**SUMMA implementation using MPI**

**Introduction**

Summa algorithm known to be very fast to compute matrix multiplication by leveraging multiple processes to run in parallel. In this report we will implement Summa algorithm using Message Passing Interface (MPI). Here we will compute **C** = **C** + **A** \* **B**, where **A**, **B**, **C** are NXN matrices where maximum N is 216=16384. To use MPI, we will use rXc=P process where r=c and we will go up to 16X16=256 processes.

**Algorithm**

The Summa algorithm is provided below:

for k=0 to n/b-1

for all i = 1 to P1/2

owner of A[i,k] broadcasts it to whole processor row

for all j = 1 to P1/2

owner of B[k,j] broadcasts it to whole processor column

Receive A[i,k] into Acol

Receive B[k,j] into Brow

C\_myproc = C\_myproc + Acol \* Brow

**Design and implementation**

MPI has rich set of libraries which are used appropriately to implement Summa algorithm. After initializing the matrices **A**, **B**, **C**, the first thing we need to do is to divide each matrix to P processes so that each process got N/P1/2 X N/P1/2 elements. Here MPI\_Scatterv function is used to get these local copies of **A**, **B**, **C**. Hence we are dealing with 2D algorithm, it would be nice if we could also rank the processes using virtual topology. To do that, MPI\_Cart\_create and MPI\_Cart\_coords function is utilized. The algorithm needs to broadcast elements row and column wise and to create sub communicator MPI\_Cart\_sub function is used. While doing the actual matrix multiplication Cray LibSci’s dgemm function is used to get best performance. The execution time is calculated using MPI\_Wtime function and MPI\_Reduce function is used to get the maximum execution time from all the processes. MPI\_Barrier function is used to calculate the time properly and also to print properly for debug purpose. Finally to test the correctness of the result, the local copy of C (which is already matrix multiplicated) from all processes is combined using MPI\_Gatherv function and check against the actual result of whole matrix multiplication. The pseudo code is give below:

Use MPI\_Scatterv to create bXb A\_loc, B\_loc, C\_loc from nXn A, B, C

Use MPI\_Cart\_create to create 2D communicator

Use MPI\_Cart\_coords to get the (row,col) index of process from their rank

Use MPI\_Cart\_sub to create row and column communicator

MPI\_Barrier(MPI\_COMM\_WORLD)

tstart = MPI\_Wtime()

for k=0 to n/b-1

MPI\_Bcast(A\_loc, b\*b, MPI\_DOUBLE, row\_comm)

MPI\_Bcast(B\_loc, b\*b, MPI\_DOUBLE, col\_comm)

square\_dgemm(A\_loc, B\_loc, C\_loc)

tend = MPI\_Wtime()

etime = tend - tstart

MPI\_Reduce(&etime, &max\_etime, 1, MPI\_DOUBLE, MPI\_MAX, 0, MPI\_COMM\_WORLD);

Use MPI\_Gatherv to combine C\_loc from all processes to C to check against the actual result of whole matrix multiplication

**Result**

From the above implementation we got the below execution table for N=512, 1024, 2048, 4096, 8192, 16384 and P=1, 4, 16, 64, 256. When larger N (>=215=32768) is tried, OMM exception occurred during run time.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Execution Time of Serial and Parallel Summa Algorithm** | | | | | | |
| **P\N** | **512** | **1024** | **2048** | **4096** | **8192** | **16384** |
| **1** | 0.027344 | 0.178262 | 1.310302 | 9.959123 | 81.19594 | 649.1852 |
| **4** | 0.013443 | 0.089224 | 0.590113 | 4.377407 | 33.87906 | 277.3679 |
| **16** | 0.004904 | 0.030082 | 0.19831 | 1.265524 | 9.083506 | 72.93582 |
| **64** | 0.003363 | 0.014062 | 0.069333 | 0.411711 | 2.655051 | 18.88073 |
| **256** | 0.013668 | 0.02196 | 0.043033 | 0.199499 | 1.094438 | 6.566071 |

We know that, Speedup = Tserial/Tparallel and if we consider the first row of above table as serial execution time(Tserial) and rest of them are parallel execution time(Tparallel), then we got the below speedup table.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Speedup of Parallel Summa Algorithm** | | | | | | |
| **P\N** | **512** | **1024** | **2048** | **4096** | **8192** | **16384** |
| **1** | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 |
| **4** | 2.03407 | 1.997915 | 2.220426 | 2.275119 | 2.396641 | 2.34052 |
| **16** | 5.575856 | 5.925869 | 6.607342 | 7.869565 | 8.938832 | 8.900773 |
| **64** | 8.130836 | 12.67686 | 18.89868 | 24.1896 | 30.58169 | 34.38349 |
| **256** | 2.000585 | 8.117577 | 30.44877 | 49.92067 | 74.18962 | 98.86965 |

Now we know that Parallel Efficiency = Tserial/PXTparallel so using this formula we can get the below efficiency table.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Parallel Efficiency of Parallel Summa Algorithm** | | | | | | |
| **P\N** | **512** | **1024** | **2048** | **4096** | **8192** | **16384** |
| **1** | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 |
| **4** | 0.508517 | 0.499479 | 0.555106 | 0.56878 | **0.59916** | 0.58513 |
| **16** | 0.348491 | 0.370367 | 0.412959 | 0.491848 | 0.558677 | 0.556298 |
| **64** | 0.127044 | 0.198076 | 0.295292 | 0.377962 | 0.477839 | 0.537242 |
| **256** | 0.007815 | 0.031709 | 0.118941 | 0.195003 | 0.289803 | 0.38621 |

Now for the Gflop calculation, we just used the serial execution time because if we use parallel execution time then Gflop comes quite high as the execution time becomes smaller as we increase process size.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Gflops of Summa Algorithm** | | | | | | |
| **P\N** | **512** | **1024** | **2048** | **4096** | **8192** | **16384** |
| **1** | 9.816978 | 12.04678 | 13.11138 | 13.80031 | 13.54146 | 13.54944 |

Next the effect of block size, b is tested. We know that block size is related to matrix size, N and number of processes. So having various N and P, we can check effect of b. The first column is number of processes, first row is block size and the elements are run time for that P and b. The upper left blank means that those run times are very low, thus omitted, and lower right blank means that those are not tested due to large matrix size.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **P\B** | **32** | **64** | **128** | **256** | **512** | **1024** | **2048** | **4096** | **8192** | **16384** |
| **1** |  |  |  |  | 0.027344 | 0.178262 | 1.310302 | 9.959123 | 81.19594 | 649.1852 |
| **4** |  |  |  | 0.013443 | 0.089224 | 0.590113 | 4.377407 | 33.87906 | 277.3679 |  |
| **16** |  |  | 0.004904 | 0.030082 | 0.19831 | 1.265524 | 9.083506 | 72.93582 |  |  |
| **64** |  | 0.003363 | 0.014062 | 0.069333 | 0.411711 | 2.655051 | 18.88073 |  |  |  |
| **256** | 0.013668 | 0.02196 | 0.043033 | 0.199499 | 1.094438 | 6.566071 |  |  |  |  |

From the above table below chart can be drawn:

**Conclusion**

It is evident that the more processes we can use for a given matrix size, the execution time decreases. But from the speedup table, we can see that we could not get linear speedup. If we use more processes, then speedup increases for higher matrix size, N. Similar conclusion can be made for parallel efficiency, maximum achieved for P=4, N=8192 is around 60%. I think this is happening because there are parallel overhead due to row and column broadcast.