DREXEL UNIVERSITY
Electrical and Computer Engineering
College of Engineering

Drexel University

Electrical and Computer Engineering Dept.

ECEC-413

Jacobi Solver

Chris Kasper
Prof. Naga Kandasamy
DATE: 04/25/19

Multi-threaded Design

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 pthreads, 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 thread setup is as follows:

```
int i;
ARGS_FOR_THREAD *args_thread;
double diffs[num_threads];

grid_t *grid2 = copy_grid(grid);

pthread_t *worker = (pthread_t *) malloc (num_threads *sizeof (pthread_t));
pthread_barrier_init(&b1, 0, num_threads);
pthread_barrier_init(&b2, 0, num_threads);
pthread_barrier_init(&b3, 0, num_threads);

for (i = 0; i < num_threads; i++) {
    args_thread = (ARGS_FOR_THREAD *) malloc (sizeof (ARGS_FOR_THREAD));

    args_thread->threadID = i;
    args_thread->num_threads = num_threads;
    args_thread->grid1 = grid;
    args_thread->grid2 = grid2;
    args_thread->mydiff = diffs + i;

    if ((pthread_create (&worker[i], NULL, pthreads_jacobi, (void *) args_thread)) != 0) {
        printf ("Cannot create worker thread \n");
        exit (EXIT_FAILURE);
    }
}

for (i = 0; i < num_threads; i++) {
    pthread_join (worker[i], NULL);
```

The three barriers are initialized, as well as the worker tids and the additional grid. Each thread is given an ID, the total number of threads, grid pointers, and diff pointers. The diff pointers provide a location in memory so each thread can store the difference of values they computed between iterations, which can then be accessed by the first thread for convergence.

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

```
while(!pthread_done) { /* While we have not converged yet. */
    diff = 0.0;

    // First wave, update grid2 (new values)
    for (i = threadID+1; i < (grid1->dim - 1); i += num_threads) {
        for (j = 1; j < (grid1->dim - 1); j++) {
            grid2->element[i * grid1->dim + j] = 0.25 * (grid1->element[(i - 1) * grid1->dim + j] +\
                                                         grid1->element[(i + 1) * grid1->dim + j] +\
                                                         grid1->element[i * grid1->dim + (j + 1)] +\
                                                         grid1->element[i * grid1->dim + (j - 1)]);
        }
    }

    pthread_barrier_wait(&b1);

    //Second wave - update grid1 (grid) with new values from grid2 (backup)
    for (i = threadID+1; i < (grid1->dim - 1); i += num_threads) {
        for (j = 1; j < (grid1->dim - 1); j++) {

            // Get diff first
            diff = diff + fabs(grid2->element[i * grid1->dim + j] - grid1->element[i * grid1->dim + j]);
            grid1->element[i * grid1->dim + j] = grid2->element[i * grid1->dim + j];
        }
    }

    *grid_thread->mydiff = diff;
```

```
pthread_barrier_wait(&b2);

// First thread will check for covarence
if (threadID == 0) {
    double *temp;
    diff = 0.0;

    for (i = 0; i < num_threads; i++) {
        temp = grid_thread->mydiff + i;
        diff = diff + *temp;
    }

    diff = diff/num_elements;

    if (diff < eps)
        pthread_done = 1;

    printf ("Iteration %d. DIFF: %f.\n", pthread_iters, diff);
    pthread_iters++;
}
pthread_barrier_wait(&b3);
}
```

Each thread will continue to go through this logic until convergence. You can see the three barriers that all the threads must reach in order to proceed further. As noted, this logic is separated into three stages. The first stage is computing the new values and storing them in an additional grid. The second stage is copying over the new values to the main, as well as accumulating the total difference for this threads work. Finally, the third stage, that is only done by the first thread, checks for convergence, by accessing the diff values from each thread.

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	10.36s	2.45s	1.54s	1.25s
1024 x 1024	22.39s	1.95s	1.29s	0.94s
2048 x 2048	17.62s	1.84s	1.39s	1.37s

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