CSE 160 Programming Lab 4
12/6/2013
Group ID: 155-589

Authors:
Richard Bull (A09309891)
Weijei Cai (A99501208)

Introduction and Development Procedure

In this project we attempted to develop an understanding using MPI for direct message passing between processors vs. using a shared bus with cache coherency as methods for parallelization and sharing data. Simply put I absolutely hate MPI. While we both see the practicality of MPI and the use for it in highly scalable parallelism, the MPI library seems to have many problems with it. At first these problems were not clearly evident. We started off by analyzing the provided code and figuring out where to insert our parallelism safely and how to safely handle our communication needs for neighbor cells, ghost cells and etc. We were able to breeze through much of the problem with the stripping method, and we were able to do the 2D processor layout with relative ease as well. Our major problems didn't start occurring until we realized the behavior of MPI can be confusing at best. We initially started off the assignment using MPI_Send and MPI_Recv as these functions had MPI_Wait implicitly included, and since we wanted to quickly get working code, we saw our correct results with default values of N and varying processor layouts and we figured we were successful. So we then we went about optimizing our code by either moving things around or getting the most out of our message passing by trying to use up as much throughput on our message passing as possible. When we eventually got our code working predictably and accurately we began testing. This is where things got weird. During our development we tested with relatively small values of N (values < 1000) and we experienced no trouble. When we began formal testing however, our program would hang indefinitely. We scratched our heads for hours with no solution in plain sight. We decided to try using MPI_Isend and MPI_Irecv since everyone else were using these functions. Suddenly our code works flawlessly on all values of N. It turns out that MPI_Send will drop our message if our N is too large without giving any sort of indication that the message was dropped, other than a complete hang. This sort of behavior makes bug checking extremely tedious and I can kind of see why memory coherency is used more often than message passing because of this. While the idea of discrete message passing is brilliant because you are left only transferring absolutely necessary data (vs. saturating an entire bus with a barrier) the behavior and bugs in MPI code are absolutely infuriating.

Testing Procedure

Since the testing scripts were generously provided for us, figuring out which tests to write was not a burden on us for this assignment. We merely ran each of the provided scripts 3 times to make sure we were getting accurate timings However, we did notice that these tests went about testing performance of various methods of processor grid division and the execution time of communication being disabled or not.

Optimizations Done

1. Much of the code in solve.cpp was merged to eliminate unnecessary looping.
2. Created buffers to send single long messages to cut down on send & recieves required for message passing
3. Created submatrices to be used by each process that would later be joined into a larger matrix. Smaller submatrix sizes means there's a better chance of cache locality paying off.
4. Generated L2Norm and Linf using partial sum and MPI_Reduce instead of using a large matrix. This means each process only needs to get the L2Norm and Linf using their own smaller submatrices.

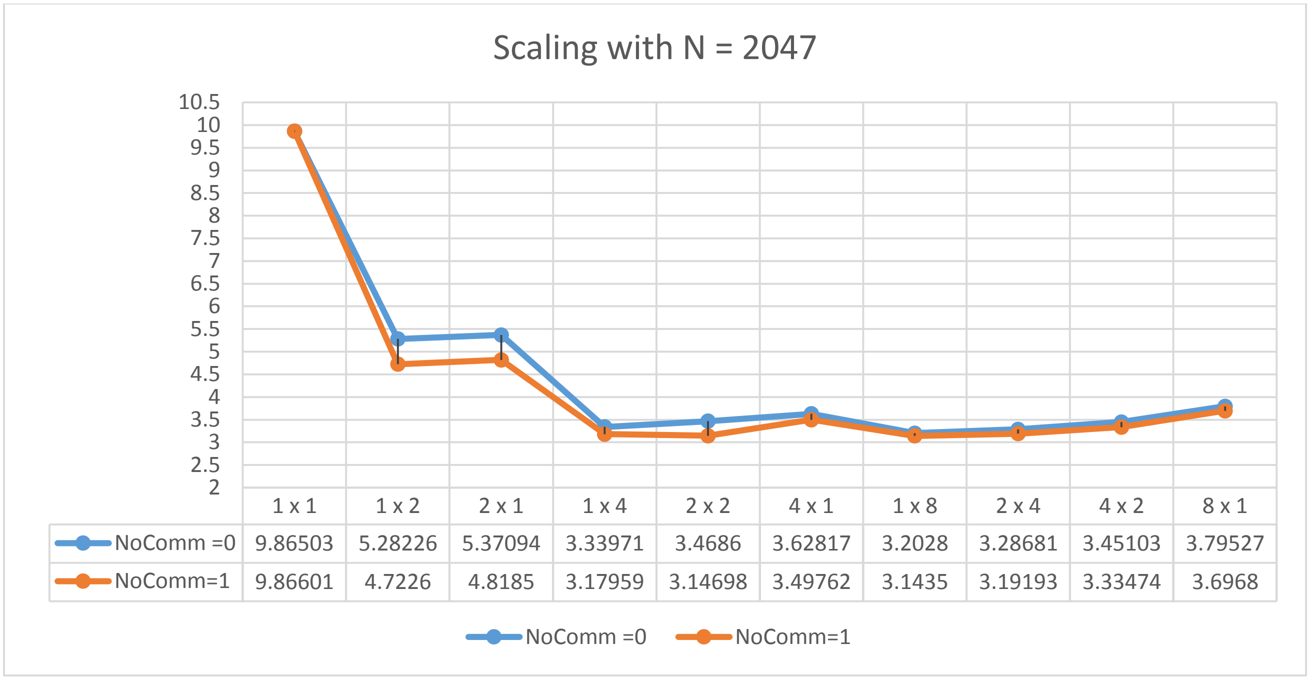
Optimizations Not Done:

1. For simplicity, we have row communication & column communication done separately. However since a 2d grid means each processor will be speaking to at least 2 other processors we could easily do row & column communication at the same time, however this would've added much more complexity to our code, and the payoff would most likely be minimal.

Problems

We ran into one severe problem with our code that we simply cannot figure out. Somewhere in our submatrix allocation in apf.cpp we are not creating submatrices large enough, thus we are walking off the edge of our allocated memory. We've looked through all of our code and cannot find the solution. We however were able to patch this problem by merely adding extra padding to our memory allocation (much more than we need). This allowed us to run our code perfectly fine on all required settings on very small & large values of N.

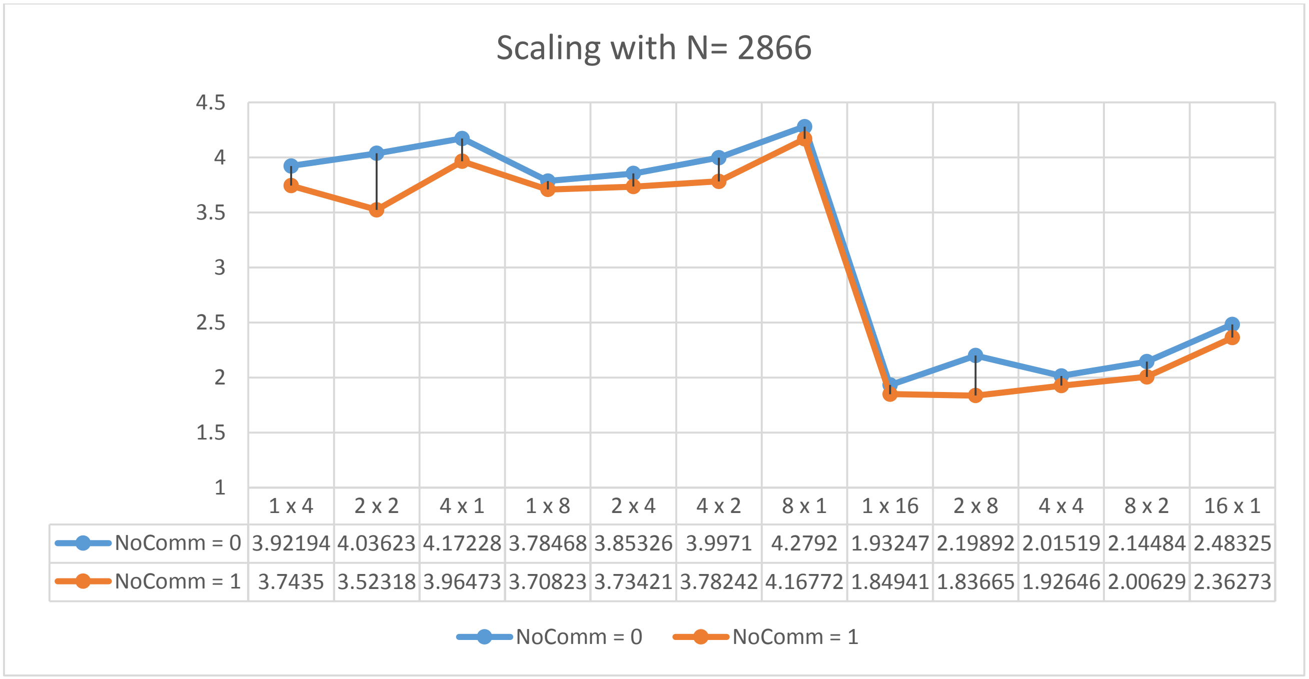
The source of this error could be from two different places. This could be an allocation error and we are simply not dividing our submatrices correctly.. however after checking our code easily 12 times, I doubt this is the source of the error. This could also be an error in init or solve where we are simply walking off the edge of our matrix and it isn't being caught.. which again is unlikely unless when we receive a message from MPI and we try to write it to our allocated space, it is doing something odd that corrupts our memory.



	1 x 1	1 x 2	2 x 1	1 x 4	2 x 2	4 x 1	1 x 8	2 x 4	4 x 2	8 x 1
NoComm =0	9.86503	5.28226	5.37094	3.33971	3.4686	3.62817	3.2028	3.28681	3.45103	3.79527
NoComm=1	9.86601	4.7226	4.8185	3.17959	3.14698	3.49762	3.1435	3.19193	3.33474	3.6968

Results from files:

apf-strong-scale.o8045, apf-strong-scale.o8047, apf-strong-scale.o8048



	1 x 4	2 x 2	4 x 1	1 x 8	2 x 4	4 x 2	8 x 1	1 x 16	2 x 8	4 x 4	8 x 2	16 x 1
NoComm = 0	3.92194	4.03623	4.17228	3.78468	3.85326	3.9971	4.2792	1.93247	2.19892	2.01519	2.14484	2.48325
NoComm = 1	3.7435	3.52318	3.96473	3.70823	3.73421	3.78242	4.16772	1.84941	1.83665	1.92646	2.00629	2.36273

Results from files:

APF-STRONG-16.o8046, APF-STRONG-16.o8049, APF-STRONG-16.o8050

Result Analysis:

For N = 2047 we see a quick distinction between serial and parallel code when switch from 1 core to 2. However, if you notice that code ran with a heavy bias on the –x core alignment we have slightly slower execution times. This is also true for N = 2866 and in fact all executions of our code. This is most likely because when we are communicating column by column our neighbor cells are no longer being sent from contiguous memory. We must go through our matrix and pack all of our ghost cells into a buffer before sending the message which is inherently slower because we cannot take advantage of any cache locality. This is in stark contrast with a heavy –y bias which has contiguous memory when passing messages so it merely needs to say how many bytes it wants to send in the message; and since all of these elements are contiguous in memory cache locality really pays off here. This sort of pattern shows up in all thread counts, as –x grows, our execution time becomes slower because we are no longer able to capitalize on cache locality.

When we turn off communication there appears to be a slight speedup, but nothing all that significant. Although this doesn't seem intuitive, it actually makes a lot of sense. The message passing only accounts for small sections in our matrices, and these small sections are always some constant in regards to N. So on average, our real gain from turning off message passing is just some constant we can slightly reduce. This however comes at the expense of accuracy, so in the end, message passing being turned off does not yield any significant benefits.

While most of these runs have expected outcomes, the two most interesting runs are when we go from 1 core to 2 cores, and when we go from 8 cores to 16 cores. The 1 to 2 core simulations can easily be explained as they are a 2x speed up and it can be easily argued that we are merely dividing up our code evenly on 2 separate processors. The 8 to 16 core simulations however also have roughly a 2x's speed up, but the problem with this is we are only supposed to be using 8 physical cores. The likely scenario is that BANG is allowing us to use another node so we have full access of another 8 cores and the two nodes are now using MPI to communicate. Since the resources for these programs exist solely on two different nodes we are getting the best speed up we can. In contrast with 1 to 2 cores and 8 to 16 cores, we have everything else. Why don't these results have the same sort of speed up? Why do we not see a 2x speed up when we step from 2 cores to 4 or from 4 to 8? The likely scenario is that unlike 1 to 2 cores and 8 to 16 cores, we have not gained access to any additional memory with these runs. With 4 and 8 cores, each core must share the same cache & memory, while with 2 and 16 cores, each processor and node has their own memory to work with as they wish. So in the end, the results that have led to a less than stellar speedup for 4 and 8 cores, is likely because of a memory bottleneck.