Performance Analysis of Fourier Transform Algorithms with and without using OpenMP

[1]Prof. Kotresh Marali, [2]Irshadahmed Preerjade, [3]Shreevatsa Tilgul, [4]Shrikrishna K, [5]Sourab M Kulkarni
[1]Asst. Professor, Department of E&CE,
[2,3,4,5] Undergraduate Students. Department of E&CE,
SDM College of Engineering & Technology Dharwad, India,

***Abstract*— This paper provides a comparative analysis of the execution speed of a multi core platform which is involved in the parallel execution of Discrete Fourier Transform (DFT) and Inverse Discrete Fourier Transform (IDFT) programs over sequential ones. In order to facilitate the parallel execution, an Application Program Interface (API) called OpenMP is used. A detailed thread level analysis of these algorithms is made and a speedup ranging from 2.9 to 4.3 over sequential execution is obtained for large number of samples ranging from 1024 to 8192.**

***Index Terms*— Parallel programming, Thread level parallelism, OpenMP, Loop-work sharing construct, Fourier Transform Algorithms.**

I. Introduction

With an evolution of multi core processors, the parallel programming becomes an appropriate choice for a programmer. This is because, if a piece of code is to be run in a sequential way, then only single core of a processor will be used and no improvement in performance can be observed, but when the execution of same code is distributed in parallel across multiple cores, certain levels of improvement in the performance can be seen. The parallel programming can performed at thread level and a class of parallelism called thread level parallelism can be obtained. This feature is offered by an API called OpenMP. This ideology of parallel programming can extended for Digital Signal Processing (DSP) applications. The choice of DSP application lies in a fact that the execution of DSP algorithms often consumes more time as it involves computations of complex floating point numbers. Using parallel programming techniques, various DSP algorithms are successfully implemented in [2] using 2nd generation Intel i-7 quad-core processor. This implementation can be an alternative for the reliance on special purpose hardware or Field Programmable Gate Arrays FPGA) for computing signal processing algorithms. This has been stated in [2].

II. Background Work

The Thread level parallelism incorporates the architectural advantages of a Multiple Instruction Multiple Data (MIMD) processor. Most of the processors that support this kind of parallelism are either Superscalar or Simultaneous Multithreading (SMT) models. This fact is illustrated in [6]. In thread level parallelism, a section of code is made to execute in parallel with each superscalar or an SMT core in a multi core chip dedicated to handle these parallel sections independently. Usually, this section of code is called user thread and the hardware resources involved in the execution of user thread form hardware thread. The kernel offers certain kernel resources to execute the user thread and these resources offered by kernel form kernel threads. This concept is given in [7].

To facilitate this kind of parallelism many APIs are used. One of the APIs is OpenMP. One of the biggest advantages of OpenMP is its portability. Further, OpenMP offers a wide range of constructs that are suitable for shared memory programming model.

```
#pragma omp parallel
{
  Body of parallel reigon
}
```

Fig. 1. Syntax of *parallel* construct in C language

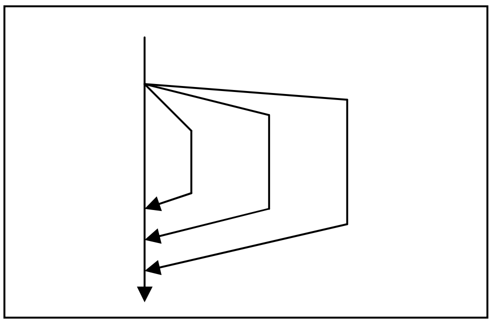

Fig.2. Fork- join model of *paralle*l construct of OpenMP. The arrows indicate threads. The vertical arrow indicates a master thread that forks rest of the threads upon encountering parallel construct

The *parallel* construct offered by OpenMP facilitates thread level parallelism. The *parallel* construct of OpenMP follows fork-join model The Fig.1 shows the syntax of the *parallel* construct in C- language. The normal process is called a master thread, upon encountering a *parallel* construct it forks rest of the threads. This method is clearly illustrated in Fig.2.
In order to further facilitate multithreading in a more appropriate manner, OpenMP offers work-sharing constructs. In [3], loop work-sharing and section-work sharing concepts are explained. The loop work-sharing constructs are used to divide the for-loop iterations into various threads. The section work-sharing constructs are used to divide various sections of code into threads. The *parallel* construct can be combined along with these constructs to parallelize for loops and sections of a code. The syntaxes of these combined constructs are described in Fig 3.

```
#pragma omp parallel for
 for ( i=0 ; i<N;i++)
{

      Body of the for -loop

}
```

```
#pragma omp parallel sections
{

       #pragma omp section
    {
       1st section
    }
      #pragma omp section
   {
       2nd section
    }

}
```

Fig. 3. a) Basic Syntax of Loop work-sharing construct b) Syntax of section work-sharing construct combined with *parallel* construct in OpenMP

The work-sharing constructs are basically exploited in [1] for implementing DSP algorithms with for loops being a primary target for parallelization. By doing so, [1] gives an important conclusion that it is of no use of parallelizing the loops with lesser number of iterations as time taken to create the threads and time spent by each thread in communicating with each other dominates over the time required to process them instead it is worth parallelizing the loops with large number of iterations. Further, [4] gives the details regarding analysis and speedup calculations of parallel programs. Some details regarding the syntaxes of parallel programming constructs are taken from [5]. Choice of Intel based processors for executing programs with parallel constructs can be found from the results of [2]. Although major DSP algorithms are tested in [1], this work gives an experimental analysis for different values of threads over sequential ones and gives an experimental evidence for choosing the number of threads for parallel programming.

III. HARDWARE PLATFORM FOR THE IMPLEMENTATION OF DSP ALGORITHMS

The hardware platform chosen in this work is 5th generation Intel i-5 5200 dual-core processor. The parallel processing techniques for implementing DSP algorithms used in [2] can be introduced in 5th generation Intel core. Further, results of [1] show that performance of Intel i7 2670QM mobile based processor is better than a digital signal processor TMS320C6678 when parallel execution is involved. This makes the choice of Intel cores as more preferable when parallel execution of a program is involved.

Intel i-5 5200 is a laptop based dual-core processor which follows Broadwell micro architecture. The details regarding the organization of Intel i-5 5200 dual core is listed in Table I.

TABLE I
ORGANIZATION OF INTEL I5 5200U LAPTOP PROCESSOR

Property	Value
Clock Speed	2.20 GHz
Sockets	1
Physical Cores	2
Logical Cores	4
Hyper-V-support	Yes
L1-cache	128 KB
L2-cache	512 KB
L3-cache	3.0 MB

One of the important factors to be concentrated upon is the support of Intel i-5 5200U for hyper-threading technology. In hyper-threading technology, it is possible to define logical processors from a physical processor. These logical processors share the execution resources and cache memory. The concept of hyper-threading is further explained in [7]. Fig 4 illustrates this concept. In Intel i-5 5200U, a single physical core can support two logical cores. Since Intel i-5 5200U is having two physical cores it certainly has four logical cores.

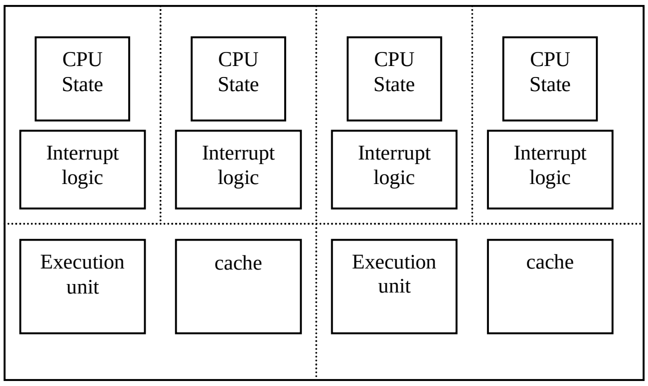

Fig 4. Conceptual model of Hyper-threading. The dotted lines indicate the separation between the cores. Each physical core has an Execution unit and cache memory. These resources are shared by two logical cores consisting of CPU state and Interrupt logic.

IV. OpenMP Clauses and Environment Variables for Scheduling and Distributing Threads

When for-loop is taken as a primary target for parallelization then it is advisable to use suitable clause along with parallel for directive. The details regarding clauses are explained in [5]. In this work, schedule clause is used. Fig 5 shows the syntax of schedule clause which is given in [5]. The first field of the syntax, *type* can be static, dynamic, guided, runtime or atomic. The second field *chunksize* value determines how to distribute the iterations of for-loop among threads in a round robin fashion. The *type* used is runtime and when runtime is used, the scheduling is determined by the status of environment variable OMP_SCHEDULE. The OMP_SCHEDULE can be set to static, dynamic or guided with suitable *chunksize* value in a bash shell. The statistical data provided in [5] says better speed is obtained when static schedule type is used and with *chunksize* value being as low as possible. Hence, OMP_SCHEDULE is set to static with chunksize value being assigned to one.

```
schedule( < type> [, <chunksize> ])
```

Fig 5. Syntax of the schedule clause

Another environment variable used is *omp_set_num_threads ( )*. This variable takes the umber of threads to be created as input arguments.
Using these clauses and environment variables parallelization of the Fourier Transform algorithms is done.

V. Implementation of Fourier Transform Algorithms

Fourier Transform algorithms are used for converting time domain samples into frequency domain samples. This is done because it is easier to analyze a signal in frequency domain than in time domain and heavy mathematical operations can be scaled down to simple ones. For example an operation like convolution which involves repetitive multiply and add operations can be scaled down to complex number multiplication. For discrete valued signals, DFT is used for converting time domain samples to frequency domain and IDFT is used to convert the frequency domain samples back to time domain samples. Parallelization of these algorithms can be of great aid especially when filtering of a signal such as an audio or an image is concerned as filtering requires repetitive domain conversion operations.

A. DFT

DFT is expressed as in (1).

$$X(k) = \sum_{n=0}^{N-1} x(n) e^{-j2\text{®}nk/N} \tag{1}$$

In order to implement (1), two nested for-loops are required. The pseudo code of the implementation is shown in Fig 6. In this figure, the innermost loop is enclosed within the function. The function is then enclosed within the for-loop. The primary advantage of this technique is that when outermost loop is parallelized then each thread can have access for more number of private variables. Using private variables in case of data independencies in turn ensures that there is no data sharing amongst the threads and latencies involved in computation can be reduced to the greater extent. Now each thread will make a function call. In Fig 6, *mac_dft ( )* is the name of the function used. The stack organization of the threads is shown in Fig 7.

```
#pragma omp parallel for schedule (runtime)
 for ( k=0; k< N ; k++)
{
     Y[k]=mac_dft ( x[0],N,k);
}
```

$$\updownarrow$$

```
 complex  mac_dft (real x[0],int N, int k )
{
   real  angle;
   complex r ,sum=0;

      for( n=0 ; n<N; n++)
     {
         angle=-2*pi*k*n/N;          // Calculate angle
         r= calc_exp( angle );       // Calculate twiddle-factor
        sum= sum +mul(x[n],r);       // multiply and add
     }
       return (sum);                 // return the value
 }
```

Fig 6   Pseudo-code of the DFT algorithm being parallelized

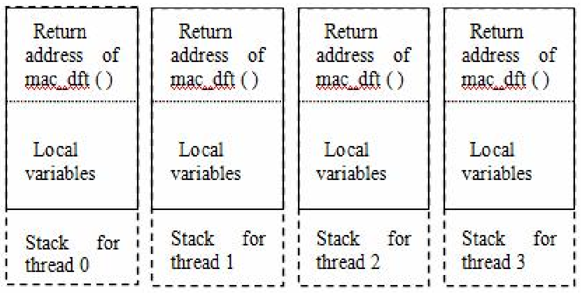

Fig 7: Stack organization of the threads. Each makes a separate function call to the function *mac_dft ( )* and keeps a copy of local variables of the function separately.

B. *IDFT*

IDFT is expressed as in (2)

$$x(n) = \frac{1}{N}\sum_{k=0}^{N-1} X\,(k)e^{\,j2®nk\,/\,N} \qquad (2)$$

By inspecting (2), it is clear that in order to implement IDFT, two for-loops are required. The approach used remains same as that of DFT. The innermost loop where most of the computation is done can be enclosed within a function. This enclosed function can be placed within a for-loop subjected to parallelization. The pseudo code for IDFT is given in Fig 8. In Fig 8, *mac_idft ( )* is the function used.

```
#pragama omp parallel for schedule (runtime )
 for ( n=0;n<N; n++)
{
        Y[n] = mac_idft (X[n], N, n);
}
```

```
complex mac_idft ( complex X[0], int N, int n )
{
      complex  r, angle, sum=0;
       int k;
         for ( k=0;k<N; k++)
        {
            angle =2*pi*n*k/N;              // calculate angle
            r= calc_exp(angle);             // calculate exponent
            sum= sum + mult (X[k], r);   // multiply and add
        }

        return (sum/N);                       // return value
}
```

Fig 8 Pseudo-code of the IDFT algorithm being parallelize

VI. Performance Analysis

Performance analysis is done by comparing the wall-clock time of the section of code which is parallelized with the same section which is not parallelized. The wall clock time is preferred because it gives the actual time taken to complete a part of a program. This fact is mentioned in [4]. In order to measure the wall clock time, *omp_get_wtime ( )* is used. As defined in [4], parallel execution speedup can be determined taking the ratio of wall clock time measured for one thread to wall clock time measured for multiple threads.

A. *Results of DFT*

The performance analysis of DFT is done for different values of threads with respect to the sequential DFT code. The results are compiled in Table II.

TABLE II
Time taken for computation of DFT Algorithm in seconds

Sample size	Sequential Execution	Parallel Execution		
		Thread=2	Thread=4	Thread=8
8	0.000087	0.000202	0.006188	0.000474
16	0.000160	0.000270	0.008423	0.000375
32	0.000624	0.000498	0.000463	0.000669
64	0.002316	0.001330	0.007613	0.001552
128	0.007425	0.00475	0.010153	0.003152
256	0.018450	0.01289	0.010834	0.007864
512	0.05093	0.030171	0.023896	0.020176
1024	0.18213	0.103147	0.062571	0.061775
2048	0.7211	0.3733	0.22182	0.224617
4096	3.7453	1.46475	0.868707	0.868707
8192	10.8727	5.8514	3.449419	3.462282

A graph is plotted based on results obtained in Table II. The graph is shown in Fig 9. The comparison of parallel code for chosen thread value is done with sequential code for DFT.

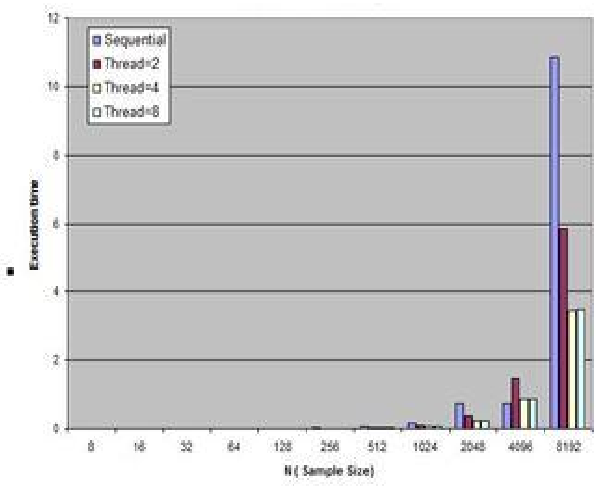

Fig 9..Results of Sequential and Parallel Implementation of DFT Algorithm

The results of Fig 9 convey that for thread value of 2 the execution time is reduced by half and for the thread value of 4, the execution time is further reduced. These results are much significant when size of samples lie between 2048 and 8192. When the size of samples is very low, the execution time of sequential code is lesser than that of parallel code. This is evident for the samples that are lesser than 128. For the remaining sample sizes, marginal reduction in execution time for parallel code is observed. As for the thread value of 8, no further performance improvement can be seen.
Parallel execution speedup is computed for the sample values of 1024, 2048, 4096 and 8192. The results of the computation are tabulated in Table III.

TABLE III
PARALLEL EXECUTION SPEEDUP FOR DFT ALGORITHM

Sample size	Parallel Speedup		
	Thread=2	Thread=4	Thread=8
1024	1.76	2.91	2.94
2048	1.93	3.25	3.21
4096	2.55	4.31	4.31
8192	1.85	3.15	3.14

From Table III, it is inferred that better speedup can be obtained for the thread values 4 and 8.

B. *Results of IDFT*

The performance analysis of IDFT algorithm is also done in a method similar to that of DFT. The execution times are tabulated in Table IV. The results of the tabulations are analyzed in the form of a graph shown in Fig 10. After the analysis, speedup obtained in by using parallel execution is computed. The computed results are given in the Table V. Better results for speedup is again obtained for thread values of 4 and 8. Again, Table IV shows the execution time of sequential to be lesser than that parallel execution of the IDFT code. Further, it can be seen that nature of results are almost same as that of DFT although the values of the execution time differ.

TABLE IV
TIME TAKEN FOR COMPUTATION OF IDFT ALGORITHM IN SECONDS

Sample size	Sequential Execution	Parallel Execution		
		Thread=2	Thread=4	Thread=8
8	0.000242	0.001003	0.006011	0.000860
16	0.000331	0.001025	0.00673	0.000917
32	0.00104	0.000816	0.015415	0.001512
64	0.003041	0.002151	0.01402	0.001965
128	0.008036	0.00602	0.009007	0.006789
256	0.02132	0.01227	0.012488	0.012280
512	0.06034	0.031686	0.02913	0.02526
1024	0.211878	0.11123	0.082721	0.07975
2048	0.81079	0.41107	0.268703	0.28106
4096	3.11974	1.61936	1.01927	1.05009
8192	12.21207	6.4526	4.0436	4.0834

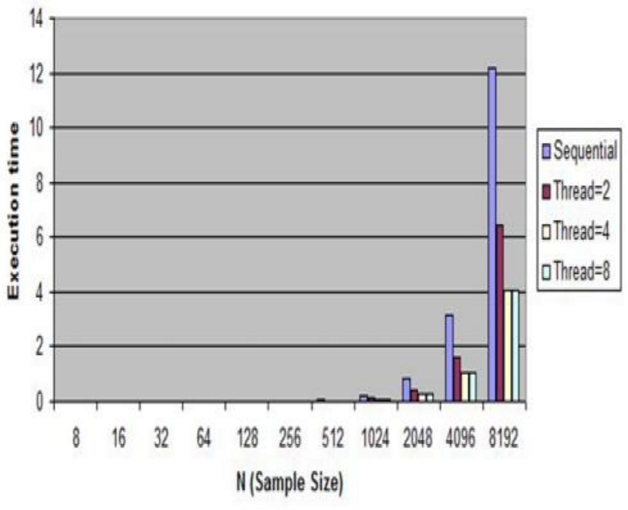

Fig 10..Results of Sequential and Parallel Implementation of IDFT Algorithm

TABLE V
PARALLEL EXECUTION SPEEDUP FOR IDFT ALGORITHM

Sample size	Parallel Speedup		
	Thread=2	Thread=4	Thread=8
1024	1.90	2.56	2.65
2048	1.97	3.01	2.88
4096	1.92	3.06	2.97
8192	1.89	3.02	2.99

VII. CONCLUSION

In a hyper-threaded processor better speedup is obtained when the number of threads is equal to the number of logical cores present in the processor. This is evident from the results of thread level analysis that is performed for DFT and IDFT algorithms. However, care should be taken by the programmer in choosing the number of threads depending on size of samples in order to ensure better performance of parallel execution. The performance analysis of the Fourier Transform algorithms can be useful for the parallel implementation of DSP applications in real time that demand faster processing of large data set.

REFERENCES

:
[1] Roman Mego and Tomas Fryza, " Performance of Parallel Algorithms Using OpenMP", 23th Conference Radioelektronika , Czech Republic, April 16-17, 2013
[2] Umberto Santoni and Thomas Long, "Signal Processing on Intel Architecture: Performance Analysis using Intel Performance Primitives", White Paper, Intel Corporation, 2011.
[3] OpenMP Application Programming Interface, OpenMP Architectural review board, version 4.5, November 2015.
[4] Barbara Chapman, Gabriele Jost and Ruud Van Der Pas, "How to get good performance by using OpenMP", *Using OpenMP, Protable Shared Memory Parallel Programming*, MIT press Cambridge, Massachusetts London England, 2008, pp 53-190.
[5] Peter Pacheo, "Shared Memory programming using OpenMP", *An Introduction to Parallel Programming*, Morgan Kufmann Publishers, 2011, pp 209-251.:
[6] William Stallings, " Parallel Processing" *Computer Organization and Architecture, Designing for performace*, 8th Edition, Prentice Hall 2008, pp 630-677.
[7] Shameem Akthar and Jason Roberts, "System overview of threading", *Multi-core programming, increasing the performance through software multithreading,* Intel press,2006, pp 21-36.

Shreevatsa Tilgul was born in Gulbarga on 11 January 1996. He finished his schooling at Maharishi Vidya Mandir High School, Gulbarga, Karnataka and is currently pursuing his Bachelor of Engineering at SDM College of Engineering and Technology, Dharwad, Karnataka, India. His major areas of interest include Digital Image Processing, Cryptography and Programming in C and C++.

Prof. Kotresh Marali is an Assistant Professor in the department of Electronics and Communication Engineering at SDM College of Engineering and Technology, Dharwad, Karnataka, India. He obtained his Bachelor of Engineering from Kalpataru Institute of Technology, Tiptur, Karnataka and a Master degree in Digital Electronics from SDM College of Engineering and Technology, Dharwad, Karnataka, India.

Shikrishna K was born in Kasaragod on 2 November 1996. He finished his schooling at Kendriya Vidyalaya Dharwad, Karnataka and is currently pursuing his Bachelor of Engineering at SDM College of Engineering and Technology, Dharwad, Karnataka, India. His areas of interest are VLSI IP Design, Computer Architecture and Parallel Processing.

Irshadahmed Peerjade was born in Dharwad on 27 August 1996. He finished his schooling at VMH school, Nesargi, Karnataka and is currently pursuing his Bachelor of Engineering at SDM College of Engineering and Technology, Dharwad, Karnataka, India. His areas of interest are ARM based applicatons, Embedded systems and Computer Networking.

Sourab M Kulkarni was born in Gadag on 9 July 1996. He finished his schooling in Shri Satya Sai Loka Seva High School ,Mudenhalli, Bangalore, Karnataka and is currently pursuing his Bachelor of Engineering at SDM College of Engineering and Technology, Dharwad, Karnataka, India. His areas of interest are Digital and Wireless Communication and VLSI based communication systems.