CS301: High Performance Computing
Course Project
Parallel Implementation of 2D FFT

Prithvi Patel (201501230)
Swastika Nayak (201501423)

November 6, 2017

Hardware Details

- Model name: Intel(R) Xeon(R) CPU E5-2640 v3 @ 2.60GHz
- Architecture: `x86_64`
- CPU(s): 16
- On-line CPU(s) list: 0-15
- Thread(s) per core: 1
- Core(s) per socket: 8
- Socket(s): 2
- NUMA node(s): 2
- CPU MHz: 1695.179
- L1d cache: 32K
- L1i cache: 32K
- L2 cache: 256K
- L3 cache: 20480K (shared)

Cache Line Size: 64 Bytes The above mentioned machine is referred as the user machine in the plots/figures.

1 Introduction

1.1 Brief Description

The project aims at parallelizing the 2D FFT which basically uses the Cooley Tukey algorithm to perform FFT on each row and each column of the image. FFT which performs Discrete Fourier Transform is used to decompose the image into its sine and cosine components. We have limited the implementation of 2D FFT to square grayscale images whose size is in powers of 2, which is also a limitation to implemenation of FFT in general.

2D FFT has many applications in Digital image processing where understanding the spatial frequency of the image is much important.

1.2 Serial Algorithm Description

The 2D DFT is defined for NXN matrix as:

$$F(u,v) = \sum_{x=0}^{N-1}\sum_{y=0}^{N-1} f(x,y)exp(-i2\pi(\frac{ux}{N}+\frac{vy}{N})) \tag{1}$$

The above given can be seen as summation over 1D FFTs of rows. Thus the implementation of sequential 2D FFT performs the following sequential tasks:

1. Converting the portable pixmap image to an array of real and complex numbers where each row is contiguous in memory as C is row major.
2. Performing FFT on each row of the complex array that is on each row of image.
3. Perform Transpose of the matrix for computing 1D FFT on columns as C is row major
4. Again performing the FFT on each row of transposed matrix
5. Perform the Transpose to get the original orientation
6. Generate the image which has magnitude of the array having complex numbers.

The last step has not been normalized and we have let the machine assign the pixel values.

The 1D FFT method employs divide and conquer strategy, using even and odd coefficients of polynomial separately. An iterative algorithm has been written. For calculating FFT,

the array at each stage is divided into array of even indexed and odd indexed elements and recursively the FFT is calculated over them as shown in Fig1 where ${y_k}^{[0]}$ denotes the even indexed elements and the ${y_k}^{[1]}$ denotes the odd indexed elements. Thus for building the iterative approach, the elements were reordered in a way that the butterfly operations were computed on them in a particular stride which increased by 2 at each stage. It is done using bit reversal. That is element at index $[001]_2$ was swapped with element at index $[100]_2$ (as seen in radix 2) and so on as shown in Fig2a.

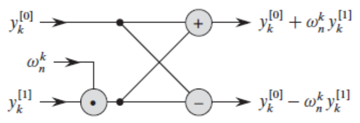

$y_k^{[0]}$
ω_n^k
$y_k^{[1]}$
+
−
$y_k^{[0]} + \omega_n^k y_k^{[1]}$
$y_k^{[0]} - \omega_n^k y_k^{[1]}$

Figure 1: Butterfly operation

The pseudo code:

```
FFT_2D (complex 2D array(NXN))
{
for (i=0,N)
    FFT(Row(i));
}
FFT(Row(i))
{
Perform Bit reversal;
z = Row(i);
stride_j=1
for k = (1,logN) {
    stride_i = 2*stride_j;
    for i = (0,(N/stride_i)){
        twiddle_index = 0;
        j = i*stride_i;
        while (twiddle_index < stride_j){
        j2 = j+stride_j;
        u = w[(twiddle_index*N/stride_i)];
        t = multiply(u, z[j2]);
        z[j2] = z[j] - t;
        z[j] = z[j] + t;
        j++;
        twiddle_index++;
        }
    }
    stride_j *= 2;
}
```

There is clearly no loop dependency in the second and the third loop of FFT.

1.3 Input Output

Input is NxN grayscale image (.ppm), where $N = 2^x$. Output is NxN image (.ppm) which is the magnitude or amplitude of the 2D FFT.

The magnitude array generated was cross checked with the fft2() and fft() performed on the transpose of the same image performed in MATLAB, which performs fft column wise.

1.4 Complexity

The FFT function called by FFT_2D for task (as given in Section 1.2) 2 and 4 is N times each if we are working with array of NxN. Thus total calls to FFT is 2N. Now, on each call, FFT works on 1D array of size N. Bit reversal take O(N). The FFT computation takes $\frac{N}{2}log_2N$ steps. Thus FFT takes O($Nlog_2N$) time. Calculation of transpose and other tasks take O(N^2) time. Thus, the complexity of sequential algorithm is O($2N^2log_2N$) that is O(N^2log_2N).

1.5 Number of memory accesses

Number of memory accesses are O(N^2)+O($3NlogN$), which is O(N^2) for each task where number of memory accesses for FFT is O($Nlog_2N$) at a time.

1.6 Profiling

The profiling information obtained using gprof shows that most of the time goes computing the FFT of rows. The flat profile for N=2048 reveals that 84.63% time goes in computing FFT and 6.35% in transposing the matrix.

From Call graph, it was observed that 2 calls to FFT_2D() takes 88.0% of time.

1.7 Number of computations

The number of FLOPs for each function are:

- Calculating Lookup Table: $\frac{3N}{2}$
- Total computation of FFT: $2N[N - 8 + (6 + \frac{17N}{2}log_2N)]$
- Calculating Amplitude of the image: $4N^2$

2 Parallel Implementation

To parallelize the 2D FFT algorithm, we have used two approaches.

1. Dividing the butterfly operations in each of the stages of 1D FFT among processors.
2. Dividing the FFT of 1D rows among the processors.

Parallelisation of transpose: We implement block method to exploit spatial locality of cache. When we access the input matrix, the rows of each block are at contiguous memory locations and hence uses spatial locality of cache. But when the result matrix, where the transpose is stored, has to be accessed, there is a cache miss of one cache line, since the columns in block are not in contiguous memory locations. The parallelisation of block method is done by dividing the blocks among the processors.

2.1 Theoretical Speedup

The tasks that can be parallelized are

- convert image into complex matrix
- creating a lookup table
- FFT implementation
- transpose of matrix using block method
- convert complex matrix to image

So, after calculating the time for each of these for problem size = 2048 and dividing by total time, we get p, i.e. parallelizable fraction as, 0.8979, so using Amdahl's Law, the speedup is 3.076 for 4 cores. Even profiling gives similar information.

2.2 Approach 1: Dividing the butterfly operations in each of the stages of 1D FFT among processors

Each independent butterfly works with a set of twiddle factors of that step.

- After Bit reversal step, each iteration of the outermost loop (refer to pseudocode of FFT(Row(i)) in Section 1.2) performs N/2 independent butterfly operations, which could run in parallel and is hence divided among the processors.
- The value of k refers to the stage of butterflies. For each k in $1, 2, 3, ...log_2n$, stage k consists of $\frac{n}{2^k}$ groups of butterflies(i.e. it gives $\frac{n}{2^k}$ 2^k- element DFT as output) with 2^{k-1} butterflies per group which is all handled by the second for loop. This loop performs the $\frac{n}{2^k}$ butterfly operations, which can be easily parallelised by using #pragma for directive before the line 13 in the pseudocode.
- For example, in Fig 2a, we have N=8. So, stage 0 out of 3 stages,which corresponds to k=1 in our pseudocode performs 4 2-element FFT here thus 4 processors are required. Stage 1 performs 2 4-element FFT here, which requires 2 processors and the last stage which is computed by 1 processor gives 1 8-element FFT as output.

2.2.1 Problems Faced and Limitations

More and more processors remain idle from the stages when $\frac{N}{2^k} < p$. This is due to overhead associated due to synchronization and cache coherence after processors compute one stage.

2.2.2 Complexity of parallel code

There are $log_2N - log_2p$ stages where p processors can approximately do proper load balance. For the next log_2p stages processors start remaining idle. Thus complexity = $O(\frac{N^2}{p}(log_2p + log_2\frac{N}{p}))$ Overhead increases during those log_2p stages.

2.2.3 Task wise speedup as observed for N = 2048

The computation of transpose and calculation of FFT as parallelized on 4 cores is given below, it is much lower than the anticipated theoretical speedup.

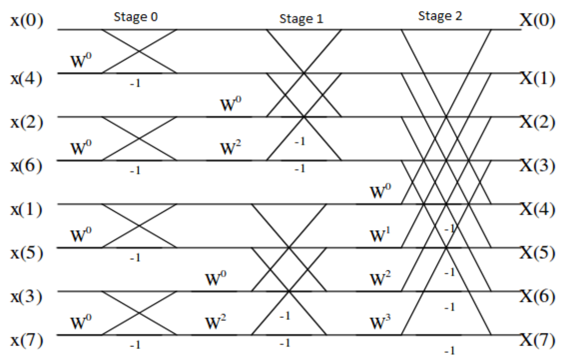
- transpose(): 2.34492
- FFT_2D(): 1.3061

2.3 Approach 2: Dividing the FFT of 1D rows among the processors

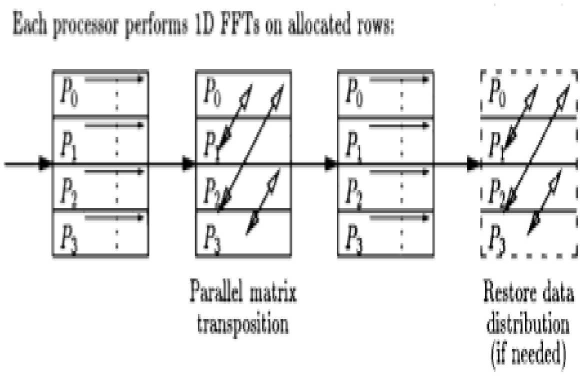
- A 2D matrix is basically an array of 1D arrays. So, in this approach, we divide these 1D arrays i.e rows among the processors.Thus we decompose the data.
- The parallelisation of rows is achieved by distributing the rows to p processors before we perform 1D FFTs of the rows,i.e. before line 3 in the pseudocode.(refer to pseudocode of FFT_2D(complex 2D array(NXN)) in Section 1.2)

2.3.1 Problems Faced and Limitations

When we divide the rows, we basically give the operation of calculating 1D FFTs of rows to a processor which has lot of serial part.So, each processor runs a lot of sequential computations.



(a) Dividing the butterfly operations in each of the stages of 1D FFT among processors


Each processor performs 1D FFTs on allocated rows:
P0
P1
P2
P3
Parallel matrix transposition
Restore data distribution (if needed)

(b) Dividing the FFT of 1D rows among processors

Figure 2: Parallelization approaches

2.3.2 Complexity of parallel code

The N rows are divided among p processors. Thus, the complexity is O($\frac{N^2}{p} log_2 N$). Thus theoretical speedup using asymptotic analysis is $O(N^2 log_2(N))/O(\frac{N^2}{p} log_2 N) = p$.

2.3.3 Task wise speedup as observed for N = 2048

The computation of transpose and calculation of FFT as parallelized on 4 cores is given below, and comparable with the theoretical speedup.

- transpose(): 2.360
- FFT_2D(): 3.0351

3 Curve Based Analysis

3.1 Time Curve related analysis

3.1.1 Dividing the butterfly operations in each of the stages of 1D FFT among processors

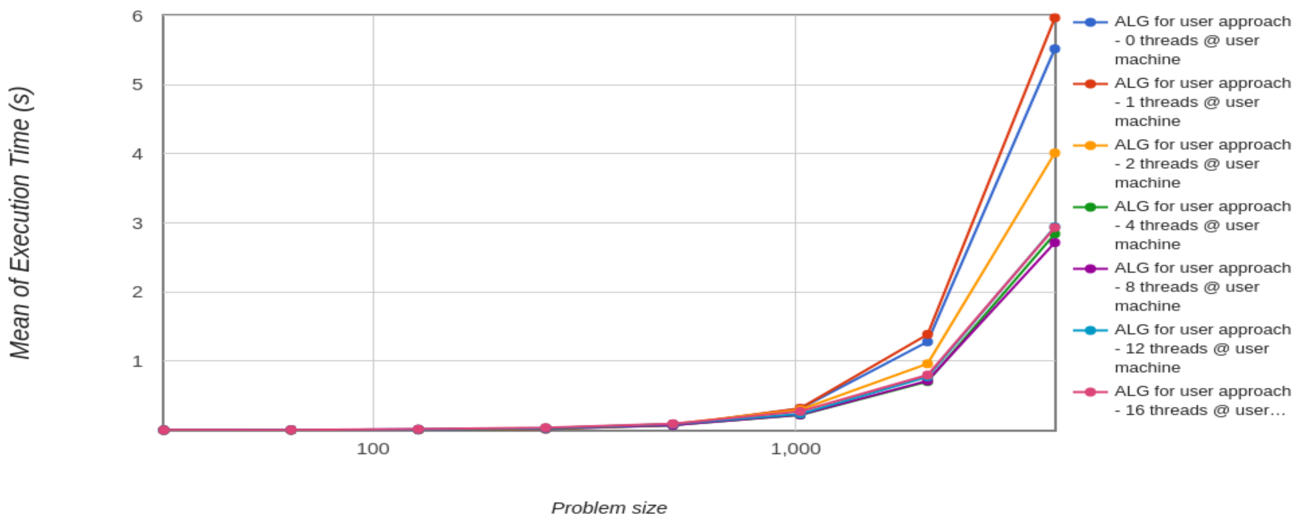


Figure 3: Time vs problemsize for Approach 1

The execution time reduces as expected as the number of cores are increased but saturates from 4 processors onwards. As the number of processors increase, more and more of the processors remain idle according to the strategy. Also as each stage k is not independent of each other, there is problem of cache coherence which could add to overhead.

3.1.2 Dividing the FFT of 1D rows among the processors

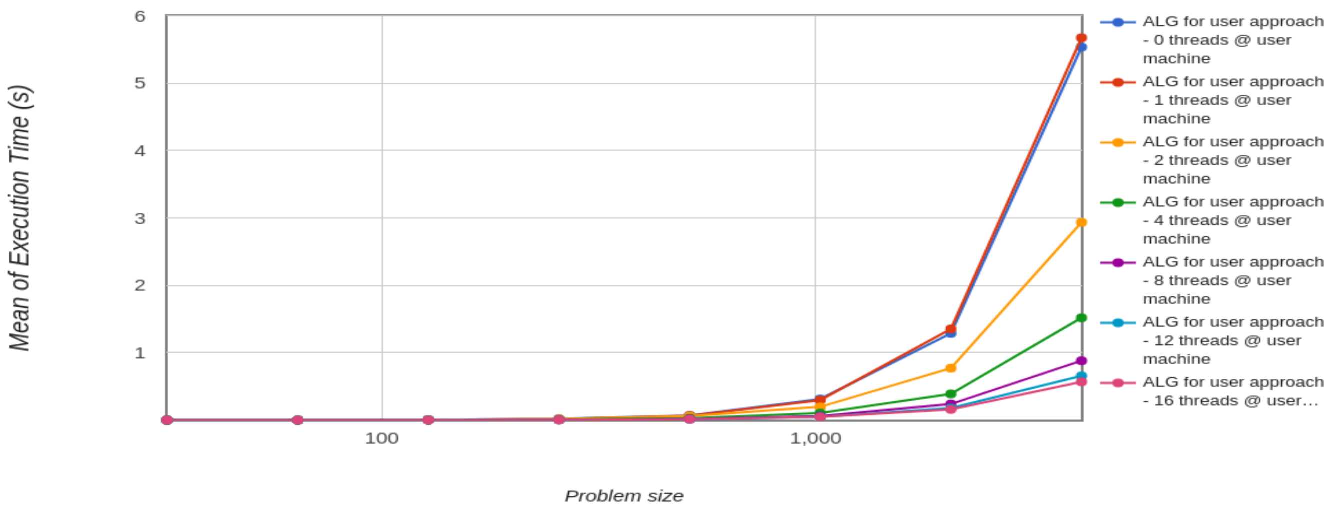


Figure 4: Time vs problemsize for Approach 2

As the N increases, the time increases. For larger image sizes, more the number of cores, lesser the time parallel code takes, that is because the execution of those operations is divided among the cores. Also the overhead grows quite slowly as compared to the computation time as the N increases.

3.2 Speedup Curve related analysis

3.2.1 Dividing the butterfly operations in each of the stages of 1D FFT among processors

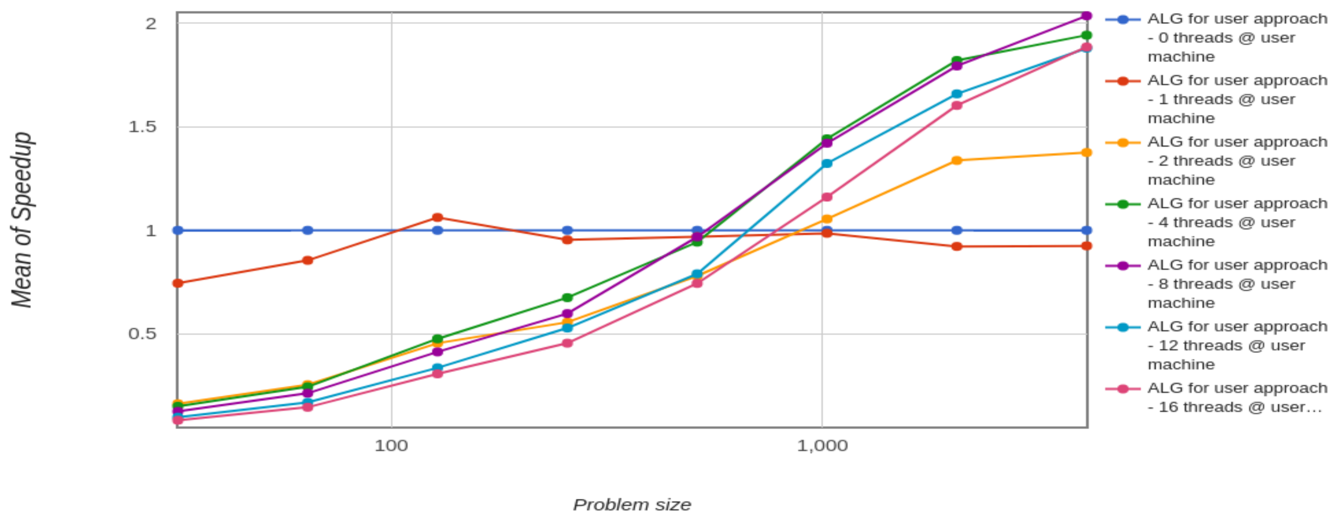


Figure 5: Speedup vs problemsize for Approach 1

- The speedup curves show that for smaller sizes, many processors remain idle and not all resources are used. We start to obtain speedup from N = 1024, as when p gets large, actually in initial stages, the processors are not idle.
- Thus communication overhead increases, once number of processors increases. N must be simultaneously increased with p to see more speedup. For N=4096, 1D FFT gives speedup of nearly 2 for $p > 4$, making 4 processors enough for computation of 2D FFT. Increasing N, can increase speedup for larger p to a certain level till N saturates.

3.2.2 Dividing the FFT of 1D rows among the processors

- As the N increases, speedup increases but the rate at which it increases decrease. We do not get good speedup for smaller problem sizes. This is due to the reason that the overhead is comparable to the number of computations for smaller size of images.
- As the number of cores increase, speedup increases but again,the increase in speedup with increase in number of cores decreases or slows down. With the increase in number of cores, the parallel overhead also increases, thus rate of increase of speedup decreases.

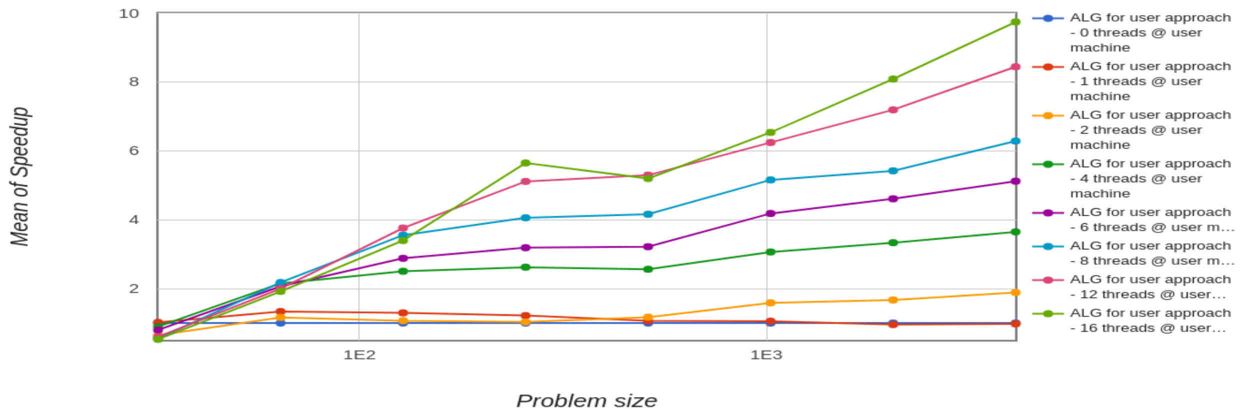


Figure 6: Speedup vs problemsize for Approach 2

- The problem is scalable for higher problem sizes when cores are less. In Figure 6, till cores = 6, the speedup is nearly equal to the theoretical speedup. The speedup for 16 cores goes till 10, it does not reach its theoretical speedup. Thus to scale the approach, N need to be increased.

3.3 Efficiency Curve related analysis

3.3.1 Dividing the butterfly operations in each of the stages of 1D FFT among processors

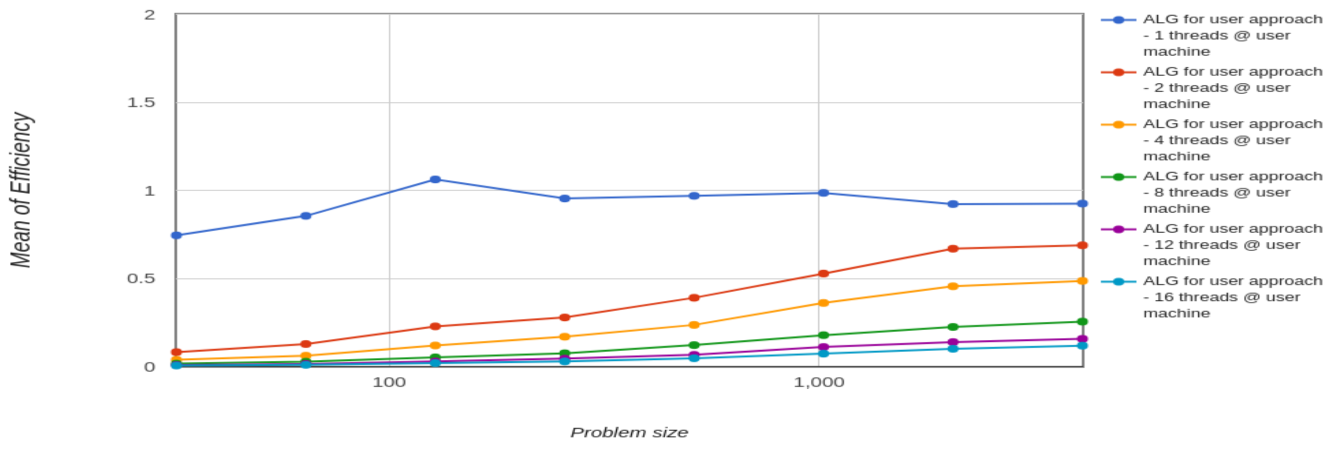


Figure 7: Efficiency vs problemsize for Approach 1

- The efficiency of this strategy decreases with increasing p, which straightway points that this approach is not scalable.
- Even on increasing N, we are not able to increase efficiency much, as overhead of the approach increases with increasing N and p.

3.3.2 Dividing the FFT of 1D rows among the processors

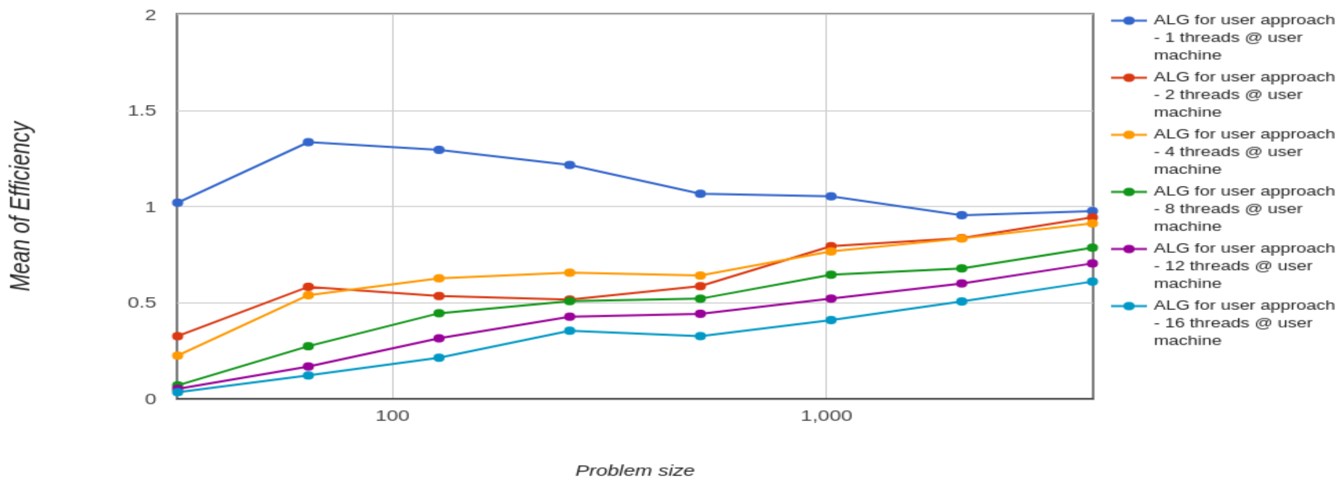


Figure 8: Efficiency vs problemsize for Approach 2

- There is a superlinear speedup when number of cores is 1 which could be because of no communication. For less number of cores, efficiency approaches to 1, for larger number of cores, efficiency does not even reach 1.
- Efficiency increases with the increase in problem size because as the problem size is increased,overhead time often grows more slowly than serial time. So,if we increase N, i.e. the problem size, the code can become scalable.

3.4 Karp Flatt Metric Analysis

For N = 4096: the experimentally determined serial fraction increases.

3.4.1 Approach 1

p	2	4	6	8	10	12	14	16
speedup	1.376	1.942	1.983	2.036	1.972	1.88	1.885	1.885
e	0.453	0.353	0.405	0.418	0.452	0.489	0.494	0.499

This implies that the overhead is increasing with number of processors Or the idle time of the processors is high as p increases or both.

3.4.2 Approach 2

p	2	4	6	8	10	12	14	16
speedup	1.886	3.646	5.118	6.284	7.442	8.44	9.039	9.741
e	0.060	0.032	0.035	0.039	0.038	0.038	0.042	0.043

The overhead is more for lesser number of cores and higher number of cores. Thus, the optimal number of cores is about 6-12 for the second approach.

4 Further Detailed Analysis

4.1 Major serial and parallel overheads

The overhead can be seen below for 1 processor for both approaches.

4.1.1 Approach 1

- For N = 2048, parallel overhead = 1.3844 - 1.2841 = 0.1003
- For N = 4096, parallel overhead = 5.9746 - 5.5238 = 0.4508

For p=8 and N = 4096, overhead = 8(2.8775) - 5.5238 = 17.4962. Thus, we observed the saturated speedup curve. For smaller N, the overhead is more than the computations.

4.1.2 Approach 2

- For N = 2048, parallel overhead = 1.318976 - 1.309104 = 0.009872
- For N = 4096, parallel overhead = 5.643441 - 5.535854 = 0.107587

The overhead in approach 2 is still less than the one in approach 1. This is due to the fact that communications are lesser in second one.

4.2 Cache coherence related analysis

4.2.1 Approach 1

The elements which are updated by butterfly operations at one stage are to be used at next stage and that too by the different processor. Cache coherence adds up to overhead as the largest N (uses (4096+2048)x2x8 = 98KB) typically used by us for FFT once can be well accommodated within L2 cache. For N = 1024, the real and complex arrays to be operated upon and the twiddle factors' array (about 25KB) fits well into L1d cache.

4.2.2 Approach 2

Here there are no dependencies, i.e. the rows are totally independent of each other. Moreover, for a problem size of 4096, which is the highest problem size here, real and imaginary parts of each row accesses memory of 98KB which fits well in L2 cache.

4.3 Load Balance related analysis

In both the approaches, block wise transpose of the matrix has been calculated. As block size of 16 elements gave better results, it has been used. The load is shared properly for $N > 256$. There is imbalance for smaller sizes.

- In approach 1, the load is not equally shared among the processors. For larger N, the work is shared equally till $\frac{N}{2^k} = p$, later on the work is not shared equally.
- In approach 2, there is load balancing, as the rows are divided more or less equally among processors.

4.4 Synchronization analysis

Synchronization is necessary for Approach 1, as the results of one stage are dependent on the result obtained in previous stage. There is implicit barrier in a way that a processor needs to wait till all processors complete computations of kth stage. In Approach 2, all rows are independently parallelised, so there is no synchronisation issue.

4.5 Granularity related analysis

Granularity is ratio of computations to communications. Approach 1 has many smaller tasks for processors but the load is not balanced and there is too much synchronisation and communication before and after every stage. Initially, the processors do less number of computations but in the last log_2p stages, computations per processor increases. Approach 2 is coarse grained as each processor does many computations sequentially.

4.6 Scalability related analysis

- Approach 1 is weakly scalable because as p is increased alongwith N, there is not much increase in efficiency.
- Approach 2 is scalable but not strongly scalable. N needs to be increased yet.

4.7 Future scope of improvement

In order to obtain more speedup or throughput, both the approaches can be implemented on the distributed system. Also, cyclic block mapping can help get us better results for the first approach mentioned as it maps data and not computations in the above context.

The output image generated can be normalized and also we can use histogram normalization for the same.

5 References

1. Understanding FFT implementation: Introduction to algorithms, Charles E. Leiserson, Clifford Stein, Ronald Rivest, and Thomas H. Cormen

2. Figures and strategies from:
   - Fig1 from Intoduction to Algorithms
   - Fig2a from Implementation and performance evaluation of parallel FFT algorithms, Somasundaram Meiyappan
   - Fig2b from An Elementary Introduction to the Discrete Fourier Transform, Chapter 23, Eleanor Chu and Alan George