Ehi Simon - 14352307 ECEC 413-001
Alisha Augustin - 14389451 04/25/2023

Project 1: Ways to SAXPY

Chunking Method

We implemented a function, *compute_chunk*, that uses the chunking method to parallelize the SAXPY loop. The function checks whether the current thread is the final thread or not by comparing its thread ID to the total number of threads - 1. If it's not the final thread, the function uses a for loop to iterate over the assigned chunk of data, updating the output vector, y, by applying the computation to each element of the chunk in turn. The computation involves multiplying each element of the input vector, x, by a scalar value, a, and then adding the result to the corresponding element of the output vector y. If the current thread is the final thread, the function iterates over the remaining elements that were not processed by the other threads. Finally, the function calls *pthread_exit* to exit the thread and return control to the calling function.

```c
void *compute_chunk(void *args){
    /* Typecast argument as a pointer to the thread_data_t structure */
    thread_data_t *thread_data = (thread_data_t *)args;

    /* Chunking is computed here */
    if (thread_data->tid < (thread_data->num_threads - 1)) {
        for (int i = thread_data->offset; i < (thread_data->offset + thread_data->chunk_size); i++)
            thread_data->vector_y[i] = thread_data->a * thread_data->vector_x[i] +  thread_data->vector_y[i];
    }
    else { /* This takes care of the number of elements that the final thread must process */
        for (int i = thread_data->offset; i < thread_data->num_elements; i++)
            thread_data->vector_y[i] = thread_data->a * thread_data->vector_x[i] + thread_data->vector_y[i];
    }

    pthread_exit(NULL);
}
```

Figure 1: Chunking method implementation in C

In *compute_using_pthreads_v1*, the computation is split into equal-sized "chunks", and each thread operates on its assigned chunk independently. The function first allocates memory on the heap for the required data structures, such as the thread IDs and thread data. Then, it initializes the thread attributes to default values. Afterwards, the function forks the program by creating worker threads using pthread_create. Each thread is assigned a unique thread ID and a set of input data. The threads execute the *compute_chunk* function, which performs the SAXPY computation on each element in the assigned chunk. The compute_chunk function receives a pointer to the structure that contains the thread's assigned chunk and input data. It updates the output vector, y, by iterating over the assigned chunk and applying the computation to each element of the chunk in turn. Finally, the function waits for the workers to finish using the *pthread_join* function and frees the dynamically allocated data structures.

```c
/* Calculate SAXPY using pthreads, version 1. Place result in the Y vector */
void compute_using_pthreads_v1(float *x, float *y, float a, int num_elements, int num_threads)
{
    Create Jira Issue
    /* FIXME: Complete this function */
    pthread_t *thread_id = (pthread_t *)malloc (num_threads * sizeof(pthread_t)); /* Data structure to store the thread IDs */
    pthread_attr_t attributes;      /* Thread attributes */
    pthread_attr_init(&attributes); /* Initialize thread attributes to default values */

    /* Fork point: allocate memory on heap for required data structures and create worker threads */
    int i;
    int chunk_size = (int)floor((float)num_elements/(float) num_threads); /* Compute the chunk size */

    thread_data_t *thread_data = (thread_data_t *) malloc(sizeof(thread_data_t) * num_threads);
    for (i = 0; i < num_threads; i++) {
        thread_data[i].tid = i;
        thread_data[i].num_threads = num_threads;
        thread_data[i].num_elements = num_elements;
        thread_data[i].vector_x = x;
        thread_data[i].vector_y = y;
        thread_data[i].a = a;
        thread_data[i].offset = i * chunk_size;
        thread_data[i].chunk_size = chunk_size;
    }

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

    /* Join point: wait for the workers to finish */
    for (i = 0; i < num_threads; i++)
        pthread_join(thread_id[i], NULL);

    /* Free dynamically allocated data structures */
    free((void *)thread_data);
}
```

Figure 2: Version 1 of using pthreads to calculate SAXPY

Striding Method

For the striding method, the computation is split into *num_threads,* equal-sized "strides", and each thread operates on its assigned stride independently. We first implemented a helper function, *compute_stride*, that contains the computation the threads will perform. It receives a pointer to the *thread_data_t* structure that contains the thread's assigned stride and input data. The function then updates the output vector *y* by iterating over the assigned stride and applying the computation to each element of the stride in turn. This is done using a while loop that increments the offset variable by the stride value until all elements of the stride have been processed.

```c
void *compute_stride(void *args)
{
    thread_data_t *thread_data = (thread_data_t *)args;
    int offset = thread_data->tid;
    int stride = thread_data->num_threads;
    int elements = thread_data->num_elements;

    while (offset < elements)
    {
        thread_data->vector_y[offset] = thread_data->a * thread_data->vector_x[offset] + thread_data->vector_y[offset];
        offset += stride;
    }

    pthread_exit(NULL);

}
```

Figure 3: Implementation of the striding method

The function *compute_using_pthreads_v2*, initializes the data structures and creates the threads for data parallelization using *pthread_create*. The *compute_stride* helper function is

an input argument for this function, showing that the computation starts when the threads are being created. Each thread is assigned a unique thread ID and a set of input data (the input and output vectors, the scalar *a*, and the number of elements and threads). The threads are then started using the *pthread_join* function and their results are combined to produce the final output.

```c
/* Calculate SAXPY using pthreads, version 2. Place result in the Y vector */
void compute_using_pthreads_v2(float *x, float *y, float a, int num_elements, int num_threads)
{
    Create Jira Issue
    /* FIXME: Complete this function */
    int i;

    pthread_t *thread_id = (pthread_t *)malloc(num_threads * sizeof(pthread_t));    /* Data structure to store thread IDs */
    pthread_attr_t attributes;                                                      /* Thread attributes */
    pthread_attr_init (&attributes);                                                /* Initialize thread attributes to default values */

    thread_data_t *thread_data = (thread_data_t *)malloc(sizeof(thread_data_t) * num_threads);
    for (i = 0; i < num_threads; i++) {
        thread_data[i].tid = i;
        thread_data[i].num_threads = num_threads;
        thread_data[i].num_elements = num_elements;
        thread_data[i].a = a;
        thread_data[i].vector_x = x;
        thread_data[i].vector_y = y;
    }

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

    /* Join point: Wait for the workers to finish */
    for (i = 0; i < num_threads; i++)
    {
        pthread_join(thread_id[i], NULL);
    }

    /* Free data structures */
    free((void *)thread_data);
}
```

Figure 4: Version 2 of using pthreads to calculate SAXPY

Performance Comparison

	Reference			Chunking			Striding		
# of Elements	**4 Threads**	**8 Threads**	**16 Threads**	**4 Threads**	**8 Threads**	**16 Threads**	**4 Threads**	**8 Threads**	**16 Threads**
10^4	0.000010s	0.000010s	0.000010s	0.000698s	0.002119s	0.002437s	0.000194s	0.000448s	0.001921s
10^6	0.000876s	0.000904s	0.000842s	0.001285s	0.001252s	0.001398s	0.001684s	0.001818s	0.003283s
10^8	0.073084s	0.075509s	0.075138s	0.070645s	0.074362s	0.056357s	0.131706s	0.144305s	0.313973s