Parallel fast Fourier transform in SPMD style of Cilk

Tien-Hsiung Weng, Teng-Xian Wang and Meng-Yen Hsieh

Department of Computer Science and Information Engineering (CSIE),
Providence University,
Taichung 43301, Taiwan
Email: thweng@pu.edu.tw
Email: g1050302@pu.edu.tw
Email: mengyen@pu.edu.tw

Hai Jiang

Department of Computer Science,
Arkansas State University,
Jonesboro, Arkansas, USA
Email: hjiang@astate.edu

Jun Shen

School of Computing and Information Technology (SCIT),
University of Wollongong,
Wollongong, NSW, Australia
Email: jshen@uow.edu.au

Kuan-Ching Li*

Hubei Education Cloud Service Engineering Technology Research Center,
Hubei University of Education,
Wuhan, China
and
Department of Computer Science and Information Engineering (CSIE),
Providence University,
Taichung 43301, Taiwan
Email: kuancli@pu.edu.tw
*Corresponding author

Abstract: In this paper, we propose a parallel one-dimensional non-recursive fast Fourier transform (FFT) program based on conventional Cooley-Tukey's algorithm written in C using Cilk in single program multiple data (SPMD) style. As a highly compact designed code, this code is compared with a highly tuned parallel recursive fast Fourier transform (FFT) using Cilk, which is included in Cilk package of version 5.4.6. Both algorithms are executed on multicore servers, and experimental results show that the performance of the SPMD style of Cilk fast Fourier transform (FFT) parallel code is highly competitive and promising.

Keywords: fast Fourier transform; FFT; single program multiple data; SPMD; Cilk; parallel programming.

Reference to this paper should be made as follows: Weng, T-H., Wang, T-X., Hsieh, M-Y., Jiang, H., Shen, J. and Li, K-C. (2019) 'Parallel fast Fourier transform in SPMD style of Cilk', *Int. J. Embedded Systems*, Vol. 11, No. 6, pp.778–787.

Biographical notes: Tien-Hsiung Weng is an Associate Professor in the Department of Computer Science and Information Engineering at Providence University, Taiwan. He received his PhD in Computer Science from the University of Houston, USA. His current research interests include parallel programming model, performance measurement, compiler analysis for code improvement, graph theory and algorithm design.

Copyright © 2019 Inderscience Enterprises Ltd.

Teng-Xian Wang is a Master student in the Department of Computer Science and Information Engineering at Providence University, Taiwan.

Meng-Yen Hsieh received his PhD in Engineering Science from the National Cheng Kung University, Taiwan, in 2007. He is currently an Associate Professor of Department of Computer Science and Information Engineering, Providence University, Taiwan. He served on symposium chairs and technical program committees for several international conferences. His research interests include, wireless network applications, security applications, and web service.

Hai Jiang is a Professor in the Department of Computer Science at Arkansas State University, USA. His current research interests include parallel and distributed systems, computer and network security, high performance computing and communication, and modelling and simulation. He is a professional member of ACM and IEEE Computer Society.

Jun Shen was awarded PhD degree in 2001 at the Southeast University, China. He is an Associate Professor at the University of Wollongong in Wollongong, NSW, Australia. He has published more than 160 papers in journals and conferences in computer science and information system area. His expertise is on cloud computing and big data. He has been an Editor, PC Chair, Guest Editor, PC member for numerous journals and conferences published by IEEE, ACM, Elsevier and Springer.

Kuan-Ching Li is a Professor at the Providence University, Taiwan. He is a recipient of awards and funding support from a number of agencies and industrial companies, as also received guest and distinguished chair professorships from universities in China and other countries. He has been actively involved in many major conferences and workshops in program/general/steering conference chairman positions and as a program committee member, and has organised numerous conferences related to high-performance computing and computational science and engineering. Not only publications in technical journals and conferences, he is author/co-author and Editor/Co-editor of several technical professional books published by CRC Press, McGraw-Hill, IGI Global and Springer. His topics of interest include cloud and GPU computing and big data. He is a senior member of the IEEE and a Fellow of the IET.

This paper is a revised and expanded version of a paper entitled 'Performance of parallel bit-reversal with Cilk and UPC for fast Fourier transform' presented at Grid and Pervasive Computing (GPC) 2010, Hualien, Taiwan, 10–13 May 2010.

1 Introduction

The fast Fourier transform (FFT) (Cooley and Tukey, 1965) has been one of the most frequently used and foremost important algorithms in many fields of science and engineering applications such as in partial differential equations, signal processing for spectral analysis of speech, image processing, fluid dynamics for solving partial differential equations (PDEs), seismic and vibration detection and other related fields applications.

In this paper, we implemented parallel non-recursive version of FFT using single program multiple data (SPMD) style of Cilk. Cilk (Frigo et al., 1998; Frigo, 2007; The MIT Cilk Project, http://supertech.csail.mit.edu/cilk/) is a multi-threaded parallel programming language extension to standard C and C++, consisting of *cilk*, *spawn*, *sync*, *inlet*, and *abort* keywords, which are easier to use for parallel programmers to express task and data parallelism as well as enabling parallel programmers to parallelise the application written in recursive programs. Its runtime system supports dynamic workload balancing capability with efficient randomised work-stealing scheduler, synchronisation, and communication protocols, once also called as Cilk Art, founded by Charler E. Leiserson of Massachusetts Institute of Technology (MIT), and is moving to support industrial and commercial developer of parallel applications that can be executed on Linux, Windows, OS X operating systems and other platforms. In addition, Cilk's runtime system also guarantees efficient and predictable performance such that provides better performance scalability for parallel applications that have irregular computations, and more robust performance for applications running in multi-programmed environments. Cilkc is a source-to-source translator used to convert parallel Cilk code to standard C, and after that, it compiles the resulting C source into the executable. It was purchased by Intel in 2009 and turned one of its products named Intel® Cilk™ Plus (Intel Cilk Plus, https://www.cilkplus.org/). These extensions are now not only included in GCC 5.0 mainline and Intel's commercial compiler, but also a set of patches to add Cilk to LLVM clang. However, only 32 and 64-bit Intel hardware is supported by any of compilers. Recently, it can also be used in Raspberry Pi in multicore ARM architectures.

Cilk is practical and easy to understand, whereas the proposed implementation of the parallel code can be developed with less effort, and the newly designed code is fairly compact. The Cilk SPMD style of FFT code only takes about 0.3K line of code (LOC), which is shorter than the original parallel recursive Cilk FFT (3K LOC). The SPMD code can be easily ported to other well-known

programming models such as message passing interface (MPI), compute unified device architecture (CUDA), unified parallel C (UPC), OpenMP, Intel threading building blocks (ITBB), and OpenACC, which is one of major advantages. Ensuring data locality memory access has been another advantage of program written in SPMD style, where each private sub-array can be created and accessed locally within each thread to enable spreading the computation among threads in the manner to ensuring data locality.

Cilk implementation of SPMD Style of FFT code has three main steps:

1 bit-reversal operation

2 pre-computation of nth roots

3 the butterfly operations.

Bit-reversal operation rearranges the input array by reversing the binary bits of the array indices. That is, a bit-reversal is an operation to swap the data between $A[j]$ and $A[Bit\text{-}reversal[j]]$, where the value of j is from 0 to the input size m and m is usually two to the power of b. The $Bit\text{-}reversal[j]$ is obtained from reversing b bits from value of j. Bit-reversal must be designed properly since it takes about 30 percent of the total execution time in FFT (Karp, 1996).

Finding the discrete Fourier transform (DFT) of the individual value is the first stage, and it passes the values along. At each one of remaining stages, the computation for a polynomial of degree n at the n complex n^{th} roots of unity is used to compute a new value depends on the values of the previous stage, which process is called butterfly operation. The proposed approach relies on performing parallel Cilk code in SPMD style.

The rest of the paper is organised as follows. The related works is discussed in Section 2, while the design of the parallel algorithm in Section 3. Experimental results and evaluation of this parallel version of FFT on multicore servers in Section 4, and finally, we give our conclusions and future directions in Section 5.

2 Related work

In ordinary Cooley-Tukey's algorithm, the bit-reversal and butterfly operations were performed separately. Bader and Agarwal (2007) proposed non-recursive FFT based on Cooley-Tukey's, the novelty of their modified one 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 cell processor.

Data reordering in FFT program using bit-reverse of array index has been widely studied (Bollman et al., 1996; Karp, 1996; Lokhmotov and Mycroft, 2007; Rodriguez, 1988; Rubio et al., 2002; Seguel et al., 2000; Zhang and Zhang, 2000). Most of algorithms proposed were designed to be executed on single processor platforms (Karp, 1996; Lokhmotov and Mycroft, 2007; Rodriguez, 1988; Rubio et al., 2002). Lokhmotov proposed the optimal Bit-reversal using vector permutations with experiments performed on single processor (Karp, 1996). They claimed that their sequential algorithm can be parallelised without any problem or extra effort. An algebraic framework for FFT permutation algorithm using functional language SISAL was implemented, and performance measurements were done on Cray C90 and SUN Sparc5 machines (Bollman et al., 1996; Seguel et al., 2000). Namneh (Karp, 1996) implemented and compared two versions of 1D FFT algorithms, implementation based on tree and transposition of one-dimensional FFT parallelised using MPI to run on symmetric multi-processing (SMP) servers. Experimental results are done on SMP SUN SPARC servers with total of eight processors and complex numbers data up to 4094 KB. The SMP SUN SPARC servers have a crossbar processor interconnection. Franchetti used spiral, an automatic program generation, and integrated it into their proposed framework to boost optimisations (Bollman et al., 1996). The users can formally specify a digital signal processing transforms such as DFT, then spiral will rewrite and transform the DFT in the form of mathematical formula into C program that computes the specified transform, optimised to achieve load balancing and avoiding false sharing to a given platform. Furthermore, they propose to extend spiral to generate parallel multithreaded code such as OpenMP and Pthread codes.

An OpenMP implementation of a one-dimensional recursive algorithm written in Fortran 90 for parallel recursive FFT has been proposed by Takahashi et al. (2003) on shared memory parallel computers. Experiments are performed on 4-CPU DELL PowerEdge 7150 for 224 points FFT with double-precision complex data. Ouni and Mtibaa (2014) proposed a data flow graph approach with temporal partitioning algorithm for reconfiguration system, where FFT is one of the applications observed in their experiments. FFTW is a publicly available free-software written in C, developed at MIT by Frigo and Johnson (2005). It is concerned with machine architectures and adaptively tune by fully taking advantage of machine architectures to maximise the performance of FFT. It is highly tuned, hard-coded FFT that run on many machine architectures without any restrictions on input size as well as dimensionality of input array. Takahashi (2013) proposed an implementation of hybrid model 1D FFT on GPU clusters, where the algorithm is based on six-step FFT algorithm that consists of three transpositions and three all-to-all communications. Utilising the combination of MPI and cuFFT, the experimental results using large input size of 2^{34} achieved a significant speedup of about 3x compared to FFTW.

3 Implementation

The proposed implementation of Cilk FFT can be divided into three parts: the computation of Bit-reversal, the pre-computation of nth roots, and butterfly operations, as shown in Figure 1. The butterfly operations are divided into

two steps: Butterfly1 and Butterfly2. It is a non-recursive 1D FFT using Cilk in SPMD style, and named as FFTC1.

Figure 1 Main function of the FFTC1

```
FFT() {
  STEP 1 : Bit-Reversal Computation
  STEP 2 : Compute Nth-Roots
  STEP 3 : Butterfly Operation;
        -  Trans1
        -  Trans2
}
```

3.1 Bit-reversal computation

The parallel implementation of the computational bit-reversal in SPMD style of Cilk is derived from the existing sequential code proposed by Rodriguez (1988). The sequential code is shown in Figure 2, which is an improved and optimised sequential code design for FFT. It takes the parameters N and p, where N is the size of an input array, then the bit-reversal permutation is performed with the total number of bits p. In their bit-reversal computation, array permutation of index for data reordering computes only the required bit-reversal of indices, which eliminates the number of unnecessary bit-reversal and swaps. The bit-reversal is calculated as bit-reverse = $\sum_{k=0}^{p-1} b_{p-1-k} 2^k$; where b is the binary value, p is the total number of bits, and k is k^{th} position of the binary from the most significant digits. It uses solely an array A to store its input data and final results. Therefore, the data reordering must perform the exchange between elements of A.

Figure 2 An improved bit-reversal code by Rodriguez

```
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   } // end of for
9 } // end of Bit-reverse
```

Despite the swapping of an array is an exchange between two elements, it actually comprises of three assignment statements or copy actions. In the memory read and write accesses, the *swap*(A[i], A[*bitreverse*(i)]) performs the copy A[i] to Temp, then A[*bitreverse*(i)] to A[i], and Temp is copied to A[i]. The merit of their code is that there is no actual conversion from the binary representation to the decimal and vice versa. In line 2, the index upper bound for the variable is computed by last = $(N - 1 - N2)$, where $N2$ is $\sqrt{N}$ when the number of bits is even and $\sqrt{2N}$ when the number of bits is odd. As result, it eliminates the unnecessary computation of bit-reversal, which reduces number of swaps. Therefore, the total numbers of copies take about 3 * $(N - N2)$ / 2 moves, which is equal to 1.5 * $(N - N2)$.

Figure 3 Data dependence graph of Figure 2

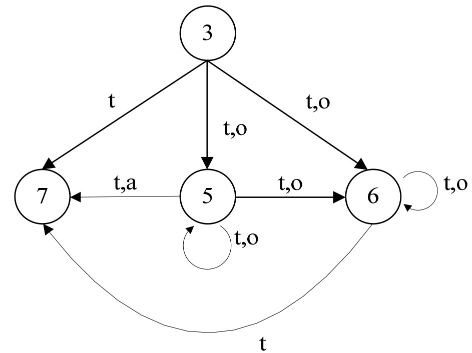


Figure 4 FFT Cilk model

```
1 cilk FFT(*A,*M,*address,*offset,n,nthreads) {
2   int chunksize;
3   chunksize = n/nthreads;
4   for(i=0;i<nthreads;i++)
5     address[i]=chunksize*i;
6   for(i=0;i<nthreads;i++) {
7     if(i==0) j==0;
8     else { k=nthreads/2;
9         while(k<=j){
10            j=j-k; k=k/2;
11        }//end while loop
12        j=j+k;
13    }//end else
14    offset[i]=j;
15  } //end for
16  for(i=0;i<nthreads;i++)
17    spawn Bit_reverse(A,M+address[i],
18              n,chunksize,offset[i]);
19  sync;
  .........
20  ...//other computation such as butterfly operation
21 }

22 cilk Bit_reverse(*A,*M,n,chunksize,offset) {
23   for(i=0;i<chunksize;i++) {
24     if(i==0) j=0;
25     else { k=n/2;
26         while(k<=j) {
27           j=j-k; k=k/2;
28         }//end while
29         j=j+k;
30     }//end else
31     M[i]=A[j+offset];
32   } // end for
33 }
```

In order to improve our parallel code, we utilised the performance-based parallel analysis toolkit proposed by Li et al. (2009), which is an effective toolkit for performance measurement and analysis for parallel application. It not only provides the measurement of the execution time, but also generates application data analysis graphs. This toolkit allows application developers to have a better understanding of the application's behaviour among selected computing nodes purposed for that particular execution. Furthermore,

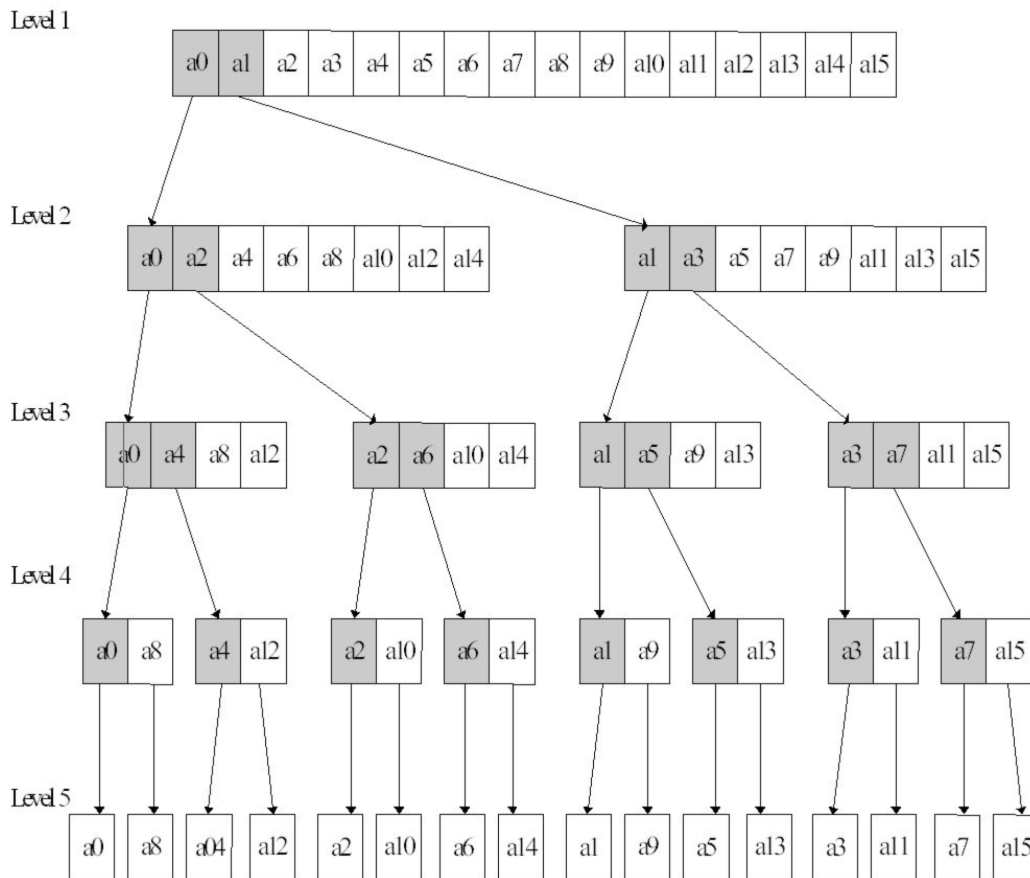
the results of multiple execution of a given application under development can be combined and overlapped, allowing the application developers to perform 'what-if' analysis, to deeper understand the utilisation of allocated computational resources. The effectiveness on the development and performance tuning of parallel applications is supported to execute on the shared memory model.

Even though bit-reversal sequential code seemed to be the best, it is implemented as a loop that has loop-carried data dependences between iterations. Hence, it is not parallelisable without complete modification to the original sequential code, as shown in data dependence graph depicted in Figure 3. It has true and output data dependences between loop iterations of statements $j = j - k$, $j = j + k$, and *if* $(i < j)$. There are also true, anti- and output dependences between the statements labelled on each edge as *t*, *a* and *o* respectively. For instance, there are true dependence between statements 3 and 7; true and output dependences between statements 5 and 6, 3 and 5, as well 3 and 6; the loop on node 6 means there is true and output dependence between statement 6 itself on different iteration of the for loop. As result, the value of variable *j* is accumulated for the entire nested for loop, what means that the computation for value of *j* is dependent on the previous value of *j*. Hence, in order to parallelise this code, we need to modify its structure.

In SPMD style of Cilk, reducing the number of cache misses and data locality are the main concerns in the design of the proposed code. The SPMD style of Cilk code is distinct from ordinary Cilk code. In most SPMD programs, shared arrays are declared and parallel for directives are used to distribute work among threads via explicit loop scheduling. We create private instances of sub-arrays to spread computation among threads to ensure the data locality, where this study has been discussed (Liu et al., 2003). Programs written in SPMD style has also been shown to provide scalable performance, which is superior to a straightforward parallelisation of loop (Weng and Chapman, 2004).

The main parallel Cilk SPMD FFT code is shown in Figure 4. When it is invoked, it starts a pre-computation of bit-reversal offset sequentially as depicted in line 6–15, after that the *Bit_reverse*() functions with keyword cilk as in line 22–33 is spawn for parallel execution as shown in line 17, then this pre-computation of offset will be used for bit-reversal computation. This pre-calculation of offset is inspired by idea of divide and conquer.

Figure 5 Bit-reversal by divide and conquer



In order to explain how the *Bit_reverse*() algorithm works, we use an input size of 16 using 4 threads as example, then the number of bits is equal to $\log_2 16$, which is equal to 4 bits. Here, chunk size is equal to the input size divided by the number of threads. It is shown in Figure 5 the top level 1 that represents the original input value of an array *A*. At each next level, it is further divided into two chunk size of same size recursively, where chunk one is obtained from the even indices, whereas the other is obtained from odd indices. At the end of the permutation as in last level (or level 5), *M*[0] = *A*[0], *M*[1] = *A*[8], *M*[2] = *A*[4], *M*[3] = *A*[12], …, *M*[14] = *A*[7], *M*[15] = *A*[15]. At each level, the first element of each chunk that is shaded in gray is derived from its parent's first and second elements. The first element of each chunk, namely offset, it will be stored in array offset. At level 2, the value of *offset*[0] is 0 and *offset*[1] is 1. At level 3, *offset*[0: 3] is equal to {0, 2, 1, 3} and *offset*[0: 7] is {0, 4, 2, 6, 1, 5, 3, 7}. Even though the precomputation of offset shown here came from divide and conquer, we implemented it iteratively instead of recursively. Since we have four threads, the number of elements for array offset is 4.

Figure 6 The pre-computation of the nth roots of unity

```
1  cilk Compute_nthroots(*A,b) {
2     j=1; nth=1;
3     for(s=1;s<=b;s++) {
4        w=e**(-2πi/nth);
5        spawn nthroot(A+j,nth,w);
6        nth=nth*2; j=j*2;
7     }
8    sync;
9   }
10 cilk nthroot(*Nth, size, w) {
11    Nth[0]=1;
12    for(ℓ=1; ℓ<size/2; ℓ++)
13       Nth[ℓ] = w** ℓ;
14 }
```

3.2 Butterfly operation

Since the bit-reversal computation takes array *A* as input and the result of permutations are stored in array *M*, as soon as this computation is completed, array *A* can then be reused to store the values of pre-computation of nth *roots*.

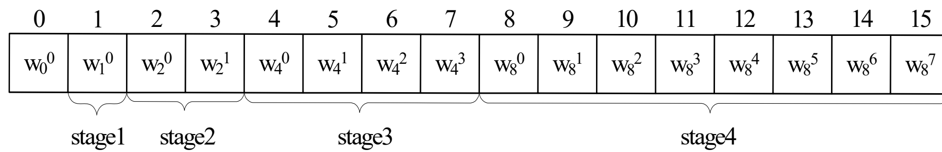
As shown in Figure 7, the pre-computation of nth *roots* is executed serially by each process when the amount of work is small, taking only very small amount of execution time. It pre-computes twiddle factor $\omega = e^{(2\pi i/2^s)}$ for each stage of the butterfly operations and the results are stored into array *Nth* as shown in Figure 6. In the *Compute_nthroots*() subprogram, there are b stages calls to *nthroot*(), where $b = \log_2 N$ and *N* is input data size. For each stage s, it computes $Nth[k] = e^{(2\pi i/2^s)\ell}$, where *k* is an array index from $2^{s-\ell}$ to $2^s - 1$, $\ell = k - 2^{s-1}$, $s = \lceil \log_2 k \rceil$, 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 performing the same calculation, reducing the number of computations. After the pre-computation of nth root is performed, each thread can access its own copy of the result as shown in Figure 7.

Next, we present butterfly operation using SPMD style of Cilk, as depicted in Figure 8. There are three inputs from the left: *a*[*i*], *a*[*i* + *twsize* / 2], and on the middle left is the twiddle factor ω_n^k. 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*].

The main butterfly operation starts by calling Trans1 and Trans2 functions of step 3, as shown in Figure 1. Trans1 performs the first stage up to *b*-$\log_2$(*nthreads*) stage, in 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 is done by accessing data within chunk size boundary, and controlled by condition (*twsize* <= *chunksize*) at line 2 of Figure 10. On the other hand, Trans2 is used to perform the sum and different operation where accessing data is outside the chunk size boundary.

By calling FFTC1(A, 16, 4), where the size of input *m* = 16, the number of bits $b = \log_2(16)$ is 4, and *nthreads* is 4. Trans1 in Figure 6 performs SPMD style Cilk where Butterfly1 function is spawned for each thread to work on different chunks of data, in which result of both sum and difference are stored within the chunk size boundary, this is shown in shaded box of stages 1 and 2 in Figure 9.

Figure 7 The result of pre-computation of nth root



Each shaded box represents each thread's chunk boundary. T_i represents Thread *i*, and line 2 of Figure 10 is the loop to iterate from stage 1 up to stage $\log_2$(chunk size). The first calls to Trans1 with the parameter *twiddle size* equal to 2 in stage 1. The parameter of each next consecutive call is the double *twiddle size*, and 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. During the pre-computation of nth roots, 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.

Figure 8 Simplified drawing of a butterfly operation

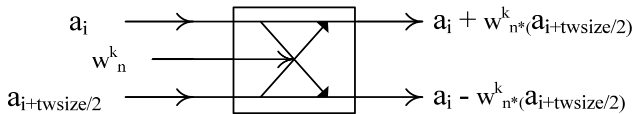
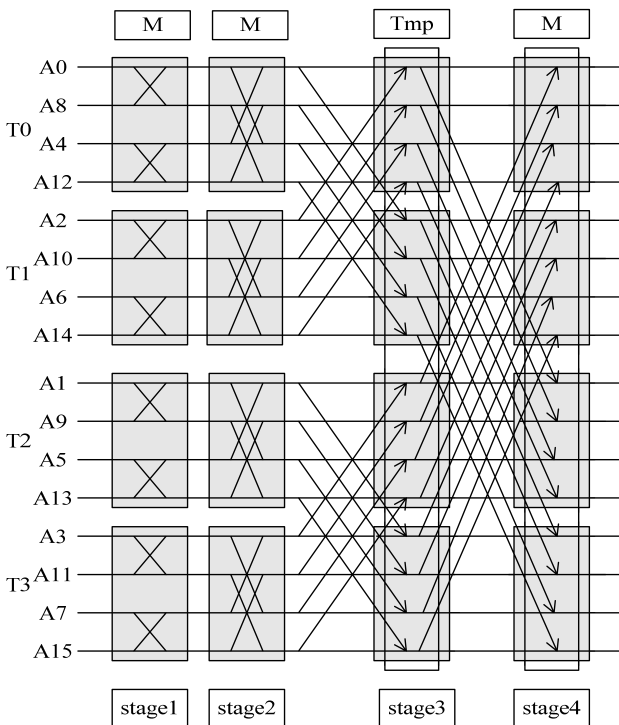


Figure 9 Butterfly operation of FFTC1 with m = 16 and 4



In Butterfly1, each thread performs sum and different *twiddle size* of sub-arrays, and each thread executes line 8 of Figure 10 (*chunksize* / *twsize*) times. Here, *M*[*j*] is stored in *temp1*. The product of twiddle factor and *M*[*j* + *twiszie* / 2] is stored in *temp2* temporarily in order to reduce from re-accessing the array *M* and re-calculating the product. Then, *M*[*j*] = *temp1* + *temp2* for the sum operation and *M*[*j* + *twsize* / 2] = *temp1* – *temp2* for the difference.

The Trans1 function iteratively spawns Butterfly1 function in SPMD style by (*b*-$\log_2$(*nthreads*)) times, where the loop index *i* is iterated from 1 to (*b*-$\log_2$(*nthreads*)) such that it can be executed asynchronously. When *i* is equal to 1, it performs butterfly operation of stage 1, and when *i* is equal to 2, then it performs butterfly operation of stage 2, and so on as shown in Figure 9. A sync statement in Line 5 is used to ensure that the execution of the current function cannot proceed until all previously spawned function calls have completed, which is similar to OpenMP barrier.

Figure 10 Trans1 function for butterfly operation of FFT1

```
1. cilk Trans1(){
2.  for( twsize=2; twsize<=chunksize; twsize*=2 ) {
3.   for( i=0; i<nthreads; i++ ) {
4.     spawn ButterFly1( M+address[i],.... );
5.     sync;  }}
6. }//end Trans1

7. cilk ButterFly1(*M,....) {
8.   for ( i=0; i<chunksize; i+=twsize ) {
9.     for( j=i; j<i+twsize; j++ ) {
10.       temp1 = M_j ; temp2 = M_{i+twsize/2} *(W_nth)^i;
11.     { M_j             = temp1 + temp2
        { M_{j+twsize/2}  = temp1 − temp2
12.   } }
13. }  //end Transl
```

Figure 11 Butterfly2 function of FFT1

```
1. cilk Trans2() {
2.   twsize=chunksize*2;  t=2;  tp=log2 nthreads
3.   for( i=0; i<tp; i++ ){
4.     if (i%2==0)  spawn Butterfly2(*M,....,i);
5.     else    spawn Butterfly2(*Tmp,....,i);
6.     sync;
7.     twsize=twsize*2;  t=t*2;
8 .  }//loop i
9. }//end Trans2

10. cilk ButterFly2( *M,....) {
11.   int nth_add[t/2];
12.   for(i=0;i<t/2;i++){
13.     nth_add[i]=i*chunksize; }
14.     mid=twsize/2;
15.     spawn twiddle(*M,*Tmp,mid,nth_add,t);
16.     sync;
17.   }//for loop i
18. } //end ButterFly2
19
20. cilk twiddle(M,Tmp,mid,nth_add,t){
21.   for (i=0;i<chunksize;i++){
22. { Tmp_i      = M_i + M_{i+mid} * (W_nth)^{i+nth_add}
    {              if (threadid%t) < t/2
    { Tmp_{i+mid} = M_{i−mid} − M_i * (W_nth)^{i+nth_add}
    {              if t/2 ≤ (threadid%t)
23. } //end twiddle
```

Trans2 is invoked to perform the stage (*b*-$\log_2$(*nthreads*) + 1) up to stage b (final stage) of butterfly operation, for which stage's computation are obtained by accessing form out of the chunk size boundary, as shown in stages 3 and 4 in Figure 9. Therefore, Trans2 function spawns Butterfly2 $\log_2$(*nthreads*) times for asynchronous execution. It is shown in lines 4 to 8 of Figure 11 that, if i%2 = 0, the execution result of Butterfly2 is writing to *Tmp* array form array *M*. If *i*%2 = 1, the execution result of Butterfly2 is writing to array *M* form array *Tmp*.

In Trans2 function at lines 4 and 5 of Figure 11, it spawns Butterfly2 function with parameter swapping between *M* and *Tmp*, when stage number is even (as shown in line 4). 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 5, 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.

There are several key points in Butterfly2 computation. First, each thread's chunk size remains the same in each stage; second, the calculation is twice as much as the previous stage; third, the read access for the calculation is twice distant from the previous stage; fourth, each thread may either perform only sum or difference in a stage, and fifth, the results in each stage of butterfly2 computation is alternately stored in array *Tmp* and *M*. Otherwise, there will be race condition.

The merits of the proposed algorithm in SPMD style is the ability to control, so that each thread computes the exact location of read accesses and the result is stored within the chunk size boundary. At the butterfly2 operation, the read accesses are from array *A*, where the pre-computation of nth root is stored. The read accesses index of pre-computation of nth root for each thread of a stage is computed in lines 11 to 13 of Figure 11. The indices are stored in array *nth_add*. Even though the sum and different are performed by separate thread, they may share the same element array of nth root.

In each stage of butterfly2, the *twiddle size* is divided by chunk size into t, which is used to determine whether a thread performing the sum or a different, and to control a thread's accesses to the correct array elements of nth root. Since the sum and difference operation may use the same elements of nth root in butterfly operation, each thread read accesses to the array n^{th} root starting from *mid* + *nth_add*[*thread id* %(*t* / 2)] up to a chunk size as shown in line 21–23 of Figure 11, where mid is half of twiddle size shown in line 14 of Figure 11.

The butterfly2 spawn twiddle function to compute sum or difference operation in line 15 of Figure 11. In twiddle function, if *thread id* % *t* is less or equal to *t* / 2, the thread does sum operation, otherwise a thread does difference operation. The parameter *t* for each next consecutive call is twice the size.

Using examples, the butterfly2 operation start its computation at stage 3 in Figure 9, thread 0 (T_0) correspond to shaded box, it computes the sum operation, they are $Tmp[0] = a_0 + (a_2 * A[4])$, $Tmp[1] = a_8 + (a_{10} * A[5])$, $Tmp[2] = a_4$+ (a6 * $A[6]$), $Tmp[3] = a_{12} + (a_{14} * A[7])$, where array *A* is the value are of twiddle factors pre-computed during computation of *nth_roots*. This sum is represented as up arrow in Figure 8. Thread 1 computes the difference, they are $Tmp[4] = a_0 - (a_2 * A[4])$, $Tmp[5] = a_8 - (a_{10} * A[5])$, $Tmp[6] = a_4 - (a_6 * A[6])$, $Tmp[7] = a_{12} - (a_{14} * A[7])$. 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*.

4 Experimental results

Experiments are performed with two versions of FFT running on a four dual-core CPU 2.8 GHz AMD-OpteronTM 8200/Dell 6950 server with 8G memory, 64KB L1 cache, 1MB L2 cache. The parallel application codes are compiled with cilkc, a Cilk compiler with gcc version 3.4.65 and running Linux Cent OS 4.7. We name the proposed version of a non-recursive 1-D FFT using Cilk in SPMD style, FFTC-1, whereas the recursive Cilk FFT implemented by Frigo is named as FFTC-2. Two application programs with input size of 2^{26} and 2^{27} are executed in this server, each executed in parallel using 1, 2, 4, 8 threads. With input data size of 2^{27}, the proposed code allocates approximately more than 3G main memory.

Tables 1 and 2 show the performance of FFTC-1 with input size of 2^{26} and 2^{27} respectively. We measure the execution time of the bit-reversal computation, computation of nth roots, and the butterfly (Butterfly1 and Butterfly2), adding up all these items into the total execution of FFTC-1. The proposed code in SPMD style of Cilk for the Bit-reversal shows to be scalable as the number of processors increases, except a little degradation on using all cores of the server. 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}. The performance of bit-reversal operation scale well up to 4 threads with speedup of 2.28x for input 2^{26} and up to 2.25x for input size 2^{27}, but slightly degraded using 8 threads, it has speedup 2.96x for input size of 2^{26} and 3.04x for input size 2^{27}. The performance of butterfly operation which consisted of Butterfly1 and Butterfly2 have a very similar performance, from which we obtained 2.81x using 8 threads for input size 2^{26} and 2.83x for input size 2^{27}. The performance of the full program only achieves up to 2.82x and 2.86x for input size of 2^{26} and 2^{27} respectively. Despite this, the execution time for the proposed sequential code is one half shorter than that of FFTC-2 as shown in Figure 12 and, even for each thread, the execution time of FFTC-1 is one half shorter than that of FFTC-2.

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 the 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 the 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 two stages (obtained from b-$\log_2(threads)$) are performed by butterfly1 and the remaining two stages (obtained from $\log_2(threads)$) are done by butterfly2.

Table 1 The execution time of FFTC-1 with 2^{26}

Size = 2^{26}				
# of thread	*Bit-reversal*	*Nth-roots*	*Butterfly*	*Total*
1	10.906423	0.413586	11.312422	22.632432
2	6.526568	0.296749	8.053472	14.87679
4	4.763673	0.317307	4.258173	9.339153
8	3.684177	0.319891	4.014965	8.019033

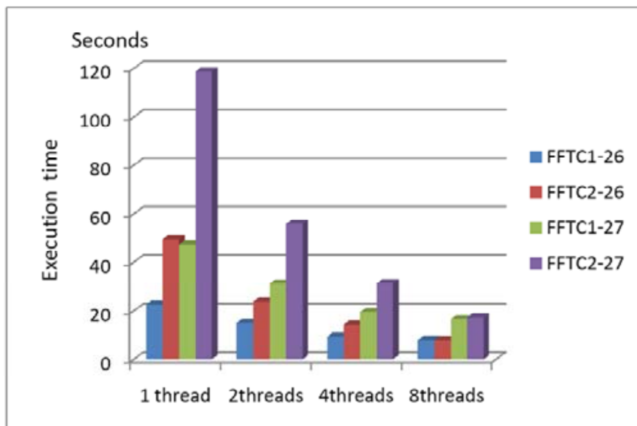
Table 2 The execution time of FFTC-1 with 2^{27}

Size = 2^{27}				
# of thread	*Bit-reversal*	*Nth-roots*	*Butterfly*	*Total*
1	22.46298	0.822327	24.126726	47.412030
2	13.69277	0.604429	16.932400	31.229597
4	9.941122	0.609798	8.917085	19.468004
8	7.371065	0.629462	8.524797	16.525323

Table 3 The execution time of FFTC-2

Input size	*Total execution*			
	Number of threads			
n	1	2	4	8
2^{26}	49.385168	23.761773	14.196570	7.947805
2^{27}	118.686967	55.894257	31.365389	17.305458

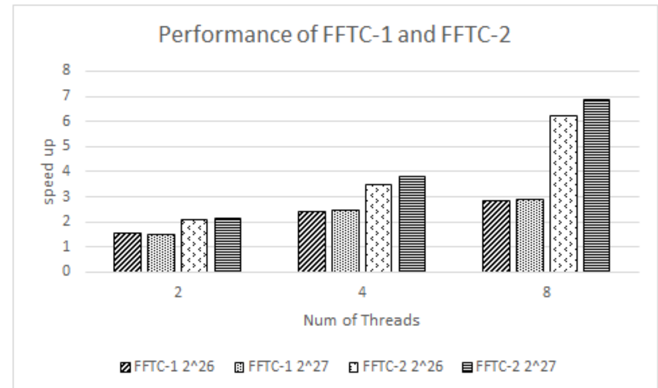
Figure 12 The execution of time FFT codes (see online version for colours)



With eight threads, Butterfly1 performs 20 stages, then Butterfly2 process 4 stages. For the input data size 2^b, 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 distant 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, causing huge number of cache misses. Due to this reason, more improvement is possible for Butterfly2 operation.

Table 3 presents the result of execution time of FFTC-2, a parallel recursive Cilk FFT implemented by Frigo that is included in Cilk 5.4.6. It is well written, highly hand-tune recursive parallel Cilk FFT code. Even though the execution time of FFTC2 on each thread is twice as much compared to those of FFTC1, the performance of FFTC-2 achieves nearly perfect linear speed up as shown in Figure 13, obtaining nearly 7x using eight threads.

Figure 13 The speedup of FFTC1 and FFTC2



5 Conclusions and future work

We have developed two versions of non-recursive SPMD style of Cilk FFT written in C. It can be easily ported to other parallel programming languages such as UPC, OpenMP, CUDA and MPI. FFTC1 performs Bit-reversal and its computed result is an input to butterfly operation, hence they operate as separate phases.

The advantage of FFTC1 is the first $\log_2(chunksize)$ stages the data accesses are within the chunksize of each thread and the remaining b-$\log_2(chunksize)$ stages access data are outside size chunk size boundary of each thread. The FFTC1 accesses the memory element outside chunk size boundary and far apart even in the early stage of butterfly operation. FFTC1 has significant overhead of cache misses compared to FFTC2. Experimental results of FFTC1 showed promise although more memory spaces are used, and has better performance compared to FFTC2. Nevertheless, more improvements are possible. The future work includes implementing FFT using CUDA for GPU on Hadoop clusters to enable massive data computation (Jiang et al., 2015; Chen et al., 2016) taking into consideration the scheduling via orchestrating the distributed servers, providing fault tolerance and redundancy.

Acknowledgements

We are grateful to National Center for High Performance Computing (NCHC), Taiwan, where the computing resources were approved for experimentations.

References

Bader, D.A. and Agarwal, V. (2007) 'FFTC: fastest Fourier transform for the IBM cell broadband engine', *The 14th International Conference on High Performance Computing (HiPC 2007), LNCS 4873*, pp.172–184.

Bollman, D., Seguel, J., and Feo, J. (1996) 'Fast digit-index permutations', *Scientific Progress*, Vol. 5, No. 2, pp.137–146.

Chen, W., Xu, S., Jiang, H., Weng, T.H., Marino, M.D., Chen, Y.S. and Li, K.C. (2016) 'GPU computations on Hadoop clusters for massive data processing', *Lecture Notes in Electrical Engineering*, Vol. 345, pp.515–521.

Cooley, J.W. and Tukey, J.W. (1965) 'An algorithm for the machine calculation of complex Fourier series', in *Math. Comput*., Vol. 19, No. 90, pp.297–301.

Frigo, M. (2007) 'Multithreaded programming in Cilk', *Proceedings of the 2007 International Workshop on Parallel Symbolic Computation*.

Frigo, M. and Johnson, S.G. (2005) 'The design and implementation of fftw3', *Proceeding of the IEEE*, Vol. 93, No. 2, pp.216–231.

Frigo, M., Leiserson, C.E. and Randall, K.H. (1998) 'The implementation of the Cilk-5 multithreaded language', in *ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, pp. 212–223.

Jiang, H., Chen, Y., Qiao, Z., Weng, T.H. and Li, K.C. (2015) 'Scaling up MapReduce-based big data processing on multi-GPU systems', *Cluster Computing*, Vol. 18, No. 1, pp.369–383,.

Karp, A.H. (1996) 'Bit reversal on uniprocessors', *SIAM Review*, Vol. 38, No. 1, pp.289–307.

Li, K.C. and Weng, T.H. (2009) 'Performance-based parallel application toolkit for high-performance clusters', *The Journal of Supercomputing*, Vol. 48, No. 1, pp.43–65.

Liu, Z., Chapman, B., Wen, Y., Huang, L., Weng, T.H., Hernandez, O. (2003) 'Analyses for the translation of OpenMP codes into SPMD style with array', Voss, M.J. (Ed.): *WOMPAT 2003*, *LNCS*, Vol. 2716, pp.26–41.

Lokhmotov, A. and Mycroft, A. (2007) 'Optimal bit-reversal using vector permutations', *ACM Symposium on the 19th Parallel Algorithms and Architectures*, pp.198–199.

Ouni, B. and Mtibaa, A. (2014) 'Emporal partitioning of data flow graphs for reconfigurable architectures', *International Journal of Computational Science and Engineering (IJCSE)*, Vol. 9, Nos. 1–2, pp.21–33.

Rodriguez, J.J. (1988) 'Improved bit-reversal algorithm for the fast Fourier transform', in *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing – Proceedings*, IEEE, pp.1407–1410.

Rubio, M., Gómez, P. and Drouiche, K. (2002) 'A new superfast bit reversal algorithm', *International Journal of Adaptive Control and Signal Processing*, Vol. 16, No. 10, pp.703–707.

Seguel, J., Bollman, D. and Feo, J. (2000) '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.

Takahashi, D. (2013) 'Implementation of parallel 1-D FFT on GPU clusters', *2013 IEEE 16th International Conference on Computational Science and Engineering*, pp.174–180.

Takahashi, D., Sato, M., and Boku, T. (2003) 'An OpenMP implementation of parallel FFT and its performance on IA-64 processors', *WOMPAT 2003, LNCS 2716*, pp.99–108.

Weng, T.H. and Chapman, B.M. (2004) 'Towards optimisation of openMP codes for synchronisation and data reuse', *IJHPCN*, Vol. 1, Nos. 1/2/3, pp.43–54.

Zhang, Z. and Zhang, X. (2000) 'Fast bit-reversals on uniprocessors and shared-memory multi-processors', *SIAM Journal on Scientific Computing*, Vol. 22, No. 6, pp.2113–2134.