ECEC 622 – Parallel Programming

Hanh Do Phung hdd29

Tam Phi thp29

Assignment 1: Parallelizing SAXPY Computations

Part 1: Chunking Method implementation

In the source code file "saxpy.c", parallelization of SAXPY is achieved via of two approaches: chunking, and stride method. The implementation is as following:

1)A struct is created to contain each thread information.

```
/* Data structure defining what to pass to each worker thread */
typedef struct thread_data_s {
    int tid;                        /* The thread ID */
    int num_threads;                /* Number of threads in the worker pool */
    int num_elements;               /* Number of elements in the vector */
    float *vector_a;                /* Pointer to vector_a */
    float *vector_b;                /* Pointer to vector_b */
    int offset;                     /* Starting offset for each thread within the vectors */
    int chunk_size;                 /* Chunk size */

} thread_data_t;
```

2) Per user input, the length of vectors to be operated on and the number of threads used for the parallel methods are specified. Then appropriate memory space is allocated for the threads and a simple for loop will pass each thread information into a struct. If user specifies 8 threads, then there will be 8 structs.

```
pthread_t *thread_id = (pthread_t*)malloc (num_threads * sizeof(pthread_t)); //allocate mem for thread
pthread_attr_t attributes; //thread attributes
pthread_attr_init(&attributes); //init thread attributes to default values

int i;
int chunk_size = (int)floor( (float)num_elements / (float)num_threads );//find chunking 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_a = x;
            thread_data[i].vector_b = y;
            thread_data[i].offset = i * chunk_size;
            thread_data[i].chunk_size = chunk_size;
}
```

Pthread_create() function will create threads and pass the information in struct necessary for computation to saxpy_compute() function.

```
for (i=0;i < num_threads;i++)
        pthread_create(&thread_id[i], &attributes, saxpy_compute, (void *)&thread_data[i]);
//wait for worker functions to finish
```

With chunking method, the division of computing tasks for each worker thread is illustrated in the table below:

Y[0]	Y[1]	Y[2]	Y[3]	Y[4]	Y[5]	Y[6]	Y[7]	Y[8]
Thr 0	Thr 0	Thr 0	Thr 1	Thr 1	Thr 1	Thr 2	Thr 2	Thr 2

The implementation of this in C code is as below

```
/* Function executed by each thread to compute the saxpy*/
void *saxpy_compute(void *args)
{
    /* Typecast argument as a pointer to the thread_data_t structure */
    thread_data_t *thread_data = (thread_data_t *)args;
         float a = 2.5;
         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_b[i] = a*thread_data->vector_a[i] + thread_data->vector_b[i];
         }
         else{
                for (int i = thread_data->offset; i < thread_data->num_elements; i++)
                         thread_data->vector_b[i] = a*thread_data->vector_a[i] + thread_data->vector_b[i];
         }
    pthread_exit(NULL);
}
```

Part 2: Striding Method

The process of implementation is similar for the stride method. The distinction is that there is no need to keep track of chunk size or offset. Instead, the number of threads is recorded as the stride distance. The struct is then smaller and contains the below fields.

```
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].x = x;
    thread_data[i].y = y;
}
```

Then in each worker thread, the elements are computed independently, no sharing between each thread and hence there exists no synchronization or critical section. The striding technique is demonstrated in the below table.

Y[0]	Y[1]	Y[2]	Y[3]	Y[4]	Y[5]	Y[6]	Y[7]	Y[8]
Thr 0	Thr 1	Thr 2	Thr 0	Thr 1	Thr 2	Thr 0	Thr 1	Thr 2

The element index for each thread to process is determine in the equation: tid + numthread*k with k € {0, 1, 2, 3, ...} until the end of the input.

```c
void* saxpy_v2(void * args) {
    thread_data_t *tdata = (thread_data_t *)args;
    int offset = tdata->tid;
    int stride = tdata->num_threads;

    while (offset < tdata->num_elements) {
        tdata->y[offset] = tdata->a* tdata->x[offset] + tdata->y[offset];
        offset += stride;
    }

    pthread_exit(NULL);
}
```

Part 3: Testing

When testing with eps = 1e-12, the 2 parallel implementations passed on every run with various vector length and number of threads

```
Checking results for correctness
TEST PASSED
TEST PASSED
```

Part 4: Execution time and Discussion

To evaluate the performance of the parallel implementation of the saxpy loop, information for each run is printed as csv in the format of:

```c
printf("%d, %d, %f, %f, %f\n", num_elements, num_threads, gold_time,\
                               chunk_time, stride_time);
```

The Runs are then automated by a simple bash script and results are in the following table and chart

num_elements	num_threads	gold_time(s)	chunk_time(s)	stride_time (s)
10000	2	5E-06	0.000439	0.000167
10000	4	5E-06	0.000536	0.000206
10000	8	5E-06	0.001734	0.000416
10000	16	5E-06	0.001105	0.000824
10000	32	5E-06	0.002678	0.002522
1000000	2	0.00172	0.001668	0.002328
1000000	4	0.001939	0.001192	0.002334
1000000	8	0.001446	0.001187	0.004637
1000000	16	0.001711	0.001615	0.007657
1000000	32	0.001931	0.002459	0.007688
100000000	2	0.383216	0.171659	0.312447
100000000	4	0.404979	0.121563	0.37413
100000000	8	0.376095	0.18284	0.686507
100000000	16	0.398835	0.108981	1.229025
100000000	32	0.266022	0.116845	1.394674

Table 1. Run time result of serial and parallel implementations of saxpy

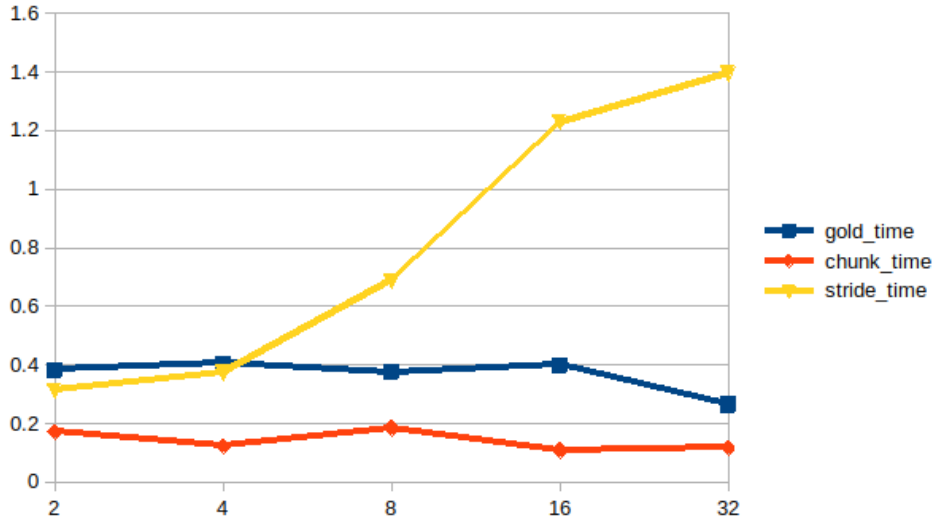

Figure 1: Plot of execution time of performing saxpy on vectors of length of 100 000 000 elements with different number of threads on different implementations.

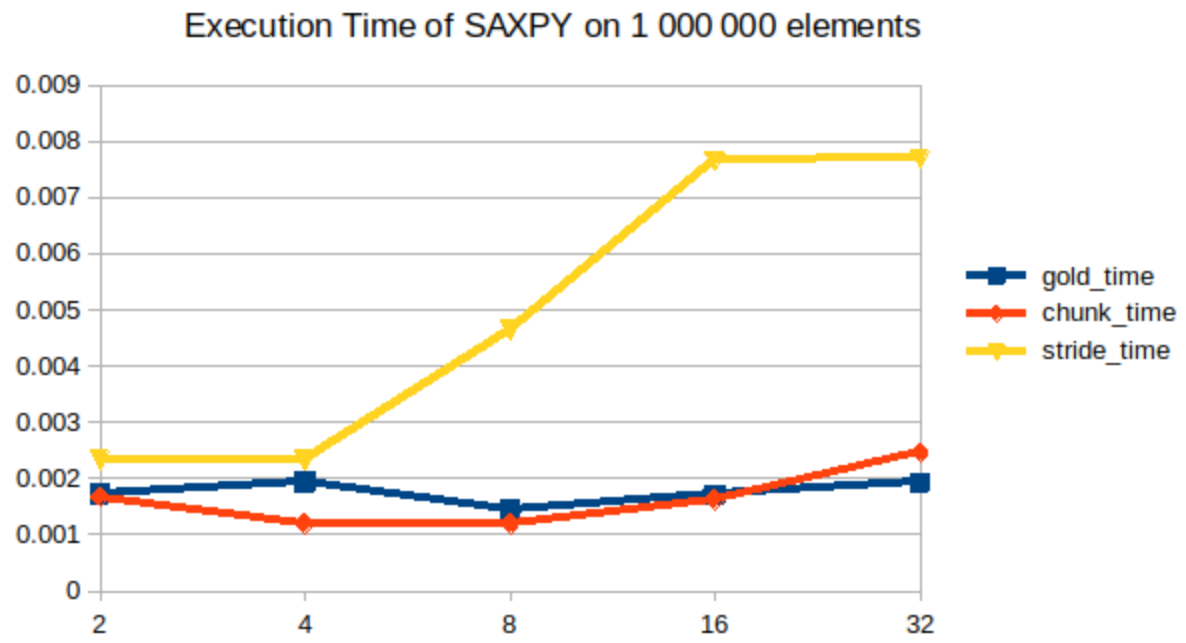

Figure 2: Plot of execution time of performing saxpy on vectors of length of 100 000 000 elements with different number of threads on different implementations.

As we can infer from Figure 1, the **chunking parallel implementation in general take around half of the time the serial method** (gold_time) would take. However, the execution seems not to change much with the change in number of threads. We hypothesize that for runs with larger number of threads, execution time shortened by the less work each thread must perform, should have decrease. However, when considering the increase in overhead of creating more worker threads and the increase in worker thread memory that gets evicted, the deficit in work per worker thread can only compensate for these overheads. Thus, explain the constant execution time of chunking method despite the change in number of threads.

With the striding method, in each time quanta, a worker thread is working on data that might resides outside of their core's L1 level cache, the execution time is significantly larger than that of the chunking method or serial implementation. The larger the number of threads is, the larger the stride is, resulting in a lack of data locality which intensifies as the number of threads grows. The time takes to fetch these data, together with the overhead in memory for each thread and the overhead of creating threads explains the significantly longer execution time of striding method when compared with serial and chunking method.