Ehi Simon - 14352307 ECEC 413-001
Alisha Augustin - 14389451 05/21/2023

Open MP: Jacobi Equation Solver

The code implementation below demonstrates the parallelization of the Jacobi iteration method using OpenMP. The compute_using_omp function takes as input the system matrix A, the solution matrix *mt_sol_x*, the right-hand side vector *B*, the maximum number of iterations *max_iter*, and the number of threads to use *num_threads*. The parallelization process begins by allocating a new matrix *new_x* to store the iteration values. The current Jacobi solution *mt_sol_x* is then initialized by copying the elements of the right-hand side vector B using a parallel loop. The main computation loop starts, iterating until convergence or the maximum number of iterations is reached. Within this loop, two parallel loops are employed. The first parallel loop calculates the new values of *new_x* in parallel by iterating over each row of matrix *A*. For each row, a sum is computed by multiplying the corresponding elements of *A* and *mt_sol_x*, excluding the diagonal element. The sum is subtracted from the corresponding element of B and divided by the diagonal element of *A* to compute the new value of *new_x.elements[i]*.

The second parallel loop is responsible for calculating the sum of squared differences (ssd) between the elements of *new_x* and *mt_sol_x*, as well as updating the elements of *mt_sol_x* with the values of *new_x*. This loop also increments the iteration count num_iter and computes the mean squared error (mse). To achieve parallelism, OpenMP directives are utilized. The *num_threads* parameter is used to specify the desired number of threads for parallel execution. The parallel for directive distributes the workload of the loops across multiple threads, allowing concurrent computation. The private and reduction clauses ensure proper memory management and reduction operations for shared variables, respectively.

```c
void compute_using_omp(const matrix_t A, matrix_t mt_sol_x, const matrix_t B, int max_iter, int num_threads) {
    int i, j;
    int num_rows = A.num_rows;
    int num_cols = A.num_columns;
    int done = 0;
    double ssd, mse;
    int num_iter = 0;

    /* Allocate n x 1 matrix to hold iteration values. */
    matrix_t new_x = allocate_matrix(num_rows, 1, 0);

    /* Initialize current Jacobi solution. */
    #pragma omp parallel for num_threads(num_threads)
    for (i = 0; i < num_rows; i++)
        mt_sol_x.elements[i] = B.elements[i];

    while (!done) {
        #pragma omp parallel for private(j) num_threads(num_threads)
        for (i = 0; i < num_rows; i++) {
            double sum = 0.0;
            for (j = 0; j < num_cols; j++) {
                if (i != j)
                    sum += A.elements[i * num_cols + j] * mt_sol_x.elements[j];
            }

            new_x.elements[i] = (B.elements[i] - sum) / A.elements[i * num_cols + i];
        }

        ssd = 0.0;
        #pragma omp parallel for reduction(+:ssd) num_threads(num_threads)
        for (i = 0; i < num_rows; i++) {
            double diff = new_x.elements[i] - mt_sol_x.elements[i];
            ssd += diff * diff;
            mt_sol_x.elements[i] = new_x.elements[i];
        }

        num_iter++;
        mse = sqrt(ssd); /* Mean squared error. */

        if ((mse <= THRESHOLD) || (num_iter == max_iter))
            done = 1;
    }

    if (num_iter < max_iter)
        fprintf(stderr, "\nConvergence achieved after %d iterations\n", num_iter);
    else
        fprintf(stderr, "\nMaximum allowed iterations reached\n");

    free(new_x.elements);
}
```

Figure 1: Implementation of Jacobi Solver

Performance (Execution Time) Comparison

	Serial				Parallel			
Matrix Size	**4 Threads**	**8 Threads**	**16 Threads**	**32 Threads**	**4 Threads**	**8 Threads**	**16 Threads**	**32 Threads**
512x512	4.382334 s	4.114499s	4.078602s	4.048388s	1.630495s	1.442992s	1.360059s	2.481225s
1024x1024	33.897705s	33.529045 s	34.649612s	33.529133 s	9.424492s	7.116407s	6.976046s	7.388496s
2048x2048	282.351196s	273.445282s	273.729095 s	268.600342s	76.839096 s	37.373932 s	38.061008s	38.275337s
4096x4096	2215.565 918s	2238.809082s	2196.681152s	2196.113525s	589.193909s	327.713898s	401.693085s	521.247803s

Speedup Comparison

Matrix Size	**4 Threads**	**8 Threads**	**16 Threads**	**32 Threads**
512x512	2.69	2.85	2.999	1.63
1024x1024	3.60	4.71	4.97	4.54
2048x2048	3.67	7.32	7.19	7.02
4096x4096	3.76	6.83	5.47	4.21

The speedup obtained measures the improvement in execution time achieved by running the matrix calculations in parallel compared to running them sequentially. In the given results above, the parallel implementation consistently demonstrates speedup for all matrix sizes and thread configurations. The highest speedup values are achieved with 16 threads, indicating that utilizing 16 threads leads to the most significant performance improvement compared to both the serial implementation and other thread configurations.