Curtis Lu

CS 331-01

11/14/13

Project 1

Project 1’s objective was to create three different algorithms to process matrix multiplication. The three different approaches were: classical, divide and conquer, and Strassen’s method. The test strategies I used for this project were to use powers of two for the sizes of the matrices.The structure of my code was geared towards only same size matrices (nxn). This worked perfectly since my test cases involved powers of two and also the fact that the sizes were even numbers. The values I used were randomly generated numbers ranging from one to one hundred. All algorithms were fairly easy to design given that you follow the instructions for the notes. The theoretical time complexities for the classical and divide and conquer are n3, while Strassen’s method is n2.81. From my data, the divide and conquer was way higher than the classical method. The only part that was difficult was partitioning the matrices into sub matrices that were later used for computing. From the data gathered, the divide and conquer method was significantly higher than the classical method followed by Strassen’s method which was slightly less than divide and conquer method. This was partially due to having to create new multiple instances of sub arrays when partitioning the matrix and copying the values as they are divided and replaced back into the original size. The partitioning was designed out of place instead of in place, which was probably why the divide and conquer and Strassen’s method had worst runtime than the classical method. Thus throughout the course of the tests, the classic method always ran faster than the other two approaches. The strengths of my work were creating the methods and understand the concept behind each one. All algorithms were designed simple to understand as if it were word for word. The program would be best used with small sizes otherwise it would take a long to process the multiplication or complete crash due to overflow. Also, even number sizes for the matrix would work better such as using powers of two since it would partition the matrix better. The constraints were the design of the algorithm when partitioning the matrix to sub matrices. As stated before, the partition was designed as out of place which caused the application to stop running due to overflowing the heap when the size of the matrix was enormous since each time the matrix is divided for instance the divide and conquer method would create eight new instances of sub arrays. Other constraints would be due to RAM size, CPU speed, or other hardware. The most important factor would be the memory used to hold the data. If possible, I would redesign the divide and conquer and Strassen’s partitioning method as in-place so it would not overload the heap space and could potentially run faster due to not creating more instances of sub arrays and copying the data into them.

|  |  |  |  |
| --- | --- | --- | --- |
| Size | Classic (nanoseconds) | Divide and Conquer (nanoseconds) | Strassen (nanoseconds) |
| 2 | 5866 | 6355 | 5377 |
| 4 | 10266 | 28841 | 46439 |
| 8 | 37151 | 228285 | 328495 |
| 16 | 548461 | 2569264 | 3101594 |
| 32 | 1750971 | 7070365 | 8344243 |
| 64 | 4488879 | 42220791 | 44305629 |
| 128 | 26460096 | 335553902 | 299816886 |
| 256 | 214120002 | 2502864026 | 2006631888 |
| 512 | 2209034757 | 19626114887 | 13889952790 |
| 1024 | 22028549334 | 1.60875E+11 | 95620984669 |
| 2048 | 1.95551E+11 | 1.25562E+12 | 6.67515E+11 |

