Daniel Garon
Dr Sanjay Rajopadhye
CS 475
2023-03-23

PA3: Polynomial Multiplication

Introduction:

I have analyzed the performance of multiple different implementations of polynomial multiplication. I recorded the execution times for each implementation running sequentially and in parallel with up to 8 threads.

Experiment Setup:

All experiment data was gathered using CS department machine “anchovy” with the following specifications:

Architecture:	x86_64	Model name:	Intel(R) Xeon(R) CPU E5-1650 v4 @ 3.60GHz
CPU op-mode(s):	32-bit, 64-bit		
Byte Order:	Little Endian	Stepping:	1
CPU(s):	12	CPU MHz:	2013.775
On-line CPU(s) list:	0-11	CPU max MHz:	4000.0000
Thread(s) per core:	2	CPU min MHz:	1200.0000
Core(s) per socket:	6	BogoMIPS:	7183.41
Socket(s):	1	L1d cache:	32K
NUMA node(s):	1	L1i cache:	32K
Vendor ID:	GenuineIntel	L2 cache:	256K
CPU family:	6	L3 cache:	15360K
Model:	79	NUMA node0 CPU(s):	0-11

Every trial was conducted seven times, with the lowest and highest recorded times discarded. The mean of the remaining times was taken and used for all comparisons in this report.

PolyMultINQ

Description:

The INQ algorithm multiplies each entry in *q* by every element in *p*, each time storing the result in the corresponding location in *r*. This loop restructuring results in greatly improved sequential execution time.

Optimal Design and Parallelization:

Different iterations of the outermost loop write to the same location in *r*. This creates the possibility of a race condition. This could be dealt with in several ways. I experimented with the following strategies:

- Enclose the loop nest in a parallel block and use an *omp for* pragma to parallelize the inner loop as each inner loop iteration writes to a different location in *r*
- Declare the write to *r* to be *critical*
- Use a *reduction* clause with the corresponding array section as the list item, which is semantically the same as a reduction clause applied to each element of the specified portion of the array

The reduction clause resulted in the best performance. I believe that each thread will make a copy of the specified array section and those are then combined once all of the iterations are complete.

Performance data:

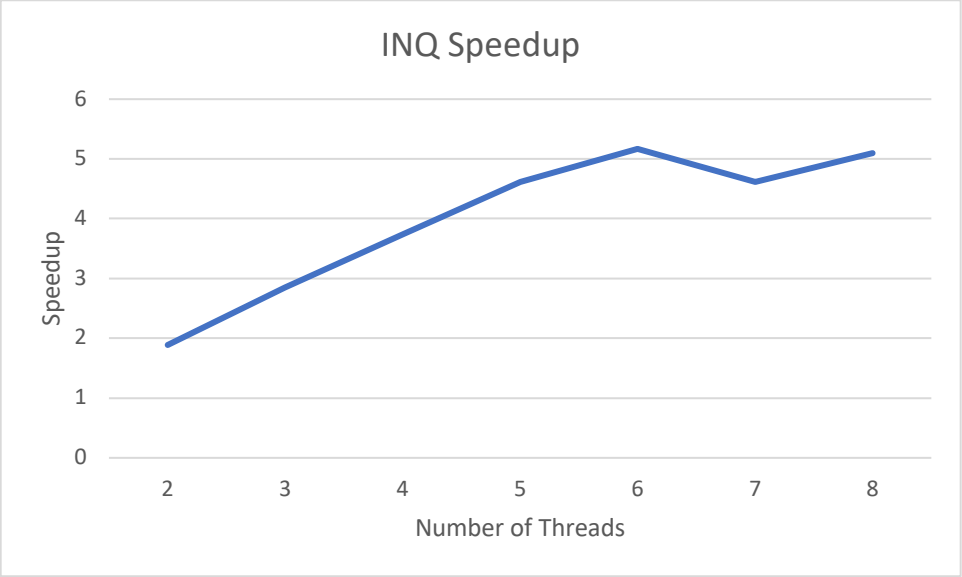

Figure 1: INQ Speedup

Number of Threads	Average Execution Time (s)
1	1.044
2	0.553
3	0.367
4	0.279
5	0.226
6	0.202
7	0.226
8	0.205

Table 1: INQ Execution Time

Observations:

In both the GSQ algorithm and the INQ algorithm, we execute a complete cycle of the inner loop iterating over two of the three arrays with one element from the third array used for the duration of the loop. In GSQ, the *r[i]* is used in each computation with different elements from *p* and *q* each iteration. In INQ, it is *q[j]* that is used for every computation, with the *r* and *p* elements changing.

Given the similar frequency of memory accesses between the two programs, the reason for the performance improvement is likely to lie elsewhere. I suspect that this structure allows the compiler to better optimize the program using vectors. I believe this is due to the independence of each computation in INQ whereas in GSQ, the result of each loop iteration depends upon the result of the previous iteration.

PolyMultOPQ

Description:

The OPQ algorithm iterates over each element of *p* and multiplies each element by every element of *q*. It stores the result in the correct place in *r*: the sum of the index values. The index values correspond to the exponent of each coefficient's respective variable and thus when the variables are multiplied, the exponents are added.

Optimal Design and Parallelization:

The constraints of parallelizing OPQ are the same as those with INQ. I chose the same strategy of parallelization using a *reduction* clause. I recorded nearly identical speedup performance.

Performance Data:

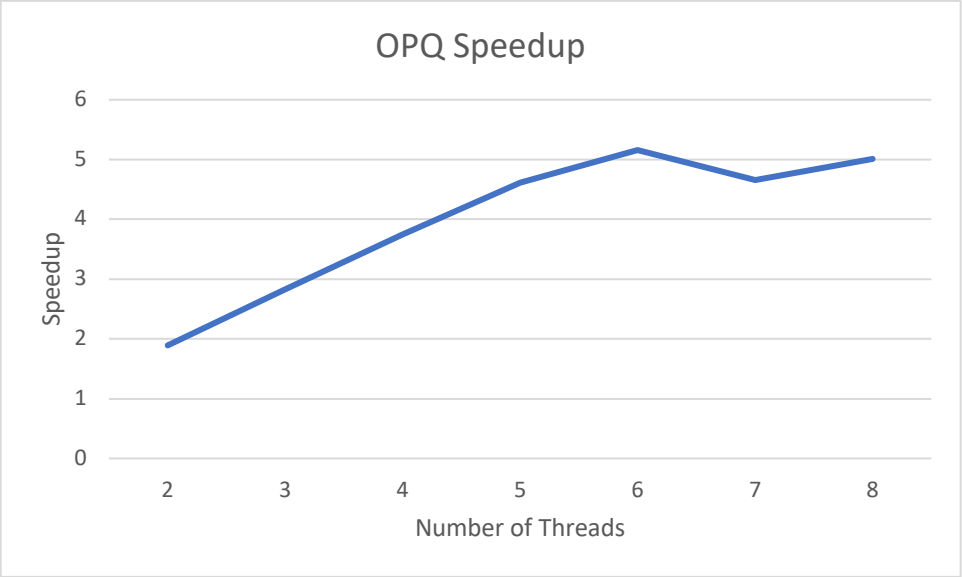

Figure 2: OPQ Speedup

Number of Threads	Average Execution Time (s)
1	1.042
2	0.551
3	0.368
4	0.278
5	0.226
6	0.202
7	0.224
8	0.208

Table 2: OPQ Execution Time

Observations:

The execution times and speedup results of OPQ and INQ are very similar. I believe that once again because the iterations of the inner loop are independent that this allows for higher performance due to compiler optimization using vector operations.

PolyMultBLQ

Description:

The BLQ algorithm divides the iterations of both loops into blocks, effectively subdividing the square iteration space into tiles. For each block of rows, we iterate over the columns, one block at a time before proceeding to the next block of rows. Inside each block, we iterate in row-major order multiplying each element of p by every element of q within the bounds of the current column block.

Optimal Design and Parallelization:

As we iterate over the blocks of columns in any given block of rows, the adjacent blocks will be writing to a common section of *r*. This constrains the parallelization possibilities because of the possible race condition.

Using a *reduction* clause with a specified array section, I was able to parallelize the outermost loop while preserving the integrity of the results. I believe this parallelization results in the best performance as it removes the need to serialize the execution of the independent groups of blocks. This comes at the cost of a higher amount of temporary storage as well as the time to combine the results.

Performance Data:

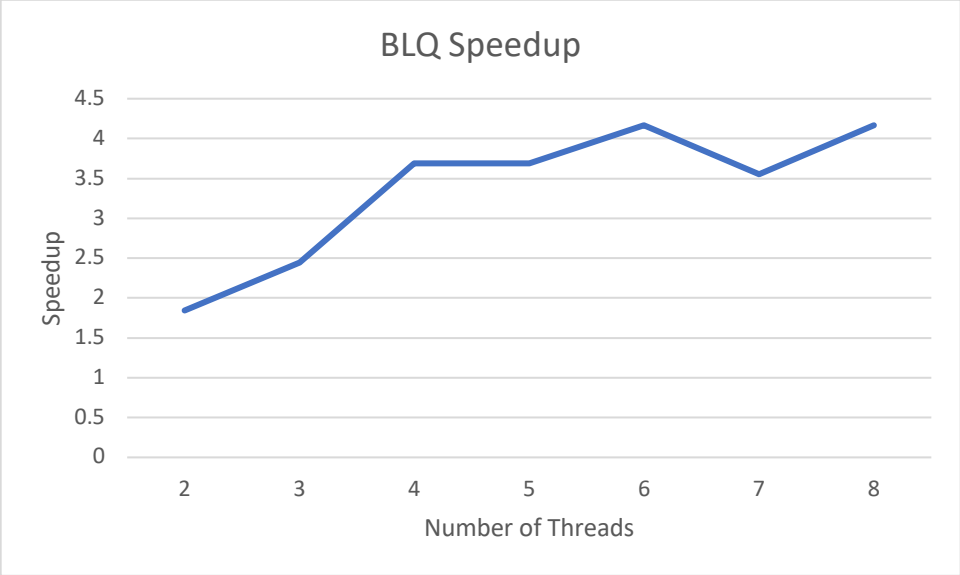

Figure 3: BLQ Speedup

Number of Threads	Average Execution Time (s)
1	0.575
2	0.312
3	0.235
4	0.156
5	0.156
6	0.138
7	0.162
8	0.138

Table 3: BLQ Execution Times

Observations:

I observed that a block size of approximately 4,000 resulted in the highest performance. I believe that this is because it maximizes the number of operations performed on the contents of the L1 cache on this specific machine, which has a size of 32 KB.

For each iteration of the innermost *j* loop, the same *p[i]* will be used for all of the computations. Equal size sections of the *q* and *r* arrays will be used during any given iteration. The elements of the arrays are floats with a size of four bytes. A block size of approximately 4,000 results in 16,000 KB of storage for each array segment, totaling approximately 32,000 KB. This allows for optimal data reuse before we encounter a cache miss.

PolyMultDCQ

Description:

The DCQ algorithm divides the polynomials in half and computes the four resulting polynomials. This is done by recursive calls to PolyMultDCQ. The leaf size for the recursive tree can be passed as an argument which determines when the function will call one of the other previously defined functions to perform the multiplication. The arrays are augmented in place using function arguments to specify the bounds.

Optimal Design and Parallelization:

I determined the bounds for each recursive call and was able to operate on the result array in place without allocating any additional storage. I experimented with multiple parallelization strategies but could not outperform BLQ.

Performance Data:

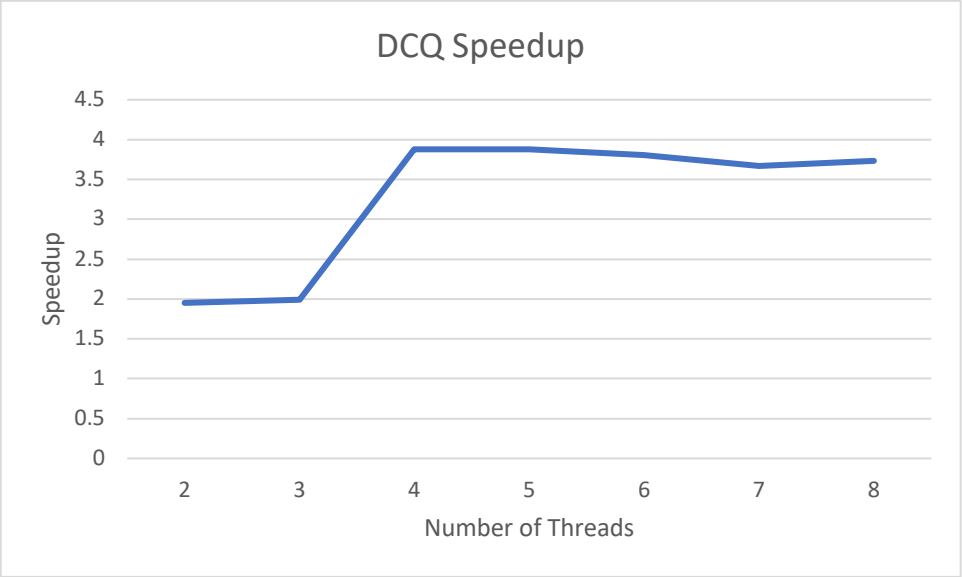

Figure 4: DCQ Speedup

Number of Threads	Average Execution Time (s)
1	0.605
2	0.31
3	0.304
4	0.156
5	0.156
6	0.159
7	0.165
8	0.162

Table 4: DCQ Execution Times

Observations:

I believe that by calling BLQ upon reaching a leaf, I was achieving the same performance but with slightly more overhead for the recursive calls.

The data dependencies are such that two quadrants can be multiplied in parallel without the possibility of a data race. By using the BLQ algorithm, either by calling BLQ or by adding the code to my DCQ function I believe I was just recursing until some point determined by *tune2* where the multiplication for those tiles would be computed.

Conclusion:

For each of the polynomial multiplication algorithms, I observed a decline in speedup with seven threads vs six threads. I believe that the work is evenly distributed among all iterations and so when seven threads are allocated, and the iterations are evenly divided among the threads, the core tasked with the extra share takes longer to complete: $\frac{2}{7} > \frac{1}{6}$.

I observed how the structure of loops allows for significant performance gains due to compiler optimization. I was also able to experiment with performance optimization by tuning some parameters to maximize the resources of the target machine.