Red and Black Gauss-Seidel In MPI

COMP3920: Parallel Computing
University Of Leeds

"Why did the functions stop calling each other?
Because they had constant arguments..."

November 9, 2016

Question 1

For N=128, the table of convergence tolerance against number of iterations:

convTol	iters
10e-1	3713
10e-2	7595
10e-3	11477
10e-4	15359
10e-5	19241
10e-6	23123
10e-7	27005

As the tolerance becomes more stringent in powers by 10, the number of iterations increases by a constant amount. In other words, the amount of iterations is logarithmically related to the convergence tolerance.

Namely, the relationship is captured by:

$$iteration = 3882 * (-logconvTol) - 169$$

which fits through all the data points with 0 error surprisingly.

As for the explaining the relation, as the tolerance becomes more stringent, more iterations are needed to minimize the error suitably hence both the iteration amount and tolerance increase positively.

Question 2

The table of the matrix size N against the final error of vector x:

N	error
64	6.45149e-11
128	0.00240055
256	0.202453
512	0.619469
1024	0.830251

The problem size is roughly linearly proportional to the error for a fixed number of iterations. One explanation to this is because the amount of error terms increases as N increases (since the error is some least squares fit)

Question 3

In chronological order of the code, the routines are:

MPI_Scatter, to efficiently strip partition the matrix A across all ranks since only a slice of the matrix is needed per rank for the subsequent calculations.

MPI_Bcast, to send vector x and b to all ranks since all elements of both vectors are needed during matrix multiplication and other calculations.

MPI_Gather, used to gather on the root rank all the updated values of x from all other ranks. This is needed since the fully updated vector x will need be broadcast to all ranks (synchronization).

MPI_Bcast, to send the synchronized vector x on the root rank to all other ranks before proceeding to the next iteration. This is necessary since each iteration requires the newly updated values of x for Gauss-Seidel

MPI_Allreduce, to gather all the errors from each rank and sum them up. This is needed later on to define the while loop in terms of a convergence tolerance.

As for the conditions for the code to work:

$N \ mod \ matrixSize == 0$, that is, the number of rows per process must be equal. Otherwise calculating the sum of the matrix multiplication will include some extra rows for the last process which has a different amount of matrix rows.

$matrixSize > p > 1$, that is, each process must have at least one row otherwise the initial calculation of $rowsPerProc = N/numprocs$ will yield 0 and hence no rows being sent out to any ranks (a bigger concern though is the fact if there are more process than matrix rows then some of the process will be idle with the current implementation of matrix multiplication).

Question 4

For a fixed $N = 1536$, $iters = 2000$, the tables below illustrate the time taken for various number of processors p and various number of machines:

p	parallel t (seconds)	ideal serial t (seconds)
1	5.242228	2.48321
1	5.231823	2.48321
1	5.202714	2.48321
2	2.847799	4.96642
2	2.835561	4.96642
2	2.884595	4.96642
4	2.413028	9.93284
4	2.400443	9.93284
4	2.426597	9.93284

Table 1: One Machine.

p	parallel t (seconds)	ideal serial t (seconds)
4	7.834163	20.968912
4	7.810804	20.968912
4	7.811297	20.968912
8	9.365273	41.937824
8	9.362385	41.937824
8	9.363021	41.937824

Table 2: Two Machines.

p	parallel t (seconds)	ideal serial t (seconds)
12	11.139044	29.79852
12	11.110144	29.79852
12	10.989798	29.79852

Table 3: Three Machines.

For a single machine, the parallel time decreases as the number of processes increase whereas the opposite holds true for the ideal serial time. This is because as the number of processes increase on the same machine, the time spent computing increases in proportion to the time spent communicating between processes.

We can note an increase in time taken for four processes when using a

single machine vs multiple (two) machines. This is due to the fact that the communication overhead now includes time taken to send information across the network as oppose to solely within the same machine hence is slower.

Additionally, as the number of processes increases on two machines from four to eight, the time taken for the parallel algorithm also increases. One explanation is that the time to communicate between processes (even internally on the same machine) scales much worse compared to the time spent computing on a single process for a fixed matrix size (i.e. the communication overheads increases faster than the time spent computing per rank as p increases for a fixed matrix size).

Lastly, we can see that on the whole, the parallel algorithm performs vastly superior compared to the ideal serial implementation. However, as p increases so does the parallel time taken which isn't favoured for a fixed matrix size. Although, if the matrix size could vary as well then the parallel time would further out perform the ideal serial time.