UPPSALA
UNIVERSITET

Final Project: Matrix-Matrix Multiplication with Strassen's algorithm
High Performance Programming

Written by: Jakob Gölén

August 19, 2022

1 Introduction

As computers get more and more powerful and more cores can be used, the advantage of running programs over multiple threads also increases. The main goal of this report is to choose a problem suitable for a computer program to solve, write the program to study the problem, optimize it serially and then parallelize it so it can run on multiple cores. The programming language used was C and OpenMP was used to parallelize the code.

2 Problem description

The problem chosen for this report concerns a matrix-matrix multiplication of square matrices, using a method called "Strassen's Algorithm". The method is used to decrease the computational complexity of a Matrix-Matrix multiplication.

The standard way to calculate a matrix-matrix multiplication is by iterating over three loops. If two square matrices of size n are multiplied, the computational complexity is $\theta(n^3)$. Another common way for square matrices where n is a power of two is to use a divide-and-conquer method. The main idea is to split the two matrices to be multiplied (denoted A and B) and the resulting matrix (denoted C) into four sub matrices as seen in equation 1.

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}, C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} \tag{1}$$

The C sub matrices can then be calculated by using equation 2.

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} \cdot B_{11} + A_{12} \cdot B_{21} & A_{11} \cdot B_{12} + A_{12} \cdot B_{22} \\ A_{21} \cdot B_{11} + A_{22} \cdot B_{21} & A_{21} \cdot B_{12} + A_{22} \cdot B_{22} \end{bmatrix} \tag{2}$$

The multiplications in the sub matrices are done recursively, so to calculate the four C sub matrices, eight recursive calls needs to be made. The computational complexity for the divide-and-conquer algorithm is $\theta(n^3)$, i.e. the same as for the iterative way with three for-loops [1].

In 1968, Volker Strassen showed that the divide-and-conquer algorithm can be rewritten so that only seven recursive calls are made. This is done by rewriting the equations to calculate the four C sub matrices used in equation 2. First, seven equations are calculated as shown in equation 3. Each equation contains a single matrix-matrix multiplication, therefore only seven recursive calls are needed.

$$\begin{cases} M_1 = (A_{11} + A_{22}) \cdot (B_{11} + B_{22}) \\ M_2 = (A_{21} + A_{22}) \cdot B_{11} \\ M_3 = A_{11} \cdot (B_{12} - B_{22}) \\ M_4 = A_{22} \cdot (B_{21} - B_{11}) \\ M_5 = (A_{11} + A_{12}) \cdot B_{22} \\ M_6 = (A_{21} - A_{11}) \cdot (B_{11} + B_{12}) \\ M_7 = (A_{12} - A_{22}) \cdot (B_{21} + B_{22}) \end{cases} \tag{3}$$

Next, equation 2 can be written using the seven equations from equation 3, which is seen in equation 4.

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 - M_2 + M_3 + M_6 \end{bmatrix} \tag{4}$$

Since only seven recursive calls are made instead of eight, the computational complexity is reduced to $\Theta(n^{log_2(7)}) \approx \Theta(n^{2.807})$ [1] [2]

The task was to create a program in C which calculates square matrix-matrix multiplications using Strassen's algorithm. An extention to the algorithm was made so that the program could handle square matrices where n was not a power of two by padding the matrices with zeroes if n was odd before splitting the matrices into sub matrices. If n was even however, then the matrices could be evenly divided into four sub matrices and no padding was necessary. The program was then parallelized using OpenMP.

3 Solution method

3.1 Implementation of the algorithm

The program was written to take three input arguments. The first argument was a name of a file where the A and B matrices were stored, where A and B are the two matrices to be multiplied. The input files contained an integer storing the size of the matrices, then the elements of the A matrix row by row, and lastly the elements of the B matrix row by row. The elements of the A and B matrices were stored as doubles. The second argument was the name of an output file where firstly the size of the resulting C-matrix was stored as an integer and then the elements of the resulting C matrix were written row by row, stored as doubles. The last input argument was the desired number of cores for running the program in parallel.

First, the input arguments were saved. Next, the size was read from the input file and memory for the A, B and C matrices was allocated. Next, the elements in the A and B matrices were read from the file and stored in the allocated matrices. A timer started and a function performing the matrix-matrix multiplication with Strassen's algorithm was ran.

The function took the A, B and C matrices as input as well as the size variable. The base case was if the inputs to the function were matrices of size one. Then A and B could be multiplied directly and the result was stored in C.

If the size was not one, a check if the size was odd or even was performed by checking the size modulo 2 and storing the result in a rest variable. If the result was zero, the size was even and the size was divided by two, otherwise one was added to the size and the result was then divided by two. Next, memory for 29 square matrices of the new size were allocated. These were the twelve sub matrices for A, B and C, seven matrices to store the M results from equation 3, and ten matrices storing the ten additions and subtractions from equation 3.

With the memory allocated, the next step was to divide the A and B matrices into their respective sub matrices. If the size had been divisible by two, i.e. the rest variable was zero, this was straight forward as shown in equation 5 with the same being performed on the B-matrix. Size in the equation refers to the size of the sub matrices, so A was of size $2 * size$.

$$\begin{cases} A_{11}[i][j] = A[i][j] \\ A_{12}[i][j] = A[i][j + size] \\ A_{21}[i][j] = A[i + size][j] \\ A_{22}[i][j] = A[i + size][j + size] \\ i = 0, 1, ..., size - 1 \\ j = 0, 1, ..., size - 1 \end{cases} \tag{5}$$

If the rest variable was one, a slight modification had to be made. First equation 5 was performed as normal but i and j went from zero to $size - 2$, i.e. all the rows and columns except the last one was transfered. The last row and column in A_{11}, the last row in A_{12} and the last column in A_{21} could be taken from A. The remaining last column in A_{12}, the last row in A_{21} and the last row and column in A_{22} was set to zero. This gives the same result as if the A matrix had been padded with one extra row and column of zeroes, and then split into submatrices, but with the method used, memory for an extra row and column in the A matrix didn't need to be allocated. The same procedure was done for B and its submatrices.

With the A and B matrices split into submatrices, the next step was to calculate the seven M matrices as seen in equation 3. This was done by first calculating the addition and subtraction, saving the result in one of the temp matrices, and then calling the Strassen's algorithm function recursively to handle the multiplication. Equation 6 demonstrates how M_1 was calculated, but the same principle was used for all seven M matrices. T_1 and T_2 in the equation represents the new A and B matrices to be multiplied when the function is called recursively, with M_1 representing the resulting C-matrix.

$$\begin{cases} M_1 = (A_{11} + A_{22}) \cdot (B_{11} + B_{22}) \\ T_1 = (A_{11} + A_{22}) \\ T_2 = (B_{11} + B_{22}) \\ M_1 = T_1 \cdot T_2 \end{cases} \tag{6}$$

With the seven M-matrices calculated, the four C sub matrices could be calculated with equation 4. The last step was to merge the four sub matrices into C. If the rest variable was 0, i.e. the size of C was even, equation 7 was used to merge the sub matrices into C. Again, size refers to the size of the sub matrices, so C is of size $2 \cdot size$.

$$\begin{cases} C[i][j] = C_{11}[i][j] \\ C[i][j + size] = C_{12}[i][j] \\ C[i + size][j] = C_{21}[i][j] \\ C[i + size][j + size] = C_{22}[i][j] \\ i = 0, 1, ..., size - 1 \\ j = 0, 1, ..., size - 1 \end{cases} \tag{7}$$

If the rest variable was 1, i.e. the size of C was odd, a slight modification was made. First, equation 7 was was performed with i and j going from 0 to $size-2$. Then, the last row and column of C_{11}, the last row of C_{12} and the last column of C_{21} were merged into C, with the last column of C_{12}, the last row of C_{21} and the last row and column of C_{22} being ignored. This gave the same result as if the four sub matrices were merged into C and then the last row and column of C was removed, but the extra row and column in C didn't have to be allocated.

With the C matrix calculated, the memory for the 29 matrices used in the function was freed and the function ended. When all the recursive calls to the function had been completed, the final C-matrix was available. The timer stopped and the time was printed with the built in fprint function. A file with the output filename was created, the size of the matrix was written and then the elements of the C-matrix were written row by row. The output file was closed and the memory for the A, B and C matrices was freed, after which the program ended.

3.2 Optimization of the code

The main optimization of the serial code was to reduce the number of malloc calls and free calls made in the function calculating Strassen's algorithm. Since the matrices could be large, the memory on the stack was insufficient, so the matrices had to be stored on the heap. Allocating and freeing memory on the heap takes a significant amount of time since the heap manager has to look for available space to store the data when data is allocated and clean up unused spaces in the heap and mark them as available when memory is freed. [3]

The number of malloc and free calls were reduced by first changing the way the matrices were stored from 2d-arrays to 1d-arrays. The elements could then be accessed by writing $A[i*size+j]$ instead of $A[i][j]$ where size refers to the matrix size. The 29 matrices could then be stored in a single array of size $29*size*size$. The matrices were stored in sequence, with pointers pointing to the first element of the respective matrix, for example A_{11} began on element 0, A_{12} began on element $size*size$, A_{21} began on element $2*size*size$ and so on. Storing the matrices in this way reduced the number of malloc calls from 29 per function call to only one. Subsequently, only one free call had to be made per function call, freeing all 29 matrices at the same time.

Another optimisation technique was to perform loop fusion. This meant to merge a number of for loops together, thus creating more work per iteration and reducing the loop control overhead, since fewer iterations overall were done. [4] Since the ten additions and subtractions in equation 3 were saved in ten independant matrices, the ten additions and subtractions could be done in a single for loop instead of having seven for loops, one for every calculation of an M-matrix. Similarly, the splitting of the A and B matrix into sub matrices could be merged into one one loop, and the merging of the C-submatrices into C could also be performed in a single for loop.

The final optimization technique was the static keyword used on the function calculating the matrix-matrix multiplication with Strassen's algorithm. This indicated that the function is only used within the .C file, since only one .C file was used. This made

it easier for the compiler to optimize the function calls. [3].

For optimization flags, -O3 enabled aggressive optimisation by the compiler. -Wall was used to catch any errors or warnings from the program. -march=native was used to make the compiler optimize the code for the processor architecture the code is compiled on [5]. -std=c99 allowed some features in the code that were disabled by default on the computer that performed the experiments, one example of this is the declaration of variables inside for loops (e.g. for(int i = 0; i¡size; i++) caused an error by default on the computer running the tests). Originally -faggressive-loop-optimizations were used which defines the bounds of loops and helps the compiler perform loop unrolling and loop exit test optimizations. [6]. The computer performing the experiments didn't recognize the flag and gave an error when the program was compiled. The flag therefore had to be removed.

When the serial program was completed, the program was checked with valgrind to ensure that the program didn't have any memory leaks or errors.

3.3 Parallelization

The program was parallelized with OpenMP due to its ease of implementation. The number of threads was chosen by the user as an input parameter to the program and set with the omp_set_num_threads function. Since the reading from the input file and the writing to the output file was done in sequence, those loops were not parallelized.

In the function which calculates the matrix-matrix multiplication with Strassen's algorithm, all for loops was parallelized with #pragma omp parallel for schedule(static). The static schedule means that the loop is divided relatively evenly between threads [7], so for example if 12 iterations are to be done with 4 thread, each thread does 3 iterations. The static schedule was chosen since the same amount of work is performed per iteration and therefore there is no issue with the load balance, so the work can be evenly divided between threads.

The seven recursive function calls inside the function was parallelized with tasks. First, the parallel section is defined with #pragma omp parallel. Next, a single thread creates the different tasks with #pragma omp single, with the tasks being defined with #pragma omp task, in this case the seven recursive calls, and then the recursive calls are performed in parallel. [8] Each task was performed independantly, i.e. no data was shared between the different tasks, so no critical section was needed.

4 Experiments

All experiments were performed on the "vitsippa" scientific linux machine hosted by Uppsala University, which runs a AMD Opteron (Bulldozer) 6282SE@2.6Ghz on Scientific Linux 6 [9]. First, small scale tests were performed to ensure the correctness of the program. The program was tested for 3x3 and 4x4 matrices. The input matrices and the results are shown in figures 1 and 2, and was produced by simply printing out the

A, B and C matrices with the built in printf function.

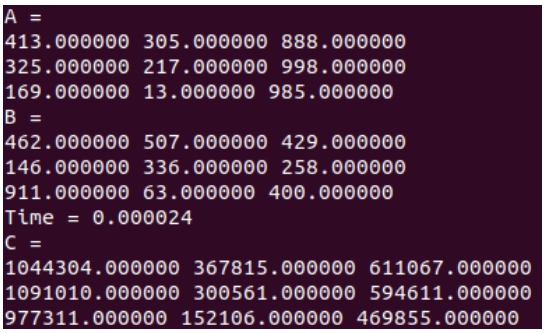

Figure 1: 3x3 matrices calculated with Strassen's algorithm

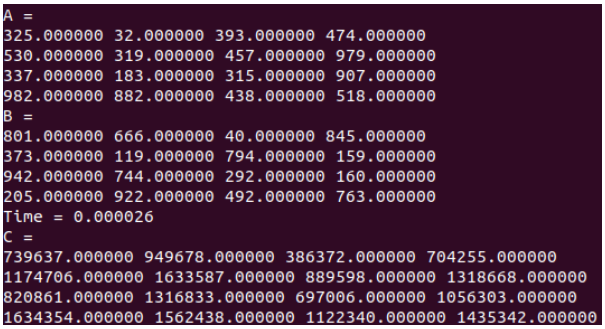

Figure 2: 4x4 matrices calculated with Strassen's algorithm

As can be seen from figures 1 and 2, the program produces the correct result of $A \cdot B = C$ for both the case with the 3x3 matrices and the 4x4 matrices.

The next step was to test the run time of the program for different sizes and different number of cores. First, matrices of power 2 were tested, the measured times can be seen in table 1. When matrices of power 2 were used, the matrices never had to be padded with zeros.

Table 1: Time measurements of matrix multiplication of matrices of sizes of power 2. The times are in seconds using the printed times from the program

Matrix size	64 x 64	128 x 128	256 x 256	512 x 512	1024 x 1024
1 core	0.140	0.977	6.826	47.216	335.705
2 core	0.090	0.632	4.371	30.680	217.571
3 core	0.068	0.479	3.313	23.249	165.256
4 core	0.048	0.336	2.329	16.330	115.484
5 core	0.047	0.330	2.290	15.998	113.876
6 core	0.046	0.320	2.220	15.608	110.450
7 core	0.026	0.176	1.224	8.529	61.034
8 core	0.026	0.177	1.210	8.479	60.767
9 core	0.026	0.176	1.204	8.543	60.200

Next, matrices that were not powers of 2 were tested. These required padding with zeros when calculating the matrix-matrix multiplication with Strassen's algorithm. The results are shown in table 2.

With the data collected, a graph was plotted of how the run time changes depending on matrix size. This can be seen in figure 3.

Table 2: Time measurements of matrix multiplication of matrices of sizes that are not of power 2. The times are in seconds using the printed times from the program

Matrix size	111 x 111	222 x 222	444 x 444
1 core	0.967	6.706	46.968
2 core	0.624	4.312	30.387
3 core	0.476	3.290	22.981
4 core	0.331	2.295	16.062
5 core	0.326	2.266	15.883
6 core	0.316	2.195	15.397
7 core	0.174	1.202	8.516
8 core	0.175	1.200	8.529
9 core	0.174	1.200	8.443

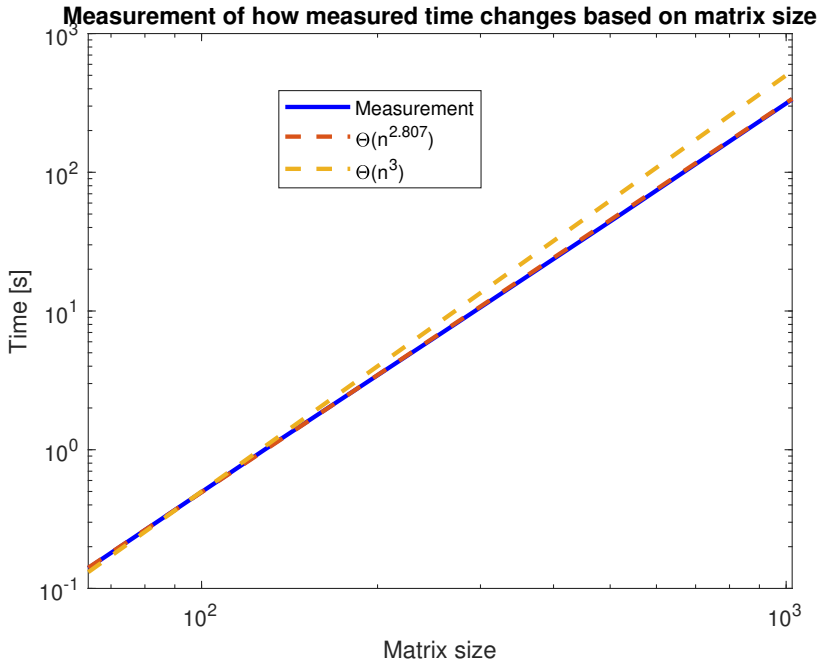

Figure 3: Runtime plotted against matrix size, with the x and y-axis scaled logarithmicly

Lastly, the speedup was plotted against number of cores used for a few matrix sizes. The speedup was calculated by dividing the parallel runtime with the serial runtime, i.e. when only one core is used. The result can be seen in figure 4.

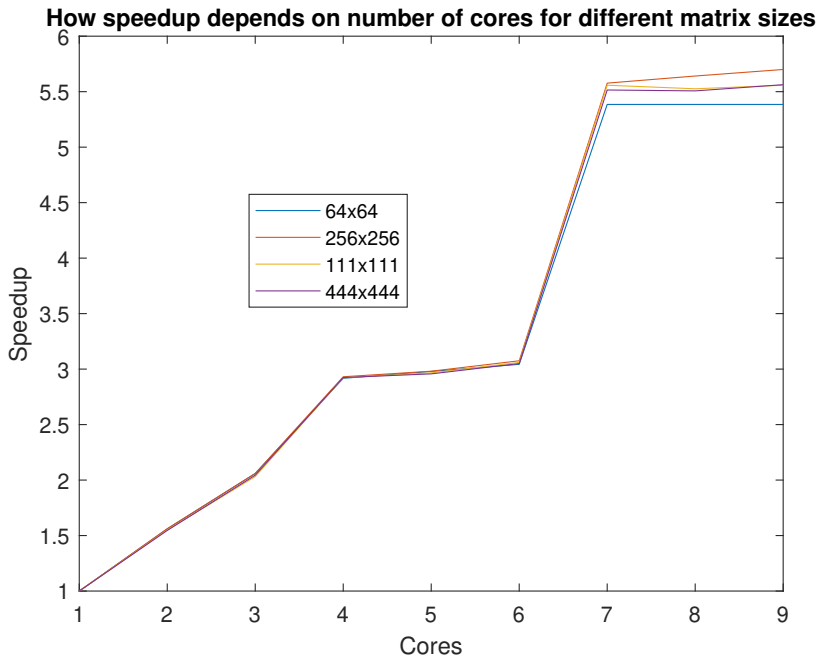

Figure 4: Speedup plotted against number of cores

5 Conclusion

The program calculated the matrix-matrix multiplication with Strassen's algorithm and a correct result was given as seen in figures 1 and 2. No memory leaks or errors were detected on the serial code when scanning the code with valgrind.

From figure 3, it could be seen that the measured time lined up perfectly with the line showing a $\Theta(n^{2.807})$. This was what was expected when using Strassen's algorithm to calculate a matrix-matrix multiplication. The time also increased slower than the line showing a $\Theta(n^3)$, which was what would be the expected increase if the matrices were multiplied the normal way with three for loops. The complexity depending on the matrix size is therefore as expected when using Strassen's algorithm.

From figure 4 an interesting behavour could be seen. The speedup increased up to around 3 when using 4 cores. Then the speedup was negligible between 4 and 6 cores, before the speedup made a large jump at 7 cores where it then leveled out. Tests were done with up to 16 cores but no extra speedup could be seen so the graph only showed up to 9 cores. The behaviour seemed to be the same regardless of matrix size between 1 and 6 cores, with a slight variation between 7 and 9 cores. The reason why 7 cores appeared to be optimal was probably due to the 7 tasks performing the recursive matrix-matrix multiplications inside the function calculating the matrix multiplication using Strassen's algorithm. This could also explain why the speedup was negligible between 4 and 6 cores. When 4 cores was used, the first four tasks were performed by four cores, and then the last three tasks were performed by three cores while one core waited. If 5 cores were used however, the first five tasks were performed by five cores, and then the last two tasks were performed by two cores while three core waited. Since the tasks were done in two "batches" regardless if four, five or six cores were used, the speedup was

minimal between four and six cores, with the small speedup probably being because of the parallel for loops. If seven cores were used, then the seven tasks could be done in one single "batch", so the speedup was very large compared to the speedup at six cores. This also explained why the speedup leveled out at seven cores, since the tasks were done in one "batch" regardless if more cores were used.

One improvement that could be done with the code would have been to activate recursion within OpenMP. By default, OpenMP does not parallelize inside recursive function calls, but this can be changed by calling omp_set_nested(1). The reason why this wasn't done was because if nested parallelism was turned on, the program took longer to run the more cores was used, so the fastest runtime was to run the code in serial. Attempts were done to check the recursion level by having it as an input to the matrix-matrix multiplication function and turn off parallelization after a certain recursion depth was reached by using if-statements, but this still lead to the code running slower in parallel than in serial. One possible solution to this could have been to have two matrix-matrix multiplication functions, where one function was parallel and the other function was completely serial. First the parallel function is called and after a certain recursion level determined by the number of cores used, only the serial function is called. This could make it possible to increase the speedup with more cores, as the current program is optimal when using seven cores.

Another change that would need to be made if the matrix-matrix multiplication were to be calculated for larger matrices was to change the data type of the size variable to a long. Since 29 matrices are allocated at once, the largest matrix-matrix multiplication that can be calculated with the current code are matrices around a size of 8600. If larger matrices are used, the size variable will overflow since it is an int, probably leading to a segmentation fault due to an unmapped location being called.

The last improvement that could be done to the program would be to reduce the number of temp matrices used, since a lot of memory could be used by allocating 29 matrices. Initially when the serial code was written, only two temp matrices were used since only two additions or subtractions were used for every calculation of the M-matrices. If the M-matrices were calculated in serial, the temp matrices could be reused. This posed a problem however when the code was parallelized, since multiple sections could change the temp matrices and thereby giving the wrong input matrix into the recursive function. This was why ten independant temp matrices were used in the parallel code. It is probably possible to reduce this number by first calculating the first five temp matrices, calculating the first four M-matrices in parallel and possibly with nested parallelism enabled, and then reuse the five temp matrices to calculate the last four M-matrices in parallel. This would decrease the required number of matrices from 29 to 24, thus saving memory.

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 2009.

[2] Volker Strassen. Gaussian Elimination is not Optimal. *Numerische Mathematik*, 13:354–355, 1969.

[3] Agner Fog. *Optimizing software in C++*. Technical University of Denmark, 2021.

[4] João M.P.CardosoJosé, Gabriel F.Coutinho, and Pedro C.Diniz. *Embedded Computing for High Performance*. Morgan Kauffman, 2017.

[5] The GCC Team. x86 Options. *GCC, the GNU Compiler Collection*, 2022. https://gcc.gnu.org/onlinedocs/gcc/x86-Options.html (retrieved 2022-08-14).

[6] The GCC Team. Options That Control Optimization. *GCC, the GNU Compiler Collection*, 2022. https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html (retrieved 2022-08-14).

[7] Peter Pacheco. *Introduction to Parallel Programming*. Elsevier, 2011.

[8] Mike Bailey. *Tasks*. Oregon State University, 2022. https://web.engr.oregonstate.edu/∼mjb/cs575/Handouts/tasks.1pp.pdf (retrieved 2022-08-17).

[9] Uppsala University. Linux hosts. https://www.it.uu.se/datordrift/maskinpark/linux (retrieved 2022-08-16).