Aleksandar Aleksandrov

ECEC 413

Jacobi Equation Solver Report

1) **Design of Multi-Threaded Implementation:**
   For this assignment, a parallel version of the Jacobi method was created using pthreads. In this specific assignment, the Jacobi iterative algorithm will be implemented to solve for the temperatures on a grid based on fixed initial conditions on the top row of the grid. The grid represents a metal plate in which the grid points are uniformly spaced where heat is applied to the top row only. The figure below is an example of what the grid would look like (screenshot from assignment).

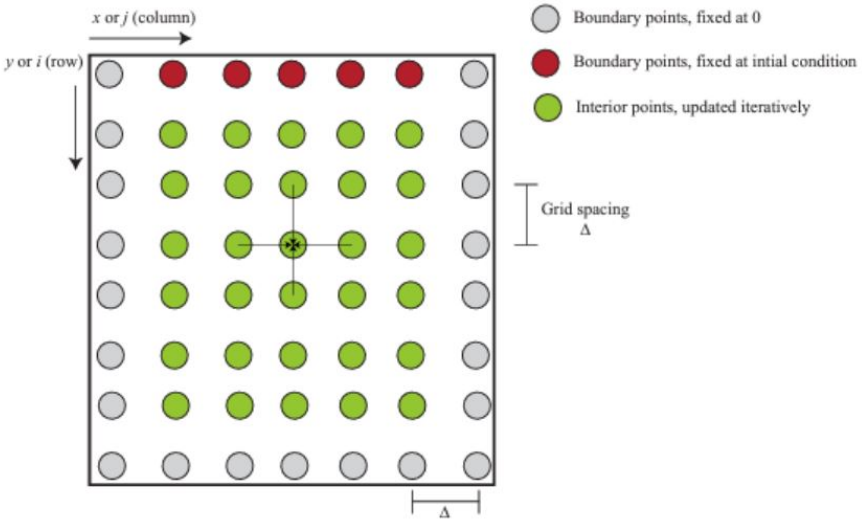

Figure 1 - assignment example grid

For the multithreaded implementation of the algorithm, the chunking method is used to assign to every thread a portion of the grid. Since the outer boundaries of the matrix are fixed and the algorithm doesn't have to iterate over those parts the offset for every thread is set with the following formula:

(Thread ID)*(Chunk Size) + 1

This way the first thread will always ignore the first row of the matrix.

The thread structure contains the following elements:

```c
typedef struct thread_data_s {
    int tid;
    int num_threads;
    double num_elements;
    grid_t *dest_grid; //Store pointer to a source grid
    grid_t *src_grid;  //Store pointer to a destination grid
    int offset;
    int chunk_size;
    double *diff;
    double *num_iter;
    pthread_mutex_t *mutex_for_diff;
    pthread_mutex_t *mutex_for_num_iter;
} thread_data_t;
```

There are two pointers for a destination grid and a source grid. The source grid is copied using the "copy_grid" function. According to the iterative algorithm, to compute the correct values it is enough to use the values computed from the previous iteration. Therefore, the source grid contains the values computed for iteration "i-1" and the destination grid contains the values computed in the current iteration. The threads also share a global variable "diff" which stores the total difference accumulated from all threads. Since "diff" is a global variable for all the threads, a mutex is initialized.

The barriers in this assignment are implemented using semaphores. The following structure is created:

```c
/* Structure that defines the barrier */
typedef struct barrier_s {
    sem_t counter_sem;          /* Protects access to the counter */
    sem_t barrier_sem;          /* Signals that barrier is safe to cross */
    int counter;                /* The value itself */
} barrier_t;

barrier_t barrier1;
barrier_t barrier2;
void barrier_sync(barrier_t *, int, int);
```

The implementation uses two barriers, therefore, barrier1 and barrier2 are declared and initialized in the thread creation function.

The worker function is passed to every thread and executes the parallel Jacobi Algorithm.

First the data structure that passes all the variables is casted:

```c
thread_data_t *thread_data = (thread_data_t *)args;
```

Then every thread has the following local variables:

```
    int done = 0;
    int i,j;
    double p_diff;
    float old, new;
    double eps = 1e-6;
    int num_elements;
```

"p_diff" is used to accumulate the partial diff for the corresponding thread.

Afterwards, all threads go into the while loop that runs until the total difference of all datapoints has become less than e^{-6}.

The first step of the loop is to set the partial difference and the number of elements to 0. Then a barrier is placed to make sure that all threads have finished executing the previous iteration. Thread 0 then sets the global "diff" variable to 0 by using the mutex lock.

Then the threads go into the if-else statement and perform the calculation on their respective chunk. The last thread goes to the end of the matrix rows. Every thread has a set number of rows according to its chunk size. Therefore, the outer for loop goes through the necessary rows and the inner for loop goes through every element in the respective row:

```
for (i = thread_data->offset; i < (thread_data->offset + thread_data->chunk_size); i++)
```

```
for (j = 1; j < (thread_data->src_grid->dim - 1); j++)
```

The inner for loop always ignores the first and last elements of the row (grey dots on the schematic).

After the thread has gone through its chunk of data it adds its partial difference to the global difference. In this case, however, the partial difference is divided by the number of elements that the thread has iterated through. Then the thread hits the second barrier and waits for all other threads to complete their execution.

Then every thread uses the updated global difference value to evaluate if the difference is less than e^{-6}. The difference is divided by the number of threads, since we need to make sure we have the average across all threads. If the if statement passes, all threads exit the while loop. Else the two grid pointers are switched, and the current destination grid becomes the source for the next iteration.

2) **Speedup Obtained Over Various Serial Versions:**

Below are tables of the time it takes to calculate the interior points of each grid depending on the number of threads and its grid size.

Temperature Range	Grid Dimensions	Single Thread (s)	4 threads (s)	8 threads (s)	16 threads (s)
100-150	**128x128**	1.392	0.927	1.206	1.052
100-150	**256x256**	19.01	6.944	5.895	4.031
100-150	**512x512**	253.067	73.703	50.291	20.166

Based on the tables, we can see that the larger the grid becomes, the more substantial of a speedup there is when computing the interior points of the grid for a multi-threaded process. When the grid is smaller, there doesn't seem to be a noticeable speedup as using multiple threads adds too much of an overhead to notice any improvement. It could be said that for small grids the single threaded version is sufficient. It can also be noticed that for smaller grids, increasing the threads decreases the performance of the multithreaded program because of the added overhead. The most significant performance improvements are for the grid of size 512x512. Increasing the number of threads lead to higher performance improvements for the bigger matrices. With 16 threads for the 512x512 grid the multithreaded solution is over 12 times faster.

The resulting MSE between the multithreaded version and the synchronous version is always very small (less than 1) which leads to the conclusion that the threads implementation returns the correct results.