Fine-Grain Multiprocessing Of Fast Fourier Transform Algorithm

Felicia Ionescu and Bogdan Popescu

Applied Electronics and Information Engineering Department
Faculty of Electronics and Telecommunications
"Politehnica" University of Bucharest
1-3 Iuliu Maniu Street
Bucharest
ROMANIA
E-mail: fionescu@atm.neuro.pub.ro

Abstract: In multithreaded execution on shared memory multiprocessors, the Fast Fourier Transform algorithm (FFT) is partitioned into a number of threads which concurrently run on the available processors of the system. Each thread computes a partition of the data vector and two or more threads communicate with each other through shared memory. The analysis of sequential algorithm points out different types of data dependencies that are removed by insertion of synchronization points between threads. For the parallel algorithm, an estimation of its performance parameters is done in order to determine the performance of the parallel execution.

Keywords: FFT algorithm, shared memory systems, multithreaded execution, scalability of parallel systems

Felicia Ionescu received her Electronics Engineering degree and Ph. D degree from Polytechnical Institute of Bucharest. She is now Professor at Applied Electronics and Information Engineering Department, Faculty of Electronics and Telecommunications. Her main didactic and research areas are real -time image generation, virtual reality, parallel and distributed processing, object-oriented analysis and design.

Bogdan Popescu received his Electronics Engineering degree from the "Politehnica" University of Bucharest in 1998. He is now system engineer at Softnet Company, and Associated Assistant at Applied Electronics and Information Engineering Department, Faculty of Electronics and Telecommunications, "Politehnica" University of Bucharest. His main interest areas are parallel and distributed systems, computer networks and databases.

1. Introduction

The *Fast Fourier Transform* algorithm (FFT) is one of the most frequently used algorithms in digital signal processing applications: audio processing, image processing, scientific/statistical applications, etc. Here are only a few examples in which high speed of FFT computation is required. One way to achieving this goal is that of multiprocessing in parallel systems, a subject to be dealt with in the following.

For a given sequence of points $X = (X[0], X[1], \ldots X[n-1])$, *Discrete Fourier Transform*, (DFT) is a sequence $Y = (Y[0], Y[1], \ldots Y[n-1])$, with the same dimension n, in which elements $Y[i]$ have the following expression:

$$Y[i] = \sum_{k=0}^{n-1} X[k]\omega^{ki}\ ,\ 0 \le i < n \qquad (1)$$

where $\omega = e^{2\pi\sqrt{-1}/n}$ is the n-th root of the unit in the complex plane. From Equation (1), the evaluation of each element $Y[i]$ requires n complex operations, so that the execution time of the n-points DFT algorithm is in $O(n^2)$.

The *Fast Fourier Transform* (FFT) algorithm reduces this execution time to $O(n \log n)$ [1]. For obtaining the FFT algorithm, each element of n-points DFT is written as a sum of two elements of an $n/2$ – points DFT as follows:

$$\begin{aligned} Y[i] &= \sum_{k=0}^{(n/2)-1} X[2k]\omega^{2ki} + \sum_{k=0}^{(n/2)-1} X[2k+1]\omega^{(2k+1)i} \\ &= \sum_{k=0}^{(n/2)-1} X[2k]e^{2(2\pi\sqrt{-1}/n)ki} + \sum_{k=0}^{(n/2)-1} X[2k+1]\omega^{i}e^{2(2\pi\sqrt{-1}/n)ki} \\ &= \sum_{k=0}^{(n/2)-1} X[2k]e^{2\pi\sqrt{-1}ki/(n/2)} + \omega^{i}\sum_{k=0}^{(n/2)-1} X[2k+1]e^{2\pi\sqrt{-1}ki/(n/2)} \end{aligned} \qquad (2)$$

In this expression, n is considered as a power of 2. Using the notation $\varpi = e^{2\pi\sqrt{-1}/(n/2)} = \omega^2$, i.e. ϖ is the $n/2$ root of the unit, Equation (2) can be written:

$$Y[i] = \sum_{k=0}^{(n/2)-1} X[2k]\varpi^{ki} + \omega^{i}\sum_{k=0}^{(n/2)-1} X[2k+1]\varpi^{ki} \qquad (3)$$

The multiprocessing system herein considered for parallelization of the FFT algorithm is a multiprocessor with shared memory, which may have up to p processors. The execution of the

algorithm is partitioned into a number p of threads, which concurrently run on the available processors. In this multiprocessor system, the communication among threads is implemented as fast, shared access to global variables, which involves low parallel overhead, efficiency and scalability.

2. Sequential FFT Algorithm

Sequential FFT algorithm can be expressed in a recursive or iterative form. The simplest parallelization of FFT can be obtained based on iterative sequential algorithm (known as Cooley-Tukey algorithm[2]), presented in Figure 1, as C-like pseudocode.

```
ITERATIVE_FFT (X, Y, n)
{
      /* Initialization of vector R */
      for (i = 0; i < n; i++)
         R[i] = X[i];
      r = log n;
      /* Outer loop */
      for (m = 0; m < r; m++){
         /* Step 1: First inner loop */
         for (i = 0; i < n; i++)
          S[i] = R[i];
         /* Step 2: Second inner loop */
         for (i = 0;i < n; i++) {
           i = (b0b1 ... br-1) /*binary representation of i */
           i1 = (b0 ... bm-10bm+1 ... br-1);
           i2 = (b0 ... bm-11bm+1 ... br-1);
           R[i] = S[i1] + S[i2] x ω exp(bmbm-1 ... b00 ... 0);
         }
      }
      /* Update output vector Y */
      for (i = 0; i < n; i++)
        Y[i] = R[i];
}
```

Figure 1. Sequential Iterative FFT Algorithm

The program uses two vectors R and S with dimension n for the computation of the coefficients. Ignoring the initialization and the final step of the program, the iterative algorithm contains an outer loop with $\log n$ iterations, each iteration containing two inner loops. The first inner loop is a step that updates the n-points vector S from the corresponding elements of vector R, computed in the previous iteration. The second inner loop is a step at which each element $R[i]$ of vector R is computed combining two different components, $S[i_1]$ and $S[i_2]$ from vector S.

At each step, n operations are executed, and the outer loop has $\log n$ iterations, so that the execution time of sequential FFT algorithm is in $O(n \log n)$.

The diagram in Figure 2 illustrates the steps of computation of the output sequence Y from input X, for $n = 16$ points.

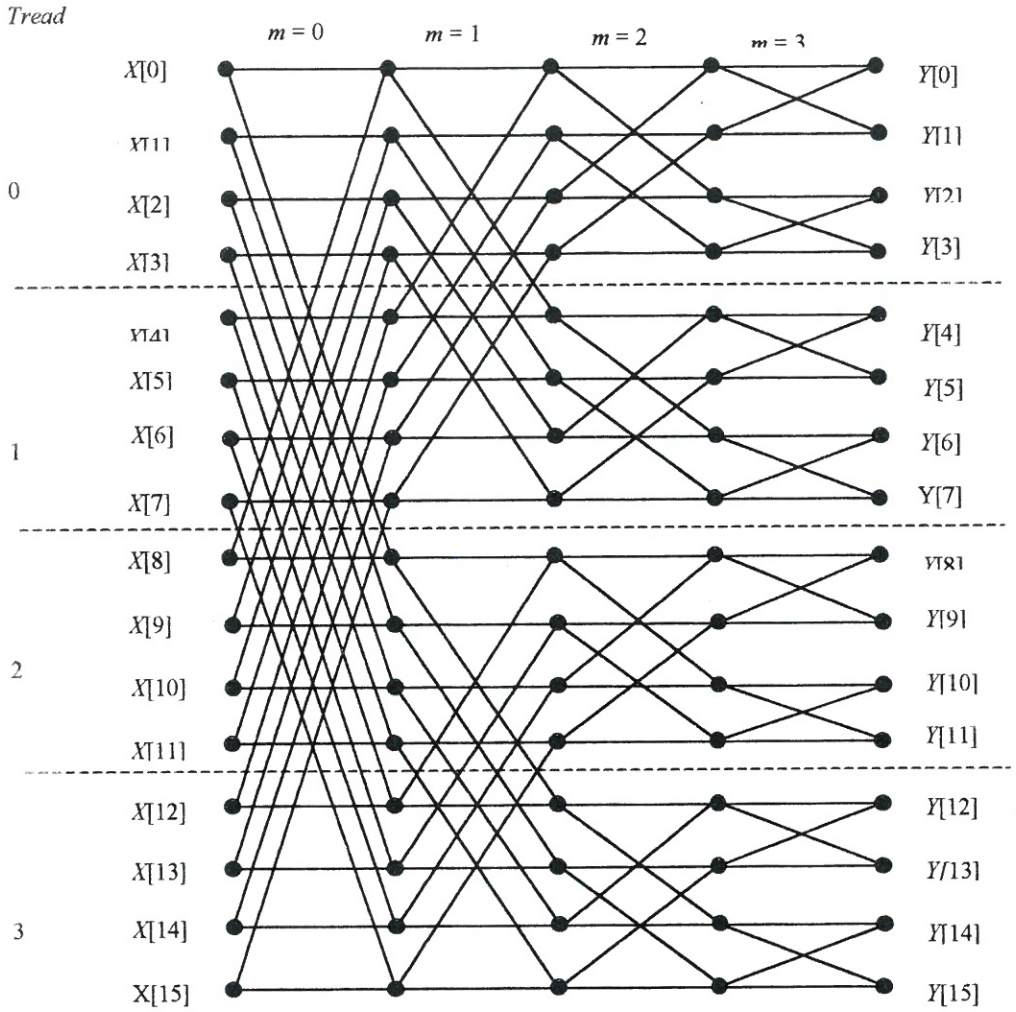

Figure 2. The Computation Diagram of FFT for $n = 16$ Points

3. Fine-grain Parallelization of FFT

For parallelization of the FFT model we consider a mathematical representation of the model as algebraic functions that operate on data-vectors and the parallelization efforts will be directed towards an efficient distribution of loops that implement these functions. In this approach, the FFT algorithm can be viewed as three nested loops (an outer loop and two inner loops, see Figure 1) and the data-dependencies analysis must find out which one of these loops can be parallelized.

The essential condition which needs be met in parallelizing a loop in a shared-memory multiprocessing is that each iteration of the loop is independent of all other iterations [3]. If the loop meets this condition, then the order in which iterations of the loop get executed does not matter any more: they can be executed either sequentially or concurrently, and the result is the same. This property is known as data-independence. For a loop to be data-independent, no iteration of the loop can write into a shared location that is read or written by the other iteration of that loop.

The outer loop of the FFT program is a data-dependent loop because each iteration uses elements of the vectors (R and S) updated in a previous iteration. As consequence, this loop cannot be distributed for parallel execution on different processors.

Each of the inner loops of the algorithm is a data-independent loop, because each iteration of these loops updates a different element of vector S, respectively R (in the iteration i, elements $R[i]$, respectively $S[i]$ are updated). Consequently, each inner loop can be correctly parallelized, by distributing the iterations to a number of p threads, executed on different processors. The distribution of the inner loops is equivalent to a partition in the data space, i.e. each processor, running a different thread, will be assigned n/p elements of vectors R and S. Running the inner loop of an algorithm in parallel makes a class of parallelism called "fine-grain parallelism" because the size of the computational grain distributed to concurrent threads is the smallest one in the given algorithm.

Even though these inner loops are data-independent, other type of data-dependency appears in the FFT model and these must be removed, for a correct parallel execution of the algorithm. This dependency is carried on at consecutive steps, when a step requires elements of the vectors which have been computed by multiple threads at a previous step. In this case, each thread must wait for other threads to complete execution of the step. This waiting is a synchronization point between threads: no thread can continue the execution until all threads have reached this point. This synchronization mechanism is a barrier, and it removes the data-dependencies between consecutive steps executed on multiple threads.

4. Implementation and Performances' Evaluation of the Parallel FFT Algorithm

In multithreaded execution on an p processors system, the main thread of the program launches p workers threads and, after that, waits for completion of these. The coefficient's vectors R and S, as well as input vector X and output vector Y are global, shared variables and are

```
void *thread_FFT (void *param) {
      int tid = *(int *)param;
      long s = n/p;

      /* Initialization of vector Ŕ */
      for (i = tid*s; i < tid*(s + 1); i++)
            R[i] = X[i];


      for (m = 0; m < r; m++){

         if (m < log p) barrier(p);
         for (i = tid*s; i < tid*(s + 1); i++)
            S[i] = R[i];
         if (m < log p) barrier(p);

         for (i = tid*s; i < tid*(s + 1); i++) {
            i = (b0b1 ... br-1) /*binary representation of i */
            i1 = (b0 ... bm-10bm+1 ... br-1);
            i2 = (b0 ... bm-11bm+1 ... br-1);
            R[i] = S[i1] + S[i2] x ω exp(bmbm-1 ... b00 ... 0);
         }
      }
      /* Update output vector Y */
      for (i = tid*s; i < tid*(s + 1); i++)
            Y[i] = R[i];
      }
```

Figure 3. Thread Function in Parallel FFT Algorithm

partitioned between the p executive threads that concurrently run on p processors. Each thread determines its partition of $s = n/p$ coefficients, by using the total number of coefficients (n), the number of threads (p) and an index of its own,called *tid* (0 ... p). In Figure 2, for $n = 16$ points, the coefficients (0,1,2,3) are assigned to the thread with *tid* = 0, the coefficients (4, 5, 6, 7) are assigned to the thread with *tid* = 1, a.s.o.

The function executed by one thread, with index *tid* is presented in Figure 3. In this function, two barriers are needed for the first ($\log p$) iterations of the outer loop (i.e. for $m = 0, 1, \ldots, \log p - 1$). The first barrier inserted before the first inner loop ensures that all threads have completed step 2 of the precedent iteration. The second barrier inserted before the second inner loop ensures that all threads have finished step 1 of the current iteration. After ($\log p$) iterations of the outer loop, all coefficients used by each thread belong to the same partition, assigned to a single thread, and synchronization points are no more required.

The initialization of vector R from input vector X, the updating of output vector Y, are also distributed between worker threads, each of them updating a partition of the vectors.

A theoretical analysis of the parallel FFT algorithm determines a parallel computing time (T_P), speed-up (S) and efficiency (E) of the algorithm [2].

The parallel computing time has two components. The first component (T_{PE}) represents the effective time consumed with multiplications and summations of coefficients on p processors. As the external loop is computed $\log n$ times and the internal loop is computed by each processor for n/p coefficients, here results:

$$T_{PE} = O\left(\frac{n}{p}\log n\right) \qquad (4)$$

If we assume that the time spent by each processor at a synchronization point (barrier) is a polynomial (possibly linear) function of the number of processors, then the synchronization time is $T_{PS} = O(p \log p)$ because there are ($2 \log p$) synchronization points. So, the parallel computing time is:

$$T_P = O\left(\frac{n}{p}\log n + p\log p\right) \qquad (5)$$

The speed-up S and efficiency E result:

$$S = \frac{T_S}{T_P} = O\left(\frac{p}{1+\frac{p^2\log p}{n\log n}}\right);$$

$$E = \frac{S}{p} = O\left(\frac{1}{1+\frac{p^2\log p}{n\log n}}\right). \qquad (6)$$

As an observation, if $n >> p$ (for example $n = 2^{25}$ and $p = 4$) we can obtain a high speed-up and efficiency ($S \approx p$ and $E \approx 1$), which was proved by experimental results.

5. Experimental results and conclusions

The parallel algorithm was implemented in POSIX 1003 standard on two systems: a Silicon Graphics Onyx2 with four processors under IRIX 6.4 operating system and a IBM RS/6000 with two processors under AIX 4.3 operating system [3]. As input data were used vectors with different dimensions. Table 1 represents the speed-up versus vector dimension (n), for different number of processors (p).

Table 1. The Speed-Up As A Function of the Number of Points of the FFT

Number of processors (p)	$n = 2^5$	$n = 2^{10}$	$n = 2^{15}$	$n = 2^{20}$	$n = 2^{25}$
1	1	1	1	1	1
2	1.69	1.92	1.92	1.94	1.94
4	3.4	3.82	3.91	3.92	3.93

The results obtained in a multithreaded implementation of FFT algorithm prove that the speed of computation increases by using a multiprocessor system. Even if the number of processors on such a system is, in general, small, the high-performance processors and the short time consumed for communication via shared memory leads to a significant decrease of computational time reporting to a sequential system.

REFERENCES

1. KUMAR, V., GRAMA, A., GUPTA, A. and KARPIS, G., **Introduction to Parallel Computing**, THE BENJAMIN/CUMMINGS PUBLISHING COMPANY, INC., Redwood City, CA, 1994.

2. QUINN, M., **Parallel Computing- Theory and Practice**, MCGRAW-HILL, New-York, 1994.

3. HWANG, K., **Advanced Computer Architecture: Parallelism, Scalabilily, Programmability**, MCGRAW-HILL, New-York, 1993.

4. IBM, **AIX Version 4.3 General Programming Concepts**, 1997.