Journal of Applied Sciences

ISSN 1812-5654

science alert

ANSInet
an open access publisher
http://ansinet.com

Journal of Applied Sciences 12 (20): 2115-2127, 2012
ISSN 1812-5654 / DOI: 10.3923/jas.2012.2115.2127

Software Refactoring: Solving the Time-Dependent Schrodinger Equation via Fast Fourier Transforms and Parallel Programming

[1]Ali Khwaldeh, [2]Amani Tahat and [3]Jordi Marti
[1]Department of Computer Engineering, Faculty of Engineering, Philadelphia University, 19392 Amman, Jordan
[2,3]Department of Physics and Nuclear Engineering, Technical University of Catalonia-Barcelona Tech, North Campus, UPC, 08034 Barcelona, Catalonia, Spain

Abstract: In this study a multiprocessor C++ message passing interface implementation of a new bit-reversal algorithm to numerically solve the time dependent Schrodinger equation using a spectral method based on Fourier transform was presented. The major issues of parallel computer performance were discussed in terms of efficiency; speed up, cost and fraction of the execution time that could be parallelized. The scalable performance to a very high number of processors was addressed as well as compared with ideal values of Amdahl's Law when presenting the parallel performance of the new developed algorithms. The results showed that message passing interface was an optimal method of implementing a parallelized bit-reversal algorithm.

Key words: Software refactoring, fast Fourier transform, parallel computing, schrodinger equation, spectral method, Amdahl's law

INTRODUCTION

Refactoring of programming codes is a very useful technique to improve the quality of existing computational codes (Fowler, 1999). A series of small steps has to be applied for changing the internal structure of the required code during refactoring process. On the other hand, the system functionality and the main code external behavior would not be affected (Griswold, 1991). Refactoring improves the structure of a code, making it easier to maintain and to extend. So, the activity of refactoring has two general categories of benefits: maintainability and extensibility of a code. The first category can be achieved by reducing large well-named, single-purpose methods as well as monolithic routines into a set of individually concise ones, through moving a method to a more appropriate class or by removing misleading comments. Then processes of bug fixing would be effortless and the modified code would be more readable, easier to understand and grasp (Robert, 2008). Extensibility means that capabilities of the application could be easier to be extended by using recognizable design patterns which provide flexibility where never before may have existed (Kerievsky, 2004). Some types of refactoring techniques are illustrated in Table 1. Some of these might be only applied to certain languages or language types. A longer list can be found in Fowler's Refactoring book (Fowler, 1999). This study presented the work of the authors on refactoring codes by using the programming language C++ via a message passing interface implementation of a new bit-reversal algorithm on multiprocessors. The new code outperforms an existing one included in the Fast Fourier Transform (FFT) program (Cooley and Tukey, 1965). Further, Sarkar *et al.* (1986), Peng *et al.* (2009), Jing *et al.* (2010), He *et al.* (2011) and Arioua *et al.* (2012), presented some applications of FFT algorithms in solving real-life problems. For an efficient implementation of the Discrete Fourier Transform (DFT) to an array of 2^N samples, FFT could be considered as an optimized computational algorithm. DFT of N data samples can be mathematically defined as follows:

$$X_K = \sum_{i=0}^{N-1} x_i \cdot W_N^{iK} \quad (1)$$

Hence:

$$W_N = e^{-j(2\Pi/N)} \quad (2)$$

Herein, W_N is the Nth primitive root of unity, x is the original sequence and X is the FFT of x, k = 0, 1, 2,..., N-1.

Corresponding Author: Amani Tahat, Department of Physics and Nuclear Engineering, Technical University of Catalonia-Barcelona Tech, North Campus UPC, 08034 Barcelona, Catalonia, Spain

Table 1: Types of refactoring techniques adopted from Fowler (1999)

Techniques type for:		
Breaking code apart into more logical pieces	Improving names and location of code	Allowing for more abstraction
Componentization breaks code down into reusable semantic	Moving method or moving field-move to a more appropriate class or source file	Encapsulate field-forcing the source code to access the field with getter and setter methods
Extracting class moves part of the code from an existing class into a new class	Renaming method or renaming field-changing the name into a new one that better reveals its purpose	Replacing conditional with polymorphism
Extracting class moves part of the code from an existing class into a new class	Pulling up-in OOP, move to a super class	Generalizing type-creating more general types to enable more source code sharing
Extracting method, in order to turn part of a larger method into a new method. By breaking down code in smaller pieces, it is more easily understandable. This is also applicable to functions	Pushing down-in OOP, move to a subclass	Replacing type-checking code with state/strategy

W_n^{iK} is defined as a NxN matrix. Generally, X, x and W are complex numbers. This can be commutated by using matrix-vector multiplication. This method required O (N^2) complex multiplications. But using FFT can enhance fast DFT calculations. It works by dividing the input points into two sets (k sets in general), calculating the FFT of the smaller sets recursively and combining them together to get the Fourier transform of the original set. That will reduce the number of multiplications from O (N^2) to O (N log N). Furthermore, the total time must be proportional to N^2, i.e., it is an O (N^2) algorithm. However, the algorithm could be optimized down to O (N log N) based on rearranging these operations, what makes a huge difference for large N. As far as FFT must be calculated into two steps, the first step would transform the original data array into a bit reverse order and then its implementation required the so-called bit-reversal reordering of the data for making the mathematical calculations of the second part much easier. But if bit-reversal is not done, performing the FFT could take a substantial fraction of the total computational time. For example, FFT has been widely used in computer based numerical techniques for solving electronic structure where bit-reversal algorithm could be considered as main part of any atomic structure software. If the bit-reversal code was not properly designed, FFT could take about 10-30% of the overall computational time, delaying calculations (Saad *et al*., 2010). Therefore, bit-reversal program must be carefully designed to enhance the quality of the code. In this study, a performance analysis of a parallel programming implementation of new bit-reversal algorithm (Kumar *et al*., 1994) on multiprocessors, has been developed via code refactoring of an existing library programs recently used in the computational process of solving the electronic atomic structure of an online atomic database (Tahat and Salah, 2011; Tahat *et al*., 2010; Tahat and Tahat, 2011).

So, the main goal of this study was the improvement of numerical techniques able to solve Schrodinger's time-dependent equation, achieving higher speed and efficiency.

THEORY AND METHODS

Discrete Fourier transform using assumptions: Rewriting Eq. 1 and 2 by using a complex-valued function $\psi(X)$ instead of X and implementing periodic boundary conditions $\psi(X+L) = \psi(X)$ is shown in Fig. 1, including discrete function values, where the propagation of the wave function has been set to be in X direction in the range $X\epsilon[0, L]$. Also $\psi(X)$ has been discretized on N mesh points equally spaced by ΔX as: $X_j = j\Delta X$, with $j = 0,\ldots, N-1$ (Fig. 1). Further, L, X, j, N, X_j represented, the length of the box where ψ is defined (in A°), the spatial coordinate (in atomic units), a counter index, the total number of spatial mesh points of the sample and the discretized step distance along X, respectively. On the other hand, Wave functions $\psi(X)$ at coordinate and momentum spaces must satisfy boundary conditions $\psi(X+L) = \psi(X)$ in the interval $X\epsilon[0, L]$. So that $\psi(0) = \psi(L)$ and $\psi(+/-\pi N/L) = 0$, respectively, so that the meaning of each term (Fig. 1a, b), (L, X, j, N, X_j) should be the same as before.

The expression of Eq. 3, shows that DFT presents the wave function $\psi(X)$ as a linear combination of trigonometric components [exp(iKX) = cos(KX)+i sin(KX)] with different wave numbers:

$$\psi_j = \sum_{m=0}^{N-1} \tilde{\psi}_m \exp(iK_m X_j) \quad (3)$$

Here, K_m represents discrete wave numbers and can be defined as:

$$K_m = \left\{ \begin{array}{l} \frac{2\pi m}{L}; m = 0,1,\ldots,\frac{N}{2-1} \\ \frac{2\pi(m-L)}{L}; m = \frac{N}{2},\frac{N}{2+1},\ldots,N-1 \end{array} \right\} \quad (4)$$

Equation 3 has periodicity L with the expansion coefficients $\tilde{\psi}_m$; where:

$$\tilde{\psi}_m = \frac{1}{N}\sum_{j=0}^{N-1} \psi_j \exp(-iKmXj) \quad (5)$$

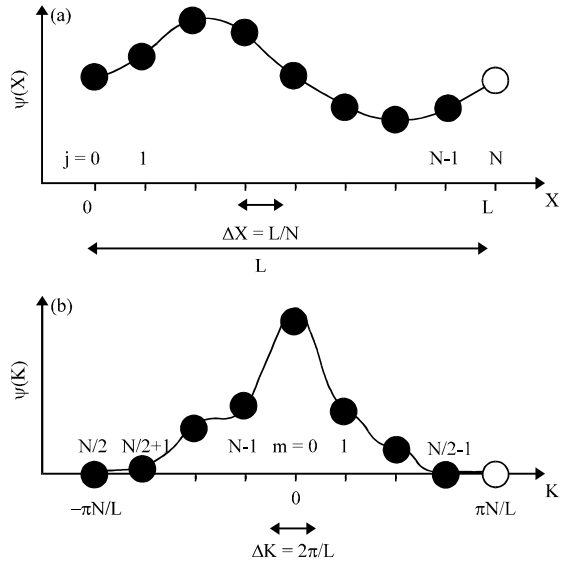


Fig. 1(a-b): Discrete function ψ(X) values and boundary conditions, (a) Discrete function ψ(X) values along X axis on N mesh points and equal mesh spacing ΔX = L/N, using a discrete step of $X_j = j\Delta X$ (j = 0,....., N-1) and (b) Discrete function ψ(K) values at different wave numbers (K) in the range of $[-\pi/\Delta X, \pi/\Delta X]$ at ΔX = L/N and even number N; using a discrete step of $\Delta K = 2\pi/L$. Here, (m) denoted the wave numbers for the higher indices and blank wholes indicate periodic boundary conditions both in coordinates and momentum spaces

Accuracy of the assumed Fourier expansion: The choice of wave numbers in Eq. 4 guarantees that ψ(X) has L periodicity. Wave numbers K_n have been separated by:

$$\frac{2n\pi N}{L} = \frac{2\pi n}{\Delta X}; (n = \pm 1, \pm 2, \pm 3, \pm 4, ...)$$

therefore, equivalent wave numbers would be expected to be found, since because of the discrete sampling all are equivalent in real space (e.g., high values of wave numbers oscillate more but come back to the same value as their lower wave number counterparts at X_j). Moreover, mathematically, the discrete mesh points in real space cannot represent higher wave numbers. But physically the smallest-magnitudes of wave numbers where involved in these assumptions as far as they represent the lowest-energy state. For that reason, the wave numbers for the higher indices in Eq. 4 must be folded back by 2πN/L. For simplicity, it has been assumed that N is an even number. From Fig. 1b, all wave numbers are in the range:

$$\left[\frac{-\pi}{\Delta X}, \frac{\pi}{\Delta X}\right]$$

Considering the discrete function ψ_j, as a vector in N-dimensional vector space, $|\psi\rangle = \psi_0, \psi_1, \psi_2, ..., \psi_{0N-1}$, must be convenient to prove the correctness of Eq. 5. The plane-wave basis set of Eq. 6 is defined in this vector space:

$$\{|m\rangle = b_m(X_j) = \frac{1}{\sqrt{N}} \exp(iK_m X_j) \Big| m = 0,1,..., N-1\} \quad (6)$$

Equation 6 is orthonormal, i.e., inner products of the basis functions are:

$$\langle m|n\rangle = \sum_{j=0}^{N-1} b_m^*(X_j) b_n(X_j) = \delta_{m,n} = \begin{cases} 1; m = n \\ 0; m \neq n \end{cases} \quad (7)$$

Then for m≠n, the sum of the geometric series is carried out; otherwise (m = *n*), all *N* summands must be 1/N, Eq. 8 and 9:

$$\langle m|n\rangle = \frac{1}{N}\sum_{j=0}^{N-1} \exp(i(K_n - K_m)X_j) = \frac{1}{N}\sum_{j=0}^{N-1} \exp(i\frac{2\pi}{N}(n-m)j) \quad (8)$$

$$= \begin{cases} \frac{1}{N}\frac{\exp(i2\pi(n-m)-1)}{\exp(\frac{i2\pi}{N}(n-m))-1} = 0; m \neq n \\ \frac{1}{N} \cdot N = 1; m = n \end{cases} \quad (9)$$

For the complete basis set, a discrete function ψ_j in N-dimensional vector space can be represented as a linear combination of N basis set functions $b_m(X_j)$ in particular:

$$|\psi\rangle = \sum_{m=0}^{N-1} |m\rangle\langle m|\psi\rangle \quad (10)$$

Or:

$$1 = \sum_{m=0}^{N-1} |m\rangle\langle m| \quad (11)$$

So that, suppose the function is expanded as:

$$|\psi\rangle = \sum_{n=0}^{N-1} C_n |n\rangle \quad (12)$$

Multiplying both sides by, ⟨m| and using the orthonormality condition, Eq. 7, leads to:

$$\langle m|\psi\rangle = C_m \quad (13)$$

Therefore, Fourier coefficients $\tilde{\psi}_m$ in Eq. 5 can be readily obtained from Eq. 10 by substituting the definitions of the basis functions and the inner product in Eq. 10. This leads to obtain:

$$\psi_j = \sum_{m=0}^{N-1} \exp(iK_m X_j) \frac{1}{N} \sum_{L=0}^{N-1} \exp(-iK_m X_L)\, \psi_L \quad (14)$$

Identification of the expansion coefficients $\tilde{\psi}_m$ in Eq. 3 and 5 can be done by comparing Eq. 14 with 3.

Solving time-dependent Schrodinger equation using spectral method: Most of the computational effort in electronic structure calculations development had often been focused on solving the time-dependent Schrodinger equation:

$$i\hbar \frac{\partial}{\partial t} \psi = \hat{H}\psi \quad (15)$$

For the purpose of solving this equation numerically, DFT and FFT are the two required computable types of Fourier transformation. Here, $\hat{H}$ represents the Hamiltonian matrix consisting of the sum of the kinetic T and potential operators V and can be defined as:

$$\hat{H} = T + V \quad (16)$$

The ongoing work presents such numerical solution by re-factoring existing software based on a numerical technique, the so called spectral method (Canuto *et al*., 2006) which depends on Fourier transformation (Bracewell, 2000). Here, the exposition of Fourier transform of discretely sampled data and fast Fourier transform from Press *et al*. (1992) will be followed.

Consider the time-dependent Schrodinger equation (Eq. 15), to be implemented in one dimension in atomic unit. Thus:

$$i\hbar \frac{\partial}{\partial t} \psi(X,t) = H\psi(X,t) \quad (17)$$

Here, Hamiltonian operator $\hat{H}$ in Eq. 14 can be defined in one dimension based on the kinetic and potential-energy operators, within the pseudopotential approximation (Heine, 1970) and the momentum-space formalism (Ihm *et al*., 1979), in which plane waves are used as the basis set for the eigenfunctions:

$$\hat{H} = -\frac{h^2}{2m} \frac{\partial^2}{\partial x^2} + V(x) \quad (18)$$

In momentum-space, the kinetic energy operator is diagonal and hence it trivially applies to a single trial eigenvector. However, the calculation of the potential energy would include multiplication in real-space as well as expensive convolution in momentum-space. The kinetic energy operator is diagonal in the Fourier (or momentum) space. To see this, in the present work T was operated on the wave function in its Fourier representation using the assumed discrete Fourier transform of Eq. 5:

$$-\frac{1}{2} \frac{\partial^2}{\partial t^2} \sum_{m=0}^{N-1} \tilde{\psi}_m \exp(iK_m X) = \sum_{m=0}^{N-1} \frac{K_m^2}{2} \tilde{\psi}_m \exp(iK_m X) \quad (19)$$

i.e., the kinetic energy operator multiplied the factor, $K^2_m/2$, applies to the Fourier coefficient of the wave function:

$$\tilde{\psi}_m \xrightarrow[T]{} \frac{K_m^2}{2} \tilde{\psi}_m$$

On the other hand, recalling the potential energy operator is diagonal in the real space means that it multiplies a factor $V_j = V(X_j)$ to the wave function:

$$\tilde{\psi}_j \xrightarrow[V]{} V_j \tilde{\psi}_j$$

This suggests an efficient algorithm for the time evolution of the wave function, where the kinetic and potential energy operators are diagonal in real and momentum space, respectively. Then Trotter expansion Xiao-Ping and Broughton (1987) can be used.

Trotter expansion and spectral method: Recalling the Trotter expansion technique (Xiao-Ping and Broughton, 1987), when used in the current simulation to compute the energy leads to:

$$\psi(X, t+\Delta t) = \exp\left(\frac{-iV\Delta t}{2}\right) \exp(iT\Delta t) \exp\left(\frac{-iV\Delta t}{2}\right) \psi(X,t) + O\left\{[\Delta t]^3\right\} \quad (20)$$

Where:

$$\exp\left(\frac{-iV\Delta t}{2}\right)$$

can be defined as the time evolution operator that had arisen from the potential energy V and it can be easily operated in the real space as:

$$\exp\left(\frac{-iV\Delta t}{2}\right) \psi_j = \exp\left(\frac{-iVj\Delta t}{2}\right) \psi_j \quad (21)$$

On the other hand, the operator $\exp(iT\Delta t)$ can be defined as the time evolution operator that had arisen from the kinetic energy T. This operator can be operated in the Fourier space as:

$$\exp(-iT\Delta t)\tilde{\psi}_m = \exp(\frac{-iK_m^2\Delta t}{2})\tilde{\psi}_m \quad (22)$$

Fast Fourier transforms: In view of the fact that the computation of each of the N Fourier coefficients ψ_m involved summation over N terms, the computational time grows as $O(N)^2$. Then computational cost associated with DFT would be one of the most important obstacles of implementing the spectral method. Herein, using the FFT algorithm will improve the computational effort by reducing the complexity to O(N log N) (Cooley and Tukey, 1965). FFT will allow the cheaply eigenvector transformation between real and momentum space. So potential energy operator can also be applied efficiently, making the quantum dynamics simulation less intensive from the computational point of view. Therefore, FFT has to be involved in the current numerical solution. Equation 20 can be rewritten in terms of forward (F) and inverse Fourier transformation (F^{-1}) operators (Press *et al*., 1992) as follows:

$$\psi(t+\Delta t) = \exp(\frac{-iV\Delta t}{2})F\exp(iT\Delta t)F^{-1}\exp(\frac{-iV\Delta t}{2})\psi(t)) \quad (23)$$

Where:

$$\psi_j \xrightarrow{F^{-1}} F^{-1}\psi_j = \tilde{\psi}_m = \frac{1}{N}\sum_{j=1}^{N}\psi_j\exp(-iK_mX_j) \quad (24)$$

$$\tilde{\psi}_m \xrightarrow{F} F\tilde{\psi}_m = \psi_j = \frac{1}{N}\sum_{m=1}^{N}\psi_m\exp(iK_mX_j) \quad (25)$$

The adopted FFT program being re-factored in this work follows Danielson-Lanczos procedures (Danielson and Lanczos, 1942). More detailed description of the each step in our adopted FFT can be found in reference (Press *et al*., 1992). In short, the input wave function values of the FFT array were first re-ordered by applying the bit-reversal operation to each wave function index. The Danielson-Lanczos procedures, such as in Eq. 26 and 27, then applied recursively, starting from the smaller sub-arrays up:

$$\psi_j = \psi_j^0 + W_N^j\psi_j^1 \quad (26)$$

Where:

$$\begin{cases}\psi_j^0 = \sum_{j=0}^{N/2-1}\tilde{\psi}_{2m}\exp(i2\pi mj/(N/2))\\ \psi_j^1 = \sum_{j=0}^{N/2-1}\tilde{\psi}_{2m+1}\exp(i2\pi mj/(N/2))\\ W_N = \exp(i2\pi/N)\end{cases} \quad (27)$$

ψ_j^0 and ψ_j^1 represent N/2 element Fourier transforms consisting of even and odd sub-arrays, respectively. Since there are $\log_2 N$ recursive steps, the number of complex floating-point operations in the FFT algorithm would be $2\log_2 N$.

Computing the energy: Total energy can be computed by using a set of equations based on the time-dependent Schrodinger equation appearing in Eq. 17 which presented the total energy as a conserved quantity. Thus, by discretizing:

$$\langle H\rangle = \langle V\rangle + \langle T\rangle \quad (28)$$

$$= \int dX\,\psi^*(X)\left(-\frac{1}{2}\frac{\partial^2}{\partial X^2}\right)\psi(X) + \int dX\,\psi^*(X)V(X)\psi(X) \quad (29)$$

$$= dX\sum_{j=0}^{N-1}\psi_j^*\left(-\frac{1}{2}\frac{\partial^2}{\partial X^2}\right)\psi_j + dX\sum_{j=0}^{N-1}V_j|\psi_j|^2 \quad (30)$$

Here, expansion of the wave function in terms of its Fourier components (Eq. 3) is required for calculating the first term (i.e., the kinetic energy $\langle T\rangle$):

$$\langle T\rangle = dX\sum_{j=0}^{N-1}\sum_{m=0}^{N-1}\tilde{\psi}_m^*\exp(-iK_mX_j)\left(-\frac{1}{2}\frac{\partial^2}{\partial X^2}\right)\sum_{n=0}^{N-1}\tilde{\psi}_n\exp(iK_nX_j) \quad (31)$$

$$= dX\sum_{j=0}^{N-1}\sum_{m=0}^{N-1}\tilde{\psi}_m^*\exp(-iK_mX_j)\left(\frac{K_n^2}{2}\right)\sum_{n=0}^{N-1}\tilde{\psi}_n\exp(iK_nX_j) \quad (32)$$

$$= dX\sum_{m=0}^{N-1}\tilde{\psi}_m^*\left(\frac{K_n^2}{2}\right)\sum_{n=0}^{N-1}\tilde{\psi}_n\sum_{j=0}^{N-1}\exp(i(K_n-K_m)X_j) \quad (33)$$

$$= dX\sum_{m=0}^{N-1}\tilde{\psi}_m^*\left(\frac{K_n^2}{2}\right)\sum_{n=0}^{N-1}\tilde{\psi}_nN\delta_{m,n}) \quad (34)$$

$$= dXN\sum_{m=0}^{N-1}\tilde{\psi}_m^*\left(\frac{K_m^2}{2}\right)\sum_{n=0}^{N-1}|\tilde{\psi}_m|^2 \quad (35)$$

Finally, by substituting Eq. 33 in Eq. 28, 29 could be rewritten as follows:

$$\langle H\rangle = dX\cdot N\sum_{m=0}^{N-1}\frac{K_m^2}{2}|\tilde{\psi}_m|^2 + dX\sum_{j=0}^{N-1}V_j|\psi_j|^2 \quad (36)$$

Parallel Algorithm for spectral method: Equation 23 and 36 present the algorithm for the spectral method which is the same for sequential and parallel codes. Parallel implementation of the algorithm involved parallelizing each of the following four steps: (1) nonlinear step, (2) forward F Eq. 25, (3) linear step and (4) backward F^{-1}

Eq. 24. The difficulty of parallelizing spectral method algorithms arises in steps 2 and 4, because there are nontrivial data dependences over the entire range $0 \le L \le N$, involving forward and backward Fourier transforms (FFT and butterfly transform, BFT), respectively. On the other hand, steps 1 and 3 could be trivially evolved because of the natural independence of the data. The problem of parallelizing the one-dimensional FFT has been of great interest for vector (Averbuch *et al*., 1990; Swartztrauber, 1987) and distributed memory computers (Dubey *et al*., 1994). Moreover, communication issues and memory hierarchy are the most two important parameters to be recognized in the paralyzed algorithms in order to achieve an enhanced speed up of the one dimensional FFT.

These one-dimensional algorithms are architecture dependent and involve efficient methods for the data rearrangement and transposition phases (Averbuch *et al*., 1990). The complete parallel algorithm consists of the following steps:

- Nonlinear step
- Row FFT
- Multiplication by $\tilde{\psi}_m$
- Bit-reversal algorithms with respect column FFT
- Linear step (transposed linear operator)
- Column BFT
- Multiplication by the complex conjugate of $\tilde{\psi}_m$, ($\tilde{\psi}_m^*$)
- Row BFT

The parallelism would be due to the row and column subarray FFTs in steps 2, 4, 6 and 8 in addition to the independent operations in steps 1, 3, 5 and 7.

Distributed-memory approach: The Message Passing Interface (MPI) (Karniadakis and Kirby, 2003) can be considered as a tool for distributed parallel computing that has become an “ad hoc” standard (Gropp *et al*., 1998). In distributed parallel programming, different processors would work on completely independent data and explicitly used to send and receive library calls to communicate data between processors (Snir *et al*., 1996; Nehra *et al*., 2007). To implement the distributed parallel algorithm for the time-dependent Schrodinger equation, in this work the rows of FFT arrays will be distributed among 4 processors. Fast communication between them would be the main reason of successful implementation of the algorithm. The large performance cost in this algorithm is due to the redistribution of data between row and column operations. If the row and column computational stages result in significant speed up compared with the communication expense incurred in redistributing the matrix data, then the algorithm will be successful. When speeding up the process of commutation of the current parallel code, bit-reversal problem can be solved by swapping columns and/or rows of a matrix of size X $(n)^{(n)}$ in the FTT arrays. In particular, bit-reversal swapping can be done in a specific time, causing a problem with communication between parallel processes. Consequently, communication took a longer time than the bit-reversal time. The present work has proposed a solution for this problem based on a new algorithm of bit reversal procedure of FFT arrays, by solving the conflict in time between the speed of bit-reversal swapping and the communication between parallel processes on multiprocessors environments. The new algorithms patter performance with parallel processes. In parallel programming, the need for communications between tasks depends upon the nature of the proposed tasks. To implement the final step, the scatter method will be adopted. In the scatter operation, a single node sends a unique message of size m to every other node (also called a one-to-all personalized communication), in order to distribute the data into processors. A unique message from each node would be collected by a single node during the gather operation. In the meantime, gathered operation could be defined as the inverse of the scatter operation and could be executed as such (Fig. 2).

Example: In order to simply explain the proposed algorithms, a compression of the new developed bit-reversal algorithm and the current used algorithm is presented in Appendices 1 and 2. In Appendix 1, it is reported how to distribute the following array of input data: A = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15] into two processors based on the new algorithms. In the meantime, this array can also be distributed into 4 processors as shown in Fig. 3. Further, in Appendix 3 the parallel source code of bit-reversal algorithm is presented.

Advanced example of distributing data on 4 processors using the new bit-reversal algorithm: Breaking the problem into discrete "chunks" of work, must be the first

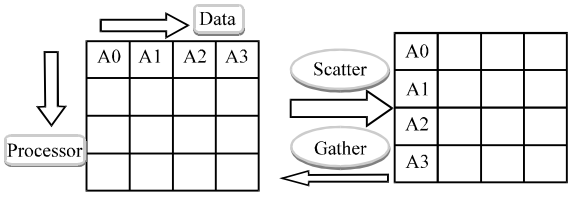


Fig. 2: The scatter method (Scatter and Gather)

step of designing a parallel program, these chunks could be distributed to multiple tasks as well (Kumar *et al*., 1994). The steps of distributing the data of a matrix of 8×8 sizes on 4 processors are presented in Fig. 4 based on four proposed chunks as follows:

- The one dimensional array fulfilled with the input data could be arranged in columns as shown in Fig. 4a

Process No. 0:

0	8	4	12

Process No. 1:

2	10	6	14

Process No. 2:

1	9	5	13

Process No. 3:

3	11	7	15

Fig. 3: Distributing the vector A = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15] in 4 processors

- The bit reversal for each column would be found, simply as shown in Fig. 5a, along with swapping between the columns as shown in Fig. 5b, the significant results of these two processes are illustrated in Fig. 4b
- Distributing the data into specified processors as appeared in Fig. 4c
- Finding the bit reversal for each row then swapping between the rows must be done. The final results would be appeared in the last step as shown in Fig. 4d

On the other hand, number of process must be in base 2×2"; (n = 0, 1, 2, 3.., etc.) for any arrangement (e.g., 2, 4, 8, 16, 32, 64, 128, 256, etc.); (Kumar *et al*., 1994). Herein, the data of the (8×8) matrix could be also distributed into 8 processors. Figure 6 presented the significant results of the redistributed data.

(a)

Binary No.	000	001	010	011	100	101	110	111
000	0	8	16	24	32	40	48	56
001	1	9	17	25	33	41	49	57
010	2	10	18	26	34	42	50	58
011	3	11	19	27	35	43	51	59
100	4	12	20	28	36	44	52	60
101	5	13	21	29	37	45	53	61
110	6	14	22	30	38	46	54	62
111	7	15	23	31	39	47	55	63

(b)

Binary No.	000	001	010	011	100	101	110	111
000	0	32	16	48	8	40	24	56
001	1	33	17	49	9	41	25	57
010	2	34	18	50	10	42	26	58
011	3	35	19	51	11	43	27	59
100	4	36	20	52	12	44	28	60
101	5	37	21	53	13	45	29	61
110	6	38	22	54	14	46	30	62
111	7	39	23	55	15	47	31	63

(c)

Process No. 0:

0	32	16	48	8	40	24	56
1	33	17	49	9	41	25	57

Process No. 1:

2	34	18	50	10	42	26	58
3	35	19	51	11	43	27	59

Process No. 2:

4	36	20	52	12	44	28	60
5	37	21	53	13	45	29	61

Process No. 3:

6	38	22	54	14	46	30	62
7	39	23	55	15	47	31	63

(d)

Process No. 0:

0	32	16	48	8	40	24	56
4	36	20	52	12	44	28	60

Process No. 1:

2	34	18	50	10	42	26	58
6	38	22	54	14	46	30	62

Process No. 2:

1	33	17	49	9	41	25	57
5	37	21	53	13	45	29	61

Process No. 3:

3	35	19	51	11	43	27	59
7	39	23	55	15	47	31	63

Fig. 4(a-d): Example of an 8×8 matrix, (a) Arrange table, (b) Take the bit reversal of column and then swap the columns, (c) Distribute data on several processes (4 processes) and (d) Communication between the processors takes the bit reversal of row and then swap the rows

(a)

Binary No.	Bit reversal
000	000
001	100
010	010
011	110
100	001
101	101
110	011
111	111

(b)

Column No.	Swap with
1	Not swap
2	5
3	Not swap
4	7
5	2
6	Not swap
7	4
8	Not swap

Fig. 5(a-b): Explanation of bit reversal and swap of columns, (a) Bit-reversal of columns and (b) Swap between the columns

Process No. 0:							
0	32	16	48	8	40	24	56
Process No. 1:							
4	36	20	52	12	44	28	60
Process No. 2:							
2	34	18	50	10	42	26	58
Process No. 3:							
6	38	22	54	14	46	30	62
Process No. 4:							
1	33	17	49	9	41	25	57
Process No. 5:							
5	37	21	53	13	45	29	61
Process No. 6:							
3	35	19	51	11	43	27	59
Process No. 7:							
7	39	23	55	15	47	31	63

Fig. 6: Distributing the data of the matrix (8×8) into 8 processes

RESULTS

After refactoring the existing code and including the new bit-reversal algorithms for reordering the input data and to do the FFT matrix in the spectral method, the one-processor implementation of the bit reversal parallel code has become 8.7 to 17.3% faster than that of the sequential code (Table 5). The produced parallel code has been tested by using a portable implementation of MPI, called MPICH (http://www.mcs.anl.gov/research/projects/mpich2/).

Program input parameters:

- Matrix dimension
- A (1, 1) = 0, 1, 2… any random number
- Fill raw/column
- Number of processors (n)

Program outputs: Table 2-5 present the computed results for distributed memory and MPI implementations of the new bit-reversal algorithm for different square matrix of size X $(n)^{(n)}$ of FTT arrays. They have been produced and tested using three methods for computing the parallel speed up, in order to measure the program scalability and to show how program could scale as more processors would be used.

Speed up for various matrix size X $(n)^{(n)}$ at different fractions (P) and fixed number of processors (n = 4): As shown in Table 2, with four processors (n = 4), the speed up increases with the working set size X(n). The measured results showed that the speed up eventually decreased with larger X(n) at various values of fraction p. The maximum and minimum dimensionless speed up ratios have been 3.8352 and 2.9110 occurred at p = 0.9856; X (n) = 2×2 and p = 0.8753; X (n) = 19×19, respectively. The speed up values can be very close to each other at same values of parallel fractions p, far away from the matrix size. For example the speed up was equalled (3.7352) at X (n) = 3×3, p = 0.9763 which was very close to the speed up of X (N) = 4×4; (3.7123) at p = 0.9741. The measured efficiency is closely related to speed up at fixed n and dependent proportionally of matrix size. All values had been less than one and ranged between 0.9588 and 0.7277. The communication cost for the MPI code as a function of efficiency $T_1/E(n)$ has become close to perfect speed up showing an excellent agreement with the expected ideal cost that may be calculated by multiplying the number of used processors (n = 4) by parallel processing time (T_4).

Speed up at fixed fraction (P) and fixed matrix size X $(n)^{(n)}$ on different number of processors (n): For fixed parallel fraction (p = 0.9545), the speed up and the cost rapidly increased with larger number of processors for the same matrix size (8×8), with decreasing efficiency. The maximum speed up has been 20.3134 and occurred at the highest number of processors n = 256 (Table 3).

Speed up at large matrix size X $(n)^{(n)}$ with various fractions (P): The results of Table 4 and 5 revealed that the speed up increased as the problem size X (n) became larger, for high values of parallel fraction; (e.g., p ≈ 0.9554) with increasing the number of processors. However, speed up eventually decreased with larger X (n) on the

Table 2: Results for distributed-memory and MPI implementations of $(2\times2)\geq X(n)^n\leq(19\times19)$, at n = 4, A (1, 1) = 0, with filling columns, repeat = 1000

Matrix dimension ×n	Processing time (10^{-3} sec) Single processor T_1	4 parallel processors Tn	Speedup $S(n)=\frac{T_1}{Tn}$	Efficiency E(n) = S(n)/n	Fraction of the code can be made parallel (P)	Remaining fraction of the code must run serially (1-P)	Cost $T_1/E(n)$
2×2	0.0831	0.0216	3.8352	0.9588	0.9856	0.0143	0.0866
3×3	0.2842	0.0760	3.7352	0.9338	0.9763	0.0236	0.3043
4×4	0.5806	0.1563	3.7123	0.9280	0.9741	0.0258	0.6255
5×5	1.1055	0.3034	3.6432	0.9108	0.9673	0.0326	1.2137
6×6	1.8533	0.5119	3.6201	0.9050	0.9650	0.0349	2.0477
7×7	3.0048	0.8516	3.5281	0.8820	0.9554	0.0445	3.4066
8×8	4.5139	1.2823	3.5201	0.8800	0.9545	0.0454	5.1292
9×9	6.3657	1.8254	3.4872	0.8718	0.9509	0.0490	7.3017
10×10	8.7578	2.5163	3.4803	0.8700	0.9502	0.0497	10.0654
11×11	11.736	3.5503	3.3056	0.8264	0.9299	0.0700	14.2013
12×12	15.158	4.5774	3.3114	0.8278	0.9306	0.0693	18.3098
13×13	19.286	5.8974	3.2702	0.8175	0.9256	0.0743	23.5899
14×14	24.717	7.6779	3.2192	0.8048	0.9191	0.0808	30.7116
15×15	29.555	9.2355	3.2001	0.8000	0.9166	0.0833	36.9422
16×16	35.339	11.5975	3.0471	0.7617	0.8957	0.1042	46.3900
17×17	42.329	13.9967	3.0242	0.7560	0.8924	0.1075	55.9868
18×18	50.270	17.2598	2.9125	0.7281	0.8755	0.1244	69.0393
19×19	59.500	20.4394	2.9110	0.7277	0.8753	0.1246	81.7577

Table 3: Speed up for implementing a matrix of size (8×8) at (p = 0.9545; n = 4)

No. of processors CPUs	Speed up S(n)	Efficiency $\left(E(n)=\frac{S(n)}{n}\right)$	Cost $\left(Cost=\frac{T_1}{E(n)}\right)$
4	3.5195	0.8799	5.1300
8	6.0675	0.7584	5.9516
16	9.5096	0.5944	7.5946
32	13.2752	0.4149	10.8808
64	16.5524	0.2586	17.4530
128	18.8832	0.1475	30.5975
256	20.3134	0.0793	56.8864

Table 4: Speed up at large matrix size

No. of processors	Speed up (8×8) p = 0.9554	Speed up (15×15) p = 0.9166	Speed up (17×17) p = 0.8924	Speed up (18×18) p = 0.8755	Speed up (19×19) p = 0.8753
4	3.5195	3.1995	3.0239	2.9123	2.9110
8	6.0675	5.0511	4.5631	4.2746	4.2715
16	9.5096	7.1080	6.1209	5.5798	5.5739
32	13.2752	8.9251	7.3808	6.5850	6.5766
64	16.5524	10.2331	8.2275	7.2370	7.2267
128	18.8832	11.0423	8.7281	7.6138	7.6023
256	20.3134	11.4968	9.0020	7.8174	7.8052

Table 5: Output of running the re-factored code after including new bit-reversal algorithm

×n	T1 (sec)	T2 (sec)	T4 (sec)	Speed up T1/T2	Speed up T1/T4
12×12	36.8	23.9	17.9	1.539749	2.055866
14×14	43.7	24.6	15.7	1.776423	2.783439
16×16	58.7	33.7	19.8	1.74184	2.964646
18×18	91.8	64.8	25.6	1.416667	3.585938

same processor at lowered values of parallel fraction (p = 0.8753). Generally speaking, all measured speed up of the large matrix has increased when increasing the number of processors based on p. In all cases p was less than one and maximum speedup was around 20.

DISCUSSION

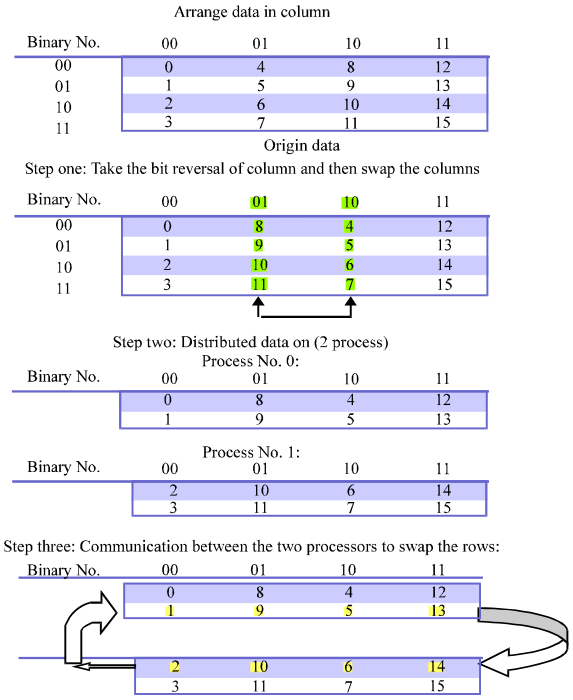
Parallel performance analysis: The reported results in this work indicated that parallel implementations of distributed memory for the bit-reversal of FFT arrays over the range $(2\times2)\geq X(n)^n\leq(19\times19)$, produced important speed up of the calculations, with maximum values at high parallel fraction, beyond which the communication cost increased and the computation/communication ratio decreased. The communication cost for the MPI code has been a result of "communication stages"; the less than perfect speed up is thus due to the volume of data communicated between processors in redistribution stages and not to the constant sharing of small sub-array data. Many studies in the literature had covered such topic. For example, Yan and Zhang (1999), focused on investigating the relationship between cost-effective parallel computing along with profit-effective parallel computing. In the present work the cost results match their ideal result. With four processors (n = 4), the speed up increased with the working set size X(n) based on (p). This has been due both to the faster computation stages and to the reduced volume of data communicated between single processors in the redistribution stage. The division of the problem among four processors would not be beneficial in the case of small problem sizes, because there would not be enough work to make. In the main time, the speed up could be attributed to data independency contained in the processor's local cache between stages of data rearrangement. These observations are in excellent agreement with the qualitative trends seen in Amdahl's law (Amdahl, 1967), empirical data on speed up for parallel

computing method, where the maximum speed up was attained with a problem size. Therefore, the current results showed that if the fraction p is not high enough for effective scaling, there is no point in further tuning until it had been increased (e.g., 8×8 matrix, p = 0.9545), increasing the number of processors will not affect the value of the speed up at very low fractions (e.g., p = 0.1). For this reason, parallel computing will be only useful for either small numbers of processors (n = 4) or problems with very high values of p. The present results also show a good agreement with those reported by Gustafson (Gustafson, 1988). The efficiency estimated the processors affectivity in solving problems, compared to the wasted effort in the communication and synchronization processes. In addition, efficiency had to be ranged between 0 and 1. Thus, this work showed that all measured values of efficiency were less than one. This showed a good agreement with expected efficiency according to Alba's work (Alba, 2002). Finally a comparison between spectral method techniques of solving the time dependent Schrodinger equation can be found in (Feit *et al*., 1980; Pshenichnov and Sokolov, 1964; Zhirnov and Turev, 1980). The difference between these studies and the current work is the present use of the distributed memory parallel programming in solving a critical part of the fast Fourier transform that reduces the time of solving the equation which has lead to increase the speed up and to reach the ideal value of Amdahl's law (Amdahl, 1967), where S(n) must equal the number of processors (n). In this work all measured speed up on 4 processors have been very close to 4. On the other hand this work could be retested by using the sharing memory approach in solving the Schrodinger equation other than the distributed parallel programming.

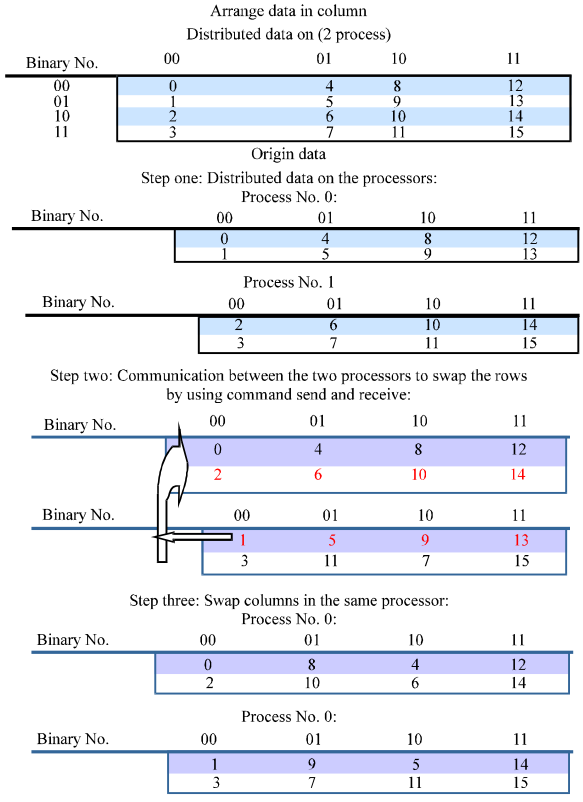
The spectral method commonly used in the numerical solution of the time-dependent Schrodinger equation often proved very slow in serial versions, even on the fastest workstations. This study has successfully implemented and tested the parallelization of new bit-reversal algorithms that have been shown to be appropriated for multiprocessors that outperformed an existing one. Distributed bit-reversal FFT speed up is a function of the number of processors (n) which reduces both the computational time and the communication volume between single processors. The speed up of the parallel algorithm has been strongly dependent on reductions in both communication time (Tn) and contention in the multiprocessor. Thus, MPI has been revealed to be an optimal method of implementing parallelized bit-reversal algorithms.

APPENDIX

Appendix 1: Bit-reversal algorithm after refactoring



Appendix 2: Bit-reversal algorithm before refactoring



Appendix 3: C++ (MPI) code

```
#include "stdafx.h"
//#include<stdio.h>
//#include<stdlib.h>
//#include<math.h>
//#include<conio.h>
#include <mpi.h>
#include<iostream>
using namespace std;
#define Asize 8*8
#define Asize1 8
#define process 2
#define part (Asize1/process)
#define MASTER 0
int apart[part][Asize1];
int temp1[Asize1], temp2[Asize1];
//-------------------------------------------------------------
void swap(int and a,int and b){
        int  tempr;
        tempr = a;
        a = b;
        b = tempr;
}
//-------------------------------------------------------------
void bitreversal(int data1[],unsigned long nn)
{
        unsigned long n,m,j,i ;
          n = nn ;
          j = 0;
          for(i = 0;i<n-2;i+ = 1){
           if (j>i){
             /*                  swap(data[j],data[i]);   */
                              swap(data1[j],data1[i]);
                }
           m = n >> 1;
           /* cout<<" m"<<m<<endl;     */
            while( j > = m){
                                j - = m;
                                m >> = 1;
                                }
            j + = m;
                }
}
//-------------------------------------------------------------
int main(int argc,char *argv[])
{
        int numtasks,taskid,i,j,k; //y
        MPI_Status status;
        MPI_Request request;
        double time1,time2;
        MPI_Init(&argc,&argv);
        MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
        MPI_Comm_rank(MPI_COMM_WORLD,&taskid);
        int data[8][8];
        unsigned long n = 16;
        int n1 = 8;int bit_rev[8];
        //int y4,p,p1;
        if (taskid==MASTER)
        {
                for(i=0;i<n1;i++)
                        for(j = 0;j<n1;j++)
                                data[j][i] = j + n1*i;
        }
        MPI_Scatter(data,part*Asize1,MPI_INT,apart,part*Asize1,M
        PI_INT,MASTER,MPI_COMM_WORLD);
        // initailze the input data
        for (I = 0;i<n1;i++)
                bit_rev[i] = i;
         bitreversal(bit_rev,n1);
```

Appendix 3: Continue

```
// print input data in two dimension array ( data shpuld be saved in column
)
/*      if(taskid==0){
         cout<<" the apart";
            for(p1 = 0;p1<part;p1++)
        {     cout<<endl;
           for( p = 0;p<n1;p++)
           cout<<apart[p1][p]<<" ";
         }
        }
                cout<<endl;
*/
        time1 = MPI_Wtime();
        for(k = 0;k<100;k++)
        {
                for(i = 0;i<part;i++)
                                bitreversal(apart[i],n1);
                for(i = part*taskid;i<part*taskid+part;i++)
                {
                                j = bit_rev[i];
                                //              if(taskid==0){
                                //cout<<"i="<<i<<"j="<<j<<endl;
                                //}
                                if(j==i);
                                else
                                {
                                                int id=j/part;
                                                int index=i%(part);
                                                //              if(taskid==0){

//cout<<"id="<<id<<"index="<<index<<endl;
                                                //}

MPI_Irecv(temp2,Asize1,MPI_INT,id,100,MPI_COMM_WORLD,&req
uest);
                                                for(int u =0; u<Asize1; u++)
                                                        temp1[u]=apart[index][u];
MPI_Send(temp1,Asize1,MPI_INT,id,100,MPI_COMM_WORLD);
                                                MPI_Wait(&request,&status);
                                                for(u = 0; u<Asize1; u++)
                                                        apart[index][u] = temp2[u];
                                                //++++++++++++++++++++++++
MPI_Irecv(temp2,Asize1,MPI_INT,id,100,MPI_COMM_WORLD,&req
uest);
                                                for(u =0; u<Asize1; u++)
                                                        temp1[u]=apart[u][index];
MPI_Send(temp1,Asize1,MPI_INT,id,100,MPI_COMM_WORLD);
                                                MPI_Wait(&request,&status);
                                                for(u=0; u<Asize1; u++)
                                                        apart[u][index] = temp2[u];
                                }
                        }
        }
        if (taskid==MASTER) {
                time2=MPI_Wtime();
                cout<<" the total time is ="<<(time2-time1)/100;
        }
        MPI_Finalize();
        return 0;
}
```

REFERENCES

Alba, E., 2002. Parallel evolutionary algorithms can achieve super-linear performance. Inform. Process. Lett., 82: 7-13.

Amdahl, G.M., 1967. Validity of the single processor approach to achieving large scale computing capabilities. Proc. AFIPS Spring Joint Comput. Conf., 30: 483-485.

Arioua, M., S. Belkouch and M.M. Hassani, 2012. Efficient 16-points FFT/IFFT architecture for OFDM based wireless broadband communication. Inform. Technol. J., 11: 118-125.

Averbuch, A., E. Gabber, B. Gordissky and Y. Medan, 1990. A parallel FFT on an MIMD machine. Parallel Comput., 15: 61-74.

Bracewell, R.N., 2000. The Fourier Transform and its Applications. 3rd Edn., McGraw-Hill, Boston, ISBN: 0073039381.

Canuto, C., M.Y. Hussaini, A. Quarteroni and T.A. Zang, 2006. Spectral Methods: Fundamentals in Single Domains. Springer-Verlag, Berlin, ISBN: 9783540307266, Pages: 563.

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

Danielson, G.C. and C. Lanczos, 1942. Some improvements in practical Fourier analysis and their application to X-ray scattering from liquids. J. Franklin Inst., 233: 365-380.

Dubey, A., M. Zubair and C.E. Grosch, 1994. A general purpose subroutine for fast Fourier transform on a distributed memory parallel machine. Parallel Comput., 20: 1697-1710.

Feit, M.D., J.A. Jr Fleck and A. Steiger, 1980. Solution of the schrtidinger equation by a spectral method. J. Comput. Phys., 47: 412-433.

Fowler, M., 1999. Refactoring: Improving the Design of Existing Code. Addison-Wesley, New York, USA., ISBN-13: 9780201485677, Pages: 431.

Griswold, W.G., 1991. Program restructuring as an aid to software maintenance. Ph.D. Thesis, University of Washington

Gropp, W., S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir and M. Snir, 1998. MPI: The Complete Reference. The MPI-2 Extensions. vol. 2, MIT Press, Cambridge, MA., ISBN-13:9780262571234, Pages: 344.

Gustafson, J.L., 1988. Reevaluating amdahl's law. Commun. ACM, 31: 352-533.

He, C., F. Lang and H. Li, 2011. Medical image registration using cascaded pulse coupled neural networks. Inform. Technol. J., 10: 1733-1739.

Heine, V., 1970. The Pseudopotential Concept. In: Solid State Physics, Ehrenreich, H., F. Seitz and D. Turnbull (Eds.). Vol. 24, Academic Press, New York, USA., pp: 1-36.

Ihm, J., A. Zunger and M.L. Cohen, 1979. Momentum-space formalism for the total energy of solids. J. Phys. C: Solid State Phys., 12: 4409-4422.

Jing, M., Z. Nai-Tong and Z. Qin-Yu, 2010. IR-UWB waveform distortion analysis in NLOS localization system. Inform. Technol. J., 9: 139-145.

Karniadakis, G.E. and R.M. Kirby, 2003. Parallel Scientific Computing in C^{++} and MPI: A Seamless Approach to Parallel Algorithms and their Implementation. Cambridge UniversityPress, UK., ISBN-13:9780521817547, Pages: 628.

Kerievsky, J., 2004. Refactoring to Patterns. 1st Edn., Addison Wesley, USA., ISBN-13: 978-0321213358, Pages: 400.

Kumar, V., A. Grama, A. Gupta and G. Karypis, 1994. Introduction to Parallel Computing: Design and Analysis of Algorithms. Benjamin/Cummings Publishing Co., IPC., North-Holland, ISBN: 9780805331707, Pages: 597.

Nehra, N., R.B. Patel and V.K. Bhat, 2007. Load balancing with fault tolerance and optimal resource utilization in grid computing. Inform. Technol. J., 6: 784-797.

Peng, W., X. Wang, K. Chen and B. Tang, 2009. The analysis of the synthetic range profile based on doppler filter bank using FFT. Inform. Technol. J., 8: 1160-1169.

Press, W.H., B.P. Flannery, S.A. Teukolsky and W.T. Vetterling, 1992. Numerical Recipes in Fortran the Art of Scientific Computing. 2nd Edn., Cambridge University Press, New York.

Pshenichnov, E.A. and N.D. Sokolov, 1964. Eigenvalues and quantum transition probabilities for an asymmetrical double potential well. Optics Spectrosc., 17: 183-183.

Robert, M., 2008. Clean Code: A Handbook of Agile Software Craftsmanship. 1st Edn., Prentice Hall, New York, ISBN-10: 0132350882.

Saad, Y., J.R. Chelikowsky and S.M. Shontz, 2010. Numerical methods for electronic structure calculations of materials. SIAM Rev., 52: 3-54.

Sarkar, T., E. Arvas and S. Rao, 1986. Application of FFT and conjugatevgradient method for the solution of electromagnetic radiation from electrically large and small conduction bodies. Trans. Antennas Propagat., 34: 635-640.

Snir, M., S. Otto, S.H. Lederman, D. Walker and J. Dongarra, 1996. MPI: The Complete Reference. MIT Press, London, ISBN-10: 0262692155.

Swartztrauber, P.N., 1987. Multiprocessor FFTs. Parallel Comput., 5: 197-210.

Tahat, A. and W. Salah, 2011. Comprehensive online atomic database management system (DBMS) with highly qualified computing capabilities. Int. J. Data Database Manage. Syst., 3: 1-20.

Tahat, A. and M. Tahat, 2011. Python GUI scripting interface for running atomic physics applications. Python Paper Source Codes, 3: 1-7.

Tahat, A.N., W. Salah and A.B. Hallak, 2010. PASS (Pixe analysis shell software): A computer utility program for the evaluation of PIXE spectra. Int. J. Pixe, 20: 63-76.

Xiao-Ping, L. and J.Q. Broughton, 1987. High-order correction to the Trotter expansion for use in computer simulation. J. Chem. Phys., 86: 5094-5100.

Yan, Y. and X. Zhang, 1999. Profit-effective parallel computing. IEEE Concurrency, 7: 65-69.

Zhirnov, N.I. and A.V. Turev, 1980. New approximate solution of the anharmonic oscillator problem. Optics Spectrosc., 49: 142-144.