Parallel Computing

Matrix Multiplication

Daniel Weitman

1. Introduction

The initial goal of this project was to learn about parallel computing and analyze the effects of parallelizing algorithms in regard to time complexity. The first stage was research, for an overview and general background information on parallel computing the provided article found at <https://computing.llnl.gov/tutorials/parallel_comp/> proved sufficient. Next, an algorithm needed was chosen to be parallelized, this became the naive matrix multiplication via nested for loops. This algorithm runs in O(n3) time, due to its nested loops. This algorithm was picked regardless that there are better-optimized matrix multiplication algorithms that employ divide and conquer techniques, such as Strassen’s algorithm. However, the speedup after parallelization was the primary focus, rather than the algorithm’s performance itself. Learning **OpenMP**, an open-source API used for parallel programming, was the next step so computational work could be divided amongst several threads. Then, a video course consisting of around twenty-five lectures covering the syntax and concepts of OpenMP provided enough information for parallelizing the matrix multiplication algorithm. Lastly, work on implementing the algorithm and parallelizing would begin.

1. Parallelism

Parallel systems are designed to yield quicker results to algorithms by dividing the computational work between different processors (nodes). The multiple nodes of these systems can be either built into the hardware, or in some cases, distributed across a network. Contrastingly, concurrent applications differ from parallel applications as they are entirely or partially held within memory, but are not actually being computed at any simultaneous instance. Parallelism, on the other hand, can perform a simultaneous computation per node in the system. Furthermore, a single process can be broken down into sub-processes to be run concurrently, parallely, or utilizing both strategies. Processes are able to divide into these subtasks by creating different threads of control each of which can work on its particular subtask. Threads then work within the same process’s virtual space and perform their subtask independent of the surrounding threads of control. Another important component of parallel systems is their classification sets. Flynn’s taxonomy distinguishes parallel systems by grouping them by the methods they enact on data sets and instruction streams. The four classes are **SISD** Single instruction, single data (Serial computer), **SIMD** Single instruction, multiple data (Modern personal computers), **MISD** Multiple instruction, single data (Multiple jobs, same data set), and **MIMD** Multiple instruction, Multiple data (Modern supercomputers). These classifications are useful in evaluating the utility of different systems depending on the problem that is presented. For illustration, this project uses the **SIMD** class, the instructions are the same for the subsets of matrices but the data held within each subset depends on the data in the matrix itself. Therefore, the testing of the algorithm is performed on a personal computer and in Salisbury University’s High Performance Computing Lab (HPCL).

1. OpenMP

OpenMP is a parallel programming API comprised of compiler directives, library functions, and environment variables and is compatible with C/C++/Fortran. The first aspect of OpenMP, the compiler directives, are used when including the API in the program, but more crucially compiler directives signify to the compiler where a parallel section is going to begin. This is essential for the system to know where it needs to allocate and drop multiple threads of control. Parallel sections are the only blocks of code that should require several threads, the remainder can be run serially. OpenMP also provides users with many helpful functions to simplify the manipulation and execution of parallel sections. This allows programmers to specify exactly how they want to divide the computations between threads. For instance, a programmer can get an ID number for threads to differentiate between them, set the number of threads (assuming they are available within the system), as well as call a function to return a timestamp for parallel timing analysis. Similarly, environment variables in OpenMP refer to the thread counts in a system and other descriptors accessed through OpenMP’s library functions.

1. Matrix Multiplication

Matrix multiplication is an algorithm that has countless applications in numerous fields of study, including but not limited to, computer graphics, linear algebra, and physics. Interestingly, matrix multiplication is defined by an abstract standard and is not a base operation. To begin matrix multiplication some conditions must hold, first the columns of the left-side matrix operand must equal the rows of right-side. Next, the resultant matrix will be in the dimensions of the rows of right-side by the columns of left-side. The computation is then a series of “Dot Products” on the vectors within the matrix. For this project the simplest matrix multiplication algorithm was used yielding O(n3) time, assuming all matrices have nxn dimensions. While, the divide and conquer matrix multiplication algorithms may have a smaller Big-O, the parallel sections themselves speed up each algorithm proportionately. Therefore, this means that any algorithm will effectively illustrate the effects parallel threads have on time complexity. Additionally, the optimal matrix multiplication algorithm has yet to be proven.

1. Timing Analysis

To preface this section the time results of the algorithm are on a PC vs HPCL are displayed below.

**n: Number of rows/columns ||S: Time serially computed||#: Time computed using # threads**

PC

| n | S | 2 | 4 | 8 | 16 |

| 10| 0.000014| 0.000074| 0.000100| 0.000170| 0.000335|

| 100| 0.013553| 0.013145| 0.011914| 0.010769| 0.011664|

| 1000| 11.322939| 12.004302| 11.453130| 11.414855| 11.517882|

| 2000| 147.770741| 144.544584| 135.479945| 134.855870| 138.615018|

| 3000| 575.972633| 556.901232| 510.233801| 502.592970| 512.358843|

HPCL

| n | S | 2 | 4 | 8 | 16 |

| 10| 0.000032| 0.000164| 0.000110| 0.000847| 0.000961|

| 100| 0.025608| 0.015122| 0.008320| 0.005651| 0.007415|

| 1000| 19.140239| 10.315732| 5.137203| 3.840751| 2.391150|

| 2000| 182.503947| 95.196258| 48.192473| 32.737455| 22.871749|

| 3000| 723.096576| 297.284493| 135.146529| 71.187891| 52.004610|

Both of these data tables clearly show the relation between adding threads versus the serial computation on an algorithm. The first thing that someone analyzing these tables may notice is that the serial computations appear to run faster than when run with multiple threads, however upon further inspection, it becomes clear that as the input size of the matrix grows the serial iteration becomes progressively slower than the parallel iterations. The reason that this happens is because the functionality of OpenMP results in a significant amount of overhead in the processors when creating threads and dividing work. This means that for applications with small input, parallelism will actually slow down runtime of a program. The next analytic is the n=3000, S vs 16-thread, in the HPCL table, the serial version takes over twelve minutes to complete and under a minute when using sixteen threads. While this may seem negligible for many applications, the matrices at large technology companies can reach into the billions. Using an n in the billions running in O(n3) instead of 3000 results in wildly different completion times. Lastly, one may observe that the more threads added gives diminishing returns on the completion time. For example, the HPCL S to 2-thread, cuts the time by well over 50%, but the 8-thread to 16-thread in HPCL table cuts the time by under 30%. This provides evidence that the scalability of parallel algorithms is not a solution for improving the time complexity of algorithms. Despite that in real-world applications parallel systems can give much faster results, the best way to improve the runtime of algorithms is to find a better algorithm in general as more nodes only speeds algorithms to a certain point. In conclusion, for specific applications parallel computing can dramatically improve runtime, but adding more processing power will not scale well to the number of nodes added to a system.