The Ministry of Education and Science of the Russian Federation

Lobachevsky State University of Nizhni Novgorod

Computing Mathematics and Cybernetics faculty

The competitiveness enhancement program

of the Lobachevsky State University of Nizhni Novgorod

among the world's research and education centers

Strategic initiative

“Achieving leading positions in the field of supercomputer technology

and high-performance computing”

TBB-Based Parallel Programming

Practice 2. Parallel Computation of the Fast Fourier Transform

Nizhni Novgorod

2014

Practice 2. Parallel Computation of the Fast Fourier Transform

Objectives

The purpose of this practice is to demonstrate recursive parallel algorithms as illustrated by the problem of the Fast Fourier Transform computation.

Abstract

This practice formulates the problem of the Fast Fourier Transform Computation. It features a sequential algorithm implementation. It describes parallel algorithm implementation versions and implements the iterative algorithm version based on parallel_for and the recursive version based on logical tasks.

Guidelines

The Fourier Transform is based on a simple idea that any periodic function can be decomposed into harmonic constituents (sine and coside waves of different amplitude, period and frequency).

The undeniable advantage of Fourier Transform is its flexibility: it is applicable to both continuous and discrete functions. In the latter case, it is called the Discrete Fourier Transform, or DFT.

Forward DFT:

$$y_p \equiv \sum_{k=0}^{n-1} x_k \exp\left(-\frac{kp}{n} 2\pi i\right), \quad p = \overline{0, n-1}.$$

Backward transform will be expressed as:

$$x_k \equiv \frac{1}{n} \sum_{p=0}^{n-1} y_p \exp\left(\frac{kp}{n} 2\pi i\right), \quad k = \overline{0, n-1}.$$

DFT computation, most often as FFT, is implemented in many mathematical libraries such as Intel MKL and Intel IPP. Fourier Transform may be generalized for multidimensional functions. Currently, there are several Fast Fourier Transform algorithms. This practice describes one of the most common algorithm implementations proposed by J.W. Cooley and John Tukey.

The Cooley-Tukey algorithm divides the initial set of points into two equal subsets (the input data size must be equal to a power of two). Each of subsets is again split into two parts etc. Each of the resulting sets is subject to FFT. Computations can be executed independently; this will be used for parallel version implementation.

The Cooley–Tukey algorithm has two steps:

1. input array permutation (*bit reversal*);
2. FFT computation.

Step 1 consists in initial data permutation to simplify thei handling in the future. In some FFT algorithms this step is omitted and included into the computation step.

Bit reversal is binary number conversion by digit-reversal permutation. Bit reversal is applicable only to array element indices and is intended for permutation of these elements while their values remain unchanged.

Let us consider the following example of the algorithm. Let initial array contain 16 elements; then transformation of indexes will take place as follows:

0	0	0	0	= 0	→	0	0	0	0	= 0	1	0	0	0	= 8	→	0	0	0	1	= 1
0	0	0	1	= 1	→	1	0	0	0	= 8	1	0	0	1	= 9	→	1	0	0	1	= 9
0	0	1	0	= 2	→	0	1	0	0	= 4	1	0	1	0	= 10	→	0	1	0	1	= 5
0	0	1	1	= 3	→	1	1	0	0	= 12	1	0	1	1	= 11	→	1	1	0	1	= 13
0	1	0	0	= 4	→	0	0	1	0	= 2	1	1	0	0	= 12	→	0	0	1	1	= 3
0	1	0	1	= 5	→	1	0	1	0	= 10	1	1	0	1	= 13	→	1	0	1	1	= 11
0	1	1	0	= 6	→	0	1	1	0	= 6	1	1	1	0	= 14	→	0	1	1	1	= 7
0	1	1	1	= 7	→	1	1	1	0	= 14	1	1	1	1	= 15	→	1	1	1	1	= 15

Fig. 1 Example of bit reversal

In the general case, when input data size is equal to a power of two ($n = 2^m$), the number of FFT computation steps is equal to $\log n = m$. Each step involves repeated execution of one and the same operation which is often called butterfly. The butterfly inputs are two numbers, *a* and *b*. The outputs are two different numbers, $a + bU$ and $a - bU$. U is the *rotation coefficient*. See Fig.2 for a graphic representation of this operation to understand the origins of its name.

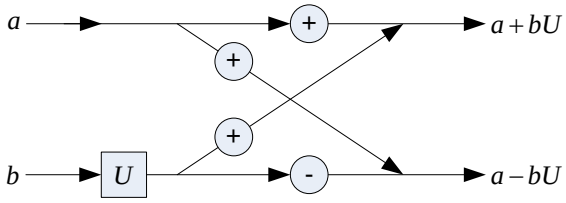

Fig. 2 Butterfly, the basic FFT computing operation

Let us review the general FFT computation algorithm. Let $n = 2^m$ be the input data size and signal be the input array of complex values.

1. bSize = 1 (butterfly step).
2. i = 0, j = 0.
3. Apply the butterfly to signal [i * bSize * 2 + j] and signal [i * bSize * 2 + j + bSize] with rotation coefficient $U = e^{-\frac{j}{bSize}\pi i}$.
4. If j < bSize - 1, then j++, go to step 3.
5. If i < bSize * n / 2 - 1, then i++, j = 0, go to step 3.
6. If bSize < n/2, then bSize = bSize * 2, go to step 2, otherwise algorithm completion.

Having performed the experiments, one can see that bit reversal requires little time as compared to other computations; this is why is it step 2, or FFT computation, that should be parallelized.

Depending on the FFT algorithm implementation, one can use either parallel_for to parallelize the iterative algorithm inner loop (outer butterfly loop must run sequentially) or logical task to parallelize the recursive algorithm.

Recommendations for Students

Cormen, Leiserson, Rivest, Stein (2009) is a recommended introduction to algorithms.

Quinn (2004) is also recommended as a description of typical problems of parallel programming.

Practice

1. Implement parallel bit reversal. Evaluate contribution of this implementation to speedup.
2. The parallel version based on logical tasks is implemented so that the last iteration is executed in a sequential manner. Implement the FFT so that the last iteration is executed in parallel, too.

Test questions

1. What is the computational complexity of the DFT algorithm?
   a. O(n).
   b. O(n logn).
   c. (+) O(n2).
   d. O(n2 logn).
2. What is the computational complexity of the FFT algorithm?
   a. O(n).
   b. (+) O(n logn).
   c. O(n2).
   d. O(n2 logn).
3. FFT is:
   a. Algorithm of plane points mapping onto a sphere.
   b. Data filtering algorithm
   c. (+) DFT computation algorithm
4. Bit reversal is:
   a. Binary number transformation by rotate bit shift.
   b. (+) Binary number conversion by digit-reversal permutation.
   c. Conversion consisting in replacing bits by reverse ones.
5. To implement a parallel FFT algorithm, one can use
   a. (+) parallel_for
   b. parallel_reduce
   c. (+) task

d. parallel_do

6. To implement a parallel iterative FFT algorithm, one can use
   a. (+) parallel_for
   b. parallel_reduce
   c. task
   d. parallel_do
7. To implement a parallel recursive FFT algorithm, one can use
   a. parallel_for
   b. parallel_reduce
   c. (+) task
   d. parallel_do
8. What FFT stage should be parallelized first?
   a. Input array permutation (bit reversal).
   b. (+) FFT computation
   c. Does not matter.
9. Pointer to the processed array in case of parallelization using parallel_for:
   a. (+) Must be stored in functor fields
   b. Must be stored in global variables.
   c. Must be sent to operator().
10. In case of parallelization using parallel_for, the functor fields must store:
   a. (+) Pointer to the processed array.
   b. Number of the currently processed array element.
   c. (+) Outer loop iteration number or butterfly step
11. Parallel implementation based on logical tasks in the execute() method:
   a. Has to allocate memory to logical tasks using new(allocate_root()).
   b. (+) Has to allocate memory to logical tasks using new(allocate_child()).
   c. Has to allocate memory to logical tasks using malloc().
12. Parallel implementation based on logical tasks:
   a. (+) It make sense to implement a sequential FFT version if the array is small enough
   b. The parallel version must be executed until a single-element array is obtained to ensure maximum scalability of the implementation.

References

1. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein C. (2009). Introduction to Algorithms, 3rd Edition. – The MIT Press.

2. Kumar V., Grama, A., Gupta, A., Karypis, G. (1994). Introduction to Parallel Computing. - The Benjamin/Cummings Publishing Company, Inc. (2nd edn., 2003)

3. Andrews, G. R. (2000). Foundations of Multithreaded, Parallel, and Distributed Programming. – Reading, MA: Addison-Wesley.

4. Quinn, M. J. (2004). Parallel Programming in C with MPI and OpenMP. – New York, NY: McGraw-Hill.