DREXEL UNIVERSITY
Electrical and Computer Engineering
College of Engineering

Drexel University

Electrical and Computer Engineering Dept.

ECEC-413

Jacobi Solver with OpenMP

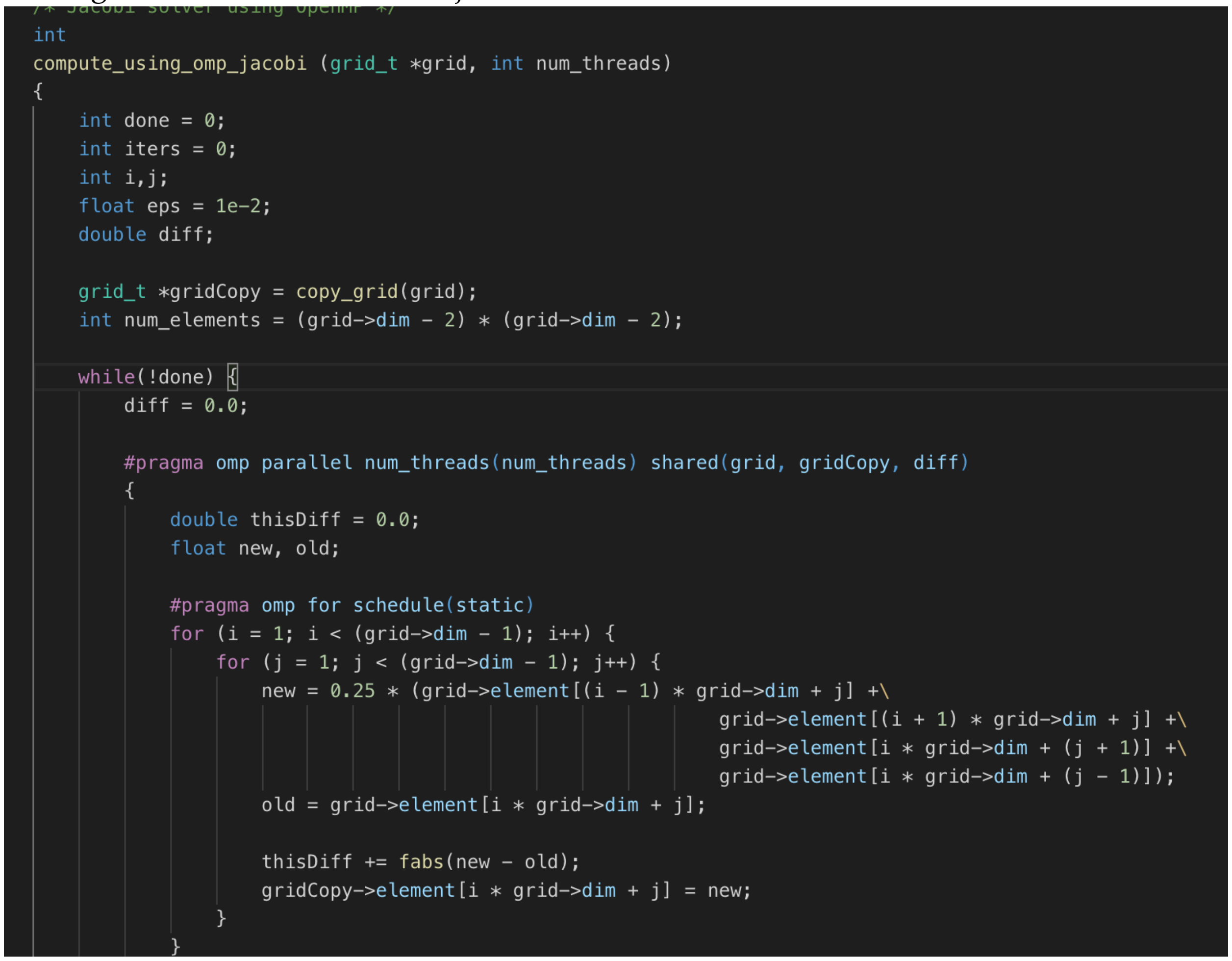
Chris Kasper
Prof. Naga Kandasamy
DATE: 05/2/19

Multi-threaded Design with OpenMP

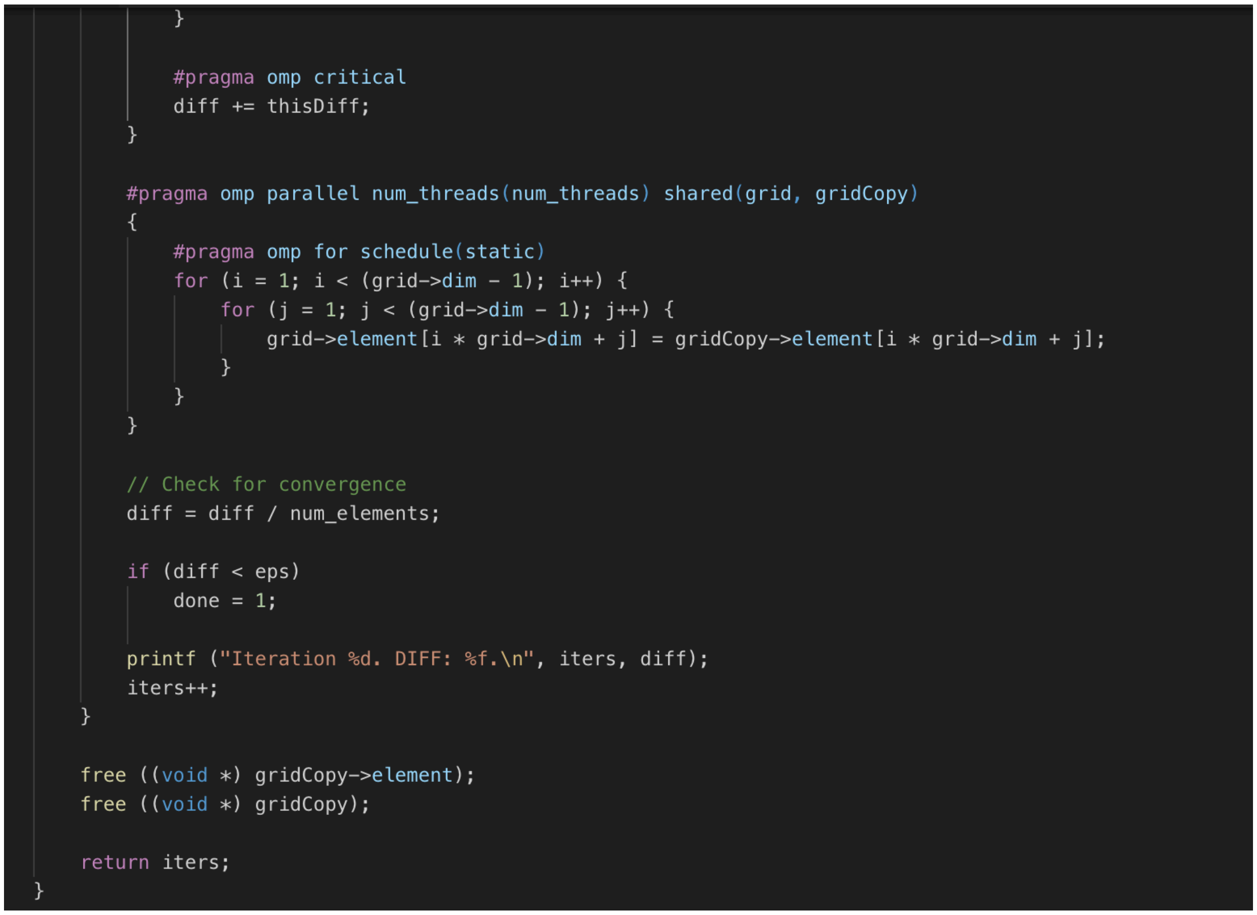
Using the Jacobi equation solver, the existing algorithm that is being used to solve the systems of equations (the steady-state temperature distribution) can be sped up in an efficient way. One of the main problems with the sequential program is the loop carried dependency. Values are computed based on a combination of the current and previous iteration.

With the multi-threaded approach, this dependency can be eliminated. The main idea is to have an additional grid to copy over the new values being computed. With using OpenMP, the total work is divided up, row-wise, by the number of threads. Each thread is responsible for computing the new values (of their allocated rows) for each iteration and copying them into the main grid. This means every computed value is done with the values of the previous iteration. However, it is done in stages using barriers.

The logic for the multi-threaded jacobi solver is as follows:



```
/* Jacobi solver using OpenMP */
int
compute_using_omp_jacobi (grid_t *grid, int num_threads)
{
    int done = 0;
    int iters = 0;
    int i,j;
    float eps = 1e-2;
    double diff;

    grid_t *gridCopy = copy_grid(grid);
    int num_elements = (grid->dim - 2) * (grid->dim - 2);

    while(!done) {
        diff = 0.0;

        #pragma omp parallel num_threads(num_threads) shared(grid, gridCopy, diff)
        {
            double thisDiff = 0.0;
            float new, old;

            #pragma omp for schedule(static)
            for (i = 1; i < (grid->dim - 1); i++) {
                for (j = 1; j < (grid->dim - 1); j++) {
                    new = 0.25 * (grid->element[(i - 1) * grid->dim + j] +\
                                        grid->element[(i + 1) * grid->dim + j] +\
                                        grid->element[i * grid->dim + (j + 1)] +\
                                        grid->element[i * grid->dim + (j - 1)]);
                    old = grid->element[i * grid->dim + j];

                    thisDiff += fabs(new - old);
                    gridCopy->element[i * grid->dim + j] = new;
                }
            }
```



```
            }

            #pragma omp critical
            diff += thisDiff;
        }

        #pragma omp parallel num_threads(num_threads) shared(grid, gridCopy)
        {
            #pragma omp for schedule(static)
            for (i = 1; i < (grid->dim - 1); i++) {
                for (j = 1; j < (grid->dim - 1); j++) {
                    grid->element[i * grid->dim + j] = gridCopy->element[i * grid->dim + j];
                }
            }
        }

        // Check for convergence
        diff = diff / num_elements;

        if (diff < eps)
            done = 1;

        printf ("Iteration %d. DIFF: %f.\n", iters, diff);
        iters++;
    }

    free ((void *) gridCopy->element);
    free ((void *) gridCopy);

    return iters;
}
```

Since the end of OpenMP parallel execution loops serve as barrier syncs, I set up two of them, one for the computation of new grid values, and another for copying over the new grid values from the backup grid. The first parallel execution has shared variables “grid”, “gridCopy”, and “diff”. The second has just “grid” and “gridCopy”. The “diff” variable in the first parallel execution is used to store the computed difference between the new and previous grid values. Each thread has a private “thisDiff” variable to store the differences with the values it has computed. Once the thread finishes its allocated work, it adds on its “thisDiff” variable to the global “diff”, which is ultimately used to check for convergence.

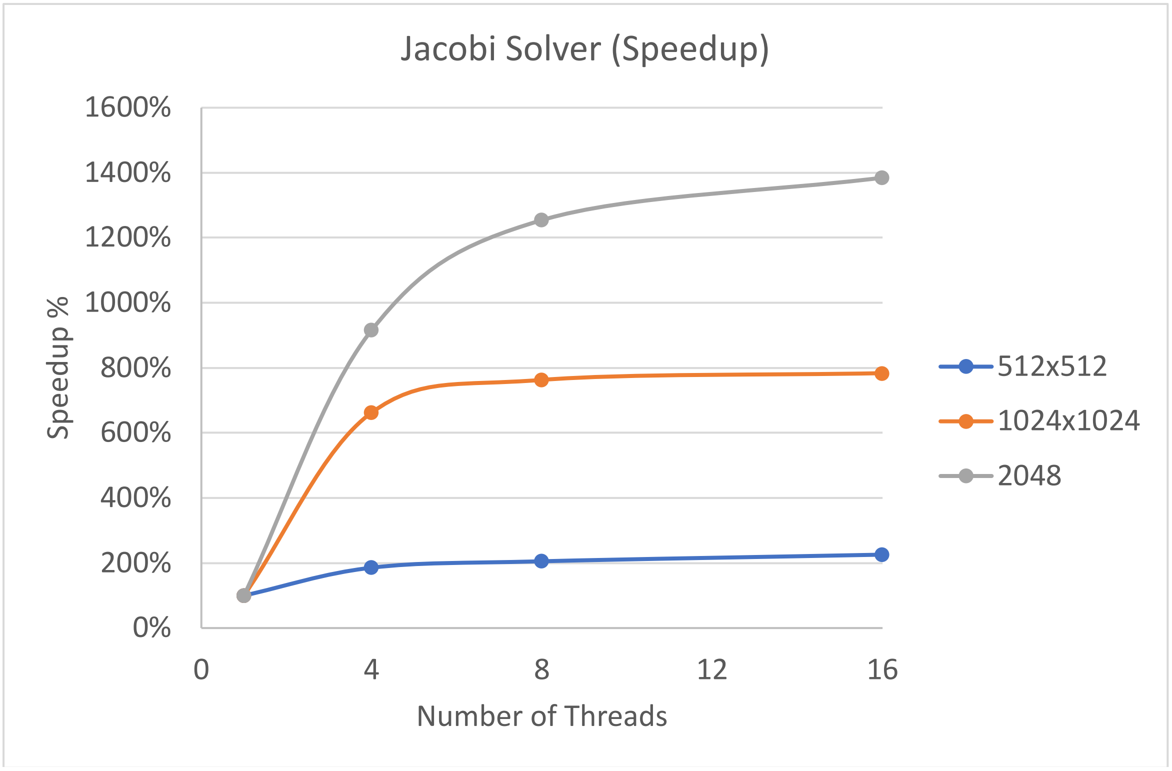
As stated above, the second parallel execution just copies over the new grid values to the main grid. Once the copying process is finished, the main thread checks for convergence. If it hasn’t yet, the whole process is repeated again. Once it has converged, the “gridCopy” is freed, and “iters” (number of iterations) is returned from the function.

Speedup

Table 1: Speedup with Jacobi Solver on xunil-05
(Temperature ranges between 1000, 1500)

Matrix Size	Single Thread	4 Threads	8 Threads	16 Threads
512 x 512	10.45s	5.63s	5.09s	4.63s
1024 x 1024	23.12s	3.49s	3.03s	2.95s
2048 x 2048	17.31s	1.89s	1.38s	1.25s

Note: Print statements in both sequential and parallel execution commented out


Jacobi Solver (Speedup)
Speedup %
Number of Threads
512x512
1024x1024