Quang Anh Hoang (qh62@drexel.edu)

Ways to SAXPY

ECEC 622: Parallel Computer Architecture
Professor Naga Kandasamy

1. Introduction

   SAXPY is a function which stands for "Single-precision AX Plus Y", and it takes two vectors of *n* floating-point values *x* and *y* as inputs in addition to a given scalar value *a*. The function multiplies each element in *x[i]* with *a*, then accumulates in *y[i]:*

```
for (i = 0; i < num_elements; i++)
        y[i] = a * x[i] + y[i];
```

   The main task is parallelizing the loop to be executed by multiple threads to gain speedup as the computation is handled concurrently. There are two methods to parallelize SAXPY loop: chunking and striding. These methods will be further discussed in this report.

2. Implementation Design

   2.1. Chunking method

   The first method to parallelizing a task is chunking: the data set is divided into smalled sets and each thread is assigned to compute on one set, and the final result is collected by the end of execution of each thread. For SAXPY loop, each thread is first assigned a thread ID (**tid**) to separate one thread from another; the magnitude of range of elements executed by one thread (**chunksize**), and this number is calculated by dividing the number of elements by the number of threads, after rounding down to the nearest integer.

   Based on **tid** and **chunksize**, each thread first computes the starting index of the array that it will compute: **starting_point = tid * chunksize**

   Since each thread takes N = **chunksize** elements from each array to execute (except from last thread), the last point is N points away from the starting point:

   ending_point = starting_point + chunksize

   For the last thread, the last point is actually the last point of vector, which is, and it makes the last thread execute a few more data points than others:

   ending_point (last thread) **= num_elements - 1**

   Each thread then compute SAXPY function on every element between the **starting_point** and **ending_point** and the result is stored in the *y* vector, which is used to test for correctness later.

   Detailed code for chunking method:

```
void *saxpy_v1(void *args)
{
        ARGS_FOR_THREAD_V1 *thread_data = (ARGS_FOR_THREAD_V1 *) args;
        int i;
        float a = thread_data→a;
        int offset = thread_data->tid * thread_data→chunksize;
        if (thread_data->tid < (thread_data->num_threads - 1)) {
                for (i = offset; i < offset+thread_data->chunksize; i++) {
                        thread_data->y[i] = a*thread_data->x[i] + thread_data→y[i];
                }
        }
        else {
                for (i = offset; i < thread_data->num_elements; i++) {
                        thread_data->y[i] = a*thread_data->x[i] + thread_data→y[i];
                }
        }
        free((void *) thread_data);
        pthread_exit(NULL);
}
```

   2.2. Striding method

   Another method to parallelizing the task is striding: each thread takes one pair of (*x[i]*, *y[i]*)

data points at a time, calculates SAXPY on that pair, then skips to the next pair (*x[i+stride]*, *y[i+stride]*) and repeats until every elements in the array is covered.
The distance between two indexes is called **stride**, and its value is equal to number of threads, as one thread will skips the exact number of threads so that the points in the between are handled by the remaining threads.
Detailed code for striding method:

```
void *saxpy_v2(void *args)
{
  ARGS_FOR_THREAD_V2 *thread_data = (ARGS_FOR_THREAD_V2 *) args;
  int i = thread_data→tid;
  while (i < thread_data->num_elements) {
        thread_data->y[i] = thread_data->a*thread_data->x[i] + thread_data→y[i];
        i += thread_data→stride;
  }
  free((void *) thread_data);
  pthread_exit(NULL);
}
```

3. Result

There are nine different configurations used in this assignment to examine the performance of parallelizing SAXPY loop, with three different values of number of threads: 4, 8 and 16; three different values of number of elements: 10^4, 10^6 and 10^8 elements. Each configuration is run for 100 times and speedups are recorded and shown in the tables below for each method:

	4 threads	8 threads	16 threads
10000 elements	0.00680490	0.00655236	0.00463370
1000000 elements	0.85778595	0.78674630	0.73454650
100000000 elements	1.06911384	1.08509227	1.03662744

Table 1: Average speedup of multi-thread SAXPY function, chunking method

	4 threads	8 threads	16 threads
10000 elements	0.03046196	0.01318314	0.00772902
1000000 elements	0.53065992	0.22189807	0.09580454
100000000 elements	0.29360031	0.14817288	0.06918099

Table 2: Average speedup of multi-thread SAXPY function, striding method

According to the collected result, striding method could not provide speedup over single-thread (higher than 1). This can be explained by false sharing, as in striding method, each thread calculates one data point and updates it on the array, which invalidate the cache line contains that data and results in high time consumption, and it makes striding method's performance worse than serial one.
On the other hand, chunking method shows better performance when dealing with 10^8 elements. It is due to the huge overhead of creating threads, assigning range of computation for each thread, and waiting for all threads to finish. However, the highest speedup the chunking method can gain could get as high as 1.494 (16 threads, 10^8 elements) or 1.487 (4 threads, 10^8 elements). To improve speedup of chunking method, we can use a larger number of elements input into the function to surpass the overhead and cut down the execution time by many times