Jaitirth Jacob 13CO125

Vidit Bhargava 13CO151

Q7. Build a pthread implementation of multiplication of large matrices (Eg. size could be 250x250 - populate your matrices using a random number generator). Repeat the experiment from the DAXPY question for this implementation. Think about how to partition the work amongst the threads - which elements of the product array will be calculated by each thread?

Additionally, what are the largest size MM programs that can be run? How much do they page fault? What are the largest size (n using matmul) programs that can be run with limited pagefaults? Using this program, investigate the limits of static, stack and heap sizes on your environment. You should also check the ulimit bash built in, and several useful system calls, such as getrlimit() and setrlimt().

Answer:

Methodology Used

Our program considers two matrices A[M][N] and B[N][P] and computes their product C[M][P]. This is done using the trivial O(n^3) mat mul algorithm.

The operation is made multithreaded by dividing the rows between different threads. If there are n threads, each thread will be responsible for npt = (M/n) rows. As all operations on A and B will be read operations, there is no worry about race condition or deadlocks occuring.

This is repeated with varying number of threads used. The execution times are recorded.

If there are n threads,

Thread 1 will be responsible for rows i=0 to i=npt-1

Thread 2 will be responsible for rows i=npt to i=2npt-1

Thread 3 will be responsible for rows i=2npt to i=3npt-1 and so on

The execution times are recorded using chrono functions and a number of iterations are performed to negate variable factors such as processor utilization, etc.

Results

The resulting output of Speedup vs No. of Threads has been compiled into a graph by us in an attached image. The output obtained will not be constant. It depends on processor usage at the given instant and a number of parameters that give varying results. We have tried to negate this by running 4 iterations for each number of threads.

However, there is a trend that is constant amongst all our results for our systems:

For our 2 matrices A[M][N] and B[N][P], the execution is fastest with (M/2 + 1) threads. This remained constant even whilst varying the number of rows and columns. Also there is a marked decrease in execution time from having it as unithreaded to introducing a few more threads as seen in the graph.

Beyond (M/2 + 1) threads the overhead for thread creation is greater than the speedup obtained from having more threads. The trend observed following this point is a linear increase in execution time as the number of threads increase.

These results depend on the parameters of the system executed on as well, such as the number of cores and the efficiency of your operating system. More powerful systems may result in a different value for the optimal number of threads.

Largest Size MM Programs and Limits of Stack, Heap and Static

Here are the execution times for a single matrix multiplication corresponding to certain values of M=N=P (square matrix multiplication):

Size of Matrix	Time
250x250	57.370 milliseconds
500x500	566.793 milliseconds
1000x1000	1.214 seconds
2000x2000	10.112 seconds
2500x2500	25.324 seconds

Increasing the size of the matrix beyond this was resulting in impractical execution times. For my system it can be said that approximately a matrix size of 2500x2500 was the practical limit for square matrix multiplication, based on execution time.

We are able to run for such large values because the arrays were declared as global variables and hence not stored in the stack or the heap. It is stored as static data. **Therefore, the size for static is limited by practicalities of execution time.**

Consider a simple MM implementation where the arrays are declared as local variables as shown in 7b.c. Here even a size of 1000 gives a segmentation fault. This is because we have three 2D arrays of approximately 4MB each which gives a total stack size of 12MB. Here we see that the maximum possible size is approximately 830x830. **This gives 830x830x3x4 = roughly 8MB which corresponds to the ulimit value of 8192KB.**

In 7c.c we allocate the matrices to the heap. **We find that once again as with static the program size is limited by the practicalities of execution time.** Therefore we conclude that the heap does not have a small easily reachable limit, and in theory the heap can take up almost the entire available memory in some cases.