Parallel Implementation of Fast Hartley Transform (FHT) in Multiprocessor Systems

Felicia Ionescu, Andrei Jalba, Mihail Ionescu

University "Politehnica" Bucharest,
Str. Iuliu Maniu Nr. 1-3, Bucharest, Romania
{fionescu, andrei, mihail}@atm.neuro.pub.ro

Abstract. The purpose of this paper is to investigate the parallelization of one-dimensional Fast Hartley Transform (FHT) algorithm on shared memory multiprocessor systems. The computational dependencies of the sequential FHT algorithm are analyzed, in order to distribute the loops of the algorithm among multiple processes (threads), executed on the available processors of the system. The outer loop of the algorithm carries data dependencies between consecutive iterations and, for parallel execution, synchronization barriers are introduced. The results show that in the parallel execution of the FHT algorithm a significant speed-up is obtained and that the speed-up increases with the size of the input sequence.

1 Introduction

Discrete Hartley Transform (DHT), like *Discrete Fourier Transform* (DFT), plays an important role in digital signal processing. DFT is very used, but it includes complex arithmetic even if the input sequence is a real one. Hence, DHT was developed to eliminate this redundancy for a real sequence of numbers. The DHT of a sequence of N real numbers X(i), i = 0,1,....,N – 1, is:

$$H(k) = \sum_{i=0}^{N-1} \left(X(i)\left[\cos\left(2\pi ki / N\right) + \sin\left(2\pi ki / N\right)\right]\right). \quad (1)$$

The computational complexities of both transformations, DFT and DHT, are in $O(N^2)$. As well as DFT, DHT has a fast version called *Fast Hartley Transform* (FHT) [1], with the time complexity in $O(N \log_2 N)$. The purpose of this paper is to investigate the parallelization of FHT algorithm on shared memory multiprocessors.

2 The Analysis of the Sequential FHT Algorithm

Several different forms of the FHT algorithm exist, but in this paper we use for parallelization a radix-2 decimation-in-time FHT algorithm [2]. Fig. 1 illustrates the computational flow graph for N = 8-points FHT algorithm.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 532-535, 2000.

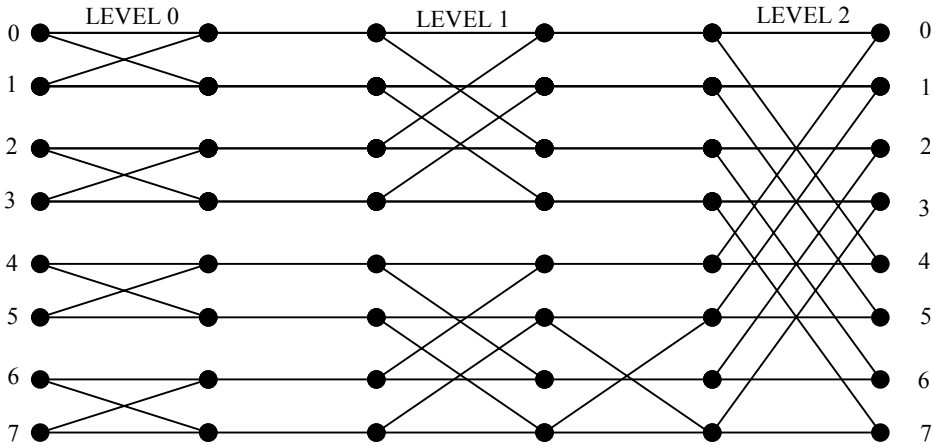

Fig.1. Computational flow graph for the 8-points FHT

An examination of Fig. 1 reveals that FHT computations at each level resemble basic FFT butterfly computations. The first level (r = 0) consists of 2-points FFT-like butterflies and the remaining levels (r = 1,2,...n – 1, where $n = \log_2 N$) consist of 4-points FHT butterflies. There are two types of FHT butterflies, which will be referred here as T1 and T2 4-points basic butterflies. Each type of FHT butterflies, which are presented in Fig. 2, is identified by a 4-tuple (p, r, q, s).

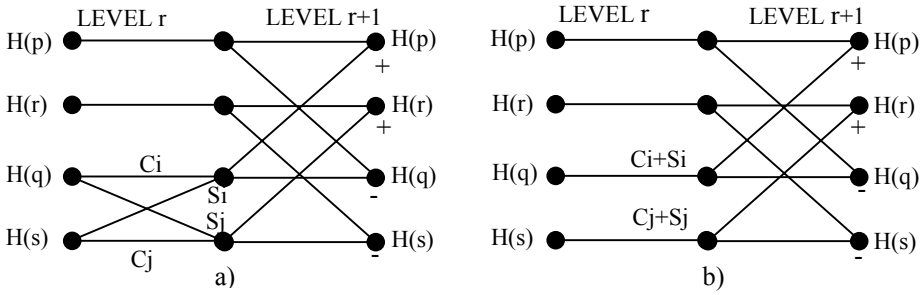

Fig. 2. Computational flow graphs for a) T1 and b) T2 4-points butterflies. Ci = cos (2πi / N), Si = sin (2πi / N)

The C-like pseudo-code of the sequential FHT algorithm is given bellow.

```
/* Sequential FHT algorithm
   Input in bit-reversed order in H[0…N-1]
   Output in normal order in H[0…N-1] */
for(i = 0; i < N/2; i++){
  temp = H[2i+1];
  H[2i+1] = H[2i] - temp;
  H[2i] = H[2i] + temp;}
```

```
for(r = 1; r < n; r++) {
  for(i = 0; i < N/pow(2,r+1); i++) {
    p2 = i*pow(2,r+1); q2 = p2 + pow(2,r);
    r2 = p2 + pow(2,r-1); s2 = q2 + pow(2,r-1);
    tmp1 = H[q2]; tmp2 = H[s2];
    H[q2] = H[p2] - tmp1; H[s2] = H[r2] - tmp2;
    H[p2] = H[p2] + tmp1; H[r2] = H[r2] + tmp2;
    for(j = 1; j < pow(2,r-1); j++) {
      p1 = p2 + j; q1 = p1 + pow(2,r);
      r1 = p2 + pow(2,r) - j; s1 = r1 + pow(2,r);
      tmp1 = Ci*H[q1] + Si*H[s1];
      tmp2 = Cj*H[s1] + Sj*H[q1];
      H[q1] = H[p1] - tmp1; H[s1] = H[r1] - tmp2;
      H[p1] = H[p1] + tmp1; H[r1] = H[r1] + tmp2;
    }
  }
}
```

As it is shown above, the first *for* loop performs computations required by the first level (r = 0), corresponding to the 2-points butterflies. The second outer *for* loop performs the computations for the remaining n – 1 levels. The first inner *for* loop iterates $N/2^{r+1}$ times to compute the T2 butterflies at each level r and the innermost *for* loop iterates $2^{r-1}-1$ times to compute the T1 butterflies. The numbers of T1 and T2 butterflies at level r are:

$$S_{T1}(r) = N/4 - N/2^{r+1};\ S_{T2}(r) = N/2^{r+1}. \quad \textbf{(2)}$$

3 Parallelization of the FHT Algorithm

For parallelization of N-points FHT on a shared memory multiprocessor system, a number of P threads run on P processors of the system, executing the code that implements the parallel version of the algorithm. The only computational dependency in the sequential version of the algorithm exists between successive levels because level-r computations depends on the level-r-1 results. For this reason, we distribute the iterations of each level loop and provide synchronization mechanisms between threads (implemented using barriers) at the beginning of each level. These barriers are needed before the beginning of the second outer *for* loop that performs the computations for levels 1,2,.... n –1 and before the first inner *for* loop that iterates $N/2^{r+1}$ times to compute the T2 butterflies.

Because the number of iterations of the *for* loops which compute the T1 and T2 butterflies are dynamically varying as a function of r (the current level), we have chosen to parallelize the *for* loop with the greatest number of iterations. The level r for which the number of iterations of these two *for* loops is the same is obtained by equaling the numbers of butterflies of type T1 and T2 given by (2) and has the value r = 2. For $r \leq 2$ the loop corresponding to T2 butterflies is parallelized; for $r > 2$ the

loop corresponding to T1 butterflies is parallelized; in this case, the computations for the T2 butterflies are done only by one process (thread).

The pseudo-code of the parallel version of the algorithm is presented bellow.

```
/* Parallel FHT Algorithm */
forall(0 ≤ p ≤ P-1) {
  for(j = p*N/(2P); j < (p+1)*N/(2P); j++) {
    temp = H[2j+1]; H[2j+1] = H[2j] - temp;
    H[2j]= H[2j] + temp;}
}
barrier synchronization;
for(r = 1; r < n; r++) {
  barrier synchronization;
  if(r ≤ 2)
    forall(0 ≤ p ≤ P-1) {
      for(i = p*N/(P*pow(2,r+1));
          i < (p+1)*N/(P*pow(2,r+1)); i++) {
        Compute T2 butterfly;
        for(j = 1; j < pow(2,r-1); j++)
          Compute T1 butterfly;
      }
    }
  else
    for(i = 0; i < N/pow(2,r+1); i++) {
      Compute T2 butterfly;
      forall(0 ≤ p ≤ P-1) {
        for(j = p*pow(r-1)/P;
            j < (p+1)*pow(2,r-1); j++)
          Compute T1 butterfly;
      }
    }
}
```

4 Results and Conclusions

For evaluation of performances, the parallel FHT algorithm was implemented on a two-processor IBM RS/6000 station, under AIX 4.3 operating system. The initial data are arrays of different dimensions and their elements are double-precision real numbers. The results show that the execution speed of the parallel algorithm can be increased using all processors in a multiprocessor system and that the speed-up increases with the size of the input sequence.

References

1. Bracewell, R.N.: The Fast Hartley Transform. Proceedings of IEEE, Vol. 72 (1984) 124-132
2. Aykanat, C., Dervis, A.: Efficient Fast Hartley Transform Algorithms for Hypercube-Connected Multicomputers. IEEE Transactions on Parallel and Distributed Systems, Vol. 6, (1995) 561-577