Assignment 2

Abishek S. Kumar

February 23, 2021

1 Design Implementation

We begin our design by creating a structure identifying all the features for a worker thread. The properties associated with a worker is its ID, the values it contains and the offset and chunk values to be used in the future. We use the said structure further elaborate on our implementation of the algorithms discussed below.

1.1 Chunking

Chunking involves splitting the vectors into smaller (sequential) parts and pass them individually to individual threads that process them simultaneously. For implementation of the SAXPY loop, we split the x and y vectors into chunks corresponding to the number of threads (provided by the user). We then pass the "chunks" of data to the thread which process the SAXPY loop on them, updates the value and then returns. In the end we join each of the threads and combine the returned results.

First thing to note is that we subdivide the array into 'k' chunks so the size of each chunk will be n/k. Granted the extra elements that are not completely subdivided by the k-threads themselves will be taken over by the the k-1th thread. Also there no specific CPU affinity so the threads are provided access to the cores as dictated by the stack.

1.1.1 Pseudocode:

Algorithm 1: Chunking Method

```
for $i \leftarrow 0$ to $k-1$ do
    if $i \neq k-1$ then
        for thread-offset to thread-offset + chunksize do
            perform SAXPY
        end
    else
        for thread-offset to end of elements do
            perform SAXPY
        end
    end
end
```

This method of chunking is fairly fast compared to the serial version because of spatial locality enforced for each of the chunks. Fewer cache misses and a slight overhead for the final thread does speed up the general process of chunking. However it is important to note that overhead is amortized in large datasets alone cannot be typically seen for small datasets.

1.2 Striding

Striding is similar to chunking in that it splits the vectors into smaller pieces. The method differs in the sense where chunking allocates sequential elements to a common thread, striding allocates sequential elements to sequential threads. In other words, if the sequential pieces of data within a vector do not end up being processed by the same processor there will be considerable overhead and cache misses. Once each of the threads are processed they are joined and the returned results are.

1.2.1 Pseudocode:

Algorithm 2: Striding Method

```
for $i \leftarrow 0$ to $k-1$ do
    offset = thread-id
    stride = no. of threads (k)
    while $offset <$ no. of elements do
        perform SAXPY
        offset += stride
    end
end
```

A stride is typically k-1 threads long and can span cache lines. If the stride spans beyond a L1 cache block it might access from L2 but this can lead to more cache misses. It depends on the stride size and the type of data being accessed. For strides less than 8 the number of cache misses is very low suggesting that there are many array elements that can be fit onto the same cache block. For large strides we might be spanning whole cache blocks. It becomes a problem of cache locality.

Also when we exceed the number of elements in the array but still stride we incur considerable overhead to manage each thread. The program works slightly better when the number of array elements are completely divisible by the number of threads invoked, indicating that there are no strided access misses and overhead.

1.3 Mutex Locks

The full form of Mutex is Mutual Exclusion Object. It is a special type of binary semaphore which is used for controlling access to a shared resource. It includes a priority inheritance mechanism to avoid extended priority inversion problems allowing for current higher priority tasks to be kept in the blocked state for the shortest time possible. However, priority inheritance does not correct priority-inversion but only minimizes its effect.

1.4 Barrier Synchronization

A synchronization barrier enables multiple threads to wait until all threads have reached a particular point of execution before any thread continues. Synchronization barriers cannot be shared across processes. Synchronization barriers are useful for phased computations, in which threads executing the same code in parallel must all complete one phase before moving on to the next.

2 Test Cases & Results

The example cases tested with our approach and their corresponding results are displayed below:

- The chunking method with barrier sync speeds up for smaller matrices but goes on indefinitely for large matrices and larger thread groups
- The number of iterations for chunking reduces for smaller matrices but not for larger matrices, striding is not completing the iterations for larger matrices
- The barrier sync is not implemented properly, the program hangs when the barrier sync is implemented before the iteration increment.