Computer Science 133, Lab 2

Michael Wu
UID: 404751542

February 19th, 2019

1 Data Partitioning and Communication

My implementation for this lab is fairly similar to my previous OpenMP implementation. First I took my OpenMP code, adjusted the block sizes to the ones that the TA recommended in discussion, and then I removed all the pragma statements. This produced code that was optimized for a single processor. From there I initialized and broadcasted copies of the input arrays from the root, split the outer for loop iterations based on process rank, and finally reduced the results into the root. So each process contains a copy of both the input arrays and calculates a set of columns in the output array. The main MPI functions that I used for communication were `MPI_Bcast` and `MPI_Reduce`.

2 Communication API Performance

Using blocking communication functions such as `MPI_Send` and `MPI_Recv` would most likely result in worse performance since each process would wait for the communication to finish. Though these functions allow for more control over what data is sent, they would be harder to write code with. I would have to calculate how to split the data across processes in a more explicit manner. The function `MPI_Bsend` is similar except it buffers the data to be sent, allowing the sender to continue with its execution. Thus this is faster than `MPI_Send`, but it uses more memory. The functions `MPI_Isend` and `MPI_Irecv` do not block either, but they do not use a buffer. Thus any modifications to the array being sent can only happen after it has been received, otherwise the communication may have an error. Luckily in my program, the input arrays are not modified at all and the output array is sent after all the calculations are finished. Thus `MPI_Isend` and `MPI_Irecv` would be the fastest since they do not have to copy data to a buffer. In my actual implementation, I chose to use `MPI_Bcast` and `MPI_Reduce` because these functions are optimized and likely offer better performance than any explicit point-to-point communication method I could implement myself. They are also easier to program with.

3 Speedup Summary

The following table summarizes the results of my implementation.

Problem Size	1024^3	2048^3	4096^3
Sequential Speed (GFlops)	0.595169	0.283576	0.195333
Parallel-Blocked Speed (GFlops)	36.0121	72.0702	76.8511
Parallel-Blocked Speedup	60.51x	254.14x	393.44x

The speedup differences occur mainly due to the sequential version performing worse as the problem size increases, since there are more cache misses in the sequential version. The parallel blocked version seemed to increase in performance as the problem size increased, since blocking works better when used on a larger array. Additionally since each process handles more data, there is less communication overhead per array element. The increase in parallel performance between the 2048^3 to the 4096^3 problem size is marginal at best, indicating that my program is operating near peak efficiency by this point.

4 Processor Scalability

The `m5.2xlarge` instance supports up to 4 processes, since there are 4 cores on the machine. Note that we were able to run 8 threads in the previous OpenMP implementation because each core could run two threads. This does not work with processes, which is why one process per core is the most optimal configuration for MPI. The following summarizes the scalability of my MPI implementation with different numbers of processes on the 4096^3 problem size.

Processes	Speed (GFlops)
1	23.6153
2	44.317
4	77.0247
8	73.1959
16	13.5458
32	2.80741

The speed scales almost linearly up to 4 processes, decreases slightly for 8 processes, then falls off rapidly for any more processes. This is most likely due to the overhead of context switching and communication between processes.

5 Optimizations

In this lab I did not add any optimizations that were significantly different from my OpenMP implementation. Most importantly, I permuted the j and k loops and used loop unrolling so that the compiler could multiply 16 elements in parallel. Then I simply split the outer loop iterations across processes. Since the machine has 4 cores, this leads to a speedup of approximately four times compared to the single process speed. I optimized the blocking by using the recommended block size given by the TA in discussion. This was a block with dimensions $(i, j, k) = (64, 1024, 8)$. These values lead to the best cache performance because the long j dimension allows for spacial locality while the overall small block size allows for temporal locality.

6 Results

My MPI program seems to become faster as the array size increases. The speedup over the sequential version increases with array size as well due to cache efficiency. My optimizations from the previous lab allows my program to achieve a speed of 23.6153 GFlops even when only using a single process. With MPI, I was able to speed this up approximately four times by simply splitting loop iterations across four processes. Adding additional processes past four results in performance decreases due to parallelism overheads.

7 Comparison with OpenMP

My MPI implementation required more explicit changes than my OpenMP implementation. I had to manually split a loop and reduce my results. OpenMP only uses compiler directives which is nice because it is easy to maintain the structure of sequential code. Luckily for this lab, I reused most of my optimizations from before so I only had to focus on parallelism using MPI. This made my work more straightforward and less time consuming than the first lab. If I had to deal with the loop unrolling and block optimization, I would probably spend more time coding with MPI than I did with OpenMP.

My best MPI code seems to perform worse than my best OpenMP implementation. With MPI I could achieve a max of approximately 80 GFlops, while with OpenMP I achieved upwards of 100 GFlops. I believe this is due to MPI only being able to use 4 processes while OpenMP can use 8 threads. Additionally MPI is slower because it has communication overheads while OpenMP threads can share data.