CPD

PROJECT 1

Performance evaluation of single core and multi-core implementations

3LEIC11

Turma 11 Grupo13
Alexandre Felgueiras de Morais up201906049
Francisco Pinto Bettencourt up202105288
Nuno Jorge da Costa Ramos up201906051

Problem Description

This project objective was to experience the impact of memory hierarchy on computational time when operating on a large scale of data. The Matrix Multiplication was used to observe this effect due to its high demand for computational power.

Algorithms

Matrix multiplication is very high demanding in computational power as such code optimization can achieve great performance improvements in performance. We developed three algorithms (Simple matrix multiplication, Line multiplication and block multiplication) with multiple in order to evaluate the significance of such optimizations.

C++ was used to implement all three algorithms. And for comparison we implemented Simple multiplication and Line multiplication in Java in order to achieve the best similarity in code and give the best possible comparison.

Simple Matrix Multiplication

The simple matrix multiplication code was given to us and it works by multiplying one line of the first matrix by each column of the second matrix.

```
for(i=0; i<m_ar; i++)
{   for( j=0; j<m_br; j++)
    {   temp = 0;
        for( k=0; k<m_ar; k++)
        {
            temp += pha[i*m_ar+k] * phb[k*m_br+j];
        }
        phc[i*m_ar+j]=temp;
    }
}
```

Line Matrix Multiplication

We implemented an optimized algorithm that multiplies each element of a row from the first matrix by the entire corresponding row of the second matrix, accumulating the results. This approach aims to improve memory access patterns and overall performance compared to the default implementation.

```
for (int i = 0; i < m_ar; i++)
{
    for (int k = 0; k < m_ar; k++)
    {
        for (int j = 0; j < m_br; j++)
        {
            phc[i * m_ar + j] += pha[i * m_ar + k] * phb[k * m_br + j];
        }
    }
}
```

Block Matrix Multiplication

The reasoning behind this algorithm is to divide the original matrices into various smaller matrices, each of these matrices are calculated independently and added in the end. This improves the cache usage and reduces memory access overhead with the downsize of not being very efficient when multiplying smaller matrices.

.

```
for (a = 0; a < m_ar; a += bkSize) {
    for (b = 0; b < m_ar; b += bkSize) {
        for (c = 0; c < m_br; c += bkSize) {
            for (int i = a; i < a + bkSize; i++) {
                for (int k = b; k < b + bkSize; k++) {
                    for (int j = c; j < c + bkSize; j++) {
                        phc[i*m_ar+j] += pha[i*m_ar+k] * phb[k*m_br+j];
                    }
                }
            }
        }
    }
}
```

Line – Parallel 1

This first multi-core implementation uses parallelization in the outer loop, specifically in the first loop.

```
#pragma omp parallel for private(i,k,j) shared(pha,phb,phc)
for(i=0; i<m_ar; i++)
{    for( k=0; k<m_ar; k++ )
    { for( j=0; j<m_br; j++)
        {
            phc[i*m_ar+j] += pha[i*m_ar+k] * phb[k*m_br+j];
        }
    }
}
```

Line – Parallel 2

This second multi-core implementation uses parallelization in the inner loop, specifically in the last loop.

```
#pragma omp parallel private(i,k)
for(i=0; i<m_ar; i++)
{    for( k=0; k<m_ar; k++ )
    {
        #pragma omp for
        for( j=0; j<m_br; j++)
        {
            phc[i*m_ar+j] += pha[i*m_ar+k] * phb[k*m_br+j];
        }
    }
}
```

Performance Metrics

To compare and evaluate the differences between the various implementations and to ensure that the data does not represent some huge discrepancies, all the testing was done on the same computer.
With the help of PAPI and OpenMP the metrics used were the following:

1. **Execution Time**
   - Time it took for the algorithm to run, in seconds.
   - Used as a baseline for other performance metrics, speedup, and efficiency.
2. **L1 and L2 cache misses**
   - Indicates how many times data was not found on L1 and L2 cache which requires fetching in slower memory.
   - High cache misses suggest bad memory access patterns that can lead to higher execution times, something that was seen in our earlier implementation of block multiplication.

These next metrics all pertain to the parallel implementations

3. **MFlops**
   - Million Floating-Point Operations per second, measures computational performance.
   - Can be calculated using:

     $$MFLOPS = \frac{2 \times Size^3}{Execution\ time \times 10^6}$$
   - Higher values can indicate a more efficient computation per unit time.
4. **Speedup**
   - Compares the single core implementation to its multi-core one
   - Can be calculated using:

     $$S = \frac{T_{serial}}{T_{parallel}}$$
   - Expected to increase with the number of threads used.
5. **Efficiency**
   - Calculates how well the parallel implementation works with multiple threads
   - Can be calculated using:

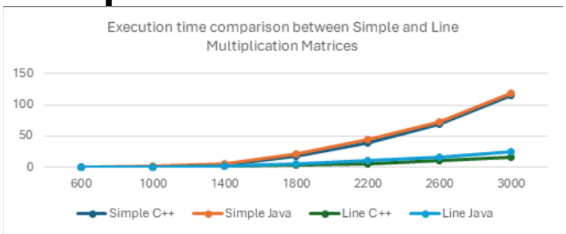
$$E = \frac{S}{Number\ Threads}$$

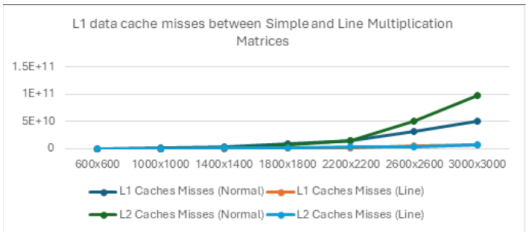
- Drops in efficiency with higher threads numbers might suggest synchronization or memory overheads.

Results and Analysis

To ensure reliable results and to prevent any abnormalities from occurring, each test was conducted five times, except for larger matrix sizes in the parallel implementations. For matrix sizes of 4096, 6144 and 8192, tests were run three times, while for 10240, twice.

Comparison between Simple and Line Matrix Multiplication:



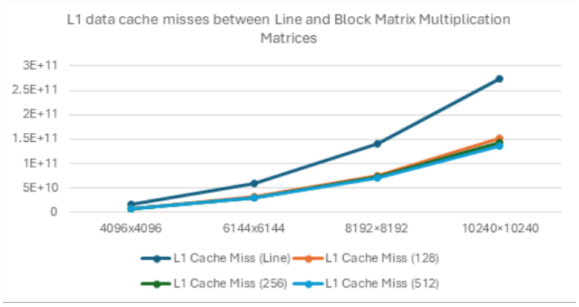


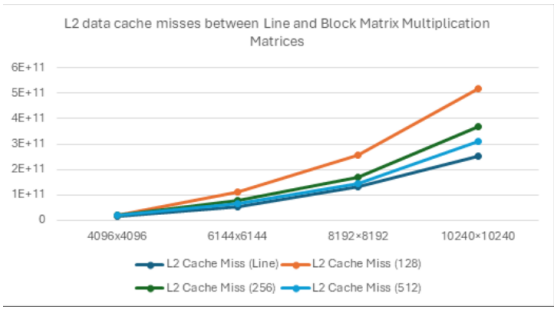
As we can observe both algorithms obtained fairly equal results when analyzing the time it took for them to run in both C++ and Java, with the slight discrepancy being explained by the memory overhead in JVM and memory efficiency and compiler optimizations offered by C++.

There is, however, a clear standout when it comes to the execution time and cache misses of the Simple algorithm in comparison to Line algorithm, as the latter ends up having not only faster speeds with fewer cache misses. Since the Line algorithm processes the matrix line by line, it makes use of spacial locality when accessing memory resulting in way lower cache misses.

We can also observe the rising execution time and cache misses in the Simple Algorithm, which become significantly more pronounced at larger matrix sizes. This trend is much less evident in the Line Algorithm, as it maintains a much lower growth rate.

Comparison between Line and Block Matrix Multiplication:





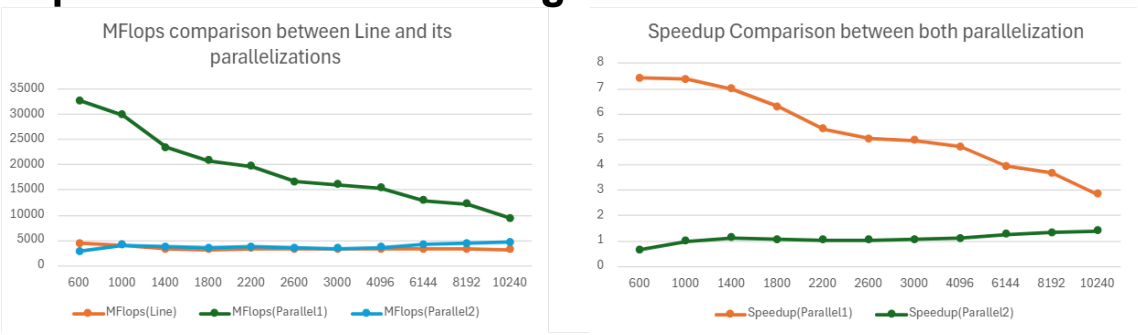
Block multiplication generally got better results when comparing execution times with the line multiplication method.

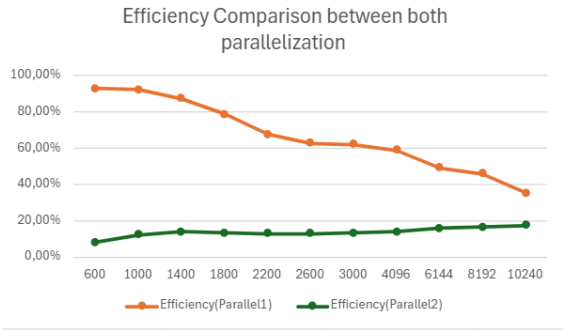
When comparing cache misses the line multiplication method performed a lot worse in L1 and quite better in L2(across all block sizes). These results were as expected as Line multiplication keeps reloading lines into L1 causing high L1 cache misses, however rows stay in L2 longer once it's loaded it is reused.

Within the block multiplication method, we got better results with 256 blocks, as the cache misses in L2 were significantly lower when compared to 128 blocks but only slightly worse than 512. This means that while not adding much overhead like 512 blocks it still got a lot better results than 128 blocks reaching a better balance.

We had an exception in the matrix sized 8192x8192 were 128 blocks performed better than any other method and 256 blocks and 512 blocks had a much worse performance, even when comparing to the line multiplication method, if comparing execution times. The cache misses results do not corroborate this, and we still don't have a concrete answer for why this happened.

Comparison between multi-core and single-core implementations of Line algorithm





Our data demonstrates that the multi-core implementation utilizing 8 threads significantly enhances performance across all matrix sizes. However, as the matrix size increases, the efficiency of the parallel implementation decreases. This trend suggests that for sufficiently large matrices, the performance gap between single-core and multi-core implementations would narrow, potentially converging.

Notably, the parallel implementations consistently achieved higher MFLOPS compared to the single-core approach, regardless of matrix size. The speedup, which ranged from 7.42 for the smallest matrix (600x600) to 2.83 for the largest (10240x10240), indicates that the performance benefits of parallelization diminish as matrices grow larger. This pattern is consistent across other metrics, reinforcing the observation that the performance difference between implementations becomes less pronounced with increasing matrix size.

Between the two parallelization strategies, the first approach delivered superior results overall. However, the second approach showed a gradual improvement in performance as matrix sizes increased, hinting that it might become more efficient for extremely large matrices. This suggests that the optimal parallelization strategy may depend on the specific matrix size and computational context.

Conclusions

In conclusion this project successfully demonstrated the importance of code and hardware optimizations. It was possible to reduce the execution time significantly by just optimizing the code and made us realize it could go much further. But even so, only when we introduced parallel computing was it possible to get reasonable times on the larger matrixes. This highlights the crucial role of both algorithmic efficiency and hardware capabilities in tackling computationally demanding problems.

Annexes

1. Single Core

1.1. Normal Multiplication

1.1.1. Execution Time C++

Size	Execution Time 2	Execution Time 3	Execution Time 4	Execution Time 5	Execution Time
600×600	0.195	0.190	0.192	0.193	0.190
1000×1000	1.291	1.175	1.065	1.102	1.294
1400×1400	3.293	3.622	3.288	3.316	3.495
1800×1800	17.873	17.925	18.343	18.473	18.369
2200×2200	39.503	39.249	39.218	39.444	39.365
2600×2600	68.821	70.698	68.619	69.997	70.013
3000×3000	114.827	114.418	114.786	114.574	116.414

1.1.2. Execution Time Java

Size	Execution Time 2	Execution Time 3	Execution Time 4	Execution Time 5	Execution Time
600×600	0,206	0,217	0,21	0,22	0,219
1000×1000	1,941	2,291	2,301	2,253	1,63
1400×1400	5,614	5,243	5,423	6,164	5,772
1800×1800	20,624	20,26	22,886	22,827	22,595
2200×2200	45,095	45,207	44,363	41,352	43,73
2600×2600	73,981	71,722	74,505	75,087	71,334
3000×3000	118,145	119,01	123,705	117,439	120,024

1.1.3. L1 Cache Miss C++

Matrix Size	L1 DCM (Run 1)	L1 DCM (Run 2)	L1 DCM (Run 3)	L1 DCM (Run 4)	L1 DCM (Run 5)
600×600	244737694	244780842	244743190	244740358	244784386
1000×1000	1240323822	1230817224	1228014552	1223782856	1224418867
1400×1400	3433922982	3498060112	3504458896	3509873372	3508218593
1800×1800	9090038251	9046696764	9046720705	9046971863	9046997741
2200×2200	17650384942	17649959394	17650453179	17649904363	17650270129
2600×2600	30897309159	30896848938	30897639504	30897582536	30896995363
3000×3000	50301796228	50302132794	50301740688	50302253488	50303091112

1.1.4. L2 Cache Miss C++

Matrix Size	L2 DCM (Run 1)	L2 DCM (Run 2)	L2 DCM (Run 3)	L2 DCM (Run 4)	L2 DCM (Run 5)
600×600	38821336	39439840	41147503	39164427	40059612
1000×1000	256849003	307873391	277245855	199145994	216534278
1400×1400	1129011551	1147403458	1319106978	1154095566	1300117713
1800×1800	6564797949	7085948321	7082613104	7819595763	7509284093

2200×2200	22773943401	22737535809	22668169381	23352960933	23124029154
2600×2600	49918028201	51191696361	51238298500	50955227864	51075757816
3000×3000	97308518769	98487686141	96617930640	96951418243	97093629116

1.2. Line Multiplication

1.2.1. Execution Time C++

Size	Execution Time 2	Execution Time 3	Execution Time 4	Execution Time 5	
600×600	0.092	0.093	0.100	0.101	0.102
1000×1000	0.484	0.481	0.512	0.485	0.509
1400×1400	1.597	1.647	1.563	1.683	1.632
1800×1800	3.520	3.466	3.536	3.538	3.468
2200×2200	6.418	6.315	6.382	6.337	6.309
2600×2600	10.550	10.925	10.509	10.889	10.937
3000×3000	16.574	16.552	16.631	16.624	16.968

1.2.2. Execution Time Java

Size	Execution Time 2	Execution Time 3	Execution Time 4	Execution Time 5	
600×600	0,121	0,121	0,145	0,118	0,14
1000×1000	0,813	0,804	0,803	0,802	0,802
1400×1400	2,486	2,511	2,551	2,502	2,435
1800×1800	5,353	5,287	5,278	5,497	5,446
2200×2200	10,096	9,991	9,954	10,107	10,081
2600×2600	16,025	16,283	16,416	16,164	16,367
3000×3000	24,944	24,727	24,938	25,006	24,917

1.2.3. L1 Cache Miss C++

Matrix Size	L1 DCM (Run 1)	L1 DCM (Run 2)	L1 DCM (Run 3)	L1 DCM (Run 4)	L1 DCM (Run 5)
600×600	27110369	27197395	27106286	27108678	27109424
1000×1000	125706551	125752512	125711324	125708769	125714983
1400×1400	346185185	346092182	346082768	346095854	346088038
1800×1800	745515967	744908351	744863627	744871601	744895127
2200×2200	2073568405	2073759119	2073826495	2073863805	2073908220
2600×2600	4412972629	4412773743	4412982154	4412779922	4412751014
3000×3000	6780798577	6780789596	6780571939	6780585482	6780557419

1.2.4. L2 Cache Miss C++

Matrix Size	L2 DCM (Run 1)	L2 DCM (Run 2)	L2 DCM (Run 3)	L2 DCM (Run 4)	L2 DCM (Run 5)
600×600	57347515	57464344	58327520	57506445	57478652
1000×1000	261350568	261602820	259420355	264377774	262461930

1400×1400	698226426	688235960	701224630	680239795	686609365
1800×1800	1407295193	1419917020	1413834841	1415882190	1422660458
2200×2200	2535788886	2548305812	2535406944	2542902160	2547702186
2600×2600	4152245146	4110739849	4157792692	4107303377	4123966569
3000×3000	6307006686	6308922217	6419056914	6418377731	6413449875

1.2.5. Execution Time Higher Values

Size	Execution Time	Execution Time 2	Execution Time 3	Execution Time 4	Execution Time 5
4096×4096	41.437	41.783	41.832	42.812	42.011
6144×6144	141.956	140.024	140.379	140.884	140.506
8192×8192	333.311	341.660	336.135	338.206	340.530
10240×10240	676.764	680.306	679.130	675.605	683.185

1.2.6. L1 Cache Miss Higher Values

Matrix Size	L1 DCM (Run 1)	L1 DCM (Run 2)	L1 DCM (Run 3)	L1 DCM (Run 4)	L1 DCM (Run 5)
4096x4096	17550740771	17560271934	17548975623	17573482941	17555893412
6144x6144	59577964857	59618472652	59553287134	59642938475	59591528743
8192×8192	1,40349E+11	1,40464E+11	1,40277E+11	1,40502E+11	1,40461E+11
10240×10240	2,73734E+11	2,738E+11	2,737E+11	2,7375E+11	2,7372E+11

1.2.7. L2 Cache Miss Higher Values

Matrix Size	L2 DCM (Run 1)	L2 DCM (Run 2)	L2 DCM (Run 3)	L2 DCM (Run 4)	L2 DCM (Run 5)
4096×4096	16197326895	16208475632	16185743218	16219238475	16193452847
6144×6144	54711622242	54748937623	54698754321	54773521894	54721093857
8192×8192	1,30524E+11	1,31736E+11	1,306E+11	1,315E+11	1,309E+11
10240×10240	2,52787E+11	2,528E+11	2,5275E+11	2,5277E+11	2,5278E+11

1.3. Block Multiplication

1.3.1. Execution Time

Size	Blocks	Execution Time 1	Execution Time 2	Execution Time 3	Execution Time 4	Execution Time 5
4096×4096	128	32.203	34.182	31.871	30.774	33.912
4096×4096	256	31.276	33.298	32.581	31.087	32.305
4096×4096	512	37.988	39.105	36.512	38.398	37.245
6144×6144	128	126.478	128.254	125.567	124.892	127.561
6144×6144	256	120.006	119.856	120.738	118.245	121.134
6144×6144	512	132.717	134.522	130.865	133.434	132.782
8192×8192	128	303.496	309.745	305.612	302.439	307.569
8192×8192	256	503.351	510.234	497.498	508.123	504.487
8192×8192	512	410.869	420.245	411.978	413.156	418.034
10240×10240	128	579.005	590.412	577.289	585.367	589.245

10240×10240	256	523.429	527.789	518.612	523.734	529.567
10240×10240	512	556.212	566.589	554.398	559.512	561.374

1.3.2. L1 Cache Miss

Matrix Size	Blocks	L1 DCM (Run 1)	L1 DCM (Run 2)	L1 DCM (Run 3)	L1 DCM (Run 4)	L1 DCM (Run 5)
4096×4096	128	8664408280	8666921060	8666911148	8670831930	8670771331
4096×4096	256	8655333434	8655330251	8655592104	8655419802	8655085496
4096×4096	512	8653500077	8654263676	8653902153	8654781921	8653998748
6144×6144	128	33000384038	33008934712	33011256789	33005678234	33009214356
6144×6144	256	30763799338	30772045612	30768932845	30767198210	30770256894
6144×6144	512	29639993707	29648021569	29645032457	29643389231	29646821573
8192×8192	128	74257882545	74269042312	74263579134	74261235698	74267284589
8192×8192	256	71987527516	71995486234	71992348657	71990127389	71993894561
8192×8192	512	70471333537	70478561234	70475489678	70473528901	70477043256
10240×10240	128	1,53004E+11	1,53024E+11	1,53016E+11	1,5301E+11	1,53019E+11
10240×10240	256	1,4239E+11	1,42411E+11	1,42401E+11	1,42396E+11	1,42406E+11
10240×10240	512	1,37093E+11	1,37112E+11	1,37105E+11	1,37098E+11	1,37109E+11

1.3.3. L2 Cache Miss

Matrix Size	Blocks	L2 DCM (Run 1)	L2 DCM (Run 2)	L2 DCM (Run 3)	L2 DCM (Run 4)	L2 DCM (Run 5)
4096×4096	128	18506767679	18352060917	18346906905	18492068294	18348578199
4096×4096	256	18188941245	18192899392	18176823521	18315089246	18216210008
4096×4096	512	17330573507	17500189641	17458023789	17389045712	17421098345
6144×6144	128	1,12398E+11	1,12511E+11	1,12476E+11	1,12432E+11	1,12489E+11
6144×6144	256	78165296201	78203456879	78189034712	78178024567	78192561345
6144×6144	512	66175633004	66213456789	66198546712	66187561234	66202347890
8192×8192	128	2,55548E+11	2,55612E+11	2,55589E+11	2,55568E+11	2,55599E+11
8192×8192	256	1,6766E+11	1,67724E+11	1,67698E+11	1,67679E+11	1,6771E+11
8192×8192	512	1,45786E+11	1,45832E+11	1,4581E+11	1,45798E+11	1,45821E+11
10240×10240	128	5,17369E+11	5,17429E+11	5,17406E+11	5,17389E+11	5,17419E+11
10240×10240	256	3,68058E+11	3,68125E+11	3,68102E+11	3,68089E+11	3,68116E+11
10240×10240	512	3,10765E+11	3,10812E+11	3,10789E+11	3,10771E+11	3,10801E+11

2. Multi-core

2.1. Parallel 1

2.1.1. Execution Time

Size	Execution Time 2	Execution Time 3	Execution Time 4	Execution Time 5
600	0.01358	0.01297	0.01348	0.01313	0.01296
1000	0.06646	0.06446	0.06530	0.06516	0.07372
1400	0.23103	0.24612	0.22608	0.23040	0.22795
1800	0.57674	0.53473	0.54699	0.58072	0.54519

2200	1.10172	1.37928	0.99103	1.37545	1.02231
2600	2.03643	2.06853	2.10236	2.33132	2.17759
3000	3.37160	3.11261	3.59388	3.62323	3.07985
4096	7.97434	9.25642	9.48218	-	-
6144	41.35256	32.88801	32.87214	-	-
8192	100.58449	90.89112	84.12406	-	-
10240	216.77812	263.32941	-	-	-

2.1.2. L1 Cache Miss

Matrix Size	L1 DCM (Run 1)	L1 DCM (Run 2)	L1 DCM (Run 3)	L1 DCM (Run 4)	L1 DCM (Run 5)
600	3391160	3390779	3527321	3527227	3527185
1000	15715820	15850122	16088981	16088748	16089451
1400	43459972	43808050	44163997	44163577	44164497
1800	93552348	94218002	93485270	93489969	93487592
2200	258553922	258365972	258579083	258373751	258505507
2600	549497157	549638315	549583746	549556994	549618287
3000	845193320	845204455	845189593	845173429	845183679
4096	2197580117	2195375405	2198508657	-	-
6144	7408298034	7414436214	7415680627	-	-
8192	17462271172	17470389443	17485558705	-	-
10240	34243983822	34189869447	-	-	-

2.1.3. L2 Cache Miss

Matrix Size	L2 DCM (Run 1)	L2 DCM (Run 2)	L2 DCM (Run 3)	L2 DCM (Run 4)	L2 DCM (Run 5)
600	7282855	7283238	7549829	7554385	7561835
1000	33226342	33675665	34056463	34037032	33516945
1400	87973540	88550234	89416321	88851072	89465074
1800	184685287	187348875	185289238	183715991	184064215
2200	332298875	316086934	335165178	313117199	331139598
2600	536466215	533799991	532579526	521114474	531499993
3000	819913779	828618825	800782561	789029912	818211481
4096	2096834130	2033942357	2078406482	-	-
6144	6743607055	7005474646	6944915072	-	-
8192	15920376451	16180836934	16472671124	-	-
10240	30187003496	28558498659	-	-	-

2.2. Parallel 2

2.2.1. Execution Time

Size	Execution Time 2	Execution Time 3	Execution Time 4	Execution Time 5

600	0.146435	0.143662	0.150617	0.144556	0.167593
1000	0.457137	0.565238	0.446924	0.540275	0.484065
1400	1.427175	1.659515	1.277717	1.275197	1.597512
1800	3.273151	3.173732	3.345245	3.413707	3.437276
2200	6.073249	5.655436	6.539692	6.785174	5.772847
2600	10.297514	10.493091	10.295505	10.594133	10.275405
3000	17.850843	15.760445	15.280903	15.283475	15.511505
4096	36.218475	41.226128	36.561543	-	-
6144	110.866074	111.181394	-	-	-
8192	256.855952	254.342872	-	-	-
10240	486.821961	490.992682	-	-	-

2.2.2. L1 Cache Miss

Matrix Size	L1 DCM (Run 1)	L1 DCM (Run 2)	L1 DCM (Run 3)	L1 DCM (Run 4)	L1 DCM (Run 5)
600	8009370	8165117	8107646	8107815	8115684
1000	29198471	29475207	29393945	29398173	29414532
1400	67995393	69266956	69355622	68898340	68656395
1800	134978164	135100522	135192935	135356500	135382155
2200	233163044	233252078	233294604	232955951	233947809
2600	365443635	367291386	367255028	367172868	367197238
3000	545833790	546980837	546444102	546671773	546516911
4096	1227662716	1230483807	1225329643	-	-
6144	4089856940	4076806461	-	-	-
8192	9253421857	9190668977	-	-	-
10240	18255092577	18204721504	-	-	-

2.2.3. L2 Cache Miss

Matrix Size	L2 DCM (Run 1)	L2 DCM (Run 2)	L2 DCM (Run 3)	L2 DCM (Run 4)	L2 DCM (Run 5)
600	33015959	33110527	32630479	32735840	32646333
1000	111661835	111034090	112277016	112025309	112474091
1400	224237054	223295362	229630162	228228223	225008875
1800	361479764	362292211	361335198	360416273	357519239
2200	568072604	564669506	567097536	563589830	572276718
2600	840302566	843999325	850592800	847826919	845521744
3000	1199007735	1192029045	1193495755	1198238677	1194531331
4096	2390187615	2404277052	2426496103	-	-
6144	7581957884	7661311902	-	-	-
8192	15770649657	15347002043	-	-	-
10240	29294638971	29577649839	-	-	-

3. Comparison Between Line Matrix Multiplication and Parallel 1 and Parallel 2

3.1. Average Execution Time

Size	MFlops(Line)	MFlops(Parallel1)	MFlops(Parallel2)
600	4408	32727	2867
1000	4049	29851	4016
1400	3324	23455	3768
1800	3114	20806	3487
2200	3345	19649	3744
2600	3297	16687	3440
3000	3227	16071	3389
4096	3325	15423	3613
6144	3310	12926	4160
8192	3252	12218	4392
10240	3157	9336	4593

3.2. Speedup

Size	Speedup(Parallel1)	Speedup(Parallel2)
600	7,42	0,65
1000	7,37	0,99
1400	6,99	1,12
1800	6,3	1,05
2200	5,41	1,03
2600	5,02	1,04
3000	4,97	1,05
4096	4,71	1,1
6144	3,94	1,27
8192	3,68	1,32
10240	2,83	1,39

3.3. Efficiency

Size	Efficiency(Parallel1)	Efficiency(Parallel2)
600	92,80%	8,10%
1000	92,10%	12,40%
1400	87,40%	14,00%
1800	78,70%	13,10%
2200	67,60%	12,90%
2600	62,80%	13,00%
3000	62,10%	13,10%
4096	58,90%	13,80%
6144	49,30%	15,90%
8192	46,00%	16,50%
10240	35,40%	17,40%