Implementing FFT using SPMD style of OpenMP

Tien-Hsiung Weng* Sheng-Wei Huang* Won Woo Ro+ Kuan-Ching Li*

*Department of Computer Science and Information Engineering, Providence University
Shalu, Taichung 43301, Taiwan
{thweng, kuancli}@pu.edu.tw
+School of Electrical and Electronic Engineering, Yonsei University
Seoul, South Korea
wro@yonsei.ac.kr

***Abstract*- In this paper, we introduce a parallel version of the Fast Fourier Transform that was created using OpenMP in SPMD style. Our implementation is non-recursive and is based on the conventional Cooley-Tukey algorithm written in C. The aim of this work is show the potential benefit of writing our FFT algorithm in SPMD style which enabled an efficient use of multi-core machines. Our experimental results are based on FFT code running on an AMD-Opteron TM8200 with four 2-core CPUs. The experimental results show that the performance of our new parallel code on an 8-core shared-memory machine is promising.**

I. Introduction

The Fast Fourier Transform (FFT) is one of the most important algorithms used in many fields of science and engineering applications, including signal processing for spectral analysis of speech, image processing, partial differential equations, fluid dynamics, seismic and vibration detection and other related fields and applications. The FFT [1] uses a divide and conquer strategy to evaluate a polynomial of degree n at the n complex nth roots of unity. Parallel FFT algorithms have been well studied and a variety of them developed on shared memory machines, but they were not written in SPMD (Single Program Multiple Data) style of OpenMP. Our implementation of the parallel FFT using OpenMP in SPMD style is practical and offers relatively easy to port to other well known programming models such as MPI, Unified Parallel C, and Cilk, since the SPMD style requires that attention be paid to data locality.

Our FFT programs consist of two main parts. First, data reordering is achieved in the Fast Fourier Transform by permuting the elements of the data array using Bit-reversal of the array index, which we call the *Bit-reversal* computation. This first stage involves finding the DFT of the individual values, and it simply passes the values along. Next, each remaining stage in the computation of a polynomial of degree n at the n complex nth roots of unity is used to compute a new value that depends on the values of the previous stage; this process, we call the *butterfly operation*.

A Bit-reversal is an operation for an exchange of data between $A[i]$ and $A[bit\text{-}reversal[i]]$, where the value of i is from 0 to the input size m where the value of m is usually 2 to the power of b. The $bit\text{-}reverse[i]$ is obtained from by b bits from the value of i. When the Bit-reversal is not properly designed, it can take about 30% of the total time to perform the FFT [3]. The paper is organized as follows: We first discuss the related work in Section 2. Next, we discuss the important aspects of the design of our parallel OpenMP-based algorithm in Section 3. Then, in Section 4, we present the results of our evaluation of this parallel version of FFT on an AMD-Opteron ^{TM}8200 platform with four 2-core CPUs. Finally in Section 5, we give our conclusions and future plans.

II. Related Works

In general, Cooley-Tukey's algorithm performs the Bit-reversal and butterfly operations separately. Bader [14] proposed a modified FFT based on Cooley-Tukey, the novelty of which is that the Bit-reversal and butterfly operation are combined and performed at once, hence saving the Bit-reversal stage computation. This algorithm has been designed well for execution on the Cell processor. As part of the work of this paper, we also rewrite it in SPMD-style OpenMP and name it FFT2, but it does not perform well, as we will discuss in our conclusions.

The data reordering in the FFT program using Bit-reversal of the array index has been well studied [2][3][5][7][8][9][11]. Most of the algorithms proposed are intended for the uniprocessor [3][5][7][8]. The optimal Bit-reversal using vector permutations have been proposed by Lokhmotov[6]; their experiments were run on a single processor, but they claimed that their algorithm can be parallelized as well. Performance measurements of the outcome of an algebraic framework for an FFT permutation algorithm using Sisal, a functional language, were carried out on a Cray C-90 and SUN Sparc5 machines [2][9]. Franchetti [2] used an automatic program generation tool, Spiral, and integrated it in their proposed framework to boost optimizations. The users then can formally specify digital signal processing transforms such as DFT, which Spiral will then rewrite and transform the DFT in the form of mathematical formula into a C program that computes the specified transform. It is optimized to achieve load balancing and avoid false sharing on a given platform. They further propose to extend Spiral to generate parallel multithreaded code such as OpenMP and Pthread code. Namneh [3] compares two versions of one-dimensional FFT algorithms. These two algorithms are implemented based on tree and transpose of one-dimensional FFT parallelized using the Message Passing Interface (MPI) to run on SMPs. The experiments were done on SUN SPARC SMPs consisting of a total of 8 processors with data size (complex numbers) up to 4094KB. The target machine had a crossbar processor interconnection. Takahashi [12] proposed an OpenMP implementation of a one-dimensional recursive algorithm for a parallel recursive FFT on shared memory parallel computers. It is written in Fortran 90, and they give experimental results that utilize a 4-CPU DELL PowerEdge 7150 for 224-point FFT on double-precision complex data.

In this paper, we propose a new OpenMP implementation of both Bit-reversal and butterfly operations for the FFT using the so-called SPMD (Single Program Multiple Data) style of OpenMP. In most OpenMP programs, shared arrays are declared and parallel loop directives are used to distribute the work in the code's loops among threads via explicit loop scheduling. The SPMD style of OpenMP programming is studied in detail in [4] and has characteristics that distinguish it from ordinary OpenMP code. In particular, in the SPMD style, systematic array privatizations, accomplished by creating private instances of sub-arrays, gives opportunities to spread the computation among threads in a manner that ensures data locality [4]. Programs written in the SPMD style have been shown to provide scalable performance that is superior to a straightforward parallelization of loops for ccNUMA systems [10]. More inherent advantages of using OpenMP to parallelize our code are the ease of use, incremental parallelization as well as its portability.

III. IMPLEMENTATION OF OpenMP

```
1 FFT(*A, m, nthreads) {
2  struct COMPLEX M[m],Tmp[m/2],Buff[m/2],*ptr[nthreads];
3  int addr_offset[nthreads], b=log2m, offset[nthreads];
5  for(i=0;i<nthreads;i++) {
6     if(i==0) j=0;
7     else {
8           k=nthreads/2;
9           while(k<=j) {
10             j=j-k; k=k/2;
11           } //end while
12           j=j+k;
13      } //end else
14      offset[i]=j;
15   } // end for
16   chunksize = m / nthreads;
17  #pragma omp parallel private(threadid)
18   { threadid = omp_get_thread_num();
19      ptr[threadid] = &(M[chunksize*threadid]);
20      Bit_reverse(A, ptr[threadid],m,
                    chunksize,offset[threadid]);
21   }
22   Compute_nthroot(A,b);
23   for(twsize=2; twsize <=chunksize; twsize *=2)
24      Trans1(M,A,chunksize,add_off, twsize);
25   Trans2(M,A,Tmp,chunksize,addr_offset,twsize,b);
26 } // end of FFT
```

Fig. 1. Main function of the OpenMP FFT1

Our implementation of FFT in OpenMP can be divided into three parts: the computation of Bit-reversal, the pre-computation of nthroots, and the butterfly operation. This version we named FFT1. In the SPMD style of OpenMP, we use only *omp parallel* directive throughout the entire parallel FFT1 code, and the distribution of work is controlled by the programmer. In contrast, most other implementation of OpenMP use *omp parallel for* directives, this means that the distribution of work is controlled by the compiler and runtime system. The main FFT1 function is shown in Fig. 1.

3.1. Our OpenMP version of Bit-reversal

Our implementation of the OpenMP SPMD Bit-reversal computation is based on the existing sequential algorithm proposed by Rodriguez [7]. Even though their sequential algorithm appeared to be the best, it is not parallelizable without completely rewriting it. An implementation of the parallel Bit-reversal computation using the SPMD style of OpenMP has been performed. The details can be found in [13]. It is shown in line 5-21 of Fig. 1 and its Bit-reversal function is given in Fig. 2. The modification from original sequential code is done by pre-computation of the permutation of array index into array *offset*. Then, a parallel region is created with the desired number of threads, where each thread executes the same function by utilizing its part of the array offset to compute different parts of its input data array *A*; the final result of permutation of the Bit-reversal computation is stored in *M*. It is a non-recursive approach, even though we use the following divide and conquer idea. We use a call to FFT1(*A*,16,4) as an example throughout this paper; with ***m***, input size of 16 and 4 threads, then the number of bits *b* can be computed as log$_2$*m*, and the chunk size is ***m / nthreads*** (the number of threads). In line 19, we use an array of pointers, such that *ptr*[*i*] stores the starting address of the portion of array *M* that belong to thread *i*. It is then passed to the Bit-reverse function.

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

Fig. 2. Bit-reversal function of FFT1

The divide and conquer characteristic is shown in Fig. 3, where the top level represents the original values of *A*. At each subsequent level, it is further divided into two equal chunk sizes recursively: chunk one is obtained from the even index that we have called *even index chunk* and the other is obtained from the odd index, or *odd index chunk*. At the end, the Bit-reversal permutation of the input data is completed. In the example, *M*[0]=*A*[0], *M*[1]=*A*[8], *M*[2]=*A*[4],.., *M*[14]=*A*[7], *M*[15]=*A*[15] shown at level 5 or the last level. The first shaded element of each chunk at each level is obtained from its parent's first two elements except at level 1. As we observed, the first element of the even index chunk is obtained from its parent first element and the odd index chunk first element is obtained from its parent second element. This first element of each chunk is the *offset* that we will store in the array offset. At level 2, we have two elements in the array offset, and two chunks that are suitable for the distribution of work to two threads. At level 3, we obtain *offset*[0:3] is {0,2,1,3} and four chunks. Similarly, level 4 is appropriate for work-sharing into eight threads and its *offset*[0:7] = {0,4,2,6,1,5,3,7}. The array offset is computed by the code shown on line 5-15 of Fig. 1.

When the number of threads, *nthreads,* is four, line 5-15 of Fig. 1 computes and obtains the offset[0:3]= {0,2,1,3}. Next, the main parallel Bit-reversal computation is performed by the code in line 17-21 of Fig. 1. In line 11 of Fig. 2, if each of the four threads executes *ptr*=*A*[*j*] without adding *offset*, then the result of the first 4 elements of each chunk is shown at level 5 of Fig. 4. By adding *offset* at line 11 of Fig. 2, as *ptr*=*A*[*j*+*offset*], the correct completed results are obtained, as shown at the last level with label "result".

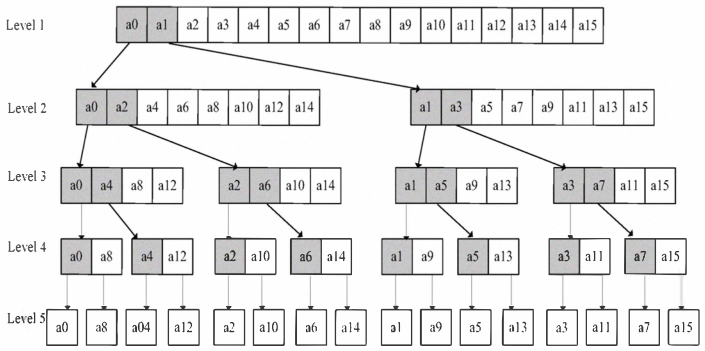


Fig. 3. Bit-reversal by divide and conquer

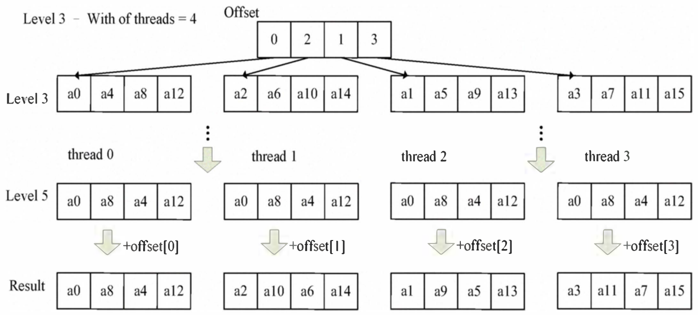


Fig. 4. The execution of Bit-reverse using four threads

Inside the parallel region, each thread computes the starting address of its data accesses by multiplying the chunk size with its thread id. Each thread then calls the Bit-reversal function with 5 parameters, as in Fig. 2. The first parameter is the start location of array *A*. The second is the start address of array *M*, where the final result of Bit-reversal is stored. The third parameter, *m,* is the size of the input data, followed by the chunk size, and finally, the *offset* that was computed earlier.

3.2. *Pre-computation of nthroots*

Initially, the input data is in array *A*. Since the Bit-reversal computation takes array *A* as input and the result of permutations are stored in array *M*, array *A* can then be reused to store the values of pre-computation of nthroots. This computation is shown in Fig. 5 and is done serially because it only takes small amount of time. It pre-computes twiddle factor $\omega = e$^ $(2\pi i/2^s)$ for each stage of the butterfly operations and the results are stored back into array *A* as shown in Fig. 6. In the Compute_nthroots() subprogram, there are *b* stages calls to nthroot(), where $b = \log_2 m$ and *m* is input data size. For each stage *s*, it computes $A[k] = e$^ $(2\pi i/2^s)$ ^ ℓ, where *k* is an array index from 2^{s-1} to $2^s - 1$, $\ell = k - 2^{s-1}$, $s = \ulcorner \log 2\ k \urcorner$, and *i* is a complex number. Later in the butterfly operation, several threads at certain stage will read reference the pre-computed values, thus prevents threads from doing the same calculation, which reduces the number of computations.

```
Compute_nthroots(*A,b) {
   j=1; nth=1;
   for(s=1;s<=b;s++) {
       w=e**(2πi/nth);
       nthroot(A+j,nth,w);
       nth=nth*2; j=j*2;
   }
}
nthroot(*A, nth, w) {
   A[0]=1;
   for(ℓ =1;ℓ <nth/2;ℓ ++)
      A[ℓ] = w**ℓ;
}
```

Fig. 5. The pre-computation of the nth root of unity.

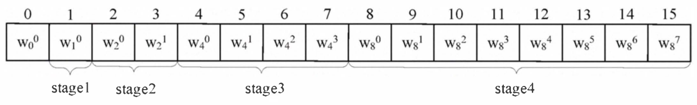

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
w_0^0 w_1^0 w_2^0 w_2^1 w_4^0 w_4^1 w_4^2 w_4^3 w_8^0 w_8^1 w_8^2 w_8^3 w_8^4 w_8^5 w_8^6 w_8^7
stage1
stage2
stage3
stage4

Fig. 6. The result of pre-computation of nthroot

3.3. *Butterfly Operation*

In this section, we discuss butterfly operation using SPMD style of OpenMP, we first use Fig. 7 as a simplified illustration of a butterfly operation. There are three inputs from the left: *a*[*i*], *a*[*i*+*twsize*/2], and on the middle left is the twiddle factor ω^k_n. The cross of two arrows in the middle of the box can be seen as two read references of the array elements into their corresponding two outputs on the right. The down arrow can be seen as the differences of *a*[*i*] and the product of the twiddle factor and a[*i*+*twsize*/2] is output into *a*[*i*+*twsize*/2]. Similarly, the up arrow represents the sum of *a*[*i*] and the product of the twiddle factor and *a*[*i*+*twsize*/2] is output into *a*[*i*].

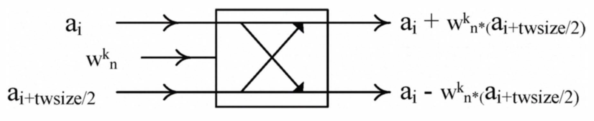
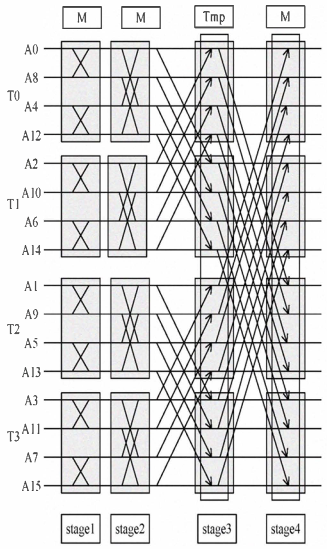

a_i
w^k_n
$a_{i+twsize/2}$
$a_i + w^k_{n*}(a_{i+twsize/2})$
$a_i - w^k_{n*}(a_{i+twsize/2})$

Fig. 7. Simplified drawing of a butterfly operation.

The main butterfly operation starts by calling *Trans*1 and *Trans*2 function at line 23-25 of Fig. 1. *Trans*1 performs the first stage up to *b*-$\log_2$(*nthreads*) stage for which the computation is within chunk size boundary where *b* is number of bits and *nthreads* is the number of threads. The sum and different operation of each thread for each stage where accessing data is within chunk size boundary is controlled by condition (*twsize* < *chunksize*) at line 23 of Fig. 1. On the other hand, *Trans*2 is used to perform the sum and different operation where accessing data is outside the chunk size boundary.



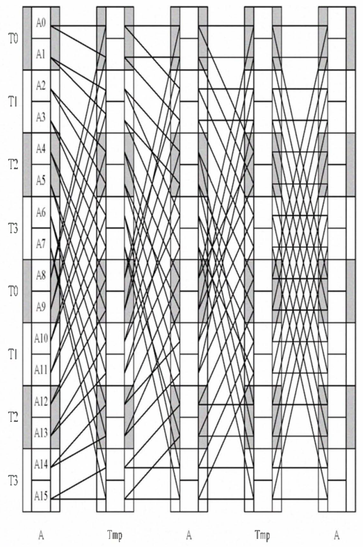


Fig. 8a. Butterfly operation of FFT1 with m=16 and 4 threads

Fig. 8b. Omega operation of FFT2 with m=16 and 4 threads

Using earlier example by calling FFT1(*A*,16,4), the size of input *m*=16, the number of bits *b*=$\log_2(16)$ is 4, and *nthreads* is 4. *Trans*1 in Fig. 9 performs SPMD style OpenMP where each thread executes *Butterfly*1 function that works on different chunks of data where the read reference of data and result of both sum and difference are stored within the chunk size boundary shown in shaded box of stage 1 and 2 in Fig. 8a. Each shaded box represents each thread's chunk boundary. T*i* represents Thread *i*. The first calls to *Trans*1 with the parameter twiddle size equal to 2 in stage 1. The parameter of each next consecutive call is the double twiddle size. The iteration is controlled so that each thread will work on the elements of array access are within the boundary of each chunk size of each thread. Since during the pre-computation of nthroots, the twiddle factor ω^0_n is computed and stored in *A[0]*, ω^1_n in *A[1]*, ω^2_n in *A[2]* and so on. In *Butterfly*1, each thread performs sum and different within the boundary of the chunk size for a thread. The *M*[*i*] is stored in *temp*1. The product of twiddle factor and *M*[*i*+*twsize*/2] is stored in *temp*2 temporarily for eliminating from re-referencing the array *M* and recalculating the product. Then, *M*[*i*]=*temp*1+*temp*2 for the sum operation and *M*[*j*+*twsize*/2] = *temp*1- *temp*2 for the difference. Using our example, *Trans*1 calls the *Butterfly*1 function using SPMD style; thread 0 will compute *temp*1 = *M*[0] and *temp2* = *M*[1]**A*[0], *M*[0] = *temp*1 + *temp*2; *M*[1] = *temp*1 - *temp*2; the second iteration, it computes *temp*1 = *M*[2], *temp*2 = *M*[3]**A*[1]; *M*[2] = *temp*1 + *temp*2; *M*[3] = *temp*1 + *temp*2; thread 1, 2, and 3 is performed the same way. This is done by the code in Fig. 9 of line 12-21. The *Trans*1 function iteratively calls *Butterfly*1 function in SPMD style by (*b* - $\log_2$(*nthreads*)) times indexed *i* from 1 to (*b* - $\log_2$(*nthreads*)). When *i* is 1, and 2, it performs stage 1, and 2 respectively as shown in Fig. 8a.

*Trans*2 is called from Fig. 1 of line 25. It is invoked to perform the stage (*b* -$\log_2$(*nthreads*)+1) up to stage *b* (final stage) of butterfly operations, for which stages' computation are obtained by accessing from out of the chunk size boundary. It is shown in stage 3 and 4 in Fig. 8a. Fig.8b is the data accesses for the four stages Omega operation of FFT2 code (rewriting FFT program proposed by Bader into SPMD) shown in Fig. 11. It accesses data outside the chunk size boundary even at the first stage (or earlier stage) of the butterfly operation.

```
1Trans1(*M,*A,chunksize,addr[],twsize){
2   int Nthreads,threadid;
3#pragma omp parallel private(threadid)
4 { nthreads = omp_get_num_threads();
5   threadid  = omp_get_thread_num();
6   Butterfly1(M+addr[threadid],A,
7              chunksize, twsize);
8 }
9}
10 Butterfly1(*M,*A,chunksize,twsize){
11   struct COMPLEX temp1,temp2;
12   for( i=0; i<chunksize; i+=twsize ){
13     k=0;
14     for( j=i; j<i+twsize/2; j++ ) {
15       temp1 = M[j];
16       temp2 = M[j+twsize/2]*A[k];
17       M[j] = temp1+temp2;
18       M[j+twsize/2] = temp1-temp2;
19       k++;
20     }
21   }
22 }
23 Trans2(*M,*A,*Tmp, add_off[], chunksize, nthreads, p ){
25   int i,twsize,t;
24   twsize=chunksize*2;
27   t=2;
28   for(i=0; i<p; i++) {
29     if(i%2==0)
30       Butterfly2(M,A,Tmp,add_off,chunksize,twsize,t);
32     else
33       Butterfly2(Tmp,A,M,add_off,chunksize,twsize,t);
35     twsize = twsize*2; t = t*2;
36   }
37 }
```

Fig. 9. Trans1 function for butterfly operation of FFT1

In *Trans*2 function at line 29-33 of Fig.9, it calls *Butterfly*2 function with parameter swapping between *M* and *Tmp*, when stage number is even as shown in line 29, it computes from accessing array *M* and output the results the butterfly computation into array *Tmp*. On the other hand, when the stage number is odd as shown in line 33, it computes from accessing array *Tmp* and write the output to *M.* In this way, we remove the race condition of the original sequential code.

In *Trans*2 function, it calls *Butterfly*2 function *p* times, where *p* is $\log_2$*(nthreads)*, to perform next *p* stages. It is stage *(b* - $\log_2$*(nthreads))* + 1 up to stage *b*. *Trans*2 is complicated by removing the race condition occurs on the later stages. Using our examples, it is stage 3 in Fig. 8a, thread 0 correspond to shaded box will compute *Tmp*[0] = a_0+a_2 * *A*[4], *Tmp*[1] = a_8 + a_{10} * *A*[5], *Tmp*[2]=a_4 + a_6 * *A*[6], *Tmp*[3] =

a_{12}+a_{14}*A[7], where array A is the value of twiddle factors pre-computed during computation of nth_roots.

```
1 Butterfly2(*M,*A,*Tmp, addr[], chunksize, twsize,t){
3   int nth_addr[t/2];
4   for(i=0;i<t/2;i++) {
5       nth_addr[i] = (i % t/2)*chunksize;
6   }
7  #pragma omp parallel private(j,threadid)
8   { nthreads = omp_get_num_threads();
9     threadid = omp_get_thread_num();
10     j=threadid % t;
11     if(j<t/2) {
12       twiddle_l(M+addr[threadid],A,Tmp+addr[threadid],
14                 chunksize, twsize/2, nth_addr[j]);
16     } //end if
17     else if(t/2<=j){
18       twiddle_r(M+addr[threadid],A, Tmp+addr[threadid],
20                 chunksize, twsize/2, nth_addr[j-t/2]);
22     } //end else if
23   } //end omp prallel
24 } //end Butterfly2
25 twiddle_l(*M,*A,*Tmp,chunksize, mid,nth_addr){
27   for(int i=0; i<chunksize; i++) {
28     Tmp[i]= M[i] + A[i+mid+nth_addr] * M[i+mid];
30   }
31 }
32 twiddle_r(*M,*A,*Tmp,chunksize, mid,nth_addr){
34    for(int i=0; i<chunksize; i++){
35     Tmp[i]= M[i-mid] -A[i+mid+nth_addr] * M[i];
37    }
38 }
```

Fig. 10. Butterfly2 function of FFT1

```
1 Butterfly2(*M,*A,*Tmp, addr[], chunksize, twsize,t){
3   int nth_addr[t/2];
4   for(i=0;i<t/2;i++) {
5       nth_addr[i] = (i % t/2)*chunksize;
6   }
7  #pragma omp parallel private(j,threadid)
8   { nthreads = omp_get_num_threads();
9     threadid = omp_get_thread_num();
10     j=threadid % t;
11     if(j<t/2) {
12       twiddle_l(M+addr[threadid],A,Tmp+addr[threadid],
14                 chunksize, twsize/2, nth_addr[j]);
16     } //end if
17     else if(t/2<=j){
18       twiddle_r(M+addr[threadid],A, Tmp+addr[threadid],
20                 chunksize, twsize/2, nth_addr[j-t/2]);
22     } //end else if
23   } //end omp prallel
24 } //end Butterfly2
25 twiddle_l(*M,*A,*Tmp,chunksize, mid,nth_addr){
27   for(int i=0; i<chunksize; i++) {
28     Tmp[i]= M[i] + A[i+mid+nth_addr] * M[i+mid];
30   }
31 }
32 twiddle_r(*M,*A,*Tmp,chunksize, mid,nth_addr){
34    for(int i=0; i<chunksize; i++){
35     Tmp[i]= M[i-mid] -A[i+mid+nth_addr] * M[i];
37    }
38 }
```

Fig. 10. Butterfly2 function of FFT1

This sum is represented as up arrow in Fig.8a. Thread 1 computes $Tmp[4] = a_0 - a_2 * A[4]$, $Tmp[5] = a_8 - a_{10} * A[5]$, $Tmp[6] = a_4 - a_6 * A[6]$, $Tmp[7] = a_{12} - a_{14} * A[7]$. The difference is represented as down arrow. Thread 2 and 3 applies the same way. In the next stage, the result of the sum and difference are stored back to array M. When the next stage is even then we stored the result in Tmp, otherwise it stored in M.

Trans2 calls Butterfly2 function p times with a decision of whether a thread in a stage is performing the sum or the difference. This is shown in Fig. 10 of line 10-22. We have thread i and stage s, then $t = 2^{s-p}$, j =*threadid* % t; where p is $\log_2$(*nthreads*), this thread performs sum if $j < (t/2)$, otherwise it performs difference. The sum is done by calling twiddle_l at line 25-31 and the difference is done by calling twiddle_r at line 32-37. Array *nth_addr* at line 5 is used to locate the starting address of those twiddle factors that belongs to each stage for each thread.

IV. EXPERIMENTAL RESULTS

We perform our experiment for the two versions of FFTs on a four 2-core CPUs 2.8 GHz AMD-Opteron™ 8200/Dell 6950 machine with 8GB of main memory, 64KB L1 cache, 1MB L2 cache. We compile the parallel version of our code with icc Intel OpenMP compiler with flag –O2 –openmp and run on Linux Cent OS. We run this program with input size of 2^{26}, and 2^{27} double-precision complex input samples respectively. Each of this was run in parallel using 1, 2, 4, 8 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 more than 3 GB main memory.

Table 1. The execution of FFT1 with 2^{26} input size

Size = 2^{26}					
# of thread	Bit reversal	Nth-roots	Butterfly1	Butterfly2	Total
1	10.550748	0.335235	10.077608	0.000000	21.662063
2	5.771677	0.336037	7.386913	0.686002	14.1804629
4	4.959388	0.337098	2.562583	0.674507	8.533576
8	3.331917	0.336793	2.106578	1.096009	6.871297

Table 2. The execution of FFT1 with 2^{27} input size

Size = 2^{27}					
# of thread	Bit reversal	Nth-roots	Butterfly1	Butterfly2	Total
1	21.498096	0.669158	21.313334	0.000000	43.480558
2	12.033417	0.670846	15.407283	1.321514	29.43306
4	9.317965	0.670651	5.422765	1.342825	16.754206
8	7.040469	0.670890	4.556342	2.207363	14.835742

Table 1 and 2 shows the performance of FFT1 with input size of 2^{26} and 2^{27} respectively. We measure the bit-reversal computation, computation of nth-roots, and the Butterfly operation is divided it into two step operations: Butterfly1 and Butterfly2, and adding up these items into the total execution of FFT1. Our SPMD style of OpenMP for the Bit-reversal shows scalable as the number of processors increases, except a little degradation on using the entire cores of the machine. The pre-computation of nth-roots is done serially, which only take 0.3 seconds for input size 2^{26} and 0.6 seconds for input size 2^{27}. Butterfly1 is used to perform the sum and difference computation for which each thread accesses their data is within its chunk size boundary; this is determined by twiddle size less or equal to chunk size of a thread. Therefore, when number of thread is one, the twiddle size will be within the chunk size boundary for all stages, hence, Butterfly2 is not performed for one thread of execution. When number of threads is 4, there will be four stages of the Butterfly operation. As we observed, our algorithm obtained the best performance for input size 2^{26} when there are 4 threads, there will be 4 stages: first 24 stages (obtained from b - $\log_2$(*threads*)) are performed by Butterfly1 and the remaining 2 stages (obtained from $\log_2$(*threads*)) are done by Butterfly2.

With 8 threads, Butterfly1 is performed for 20 stages, Butterfly2 then doing 4 stages. For the input data size 2^b, then there are b stages. The first stage all read reference to data of the computation is next to each other. At each next stage, the read reference will be twice distance apart as the previous stage. Hence, when b is large, the stage i will have a distance of 2^{i-1} for their read reference to data stored in an array, this may causes large number of cache misses. So, more improvement is possible for Butterfly2 operation.

Table 3. The execution of FFT2 with 2^{26} &2^{27} input size

Input size	Total execution time of FFT			
	Number of threads			
n	1	2	4	8
2^{26}	20.634772	16.394508	12.189186	13.142301
2^{27}	41.396297	28.652554	22.548094	24.029489

Table 3 presents the result of execution of FFT2 whose code is shown in Fig.11. It is not scale well when it is 8 threads. This is due to accessing the memory element outside chunk size boundary that the distant is far apart even in the early stage of Omega operation shown in Fig8b. This causes significant overhead of cache misses compared to FFT1. We do not discuss the FFT2 algorithm in detail, because there is space limitation of this paper.

V. CONCLUSIONS AND FUTURE WORKS

We have developed two versions of non-recursive SPMD style of OpenMP FFT written in C. It can be easily port to other parallel programming languages such as UPC, Cilk, and MPI. FFT1 performs Bit-reversal and its computed result is an input to butterfly operation, hence they operate as separate phases. The second version named FFT2 performs butterfly operation where each stage the computed data is stored at the permuted location, hence the Bit-reversal is not performed as a separate phase and this is similar to elimination of Bit-reversal operation. It originally designed to perform well on Cell processor. The advantage of FFT1 is the first log2(chunk size) stages the data accesses are within the chunk size of each thread and the remaining b-log2(chunk size) stages accesses data are distant apart out size chunk size boundary of each thread. The FFT2 accesses the memory element outside chunk size boundary and distant is far apart even in the early stage of butterfly operation. FFT2 has significant overhead of cache misses compared to FFT1. Our experimental results of FFT1 showed promising despite more memory spaces are used, and has better performance compared to FFT1. Still, there may be more improvements are possible. We continue our work on porting to MPI/OpenMP and UPC.

REFERENCES

[1] Cooley, J. W., and Tukey, J. W. An algorithm for the machine calculation of complex Fourier series. In Math. Comput, 19:297–301, 1965.

[2] D. Bollman, J. Seguel, and J. Feo, Fast Digit-Index Permutations. Scientific Progress, vol. 5, no. 2, pp. 137-146, 1996.

[3] Karp, A. H. Bit Reversal on Uniprocessors. SIAM Review, Vol. 38,pp. 289-307, 1996.

[4] Liu, Z., Chapman, B., Wen, Y., Huang L., Weng TH., and Hernandez, O. Analyses for the Translation of OpenMP Codes into SPMD Style with Array Privatization. In Proceedings of International Workshop on OpenMP Applications and Tools (WOMPAT), pp. 244-259, 2003.

[5] Lokhmotov, A., Mycroft, A. Optimal bit-reversal using vector permutations. ACM Symposium on the 19th Parallel Algorithms and Architectures, pp.198–199, 2007.

[6] OpenMP Architecture Review Board. Fortran 2.0 and C/C++ 2.0 Specifications. At www.openmp.org.

[7] Rodriguez, J.Jeffrey. 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.

[8] Rubio, M., Gómez, P., Drouiche, K. A new superfast bit reversal algorithm. International Journal of Adaptive Control and Signal Processing, vol. 16, issue 10, pp.703-707, 2002.

[9] Seguel, J., Bollman, D., Feo, J. A Framework for the Design and Implementation of FFT Permutation Algorithms, IEEE Transactions on Parallel and Distributed Systems, vol.11, no.7, pp.625-635, 2000.

[10] Wallcraft, A. J. SPMD OpenMP vs. MPI for Ocean Models. Proceedings of First European Workshops on OpenMP (EWOMP'99), Lund, Sweden, 1999.

[11] Zhang, Z., Zhang, X. Fast Bit-Reversals on Uniprocessors and Shared-Memory Multi-processors, SIAM Journal on Scientific Computing, vol.22, no.6, pp.2113-2134, 2000.

[12] Takahashi, D., Sato, M., and Boku, T. An OpenMP Implementation of Parallel FFT and Its Performance on IA-64 Processors. WOMPAT 2003, LNCS 2716, pp. 99-108, 2003.

[13] Weng, T-H, Huang, S-W, Perng, R-K, Hsu, C-H, Li, K-C. A Practical OpenMP Implementation of Bit-reversal for Fast Fourier Transform. In Proceeding of the 4th ICST 2009, Hong Kong, June, 2009.