# A Practical OpenMP Implementation of Bit-Reversal for Fast Fourier Transform

Tien-Hsiung Weng[1], Sheng-Wei Huang[1], Ruey-Kuen Perng[1], Ching-Hsien Hsu[2], and Kuan-Ching Li[1]

[1] Department of Computer Science and Information Engineering Providence University
Shalu, Taichung 43301 Taiwan
{thweng,rkperng,kuancli}@pu.edu.tw

[2] Department of Computer Science and Information Engineering Chung Hua University
Hsinchu 300 Taiwan
robert@grid.chu.edu.tw

**Abstract.** In this paper, we describe our experience of creating an OpenMP implementation of Bit-reversal for Fast Fourier Transform programs from the existing un-parallelizable sequential algorithm. The aim of this work is to present an analysis of a case study showing the development of a shared memory parallel Bit-reversal for the FFT parallel code with practical and efficient use of multi-core machines. We present our implementation and discuss the results of the case study in terms of program improvement that may be needed to help parallel application developers with similar high performance goals. Our preliminary studies, results and experiments based on FFT code running on a four 4-cores Intel Xeon64 CPUs /Dell 6850 platform. The experimental results show that the performance of our new parallel code on 16 cores shared-memory machine are promising.

**Keywords:** Shared-memory parallel programming, OpenMP, Bit-reversal, FFT.

## 1 Introduction

The Fast Fourier Transform (FFT) is one of the most important algorithms used in many fields of science and engineering, especially in signal processing and computational fluid dynamics for solving PDEs. The FFT [2] uses a divide and conquer strategy to evaluate a polynomial of degree n at the n complex nth roots of unity. FFT is easier to parallelize and many parallel FFT algorithms on shared memory machines have been well studied and developed. But practical implementation of The FFT programs consist of two main parts. First, data reordering in the Fast Fourier Transform by permuting the element of the data array using Bit-reversing of the array index. This first stage involves finding the DFT of the individual values, and it simply passes the values along. Next, each remaining stages the computation of a polynomial of degree n at the n complex nth roots of unity is used to compute a new value depends on the values of the previous stage; this process, we called butterfly operation. A Bit-reversal is an operation for an exchange data between A[i] and A[bit-reversal[i]], where the value of i is from 0 to n-1 and value of n is usually 2 to the power of b.

Then bit-reverse[i] = j, the value of j is obtained from reversing b bits from value of i. When the Bit-reversal is not properly designed, it can take a substantial fraction of total execution time to perform the FFT [4].

Our aim in this work was to realize an OpenMP implementation of the Bit-reversal for FFT from un-parallelizable sequential one. Our approach relies on performing single program multiple data (SPMD) style by adding the OpenMP parallel directives. We first discuss the related works. Later, we discuss the important design of our algorithm and compared with the original sequential algorithm. Then, we present the results of our evaluation of this parallel version of Bit-reversal on a four 4-cores Intel Xeon64 CPUs / Dell 6850 platform. We also discuss the possibilities for and challenges of further improvement of the parallel program in Section 5. Finally, we give our conclusions and future plans.

## 2 Related Work

Our aim in this work was to realize an OpenMP implementation of the Bit-reversal for FFT from un-parallelizable sequential one. Our approach relies on performing single program multiple data (SPMD) style by adding the OpenMP parallel directives. We first discuss the related works. Later, we discuss the important design of our algorithm and compared with the original sequential algorithm. Then, we present the results of our evaluation of this parallel version of Bit-reversal on a four 4-cores Intel Xeon64 CPUs / Dell 6850 platform. We also discuss the possibilities for and challenges of further improvement of the parallel program in Section 5. Finally, we give our conclusions and future plans.

The data reordering in the FFT program using Bit-reversing of array index has been well studied [3][4][6][8][9][10][12]. Most of the algorithms proposed are mainly for the uniprocessor [4][6][8][9]. The optimal Bit-reversal using vector permutations have been proposed by Lokhmotov [6]; their experiments have been run on single processor, but they claimed that their algorithm can be parallelized as well. Takahashi [13] implemented an OpenMP parallel FFT on IA-64 processors. Their code is a recursive FFT algorithm written in Fortran 90, but their bit-reversal algorithm is not presented. An algebraic framework for FFT permutation algorithm using Sisal, a functional language , an performance measurement was done on a Cray C-90 and SUN Sparc 5 machines [3][10]. Bit-reversal program must be carefully designed because it may take about 10-30% of the overall FFT computational time [6]. Not only that, it also can produce significant cache misses when its input data size is very large. This is due to the exchange between two data elements of an array that located in the distance far apart during the permutation of the data element using Bit-reversal of the array index. In addition, some of Bit-reverse sequential algorithms are non-trivial to be parallelized.

Our work is based on the sequential Bit-reversal algorithm proposed by Rodriguez [8], in which they designed an improved Bit-reversal algorithm for FFT. Even though their sequential algorithm appeared to be the best, it is not parallelizable without completely rewriting of its algorithm into a parallel one.

In this paper, we proposed our OpenMP implementation of Bit-reversal for the FFT using the so-called SPMD (Single Program Multiple Data) style of OpenMP, in which reducing number of cache misses and data locality are the main concern in the design of our code. The SPMD style of OpenMP code is distinct from ordinary OpenMP code. In most ordinary OpenMP program, shared arrays are declared and parallel for directives are used to distribute work among threads via explicit loop scheduling. In the SPMD style, systematic array privatizations by creating private instances of sub-arrays gives opportunities to spread computation among threads in the manner that ensures data locality [5]. An in depth study about SPMD style of OpenMP can be found in [5]. Programs written in SPMD style of OpenMP has been shown to provide scalable performance that is superior to a straightforward parallelization of loop for ccNUMA systems [6][11]. More advantages of using OpenMP to parallelize our code are portability, easy to use, easy to maintain as well as incremental parallelization. OpenMP [7] is an industry standard for shared memory parallel programming agreed on by a consortium of software and hardware vendors. It consists of a collection of compiler directives, library routines, and environment variables that can be easily inserted into a sequential program to create a portable program that will run in parallel on shared-memory architectures. It is considerably easier for a non-expert programmer to develop a parallel application under OpenMP than under either Pthreads or the de facto message passing standard MPI. OpenMP also permits the incremental development of parallel code. Thus it is not surprising that OpenMP has quickly become widely accepted for shared-memory parallel programming.

## 3 OpenMP Implemention

In this section, we give an overview of original un-parallelizable sequential Bit-reversal programs developed by Rodriguez [8]. Next, in Section 3.2 we present and discuss our OpenMP implementation. We describe the steps taken to create the OpenMP program as well as how we rewrote the program from the un-parallelizable one. We also explain our development of parallel Bit-reversal implementation using OpenMP SPMD style by examples.

### 3.1 Brief Overview of Sequential Bit-Reversal

Our work is based on an improved bit-reversal algorithm for the FFT by Rodriguez [8]. In their original algorithm, computation of the bit-reversal of index for data reordering calculates only the required bit-reversal of indices, which also eliminates the number of unnecessary bit-reversal and swaps. The bit-reversal is computed as bitrev= $\sum_{p=0}^{k} b_{p-1-k} 2^k$ ; this corresponds to the sequential pseudo-code as shown in Figure 1.

It uses only array A to store its input data and final results. When the algorithm uses only array A, the data reordering must perform the exchange between elements of A. Even though the swapping is only a simple exchange between the two elements of an array, it actually involves three assignment statements or copies actions. For instances, the swap(A[1],A[2]) produces the copy A[1] to Temp, then A[2] to A[1], and then Temp to A[1].

In this original sequential code as shown in Figure 1, it computes the index upper bound for the variable last = (N - 1 - $N_2$) where $N_2$ is $\sqrt{N}$ when number of bits is even and is $\sqrt{2N}$ when number of bits is odd. This eliminates the unnecessary computation of bit-reversal, which reduces number of swaps. In term of the number of moves, it is actually takes $3*(N-N_2)/2$ moves, which is $1.5*(N-N_2)$.

```
0 Bit-reverse(N,p) {
1   NV2 = N >>1;
2   last = (N-1)-(1<<((p+1)>>1));
3   j=0;
4   for(i=1;i<=last; i++)
5   {  for(k=NV2; k<=j; k>>=1) j -=k;
6      j += k;
7      if (i < j) Swap(A[i],A[j]);
8 }
```

**Fig. 1.** An Improve Bit-reversal function by Rodriguez

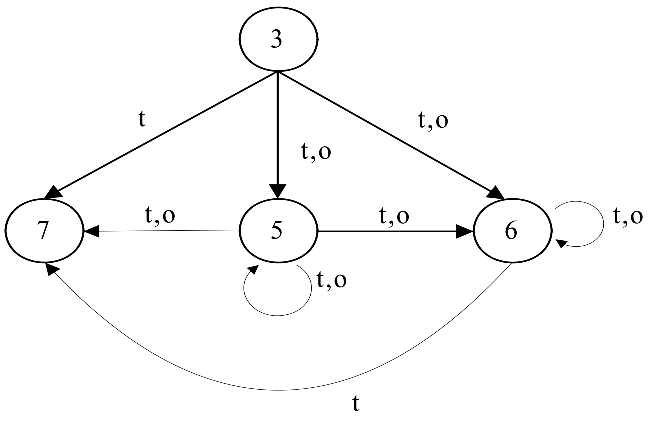

**Fig. 2.** Data dependence graph of Figure 1

In Figure 1, the program code is not parallelizable due to true and output data dependences between loop iterations of statements j = j - k, j = j + k, and if (i<j). Its data dependence graph is shown in Figure 2; there are true and output dependences between the statements labeled on each edge as t and o respectively. For instance, there are true dependence between statement 3 and 7; true and output dependences between statements 5 and 6, between 5 and 7; the loop on node 6 means there is true and output dependence between statement 6 itself on different iteration of the for loop. As the result, the value of variable j is accumulated for the entire nested for loop, this means that the computation for value of j is depend on the previous value of j. Hence, in order to parallelize this code, modification of this code is mandatory.

## 3.2 Our OpenMP Version of Bit-Reversal

We implemented our parallel code in this paper; it is quite similar to the one developed by Rodriguez, but with modification as shown in Figure 3 and 4. The original program designed by Rodriguez was not parallelizable because there are true data dependences, and therefore it is impossible to directly add the parallel OpenMP directive to this original for loop.

Figure 3 shows part of the FFT code that only computes data reordering performed by permuting the element of the data array A into $M_f$ using Bit-reversing of the array index. That is $M_f$[i]=A[Bit-reverse[i]]. Instead of putting the result in A, we store the result into $M_f$. Both arrays have size of n, where n is 2 to power of k, where k is the number of bit to be reversed. In our program, we use more memory spaces in order to parallelize the code, reduce the total number of data copy; so we trade the space for shorter execution time.

In order to write a Bit-reversal parallel code using the SPMD style of OpenMP from the existing un-parallelizable sequential code, we first remove the true data

```
FFT(*A,n,nthreads) {
struct COMPLEX Mf[n],Nth[n/2],Tmp[n/2];
chunksize = n / nthreads;
int offset[nthreads];
 for(i=0;i<nthreads;i++) {
 if(i==0) j=0;
 else {
         k=nthreads/2;
         while(k<=j) {
            j=j-k; k=k/2;
         } //end while
         j=j+k;
 } //end else
 offset[i]=j;
 } // end for
 #pragma omp parallel private(threadid, address)
   { threadid = omp_get_thread_num();
     address = chunksize * threadid;
     Bit_reverse(A,Mf+address,
                  n,chunksize,offset[threadid]);
 }
 ...
 ...
 }
```

**Fig. 3.** Main function of the OpenMP FFT

dependence between iteration of variable j, in this case, by the pre-computation of the accumulated value of j for each starting chunk of iteration that will be executed in parallel by a thread. These accumulated starting values of j for each chunk of iteration are stored in the array offset. During the parallel execution of each threads, master thread or thread 0 will access the j from offset[0], thread 1 will access the accumulated starting value of j from offset[1], thread 2 will access accumulated starting value of j from offset[2], and so on. At the first glimpse on this observation,, it seems more pre-computation for accumulated value of j to be done, in reality, it actually only compute k number of elements for array offset, where k is number of threads.

Depends on nthreads (the number of threads), we compute chunk size by dividing the input size by the number of threads. Inside the parallel region where we added the *pragma omp parallel* directive, each thread will compute its address by multiplying the chunk size with its thread id (where master thread is 0, thread 1 is 1, etc.); the two variables (treadid and address) are declared to be private to thread using *private* clause. Next, each thread will call Bit-reversal function with five parameters as shown in Figure 4. First parameter is the first location of array A. Second is address or location of the $M_f$ ($M_f$+address). Third, n is the size of input, then chunk size, and finally, the offset that have computed earlier.

In each thread, the Bit-reversal function is executed to compute different chunk of element $M_f[i]$, which is correspond to different part data element of $M_f$ that is passed from $M_f$+address. For example, with n equals to 16, and number of threads equals to 2, master thread (or thread 0) will handle the $M_f$ [0] corresponds to $M_f$ +0; thread 1 will process $M_f$ [0] corresponds to $M_f$ +8, this means $M_f[0]$ is the pointer to actual location 8 of $M_f$, $M_f$ [1] is location 9 of $M_f$, and so on.

To better illustrate our OpenMP implementation of parallel code shown in Figure 3, we use a call to FFT(A, 32, 4) as an example, that is we call FFT with size of 32 and 4 number of threads. The chunk size of 8 is calculated. We then allocate array of size

```
Bit_reverse(*A, *Mf, n, chunksize, offset) {
   for(i=0;i<chunksize;i++) {
      if(i==0) j=0;
      else {
          k=n/2;
          while(k<=j) {
             j=j-k; k=k/2;
          }//end while
          j=j+k;
}//end else
      Mf[i]=A[j+offset];
    }
  }
```

**Fig. 4.** Bit-reversal function

equal to number threads for offset, in this case, int offset[4]. Before we perform the Bit-reversal in parallel using OpenMP SPMD style to perform the permutation, we compute only four values to store in four elements of the offset, in this examples, we obtain 0, 2, 1, and 3 for offset[0], offset[1], offset[2], and offset[3] respectively. Inside the OpenMP parallel region where the pragma omp parallel directive is added, four threads will be created; each thread executes Bit-reversal function in Figure 4 with parameters to perform different part of data computation. In this case, thread 0, 1, 2, and 3 will call Bit-reverse with parameters ($M_f$+(0*8) and value of offset[0] is 0), ($M_f$+(1*8) and the value of offset[1] is 2), ($M_f$+(2*8) and offset[2] is 1), ($M_f$+(3*8) and the value of offset[3] is 3) respectively.

Hence, as shown in Figure 5, master thread(or thread 0) will handle the $M_f$[0] corresponds to $M_f$+0; then it computes $M_f$ [i]=A[j+offset] for i from 0 to number of chunk size; where the offset[0] is 0, then $M_f$[0]=A[0+0], $M_f$[1]=A[16+0], $M_f$[2]= A[8+0], $M_f$[3]=A[24+0], up to $M_f$[7]=A[28+0].

Thread 1 will process $M_f$[0] corresponds to Mf+8, is the pointer to actual location 8 of $M_f$, $M_f$ [1] is location 9 of Mf, and so on. The offset[1] is 2. So, $M_f$[0]=A[0+2], $M_f$[1]=A[16+2], $M_f$ [2]=A[8+2], $M_f$[3]=A[24+2], up to $M_f$[7]=A[28+2]. Note that $M_f$[0] is actually the $M_f$[8], $M_f$[1] is $M_f$[9], $M_f$[2] is $M_f$[10], etc. Other threads are also performed in similar manner

| Thread ID | Starting address | Value of offset | Mapping | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 16 | 8 | 24 | 4 | 20 | 12 | 28 |
| 1 | 8 | 2 | 2 | 18 | 10 | 26 | 6 | 22 | 14 | 30 |
| 2 | 16 | 1 | 1 | 17 | 9 | 25 | 5 | 21 | 13 | 29 |
| 3 | 24 | 3 | 3 | 19 | 11 | 27 | 7 | 23 | 15 | 31 |

**Fig. 5.** The result of mapping computed by threads

Note that in Figure 5, the variables n, chunksize, offset, i, j, and k are private variables, only array A and $M_f$ are shared variables. Private data is usually stored locally to thread, which promote data locality.

As the results, each of the four threads will call Bit-reverse function that is shown in Figure 5 and they are run in parallel. Thread 0 computes the mapping of 0-0, 1-16, 2-8, 3-24, 4-4, 5-20, 6-12, 7-28. During the generation of this mapping i-j, it copy $M_f$[i]=A[j]. Thread 1 computes 8-2, 9-18, 10-10, 11-26, 12-6, 13-22, 14-14, 15-30, and so on. The complete computation of the mapping of the data reordering is shown in Figure 5.

## 4 Experiments

Our experimental results based on data reordering by Bit-reversal are performed on a four 4-cores CPUs 2.6Ghz Intel Xeon64 / Dell 6850 platform with 4 GB of main memory, 16KB L1 cache, 1MB L2 cache, and 4 MB L3 cache. We compile the parallel version of our program shown in Figure 3 with icc Intel OpenMP compiler with flag –O2 –openmp and run on Linux Cent OS. We run this program with size of 2 to the power of 24, 25, 26, and 27 respectively. Each of this was run in parallel using 1, 2, 4, 8, 16 threads (one thread per core). Most of the related works have been run with data size of less than $2^{23}$. Our experiment with data size of $2^{27}$, the program approximately allocates total of 1 GB main memory.

To explain the shortcoming about the performance of our design algorithm on a uniprocessor machine, we also wrote a version of sequential algorithm, compiled with gcc with –O2 flag, and then run with different size of input data to compare with the original sequential code designed by Rodriguez. The experimental result is shown in Figure 6.

Our version of sequential code labels Wen and original sequential version by Rodriguez label Rod. Both run with input data size of $2^{20}$, $2^{21}$, up to $2^{27}$. The results show that the performance of our sequential code is around 50% percent slower than the original sequential code as the input size grows larger. In the original program, Rod, there are approximately only N/2 number of swap; even though each swap involves 3 copies or moves, so with total of 1.5N number of copies or moves, but only one cache miss on each swap will occurred, so the total of misses is N/2 times. Our sequential code involves only N copies or moves, but each move will cause cache miss, hence there will be approximately the total of N cache misses.

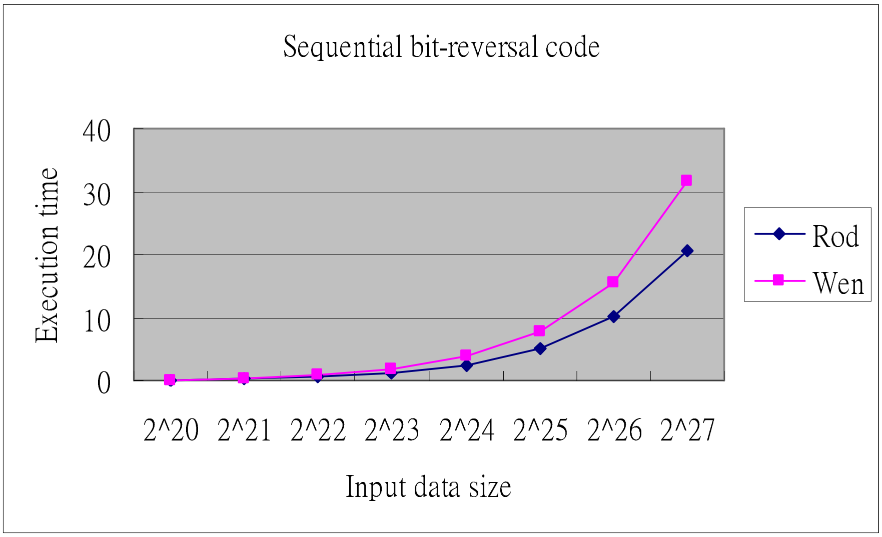

**Fig. 6.** Sequential codes run on uniprocessor

Figure 7 shows the performance of our OpenMP Bit-reversal. Our experiment is performed base on our SPMD style of OpenMP parallel code with different input sizes starting from $2^{24}$ up to $2^{27}$. The performance results of our parallel SPMD style of OpenMP code shows scalable as the number of processors increases. With number of thread equals to one, the large number of cache misses overhead occurred, this lead to longer execution time, which is twice as much as the number of size increases. This overhead is amortized by the increasing number of threads. With increased number of threads, the total number of cache misses is reduced to as the number of chunk size. For instance, with n equals to $2^{24}$, with one thread, the number of cache miss will be $2^{24}$ times, and with 16 threads, the cache misses will reduce to $2^{24}/16$ times.

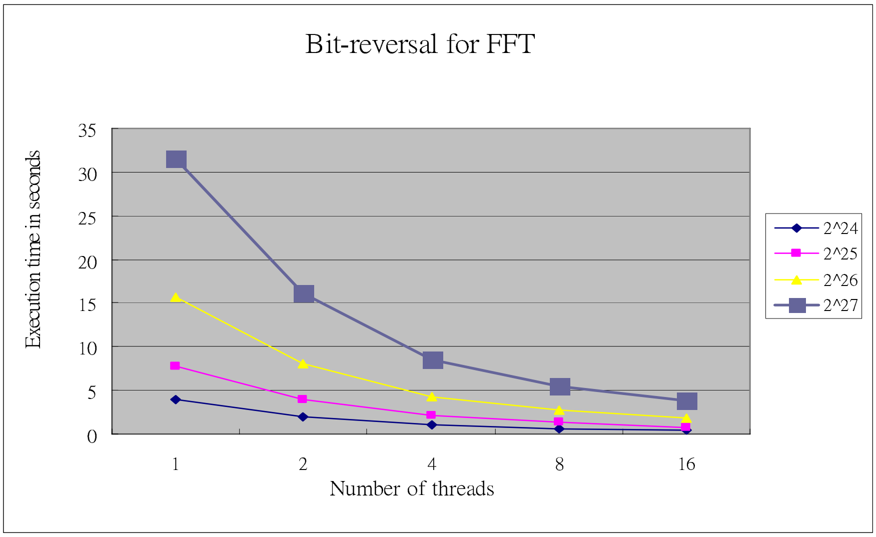

**Fig. 7.** Data reordering in FFT by permuting by bit-reversal

## 5 Possible Improvement

First we discuss the short coming encounter for the original code. In Figure 1, when we call this Bit-reverse with N=16 and p=4, we obtain the swap of A between 1-8, 2-4, 3-12, 5-10, 7-14, and 11-13. The upper bound index is 11, the value calculated for the variable, last. This program can significantly reduce the number of swaps from N to $(N-N_2)/2$. Even though it reduces the number of swaps, in reality the swap involves three copy actions. But, since the algorithm reduces the number of swap, there are only N/2 numbers of cache misses.

In our sequential code, the problem will occur when N is significantly large. Then, the swap between the two array elements involved exchange of the far distant element which causes large number of cache misses. For example, when $N=2^{26}$, then there are assignment statements as the following: $M_f[1]$ = A[33554432], $M_f$ [2] = A[16777216], and $M_f$ [3] = A[50331648], and so on. The copy of $M_f[1]$ from A[33554432], cause the first cache miss from reading from location 33554432, this causes it to load a consecutive block of array from location 33554432 into the cache probably up to the cache size. Then second assignment also cause the cache miss from reading A[16777216], which also load consecutive block of array from location 33554432 into the cache probably up to the cache size. Similar situations occur for the next iteration.

In our sequential code, it encounters more cache misses for very large input size, moreover, since we only run our code in shared-memory platform that has large (4MB) L3 cache and lower latency of memory access, the impact maybe minor. But to run on ccNUMA machines that have greater latency of remove memory access, the consequence is significant. Therefore, good data locality is needed to overcome these problems.

The possible improvement may be to use Bit-reversal data reordering by vector permutations proposed by Lokhmotov [6], They improved the cache optimal methods for Bit-reversal proposed earlier by Zhang [12], where it is designed to be cache optimal; the input source is copied into the buffer to form the tile. Finally, it copies the data from the buffer tile into the target tile. Its practicality, efficiency, and performance are still under our study and are currently our ongoing works.

## 6 Conclusions and Future Work

We have developed a practical OpenMP SPMD style Bit-Reversal for parallel FFT program from the existing un-parallelizable sequential code. It is a practical, easy to develop and maintain, as well as required only less programming effort. In our algorithm, we design to reduce the number of copy to variable and cache miss as possible when the number of threads getting large enough. Our experimental results are promising in this respect despite more memory spaces are used. Still, there may be more improvements are possible; especially the performance on ccNUMA machines where the memory latency is more significantly larger; therefore, the reduction of the number of cache misses, even for small number of threads is necessary, however, they do require more programming effort. We will continue to work to implement Bit-reversal code into the whole parallel FFT program, either iterative or parallel recursive one using Intel task queuing construct of OpenMP. We will also implement this algorithm using Cilk [14], and then compared them with other implementations.

## Acknowledgements

This paper is based upon work supported in part by Taiwan National Science Council (NSC) Grants no. NSC95-2221-E-126-006-MY3 and NSC96-2221-E-126-004-MY3. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSC.

## References

1. Chapman, B., Bregier, F., Patil, A., Prabhakar, A.: Achieving High Performance under OpenMP on ccNUMA and Software Distributed Shared Memory Systems. Concurrency and Computation Practice and Experience 14, 1–17 (2002)
2. Cooley, J.W., Tukey, J.W.: An algorithm for the machine calculation of complex Fourier series. Math. Comput 19, 297–301 (1965)
3. Bollman, D., Seguel, J., Feo, J.: Fast Digit-Index Permutations. Scientific Progress 5(2), 137–146 (1996)

4. Karp, A.H.: Bit Reversal on Uniprocessors. SIAM Review 38, 289–307 (1996)
5. Liu, Z., Chapman, B., Wen, Y., Huang, L., Weng, T.H., Hernandez, O.: Analyses for the Translation of OpenMP Codes into SPMD Style with Array Privatization. In: Voss, M.J. (ed.) WOMPAT 2003. LNCS, vol. 2716, pp. 26–41. Springer, Heidelberg (2003)
6. Lokhmotov, A., Mycroft, A.: Optimal bit-reversal using vector permutations. In: Proceedings of ACM Symposium on the 19th Parallel Algorithms and Architectures, pp. 198–199 (2007)
7. OpenMP Architecture Review Board. Fortran 2.0 and C/C++ 2.0 Specifications, `http://www.openmp.org`
8. Rodriguez, J.J.: An improved Bit-reversal algorithm for the fast Fourier transform. In: Proceedings of International Conference on Acoustics, Speech, and Signal Processing, vol. 3, pp. 1407–1410 (1988)
9. Rubio, M., Gómez, P., Drouiche, K.: A new superfast bit reversal algorithm. International Journal of Adaptive Control and Signal Processing 16(10), 703–707 (2002)
10. Seguel, J., Bollman, D., Feo, J.: A Framework for the Design and Implementation of FFT Permutation Algorithms. IEEE Transactions on Parallel and Distributed Systems 11(7), 625–635 (2000)
11. Wallcraft, A.J.: SPMD OpenMP vs. MPI for Ocean Models. In: Proceedings of First European Workshops on OpenMP (EWOMP 1999), Lund, Sweden (1999)
12. Zhang, Z., Zhang, X.: Fast Bit-Reversals on Uniprocessors and Shared-Memory Multiprocessors. SIAM Journal on Scientific Computing 22(6), 2113–2134 (2000)
13. Takahashi, D., Sato, M., Boku, T.: An OpenMP Implementation of Parallel FFT and Its Performance on IA-64 Processors. In: Voss, M.J. (ed.) WOMPAT 2003. LNCS, vol. 2716, pp. 99–108. Springer, Heidelberg (2003)
14. Frigo, M., Leiserson, C.E., Randall, K.H.: The Implementation of the Cilk-5 Multithreaded Language. In: ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation, pp. 212–223 (1998)