total = fast_sum(x, array_size, &fast_time);
        log << array_size << "\t" << binary_time << "\t" << binary_total << "\t" << parallel_time << "\t"
            << parallel_total << "\t" << fast_time << "\t" << fast_total << "\t" << binary_total - parallel_total
            << "\t" << binary_total - fast_total << "\t" << parallel_total - fast_total << std::endl;
        free(x); // Free dynamically allocated memory
    }
    log.close(); // Close file
    return 0;
}
```