Ehi Simon - 14352307 ECEC 413-001
Alisha Augustin - 14389451 05/08/2023

Project 2: Iterative Jacobi Solver

The Jacobian method is an iterative algorithm that calculates the values of each element on a grid based on the previous values of itself and its neighboring elements. Through parallelization, the computation time can be significantly reduced by dividing the workload among multiple threads. The code uses a parallel implementation by distributing the total number of rows in the grid among the threads. We parallelized the code using two different design patterns, the chunking method and the striding method.

Chunking Method

In the chunking approach, the code uses the *compute_using_pthreads_v1* function to perform the computation using multiple threads. The function starts by initializing the necessary variables and data structures. It divides the work into chunks by calculating the start and end indices for each thread based on the number of rows and the number of threads. The function then creates threads using *pthread_create* and passes the *compute_chunk* function as the thread routine. Each thread receives a specific range of rows to process. Inside the compute_chunk function, each thread enters a loop that continues until the convergence condition is met or the maximum iteration count is reached.Within the loop, the thread calculates the difference between the old and new solution vectors for the assigned range of rows. It updates the global difference by adding the local difference in a thread-safe manner using a mutex lock. Then, the thread waits for all threads to finish this step using a barrier.After the barrier, the thread calculates the mean squared error using the global difference. If the convergence condition is met or the maximum iteration count is reached, the thread sets the converged flag to 1 and updates the solution vector. Again, it waits for all threads to finish this step using the barrier. Finally, the thread swaps the old solution vector with the new solution vector for the next iteration and repeats the loop until convergence or maximum iterations are reached. Once the convergence condition is met, the thread exits.

```c
void compute_using_pthreads_v1(const matrix_t A, matrix_t mt_sol_x_v1, const matrix_t B, int max_iter, int num_threads)
{
    pthread_t *thread_id = (pthread_t *)malloc(num_threads * sizeof(pthread_t));
    pthread_attr_t attributes;
    pthread_attr_init(&attributes);
    matrix_t new_x = allocate_matrix(A.num_rows, 1, 0);

    double diff = 0.0;
    int num_iter = 0;
    int converged = 0;

    int i;
    int chunk_size = (int)floor(mt_sol_x_v1.num_rows / num_threads);
    int remainder = mt_sol_x_v1.num_rows % num_threads;

    pthread_barrierattr_t barrier_attributes;
    pthread_barrier_t barrier;
    pthread_barrierattr_init(&barrier_attributes);
    pthread_barrier_init(&barrier, &barrier_attributes, num_threads);

    pthread_mutex_t lock;
    pthread_mutex_init(&lock, NULL);

    thread_data_t *thread_data = (thread_data_t *)malloc(sizeof(thread_data_t) * num_threads);
    for (i = 0; i < num_threads; i++) {
        int start_index = i * chunk_size;
        int end_index = (i + 1) * A.num_columns;
```

```c
        thread_data[i].tid = i;
        thread_data[i].num_threads = num_threads;
        thread_data[i].A = A;
        thread_data[i].B = B;
        thread_data[i].x = &mt_sol_x_v1;
        thread_data[i].new_x = &new_x;
        thread_data[i].max_iter = max_iter;
        thread_data[i].start_index = start_index;
        thread_data[i].end_index = end_index;
        thread_data[i].barrier = &barrier;
        thread_data[i].lock = &lock;
        thread_data[i].diff = &diff;
        thread_data[i].converged = &converged;
        thread_data[i].num_iter = &num_iter;
    }

    for (i = 0; i < num_threads; i++)
        pthread_create(&thread_id[i], &attributes, compute_chunk, (void *)&thread_data[i]);

    for (i = 0; i < num_threads; i++)
        pthread_join(thread_id[i], NULL);

    free(new_x.elements);
    free((void *)thread_data);
    pthread_barrier_destroy(&barrier);

}
```

Fig. 1. Using Chunking Design Pattern to Develop Parallel Formulations of the Jacobi solver

```c
void *compute_chunk(void* args)
{
    thread_data_t *thread_data = (thread_data_t *)args;
    int tid = thread_data->tid;
    matrix_t A = thread_data->A;
    matrix_t *x = thread_data->x;
    matrix_t *new_x = thread_data->new_x;
    matrix_t B = thread_data->B;
    int max_iter = thread_data->max_iter;
    int start_index = thread_data->start_index;
    int end_index = thread_data->end_index;
    double *diff = thread_data->diff;
    int *converged = thread_data->converged;
    pthread_barrier_t *barrier = thread_data->barrier;
    pthread_mutex_t *lock = thread_data->lock;
    int *num_iter = thread_data->num_iter;
    int num_cols = A.num_columns;

    int i, j;

    while (!*converged) {
        if (tid == 0) {
            *diff = 0;
            (*num_iter)++;
        }

        pthread_barrier_wait(barrier);

        for (i = start_index; i < end_index; i++) {
            double sum = 0.0;
            for (j = 0; j < num_cols; j++) {
                if (i != j)
                    sum += A.elements[i * num_cols + j] * x->elements[j];
            }
            new_x->elements[i] = (B.elements[i] - sum) / A.elements[i * (num_cols + 1)];
        }
```

```
        double pdiff = 0.0;

        for (i = start_index; i < end_index; i++) {
            pdiff += fabs(new_x->elements[i] - x->elements[i]);
        }

        pthread_mutex_lock(lock);
        *diff += pdiff;
        pthread_mutex_unlock(lock);

        pthread_barrier_wait(barrier);

        double mse = sqrt(*diff);

        if ((mse <= THRESHOLD) || (*num_iter == max_iter)) {
            *converged = 1;
            for (i = start_index; i < end_index; i++) {
                x->elements[i] = new_x->elements[i];
            }
        }
        pthread_barrier_wait(barrier);

        matrix_t *tmp = x;
        x = new_x;
        new_x = tmp;

    }

    pthread_exit(NULL);
}
```

Fig. 2. Implementation of the Chunking Design Pattern

Striding Method

In the striding approach, the code uses the *compute_using_pthreads_v2* function to perform the computation using multiple threads with striding. The function starts by initializing the necessary variables and data structures. It divides the work among the threads by assigning each thread a stride value.

```c
void compute_using_pthreads_v2(const matrix_t A, matrix_t mt_sol_x_v2, const matrix_t B, int max_iter, int num_threads)
{

    int tid;

    pthread_t *thread_id = (pthread_t *)malloc(num_threads * sizeof(pthread_t));
    pthread_attr_t attributes;
    pthread_attr_init(&attributes);

    matrix_t new_x = allocate_matrix(A.num_rows, 1, 0);

    int converged = 0;
    double diff = 0.0;
    int num_iter = 0;
    int num_rows = A.num_rows;
    pthread_barrierattr_t barrier_attributes;
    pthread_barrier_t barrier;
    pthread_barrierattr_init(&barrier_attributes);
    pthread_barrier_init(&barrier, &barrier_attributes, num_threads);

    /*Initialize Mutex Lock*/
    pthread_mutex_t lock;
    pthread_mutex_init(&lock, NULL);

    thread_data_t *thread_data = (thread_data_t *)malloc(sizeof(thread_data_t) * num_threads);
```

```c
    for(tid = 0; tid < num_threads; tid++){
        thread_data[tid].tid = tid;
        thread_data[tid].num_threads = num_threads;
        thread_data[tid].A = A;
        thread_data[tid].B = B;
        thread_data[tid].x = &mt_sol_x_v2;
        thread_data[tid].new_x = &new_x;
        thread_data[tid].max_iter = max_iter;
        thread_data[tid].barrier = &barrier;
        thread_data[tid].lock = &lock;
        thread_data[tid].diff = &diff;
        thread_data[tid].converged = &converged;
        thread_data[tid].num_iter = &num_iter;
        thread_data[tid].start_index = tid;
        thread_data[tid].end_index = num_rows;
    }

    int i;
    for (i = 0; i < num_threads; i++){
        pthread_create(&thread_id[i], &attributes, compute_stride, (void *)&thread_data[i]);
    }

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

    free(new_x.elements);
    free((void *)thread_data);
    pthread_barrier_destroy(&barrier);

}
```

Fig. 3. Using Striding Design Pattern to Develop Parallel Formulations of the Jacobi solver

The function then creates threads using *pthread_create* and passes the *compute_stride* function as the thread routine. Each thread receives a specific stride value and processes the data based on the stride, skipping elements in between. Inside the compute_stride function, each thread enters a loop that continues until the convergence condition is met or the maximum iteration count is reached. Within the loop, the thread initializes the necessary variables and the solution vector for the assigned range of rows. It calculates the difference between the old and new solution vectors for the assigned range of rows and updates the global difference in a thread-safe manner using a mutex lock. Then, the thread waits for all threads to finish this step using a barrier.

```
void *compute_stride(void* args){

    thread_data_t *thread_data = (thread_data_t *)args;
    int tid = thread_data->tid;
    int stride = thread_data->num_threads;
    matrix_t A = thread_data->A;
    matrix_t *x = thread_data->x;
    matrix_t *new_x = thread_data->new_x;
    matrix_t B = thread_data->B;
    int max_iter = thread_data->max_iter;
    int start_index = thread_data->start_index;
    int end_index = thread_data->end_index;
    double *diff = thread_data->diff;
    int *converged = thread_data->converged;
    pthread_barrier_t *barrier = thread_data->barrier;
    pthread_mutex_t *lock = thread_data->lock;
    int *num_iter = thread_data->num_iter;
    double mse, sum;
    int i = start_index, j;
    sum = 0.0;
    int num_cols = A.num_columns;

    while(i < end_index){
        x->elements[i] = B.elements[i];
        i = i + stride;
    }

    while(!*converged) {
        if(tid == 0) {
            *diff = 0;
            (*num_iter)++;
        }

        pthread_barrier_wait(barrier);

        i = start_index;

        while(i < end_index){
            sum = 0.0;
            for(j=0; j < num_cols; j++){
                if(i != j)
                    sum += A.elements[i * num_cols + j] * x->elements[j];

            }
            new_x->elements[i] = (B.elements[i] - sum) / A.elements[i *(num_cols + 1)];

            i = i + stride;

        }

        double pdiff = 0.0;
```

```c
        i = start_index;
        while(i < end_index){
            pdiff += fabs(new_x->elements[i] - x->elements[i]);
            i = i + stride;
        }
        pthread_mutex_lock(lock);
        *diff += pdiff;
        pthread_mutex_unlock(lock);
        pthread_barrier_wait(barrier);

        mse = sqrt(*diff);

        if ((mse <= THRESHOLD) || (*num_iter == max_iter)) {
            *converged = 1;
            for (i = start_index; i <= end_index; i++)
                thread_data->x->elements[i] = new_x->elements[i];
        }

        pthread_barrier_wait(barrier);

        matrix_t *tmp = x;
        x = new_x;
        new_x = tmp;

    }

    pthread_exit(NULL);
}
```

Fig. 4. Implementation of the Striding Design Pattern

After the barrier, the thread calculates the mean squared error using the global difference. If the convergence condition is met or the maximum iteration count is reached, the thread sets the converged flag to 1 and updates the solution vector. Again, it waits for all threads to finish this step using the barrier. Finally, the thread swaps the old solution vector with the new solution vector for the next iteration and repeats the loop until convergence or maximum iterations are reached. Once the convergence condition is met, the thread exits.

Performance Comparison
Execution Time

Serial				
Matrix Size	**4 Threads**	**8 Threads**	**16 Threads**	**32 Threads**
512x512	10.998729s	9.190670s	11.605428s	7.633028s
1024x1024	45.510361s	56.668854s	51.551079s	50.904945s
2048x2048	297.955139s	299.269714s	303.491150s	303.525208s
4096x4096	2022.111450s	2113.084717s	1983.461914s	2128.862549s

Parallel	**Chunking (V1)**				**Striding(V2)**			
Matrix Size	**4 Threads**	**8 Threads**	**16 Threads**	**32 Threads**	**4 Threads**	**8 Threads**	**16 Threads**	**32 Threads**
512x512	15.345109s	14.401425s	24.163387s	47.058475s	15.532382s	19.269497s	26.672085s	48.807068s
1024x1024	56.061253s	53.646606s	56.519657s	72.622498s	90.393661s	72.027740s	65.324860s	57.623646s
2048x2048	394.064423s	320.255188s	297.177124s	271.057281s	315.810150s	309.114349s	308.967133s	224.199646s
4096x4096	1591.114380s	1147.386108s	911.639832s	1044.423950s	1058.442505s	1025.422119s	1108.799316s	984.306152s

Speedup Comparison

Parallel	Chunking (V1)				Striding(V2)			
Matrix Size	**4 Threads**	**8 Threads**	**16 Threads**	**32 Threads**	**4 Threads**	**8 Threads**	**16 Threads**	**32 Threads**
512x512	0.717	0.638	0.480	0.162	0.708	0.477	0.435	0.156
1024x1024	0.812	1.056	0.912	0.701	0.503	0.787	0.789	0.883
2048x2048	0.756	0.934	1.021	1.120	0.943	0.968	0.982	1.354
4096x4096	1.271	1.842	2.176	2.038	1.910	2.061	1.789	2.163

Based on performance and speedup,the striding method has the smallest different between the LHS and RHS in comparison to chunking and the serial algorithm. A smaller average difference between the LHS and RHS suggests that the solution obtained by the algorithm using the striding design is closer to the true MSE value. This implies that the striding design converges faster and produces more accurate results compared to chunking. The serial implementation has varying execution times depending on the matrix size and number of threads used. The execution times of the parallel implementations are higher compared to the serial implementation for most cases. The execution times of the parallel implementations also vary based on the matrix size and the number of threads used. The performance is not consistently better than the serial implementation. Speedup measures the performance improvement achieved by using parallel processing compared to the serial implementation. In most cases, the speedup values for both chunking and striding designs are less than 1, indicating that the parallel implementations are slower than the serial implementation. However, for some specific cases, particularly with larger matrix sizes, the parallel implementations achieve speedup values greater than 1, indicating improved performance compared to the serial implementation.