Faculdade de Engenharia da Universidade do Porto

Parallel and Distributed Computing
Project 1

Computação Paralela e Distribuída– 23/24– *Licenciatura em Engenharia Informática e Computação*
Turma 01 – Equipa 16

Students:
Florentia Diamantopoulou up202311252@up.pt
Maria Mitkou up202311354@up.pt

Problem Description

In this problem we studied the effect on the processor performance of the memory hierarchy when accessing large amounts of data for the first part and for the second we evaluated the performance of a multi-core implementation. For the analysis we used multiplication of two matrices which is a common operation in many engineering applications.

Algorithm Explanation

In this project, we implemented three different multiplication algorithms and measured their time and cache … in order to observe the performance of a single core. The difference between the three algorithms is how they multiply the elements of the matrices. We implemented them in C++ and Java. In more detail the three algorithms are:

1. Matrix Multiplication

This algorithm was already given in a C++ language and we translated it to Java. It is the simple Matrix Multiplication which multiplies one line of the first matrix by each column of the second matrix. In general, if the length of the matrix is N, the total time complexity would be O(N^3).

Algorithm 1: The Naive Matrix Multiplication Algorithm

```
Data: S[A][B], P[G][H]
Result: Q[][]
if B == G then
    for m = 0; m < A; m++ do
        for r = 0; r < H; r++ do
            Q[m][r] = 0;
            for k = 0; k < G; k++ do
                Q[m][r] += S[m][k] * P[k][r];
            end
        end
    end
end
```

2. Line Matrix Multiplication

In the Line Matrix Multiplication algorithm, we tried to optimize the efficiency of the simple Matrix Multiplication approach. Instead of multiplying one element from the first matrix with each column of the second matrix, we optimize by multiplying an element from the first matrix with the corresponding line of the second matrix. This adjustment maintains a time complexity of O(N^3).

For the second part we did the same but we also parallelized parts of the algorithm to optimize it even more using the command #pragma omp parallel for from the PAPI library. By dividing the workload across multiple processing units, parallel algorithms achieve faster execution times compared to traditional sequential methods.

3. Block Matrix Multiplication

In this algorithm we try to optimize the multiplication even more. It divides the matrices in blocks and uses the same sequence of computation as in the second algorithm. The advantage of this approach is that the small blocks can be moved into the fast local memory and their elements can then be repeatedly used.This technique is commonly used in high-performance computing to optimize matrix operations, especially for large matrices. Though the complexity of the algorithm is O(N^3) the same as the others.

Blocked

```
for (i = 0; i < n; i=i+bs)
for (j = 0; j < n; j=j+bs)
{
 for (ii = i; ii < i+bs && ii < n; ii++)
 for (jj = j; jj < j+bs && jj < n; jj++)
  c[ii*p+jj] = 0;
   for (k = 0; k < n ; k=k+bs)
      for (ii = i; ii < i+bs && ii < n; ii++)
      for (jj = j; jj < j+bs && jj < n; jj++)
      for (kk = k; kk < k+bs && kk < n; kk++)
       c[ii*p+jj] += a[ii*n+kk] * b[kk*p+jj];
}
```

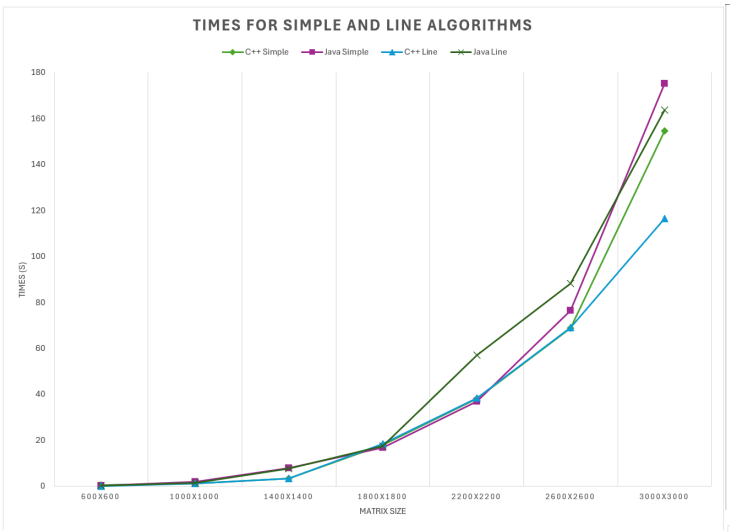
Performance Metrics

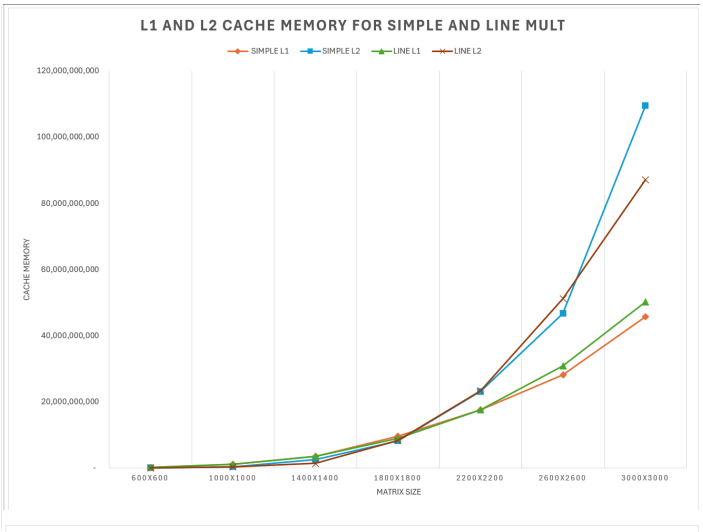
In order to register the performance metrics we used a Performance API also known as PAPI for the C++ version of the algorithms. PAPI provides a standardized interface for collecting performance counter data from diverse hardware and software components, including CPUs. It is a widely-used library for collecting hardware performance metrics, offering portability and ease of use. And for the Java version we could only calculate the first metric To evaluate and compare the performance of the different algorithm versions, we selected the following performance metrics:

1. **Execution Time:** the amount of time taken by each algorithm to complete its execution
2. **Memory Usage:** the amount of memory required by each algorithm during execution.
3. **Cache/Time Efficiency:** measures how effectively the algorithms utilize the cache memory hierarchy. We used this only for part 2.
4. **MFlops:** MFLOPS are units of measurement used to quantify the computational performance of a computer system, particularly in terms of floating-point operations per second (FLOPS), where FLOPS represents the number of floating-point arithmetic operations a computer can perform in one second. We used this formula to calculate it: 2*(n^3)/times

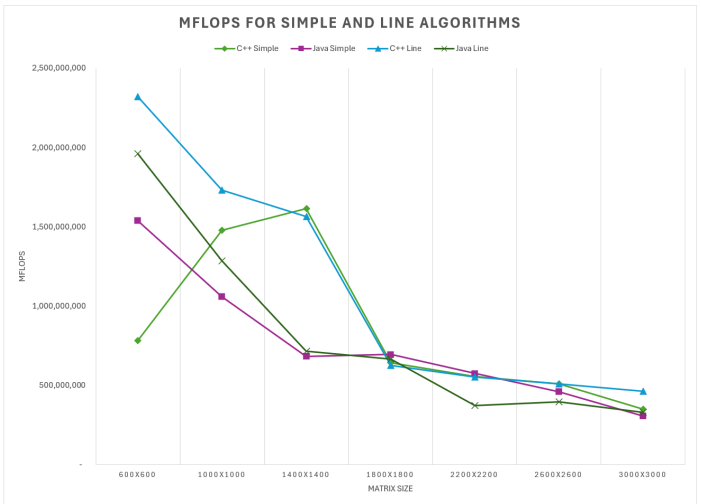
Results an Analysis

Comparison between Simple and Line Multiplication





In terms of time we notice that both Java and C++ have similar performances. The simple and Line Multiplication is similar for smaller matrices and later we observe that line multiplication has better performance. In terms of memory it seems that again for smaller matrices the two algorithms are about the same but as the matrix increases line multiplication seems to handle data better.



TIME C++ SIMPLE

	1	2	3	4	5	6	7	8	9	10	average	MFLOPS
600x600	0.546	0.546	0.542	0.543	0.552	0.579	0.553	0.552	0.547	0.553	0.551	783,602,394
1000x1000	1.213	1.384	1.383	1.364	1.36	1.363	1.366	1.359	1.375	1.359	1.353	1,478,633,742
1400x1400	3.342	3.418	3.391	3.377	3.415	3.382	3.39	3.37	3.479	3.419	3.398	1,614,925,110
1800x1800	18.072	18.125	18.055	18.103	18.236	18.174	18.259	18.141	18.055	17.987	18.121	643,683,743
2200x2200	38.387	38.056	38.358	38.181	38.289	38.226	38.251	38.16	38.367	38.15	38.243	556,867,360
2600x2600	68.744	69.181	68.283	69.301	69.118	68.461	68.449	69.452	68.834	68.686	68.851	510,552,513
3000x3000	156.665	154.22	154.235	154.867	154.603	154.675	154.776	154.5	153.842	154.189	154.657	349,159,302

L1 SIMPLE C++

	1	2	3	4	5	6	7	8	9	10	average
600x600	243502966	243536093	244046477	244053975	244067741	244014932	244058099	243691393	244060972	243858094	243889074
1000x1000	1228535910	1235831208	1235856440	1235823977	1235837293	1235851046	1235863753	1235848665	1235822085	1235838169	1235110854
1400x1400	3508168734	3518785241	3518762777	3518911375	3518934349	3518893416	3518821962	3518742672	3518956983	3518868026	3517784553
1800x1800	9089922794	9094526054	90944593440	90944590218	9094683237	9094509785	9094525968	9094459925	90944597328	9094459869	33649086861
2200x2200	17650302848	17650368173	17650324111	17650356358	17650385833	17650247803	17650259020	17650537333	17650119414	17650450460	17650335135
2600x2600	3089795539	30897049783	30898475867	30897519594	30897966922	30898240722	30898400037	30897529747	30898308184	30898307383	28117159377
3000x3000	50262899017	50263163655	50263494290	50262340757	50263055042	50262883696	50262252246	50263134417	50263907488	5026298513	45739342912

L2 SIMPLE C++

	1	2	3	4	5	6	7	8	9	10	average
600x600	40226883	41171845	39617471	40818239	41140533	39783801	39533857	40975052	40438742	41545962	40525238.5
1000x1000	340921123	313804326	294487361	299571442	309964389	303409880	330443313	291656915	294899041	336400045	311555783.5
1400x1400	1587301424	1318329741	1347377981	1378219087	1314885867	1350542953	1471056757	1390246610	1400214382	1447691766	1400586657
1800x1800	8624163393	7718560605	8229209043	8567839136	8530008063	8436743756	7892290994	8691638874	8895563290	8494225403	8408024256
2200x2200	23409079901	23102235099	23743377133	23097922757	23503755657	23165145099	23404285737	22844349508	23553666363	23276766913	23310058417
2600x2600	51298567829	50794320536	51159151094	51505616060	51131260716	50966522705	51146051227	50697554961	51254004544	51178883968	51113193364
3000x3000	96146416955	95728960435	95321160984	94693149208	96192731564	96043394813	96670027054	94767273076	95702210010	9428332537	87069365664

JAVA TIME SIMPLE

	1	2	3	4	5	6	7	8	9	10	average	MFLOPS
600x600	0.191	0.238	0.251	0.394	0.231	0.261	0.335	0.371	0.249	0.287	0.2808	1,538,461,538
1000x1000	1.79	1.744	1.748	1.924	1.777	1.926	1.97	1.975	2.152	1.995	1.889	1,058,761,249
1400x1400	5.468	5.955	8.932	8.963	9.572	9.634	9.481	9.093	9.076	9.045	8.022	684,118,674
1800x1800	10.917	11.293	18.451	18.403	17.91	18.749	16.825	18.624	17.571	18.499	16.784	694,947,569
2200x2200	34.619	35.507	39.302	40.172	39.295	42.511	35.606	37.284	37.999	35.005	37.029	575,116,800
2600x2600	69.19	82.015	71.404	77.922	70.551	75.026	72.128	78.081	83.037	85.07	76.542	459,251,130
3000x3000	172.477	172.324	176.156	177.196	175.15	170.506	176.648	175.941	178.117	176.549	175.247	308,136,516

C++LINE MULT

	1	2	3	4	5	6	7	8	9	10	average	MFLOPS
600x600	0.187	0.184	0.184	0.188	0.187	0.185	0.187	0.187	0.186	0.187	0.1862	2,320,085,929
1000x1000	1.051	1.175	1.163	1.162	1.164	1.166	1.174	1.165	1.162	1.173	1.1555	1,730,852,445
1400x1400	3.476	3.846	3.545	3.476	3.588	3.342	3.472	3.481	3.423	3.397	3.5046	1,565,941,905
1800x1800	18.172	18.496	18.928	18.857	18.639	18.612	19.337	18.551	18.349	17.996	18.5937	627,309,250
2200x2200	38.619	38.333	38.614	38.306	38.531	38.361	38.46	38.163	38.603	38.563	38.4553	553,785,824
2600x2600	69.541	68.801	69.112	69.26	68.906	68.747	68.964	68.846	69.146	69.488	69.0811	508,851,191
3000x3000	118.4	116.834	116.062	116.438	116.462	116.119	116.762	115.89	116.435	115.907	116.5309	463,396,404

C++ L1 LINE MULT

	1	2	3	4	5	6	7	8	9	10	average
600x600	244779183	244780955	244775940	244782077	244779128	244775989	244779891	244775570	244783466	244783083	244779528.2
1000x1000	1228030403	1224394567	1224407463	1224429920	1224400792	1224405665	1224398975	1224411677	1224393851	1224394185	1224766750
1400x1400	3504568279	3483253095	3483164155	3483151647	3483211701	3483094572	3483172998	3483165991	3483146208	3483161333	3485308998
1800x1800	9089482949	9060241109	9060342363	9060161986	9060259354	9060316320	9060448112	9060110326	9060144428	9060187253	9063169420
2200x2200	17627873238	17628261142	17627936836	17628591392	17628086331	17628361892	17627981130	17628577156	17628223018	17628359701	17628225184
2600x2600	30880598708	30881039571	30879781452	30880107841	30880462651	30881064787	30880780927	30880181942	30880010022	30879845550	30880387345
3000x3000	50295416527	50294590034	50294971268	50294615781	50295123653	50294842485	50295164142	50295777186	50295374159	50295257439	50295113267

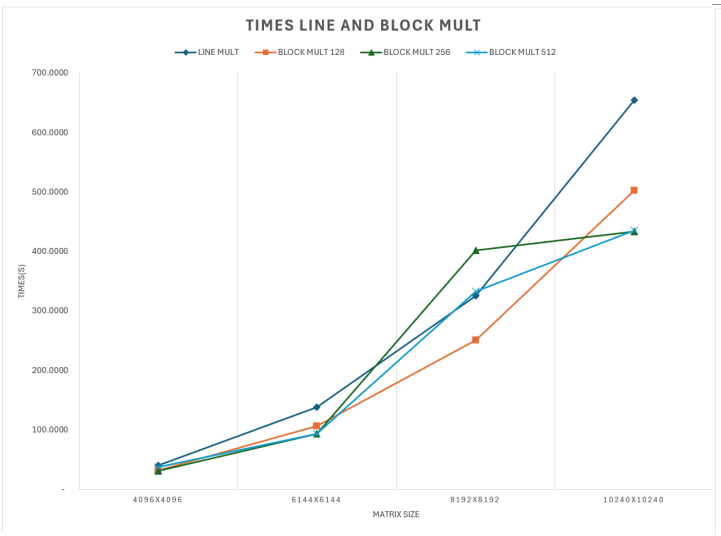
C++ L2 LINE MULT

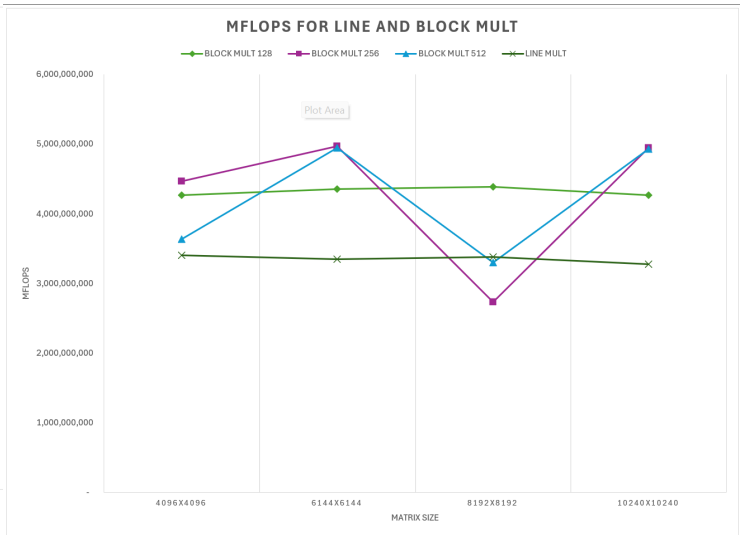
	1	2	3	4	5	6	7	8	9	10	average
600x600	40226883	41171845	39617471	40818239	41140533	39783801	39533857	40975052	40438742	41545962	40525238.5
1000x1000	340921123	313804326	294487361	299571442	309964389	303409880	330443313	291656915	294899041	336400045	311555783.5
1400x1400	1587301424	1318329741	1347377981	1378219087	1314885867	1350542953	1471056757	1390246610	1400214382	1447691766	1400586657
1800x1800	8624163393	7718560605	8229209043	8567839136	8530008063	8436743756	7892290994	8691638874	8895563290	8494225403	8408024256
2200x2200	23409079901	23102235099	23743377133	23097922757	23503755657	23165145099	23404285737	22844349508	23553666363	23276766913	23310058417
2600x2600	51298567829	50794320536	51159151094	51505616060	51131260716	50966522705	51146051227	50697554961	51254004544	51178883968	51113193364
3000x3000	96146416955	95728960435	95321160984	94693149208	96192731564	96043394813	96670027054	94767273076	95702210010	9428332537	87069365664

JAVA TIMES LINE

	1	2	3	4	5	6	7	8	9	10	average	MFLOPS
600x600	0.182	0.216	0.208	0.183	0.195	0.239	0.242	0.24	0.259	0.239	0.2203	1,960,962,324
1000x1000	1.547	1.501	1.531	1.575	1.531	1.535	1.578	1.619	1.586	1.564	1.5567	1,284,769,063
1400x1400	7.323	7.677	7.925	7.958	7.76	7.331	7.809	7.942	7.734	7.238	7.6697	715,542,981
1800x1800	16.624	16.767	16.944	16.831	16.986	17.44	17.524	17.622	18.629	19.359	17.4726	667,559,493
2200x2200	39.621	39.995	42.626	62.597	61.41	62.997	66.987	62.699	66.9	64.149	56.9981	373,626,489
2600x2600	82.404	87.767	88.752	88.562	93.31	88.998	84.811	93.195	79.247	96.085	88.3131	398,038,343
3000x3000	153.556	157.273	172.458	162.106	168.016	169.358	161.838	161.918	167.154	163.726	163.7403	329,790,528

Comparison between Line and Block Multiplication





Here we notice that overall Block Multiplication has a better performance in comparison to Line Multiplication.It's crucial to highlight that in Block Multiplication the size of blocks that we separate the matrix is also very important. Smaller matrices are not always the best options.This underscores the importance of carefully selecting the block size to optimize performance.

C++ TIME LINE

	1	2	3	4	5	6	7	8	9	10	average	r
4096x4096	40.303	40.448	40.979	40.345	40.153	40.34	40.477	40.351	40.221	40.365	40.3982	
6144x6144	138.779	137.945	139.36	138.177	139.158	138.114	138.66	138.156	138.203	138.207	138.4759	
8192x8192	325.264	324.579	325.628	325.008	324.62	325.985	324.647	324.987	325.987	325.574	325.2279	
10240x10240	653.758	644.654	665.149	667.269	652.946	652.713	649.258	655.639	653.741	655.829	655.0956	

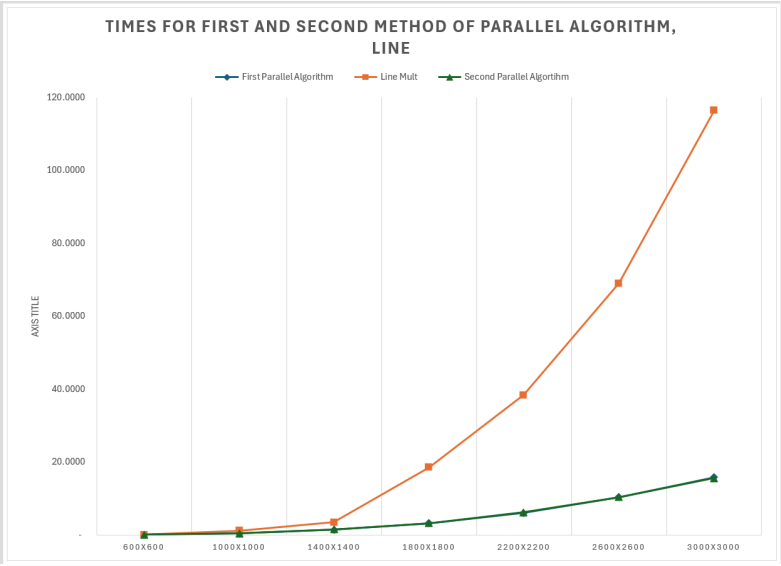
BLOCK MULT TIMES

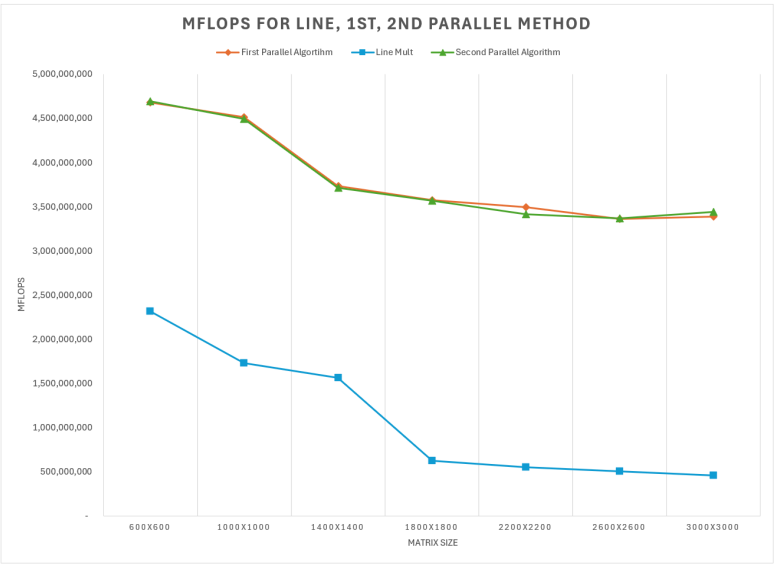
	Block Size 128(1)	Block Size 128(2)	Block Size 128(av)	Block Size 256(1)	Block Size 256(2)	Block Size 256(av)	Block Size 512(1)	Block Size 512(2)	Block Size 512(av)
4096x4096	32.375	32.05	32.2125	29.407	32.067	30.737	37.778	37.76	37.769
6144x6144	106.667	106.197	106.432	93.205	93.4	93.3025	91.5	95.875	93.6875
8192x8192	250.065	251.402	250.7335	404.746	398.673	401.7095	334.102	331.323	332.7125
10240x10240	503.025	503.225	503.125	433.32	434.443	433.8815	429.967	441.542	435.7545

MFLOPS

MATRIX SIZE	BLOCK MULT 128	BLOCK MULT 256	BLOCK MULT 512	LINE MULT
4096x4096	4,266,634,178	4,471,449,832	3,638,935,462	3,402,105,873
6144x6144	4,358,242,521	4,971,533,110	4,951,103,060	3,349,727,050
8192x8192	4,385,180,392	2,737,081,468	3,304,689,868	3,380,742,021
10240x10240	4,268,290,480	4,949,470,415	4,928,196,147	3,278,122,534

Comparison between Line and Parallel Line Multiplication





In these experiments, using parallelism makes line multiplication much faster than before. Line multiplication with parallelism outperforms its sequential counterpart significantly. By dividing the workload across multiple processing units, parallel algorithms achieve faster execution times compared to traditional sequential methods. It shows how powerful parallelism can be in making algorithms work quicker.

1ST PARALLEL METHOD TIMES

	1	2	3	4	5	6	7	8	9	10	average	MFLOPS
600x600	0.094	0.092	0.092	0.092	0.092	0.092	0.093	0.092	0.092	0.092	0.0923	4,680,390,033
1000x1000	0.447	0.442	0.445	0.439	0.44	0.443	0.445	0.441	0.441	0.446	0.4429	4,515,692,030
1400x1400	1.478	1.468	1.472	1.464	1.472	1.462	1.467	1.464	1.473	1.474	1.4694	3,734,857,765
1800x1800	3.265	3.274	3.253	3.303	3.254	3.256	3.256	3.254	3.256	3.256	3.2627	3,574,953,260
2200x2200	6.089	6.105	6.091	6.075	6.1	6.106	6.086	6.105	6.061	6.119	6.0937	3,494,756,880
2600x2600	10.386	10.494	10.454	10.572	10.676	10.542	10.515	10.267	10.223	10.358	10.4487	3,364,246,270
3000x3000	16.090	16.242	16.157	15.813	15.794	15.792	15.815	15.798	15.825	15.802	15.9128	3,393,494,545

2ND PARALLEL METHOD TIMES

	1	2	3	4	5	6	7	8	9	10	average	MFLOPS
600x600	0.093	0.092	0.092	0.092	0.092	0.091	0.092	0.092	0.092	0.092	0.092	4,695,652,174
1000x1000	0.442	0.444	0.449	0.4400.441	0.444	0.446	0.447	0.441	0.448	0.441	0.444666667	4,497,751,124
1400x1400	1.475	1.474	1.475	1.465	1.485	1.493	1.473	1.478	1.478	1.481	1.4777	3,713,879,678
1800x1800	3.245	3.284	3.296	3.287	3.299	3.273	3.253	3.25	3.248	3.25	3.2685	3,568,609,454
2200x2200	6.321	6.191	6.39	6.229	6.02	6.996	6.031	6.054	6.042	6.055	6.2329	3,416,708,113
2600x2600	10.292	10.270	10.379	10.471	10.137	10.430	10.446	10.509	10.761	10.61	10.4305	3,370,116,485
3000x3000	15.585	15.566	16.312	15.631	15.57	15.624	15.537	15.558	15.689	15.634	15.6706	3,445,943,359

Conclusions

In summary, our project explored the impact of memory management and parallelism. We implemented Matrix Multiplication, Line Matrix Multiplication, and Block Matrix Multiplication algorithms, analyzing their time complexities and memory usage. We observed that Line Matrix Multiplication, especially when parallelized, outperformed the simple Matrix Multiplication and sequential Line Multiplication algorithms, particularly for larger matrices. Additionally, Block Matrix Multiplication demonstrated superior performance overall, underlining the importance of block size selection for optimization.We also highlighted the significance of memory management techniques in enhancing program efficiency, even without parallel computing.

In conclusion, our project underscores the critical role of memory management and algorithm optimization in maximizing processor performance.