Нижегородский государственный университет им. Н.И.Лобачевского

Межфакультетская магистратура по системному и прикладному программированию для многоядерных компьютерных систем

Образовательный комплекс «Технологии разработки параллельных программ»

Лабораторная работа: Использование механизма логических задач библиотеки Intel Threading Building Blocks на примере вычисления быстрого преобразования Фурье

Разработчики: А.В. Сысоев, И.Б. Мееров, А.А. Сиднев.

Нижний Новгород
2007

Содержание

Введение

В настоящей лабораторной работе рассматривается библиотека Intel Threading Building Blocks (TBB) [16], позволяющая разрабатывать параллельные программы для систем с общей памятью. Идея, лежащая в основе библиотеки – использовании стандартного высокоуровневого языка C++ для быстрой разработки кросс-платформенных, хорошо масштабируемых параллельных приложений (см. подробнее в [1]). Кроме того, библиотека TBB предоставляет механизмы абстрагирования от парадигм многопоточного программирования, позволяя сосредоточиться непосредственно на решении прикладной задачи, что является достаточно актуальным.

Дальнейшее изучение принципов функционирования и вопросов эффективного использования TBB производится на примере вычисления быстрого преобразования Фурье (БПФ). БПФ, с одной стороны, широко используется в практических задачах, возникающих в теории автоматического регулирования и управления, в теории фильтрации, в задачах цифровой обработки сигналов и т.д., с другой, реализация эффективной параллельной версии вычисления БПФ представляет собой весьма непростую задачу.

В ходе выполнения работы предполагается, что слушатели имеют навыки разработки объектно-ориентированных программ на C++, а также владеют основами параллельного программирования в системах с общей памятью.

1. Методические рекомендации

1.1. Цели и задачи работы

Целью данной лабораторной работы является приобретение практических навыков в использовании механизма логических задач библиотеки TBB.

Данная цель предполагает решение следующих задач:

- изучение способов инициализации и завершения работы библиотеки TBB (см. также [1]);
- освоение функциональности библиотеки TBB, связанной с механизмом логических задач (см. также [1]);
- рассмотрение учебной задачи, направленной на демонстрацию использования механизма логических задач библиотеки TBB;
- самостоятельная разработка и отладка параллельных программ с использованием механизма логических задач TBB.

1.2. Структура работы

Настоящий документ состоит из введения, четырех разделов, списка дополнительных заданий и списка литературы. Во введении обосновывается актуальность использования TBB в процессе разработки параллельных программ. В первом разделе даются методические рекомендации к лабораторной работе: формулируются цели и задачи, системные требования, рекомендации по проведению занятий. Во втором разделе приводятся теоретические сведения, необходимые для дальнейшей реализации последовательного и параллельных алгоритмов преобразования Фурье. В третьем разделе описывается реализация последовательной версии вычисления БПФ. В четвертом разделе проводится реализация двух параллельных версий вычисления БПФ: с использованием функции **`tbb::parallel_for`**, а также на основе механизма логических задач библиотеки TBB. В заключение приводятся задания для самостоятельной проработки, а также использованная и рекомендуемая литература.

1.3. Системные требования

Приведем системные требования библиотеки TBB для Windows-систем. Аналогичную информацию для систем на базе Linux и Mac OS можно найти на официальном сайте TBB [16].

1.3.1. Аппаратное обеспечение

Минимальные требования

- Intel® Pentium® 4 процессор.
- 512 Мб ОЗУ.
- 300 Мб дискового пространства.

Рекомендуемые требования

- Intel Pentium 4 процессор с поддержкой технологии Hyper-Threading (HT Technology) или Intel® Xeon® процессор.
- 1 Гб ОЗУ.

Для изучения всех аспектов «реальных» параллельных вычислений желательно использование многоядерных процессоров компании Intel.

1.3.2. Программное обеспечение

- Microsoft Windows XP Professional, Microsoft Windows Server 2003, или Microsoft Windows Vista.
- Intel® C++ Compiler 9.0 for Windows или старше.
- Microsoft Visual C++ 7.1 или старше.
- Microsoft Internet Explorer 6.0 или старше.
- Adobe Reader 6.0 или старше.

1.4. Рекомендации по проведению занятий

При выполнении данной лабораторной работы рекомендуется придерживаться следующей последовательности изучения материала:

1. Изучить, руководствуясь материалом из [1], способы инициализации и завершения работы библиотеки TBB.
2. Рассмотреть, руководствуясь материалом из [1], функциональность библиотеки TBB, связанную с использованием механизма логических задач.
3. Разобрать предложенные варианты решения задачи вычисления быстрого преобразования Фурье.
4. Перейти к выполнению дополнительных заданий, состоящих в самостоятельной разработке и отладке параллельных программ, использующих механизм логических задач библиотеки TBB.

2. Схема вычисления быстрого преобразования Фурье

2.1. Теоретическая справка

В основе преобразования Фурье[1] (ПФ) лежит простая, но исключительно плодотворная идея – почти любую периодическую функцию можно представить суммой отдельных гармонических составляющих (синусоид и косинусоид с различными амплитудами, периодами и, следовательно, частотами).

Неоспоримым достоинством ПФ является его гибкость – преобразование может использоваться как для непрерывных функций, так и для дискретных. В последнем случае оно называется дискретным преобразованием Фурье – ДПФ.

ПФ часто применяется при решении задач, возникающих в теории автоматического регулирования и управления, в теории фильтрации, ДПФ – в задачах цифровой обработки сигналов и других.

Вычисление ДПФ, чаще всего в виде БПФ, реализовано во многих математических библиотеках, например, Intel MKL, Intel IPP. ПФ может быть обобщено на случай многомерных функций.

Непрерывное преобразование Фурье – преобразование, которое применяется к функции $x(t)$, заданной на интервале $(-\infty, +\infty)$. В результате получается функция $y(u)$:

$$y(u) = \int_{-\infty}^{+\infty} x(t) e^{-2\pi i u t} dt .$$

Также существует обратное преобразование, которое позволяет по образу $y(u)$ восстановить исходную функцию $x(t)$:

$$x(t) = \int_{-\infty}^{+\infty} y(u) e^{2\pi i u t} df .$$

Образ $y(u)$ является комплексной функцией вещественного аргумента, также и $x(t)$ может принимать не только вещественные, но и комплексные значения.

С непрерывным преобразованием Фурье удобно работать в теории, но на практике обычно приходится иметь дело с дискретными данными, когда задано не аналитическое выражение преобразуемой функции, а лишь набор ее значений на некоторой сетке. В таком случае принимается допущение, что за пределами этой сетки функция равна нулю, а интеграл аппроксимируется интегральной суммой:

$$y(u) = \int_{-\infty}^{+\infty} x(t) e^{-2\pi i u t} dt = \sum_{k=0}^{n-1} x_k \Delta_k \exp\left(-2\pi i u t_k\right).$$

В случае равномерной сетки формула упрощается. Также на равномерной сетке обычно избавляются от шага, чтобы получить безразмерную формулу:

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

Обратное преобразование в таком случае будет иметь вид:

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

Определенное таким образом дискретное преобразование Фурье сохраняет практически все свойства непрерывного (разумеется, с учетом перехода к дискретному множеству).

---

[1] Жан Батист Жозеф Фурье (1768 – 1830). Родился в г. Осере (Оксер), Франция, в семье портного. Остался круглым сиротой в восьмилетнем возрасте. Прошел обучение в военной школе. В 1796 году возглавил кафедру математического анализа в знаменитой Политехнической школе. Автор труда “Математическая теория тепла”, где впервые была систематически использована идея о представлении функций суммой разноамплитудных, но кратночастотных синусоид и косинусоид.

Представленная выше формула для вычисления преобразования Фурье требует значительных затрат. Трудоемкость такого алгоритма имеет порядок $O(n^2)$. На практике, вместо ДПФ, используют быстрое преобразование Фурье (БПФ). БПФ – это простой алгоритм для эффективного вычисления ДПФ с трудоемкостью $O(n\log n)$. В настоящий момент существует несколько алгоритмов быстрого преобразования Фурье. В данной лабораторной работе рассматривается одна из наиболее популярных реализаций алгоритма, предложенная Кули и Таки (Cooley-Tukey) [4, 10].

2.2. Алгоритм Cooley-Tukey

Идея БПФ состоит в сокращении числа умножений и сложений, выполняемых в исходном ДПФ. Нетрудно заметить, что при вычислении ДПФ приходится многократно считать множители $e^{-\frac{jk}{n}2\pi i}$. Например при всех $j=\overline{0,n-1}$ и $k=\overline{0,n-1}$ таких, что $(j\cdot k)\bmod n = const$ множители $e^{-\frac{jk}{n}2\pi i}$ получаются одинаковыми. Очевидно, что таких j и k достаточно много. Например, $(j\cdot k)\bmod n = 0$ достигается при $j=0$ и любом $k=\overline{0,n-1}$ или при $k=0$ и любом $j=\overline{0,n-1}$. За счет использования этого факта и происходит экономия числа арифметических операций.

Алгоритм Cooley-Tukey разбивает исходное множество точек на два равных подмножества (размер входных данных должен быть степенью двойки). Каждое из подмножеств снова делится на 2 части и т.д. Над каждым из полученных множеств выполняется БПФ. Вычисления можно выполнять независимо, этот факт будет использован при реализации параллельной версии.

Алгоритм Cooley-Tukey состоит из двух шагов:

1. преобразование входного массива данных (*бит-реверсирование*);
2. вычисление БПФ.

2.2.1. Бит-реверсирование

Данный этап заключается в изменении порядка следования исходных данных таким образом, чтобы с ними было удобно работать в дальнейшем. В некоторых алгоритмах БПФ этот этап пропускается и внедряется в вычислительный этап.

Бит-реверсирование – это преобразование двоичного числа путем изменение порядка следования бит в нем на противоположный. Бит-реверсирование применимо только к индексам элементов массива и предназначено для изменения порядка следования этих элементов, при этом значения самих элементов не изменяются.

Рассмотрим алгоритм работы на примере. Пусть исходный массив содержит 16 элементов, тогда преобразование индексов будет происходить следующим образом:

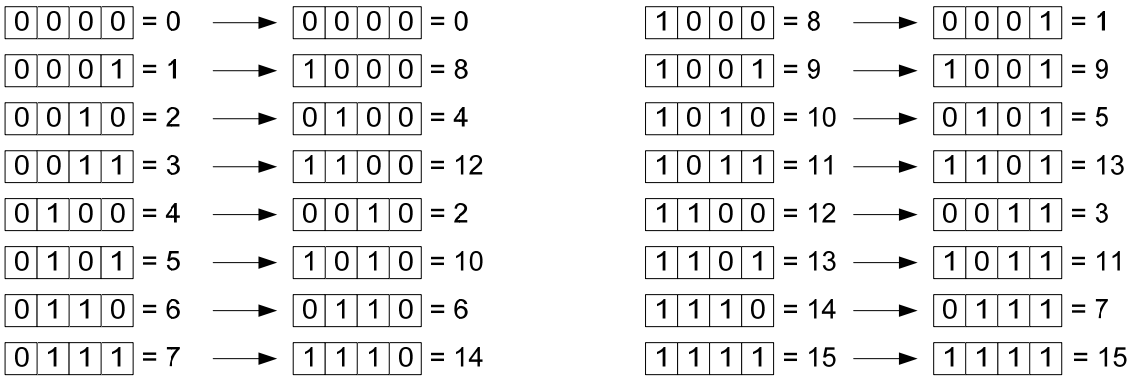

Рис. 1. Пример выполнения бит-реверсирования

Таким образом, исходный массив ($a[0], a[1], ..., a[15]$) после бит-реверсирования будет иметь вид: $a[0], a[8], a[4], a[12], a[2], a[10], a[6], a[14], a[1], a[9], a[5], a[13], a[3], a[11], a[7], a[15]$.

Программный код, выполняющий бит-реверсированиие, представлен ниже.

```
void BitReversing(complex<double> *inputSignal,
```

```
  complex<double> *outputSignal, int size)
{
  int j = 0, i = 0;

  while (i < size)
  {
    if (j > i)
    {
      outputSignal[i] = inputSignal[j];
      outputSignal[j] = inputSignal[i];
    }
    else
      if (j == i)
        outputSignal[i] = inputSignal[i];

    int m = size >> 1;
    while ((m >= 1) && (j >= m))
    {
      j -= m ;
      m = m >> 1;
    }
    j += m;
    i ++;
  }
}
```

2.2.2. Вычисление БПФ

Рассмотрим алгоритм вычисления БПФ на примере. Пусть на вход подается массив из 4-х элементов (x_0, x_1, x_2, x_3). Выполнив бит-реверсирование, получим следующий порядок элементов (x_0, x_2, x_1, x_3). Собственно вычисления проиллюстрируем на рисунке.

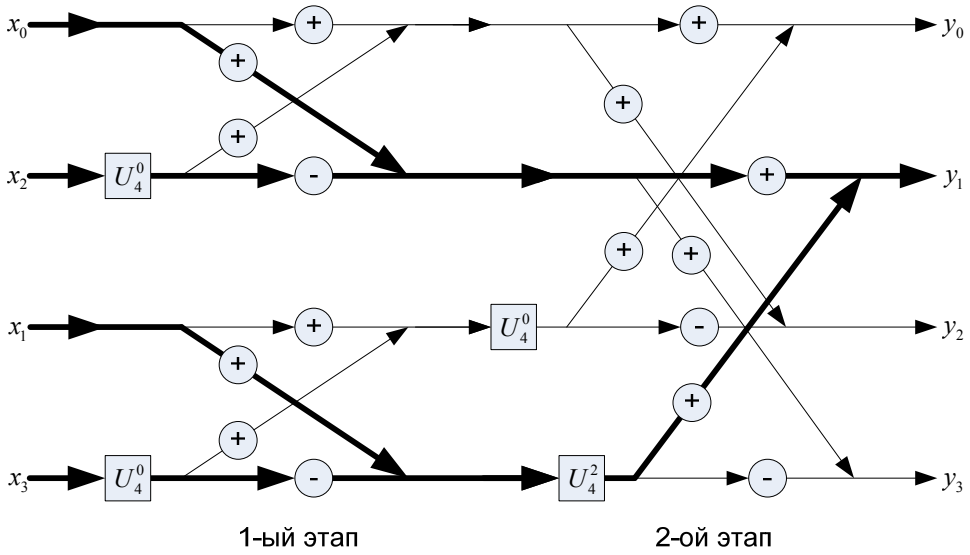

Рис. 2. Пример вычисления БПФ

Поскольку размер входного массива равен 4, вычисления состоят из двух этапов. В общем случае, когда размер входных данных равен степени двойки ($n = 2^m$), число этапов вычисления БПФ равно $\log n = m$. На каждом этапе многократно выполняется одна и та же базовая операция, которую часто называют «бабочкой». Вход «бабочки» представляют собой два числа: a и b. Выход – два других числа: $a + bU$ и $a - bU$. Число U называется *коэффициентом поворота*. На рис. 3 представлено графическое изображение указанной операции, из него читателю должно быть ясно происхождение ее названия.

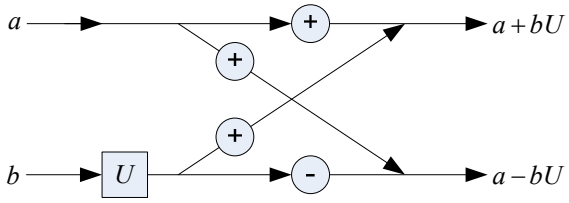

Рис. 3. Базовая операция вычисления БПФ «бабочка»

Итак, на первом этапе вычислений происходит применение «бабочки» с шагом = 1 ко всем элементам, т.е. ко всем парам, состоящим из соседних элементов входного массива. В нашем примере это пары (x_0, x_2) и (x_1, x_3) (рис. 4). Общая формула коэффициента поворота $U_{bSize}^{j} = e^{-\frac{j}{bSize}\pi i}$, где *bSize* – размер шага «бабочки», *j* – номер итерации вычислений над группой элементов, размером 2 * *bSize* ($0 \le j < bSize$). На первом этапе вычислений коэффициент поворота «бабочки» будет равен 1 ($U_1^0 = e^0 = 1$) для всех обрабатываемых пар.

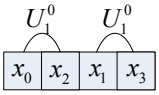

Рис. 4. 1-ый этап вычисления БПФ над массивом из 4-х элементов

На втором этапе вычислений шаг «бабочки» увеличиться в два раза, т.е. будет равен 2. Таким образом «бабочка» будет применяться к следующим парам элементов (x_0, x_1) и (x_2, x_3) (рис. 5). При этом коэффициенты поворота для этих элементов будут различны: $U_2^0 = e^0 = 1$ и $U_2^1 = e^{-\frac{\pi}{2}i} = -i$ соответственно.

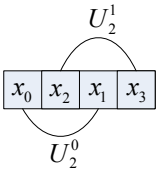

Рис. 5. 2-ой этап вычисления БПФ над массивом из 4-х элементов

Таким образом, для вычисления БПФ над массивом из 4-х элементов потребуется 4 раза выполнить «бабочку». Как видно из рис. 2 (жирные стрелки), на выходе y_1 будут присутствовать все элементы, подаваемые на вход, с разными коэффициентами (за счет разных коэффициентов поворота). Можно убедиться, что полученное значение y_1 полностью совпадает со значением, полученным при вычислении обычного ДПФ ($y_1 = \sum_{j=0}^{3} x_j e^{-\frac{j}{2}\pi i} = x_0 - ix_1 - x_2 + ix_3$). Аналогичным образом обстоит ситуация с остальными выходами.

Рассмотрим общий алгоритм вычисления БПФ. Пусть $n = 2^m$ – размер входных данных, **`signal`** – входной массив комплексных чисел.

1. **`bSize = 1`** (шаг «бабочки»).
2. **`i = 0`**, **`j = 0`**.
3. Применить «бабочку» к элементам **`signal[i * bSize * 2 + j]`** и **`signal[i * bSize * 2 + j + bSize]`** с коэффициентом поворота $U = e^{-\frac{j}{bSize}\pi i}$.
4. Если **`j < bSize - 1`**, то **`j++`**, переход на шаг 3.
5. Если **`i < bSize * n / 2 - 1`**, то **`i++`**, **`j = 0`**, переход на шаг 3.

6. Если **`bSize < n/2`**, то **`bSize = bSize * 2`**, переход на шаг 2, иначе завершение алгоритма.

3. Разработка последовательного приложения

Проведем пошагово реализацию последовательной программы вычисления БПФ. Заметим, что вариант, который будет рассмотрен, разумеется, не является единственным. Будем также считать основной целью корректность работы полученной реализации, а не ее производительность на какой-либо конкретной архитектуре.

Исходными данными для вычисления БПФ является входной сигнал, заданный в виде массива комплексных чисел. Входной сигнал будет задаваться 2-мя способами:

- специальный вид входного сигнала (для оценки корректности реализации алгоритма);
- случайная генерация данных (для проведения вычислительных экспериментов).

Для начала работы нам потребуется простейший проект, на основе которого будет выполняться разработка приложения.

3.1. Задание 1 – Открытие проекта SerialNonrecursiveFFT

Откройте проект **SerialNonrecursiveFFT**, последовательно выполняя следующие шаги:

- запустите приложение **Microsoft Visual Studio 2005**;
- в меню **File** выполните команду **Open→Project/Solution…**;
- в диалоговом окне **Open Project** выберите папку **C:\TBBLabs\SerialNonrecursiveFFT**;
- дважды щелкните на файле **SerialNonrecursiveFFT.sln** или, выбрав файл, выполните команду **Open**.

После открытия проекта в окне **Solution Explorer** дважды щелкните на файле исходного кода **SerialNonrecursiveFFT.cpp**, как это показано на рис. 6. После этих действий программный код, который предстоит в дальнейшем расширить, будет открыт в редакторе кода **Microsoft Visual Studio 2005**.

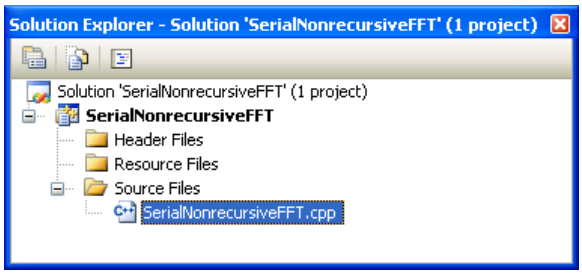

Рис. 6. Открытие файла **SerialNonrecursiveFFT.cpp**

В файле **SerialNonrecursiveFFT.cpp** подключаются необходимые библиотеки, а также содержится начальный вариант основной функции программы – функции **`main`**. Эта заготовка содержит объявление переменных и вывод на печать начального сообщения программы.

Быстрое преобразование Фурье, в общем случае, осуществляется над массивом комплексных чисел. Используем для работы с ними класс **`complex<double>`** библиотеки STL, подключив соответствующий заголовочный файл: **`#include <complex>`**. В функции **`main`** объявлены:

- **`inputSignal`** – входной сигнал, над которым будет выполняться БПФ, массив комплексных чисел **`complex<double>`**;
- **`outputSignal`** – выходной сигнал после применения БПФ, массив комплексных чисел **`complex<double>`**;
- **`size`** – размер входного и выходного сигнала (число элементов в сигнале в процессе преобразования Фурье не меняется), задает число элементов массивов **`inputSignal`** и **`outputSignal`**.

После объявления и инициализации начальными значениями переменных, выводится начальное сообщение. Перед завершением приложения добавлено ожидание ввода с клавиатуры.

```
int main()
{
  complex<double> *inputSignal = NULL;
  complex<double> *outputSignal = NULL;

  int size = 0;

  cout << "Fast Fourier Transform. Serial version." << endl;

  cin.get();

  return 0;
}
```

Теперь можно осуществить первый запуск приложения. Выполните команду **Rebuild Solution** в меню **Build** – эта команда позволяет скомпилировать приложение. Если приложение скомпилировано успешно (в нижней части окна **Microsoft Visual Studio 2005** появилось сообщение "`Rebuild All: 1 succeeded, 0 failed, 0 skipped`"), нажмите клавишу **F5** или выполните команду **Start Debugging** пункта меню **Debug**.

После запуска исполняемого модуля в командной консоли появится сообщение: "`Fast Fourier Transform. Serial version.`", после чего программа будет ожидать нажатия клавиши **Enter**.

3.2. Задание 2 – Ввод и генерация исходных данных

Для задания исходных данных последовательного алгоритма вычисления БПФ реализуем функцию **`ProcessInitialization`**. Эта функция предназначена для определения размера обрабатываемых данных (сигналов), выделения памяти под сигналы: входного **`inputSignal`** и выходного **`outputSignal`**, – а также для задания значений элементов входного сигнала. Таким образом, функция должна иметь следующий интерфейс:

```
// Function for memory allocation and data initialization
void ProcessInitialization(complex<double>* &inputSignal,
  complex<double>* &outputSignal, int &size);
```

На первом этапе необходимо определить размеры объектов (задать значение переменной **`size`**). Исходя из особенностей схемы вычисления БПФ, значение **`size`** должно быть степенью двойки. Для упрощения реализации будем полагать, что **`size >= 4`**. В тело функции **`ProcessInitialization`** добавьте выделенный фрагмент кода:

```
// Function for memory allocation and data initialization
void ProcessInitialization(complex<double>* &inputSignal,
  complex<double>* &outputSignal, int &size)
{
  // Setting the size of signals
  do
  {
    cout << "Enter the input signal length: ";
    cin >> size;

    if (size < 4)
      cout << "Input signal length should be greater or equal than four"
        << endl;
    else
    {
      int tmpSize = size;

      while (tmpSize != 1)
      {
        if (tmpSize % 2 != 0)
        {
          cout << "Input signal length should be powers of two" << endl;
          size = -1;
          break;
        }
```

```
        tmpSize /= 2;
      }
    }
  }
  while(size < 4);
  cout << "Input signal length = " << size << endl;
}
```

В функцию **main** добавим вызов функции **ProcessInitialization**:

```
int main()
{
  complex<double> *inputSignal = NULL;
  complex<double> *outputSignal = NULL;

  int size = 0;

  cout << "Fast Fourier Transform. Serial version." << endl;

  // Memory allocation and data inialization
  ProcessInitialization(inputSignal, outputSignal, size);

  cin.get();

  return 0;
}
```

Скомпилируйте и запустите приложение. Убедитесь в том, что значение переменной **size** задается корректно.

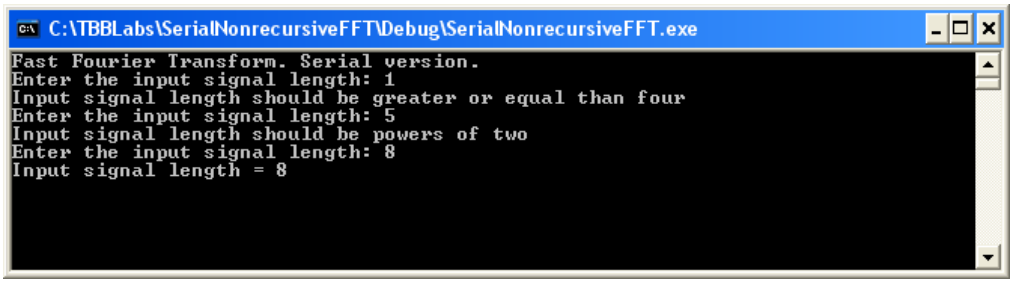

Рис. 7. Задание размера сигналов

Функция инициализации должна также выделять память для хранения объектов (добавьте выделенный код в тело функции **ProcessInitialization**):

```
// Function for memory allocation and data initialization
void ProcessInitialization(complex<double>* &inputSignal,
  complex<double>* &outputSignal, int &size)
{
  // Setting the size of signals
  do
  {
    <...>
  }
  while(size < 4);
  cout << "Input signal length = " << size << endl;

  inputSignal = new complex<double>[size];
  outputSignal = new complex<double>[size];
}
```

Далее необходимо задать значения элементов входного сигнала **inputSignal**. Для выполнения этих действий реализуем еще одну функцию **DummyDataInitialization**. Заполним массив входных данных нулями, кроме одного элемента **mas[size - size / 4] = 1.** Результатом применения БПФ к такому массиву будет массив из комплексных чисел вида $\cos(\frac{\pi}{2}k)+i\sin(\frac{\pi}{2}k)$, $k=\overline{0,size-1}$. Используя

такие начальные данные, легко визуально проверить корректность работы реализованного алгоритма при небольших входных значениях.

Интерфейс и реализация этой функции представлены ниже:

```
// Function for simple initialization of input signal elements
void DummyDataInitialization(complex<double>* mas, int size)
{
  for(int i = 0; i < size; i++)
    mas[i] = 0 ;

  mas[size – size / 4] = 1;
}
```

Вызов функции **DummyDataInitialization** необходимо выполнить после выделения памяти внутри функции **ProcessInitialization**:

```
// Function for memory allocation and data initialization
void ProcessInitialization(complex<double>* &inputSignal,
  complex<double>* &outputSignal, int &size)
{
  // Setting the size of signals
  do
  {
    <...>
  }
  while(size < 4);
  cout << "Input signal length = " << size << endl;

  inputSignal = new complex<double>[size];
  outputSignal = new complex<double>[size];

  // Initialization of input signal elements
  DummyDataInitialization(inputSignal, size);
}
```

3.3. Задание 3 – Завершение процесса вычислений

Перед собственно вычислением БПФ сначала разработаем функцию для корректного завершения процесса вычислений. Для этого необходимо освободить память, выделенную динамически в процессе выполнения программы. Реализуем соответствующую функцию **ProcessTermination**. Память выделялась для хранения входного **inputSignal** и выходного **outputSignal** сигналов. Следовательно, эти объекты необходимо передать в функцию **ProcessTermination** в качестве аргументов:

```
// Function for computational process termination
void ProcessTermination(complex<double>* &inputSignal,
  complex<double>* &outputSignal)
{
  delete [] inputSignal;
  inputSignal = NULL;
  delete [] outputSignal;
  outputSignal = NULL;
}
```

Вызов функции **ProcessTermination** необходимо добавить в самый конец главной функции **main**:

```
int main()
{
  complex<double> *inputSignal = NULL;
  complex<double> *outputSignal = NULL;

  int size = 0;

  cout << "Fast Fourier Transform. Serial version." << endl;

  // Memory allocation and data inialization
```

```
  ProcessInitialization(inputSignal, outputSignal, size);

  // Computational process termination
  ProcessTermination(inputSignal, outputSignal);

  cin.get();

  return 0;
}
```

Скомпилируйте и запустите приложение. Убедитесь в том, что оно выполняется корректно.

3.4. Задание 4 – Реализация БПФ

Выполним теперь разработку основной вычислительной части программы. Для вычисления БПФ реализуем функцию **SerialFFT**, которая принимает на вход исходный сигнал **inputSignal** размера **Size**, а также указатель на место в памяти, где должен быть сохранен выходной сигнал **outputSignal**.

В соответствии с алгоритмом вычисления БПФ, изложенным в пункте 2.2, код этой функции должен быть следующий:

```
// FFT computation
void SerialFFT(complex<double> *inputSignal, complex<double> *outputSignal,
  int size)
{
  BitReversing(inputSignal, outputSignal, size);
  SerialFFTCalculation(outputSignal, size);
}
```

Здесь функция **BitReversing** реализует первый шаг вычисления БПФ: бит-риверсирование (ее код был приведен в пункте 2.2.1), а функция **SerialFFTCalculation** – второй шаг, собственно вычисление.

В пункте 2.2.2 был, в общем виде, рассмотрен алгоритм второго шага в вычислении БПФ. Приведем теперь его реализацию.

```
// FFT computation
void SerialFFTCalculation(complex<double> *signal, int size)
{
  int m = 0;
  for (int tmp_size = size; tmp_size > 1; tmp_size /= 2, m++) //size = 2^m
    ;
  for (int p = 0; p < m; p++)
  {
    int butterflyOffset = 1 << (p + 1);
    int butterflySize = butterflyOffset >> 1;
    double coeff = PI / butterflySize;

    for (int i = 0; i < size / butterflyOffset; i++)
      for (int j = 0; j < butterflySize; j++)
        Butterfly(signal, complex<double>(cos(-j * coeff),
          sin(-j * coeff)), j + i * butterflyOffset, butterflySize);
  }
}
```

Здесь функция **Butterfly** реализует вычисление базовой операции алгоритма – «бабочки». На рис. 8 представлена схема вычисления «бабочки» с произвольным шагом (**butterflySize**) и произвольным начальным элементом (**offset**).

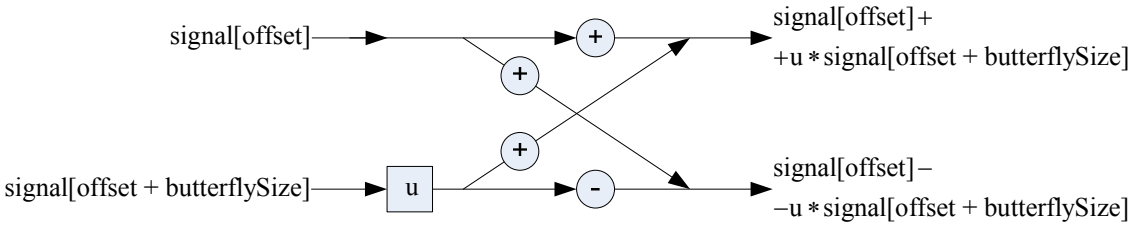

Рис. 8. Реализация «бабочки» в общем случае

Здесь **u** – комплексный коэффициент поворота, **signal** – массив комплексных чисел, над элементами которого вычисляется «бабочка». Таким образом, реализация функции **Butterfly** будет выглядеть так:

```
__inline void Butterfly(complex<double> *signal, complex<double> &u,
  int offset, int butterflySize)
{
  complex<double> tem = signal[offset +  butterflySize] * u;

  signal[offset + butterflySize] = signal[offset] - tem;
  signal[offset] += tem;
}
```

Выполним вызов функции вычисления БПФ из основной программы.

```
int main()
{
  complex<double> *inputSignal = NULL;
  complex<double> *outputSignal = NULL;

  int size = 0;

  cout << "Fast Fourier Transform. Serial version." << endl;

  // Memory allocation and data inialization
  ProcessInitialization(inputSignal, outputSignal, size);

  // FFT computation
  SerialFFT(inputSignal, outputSignal, size);

  // Computational process termination
  ProcessTermination(inputSignal, outputSignal);

  cin.get();

  return 0;
}
```

Выполните сборку и запуск приложения. Убедитесь в отсутствии ошибок и правильной работе приложения. Вывод на консоль не изменился по сравнению с предыдущей сборкой, поэтому содержимое консоли после выполнения приложения не изменится. Не указывайте большой размер входного сигнала, т.к. это может привести к долгой работе приложения.

3.5. Задание 5 – Проверка корректности

Необходимо определить правильность работы приложения. Для этого подадим на вход вычислительному алгоритму (**SerialFFT**) данные специального вида (сгенерированные с помощью функции **DummyDataInitialization**). Например, если размер входного сигнала равен 8, то после выполнения БПФ над сгенерированным массивом получим результат в виде массива из восьми комплексных чисел вида $\cos(\frac{\pi}{2}k)+i\sin(\frac{\pi}{2}k)$, $k=\overline{0,7}$.

Для отображения результатов работы алгоритма реализуем функцию вывода на экран значений массива комплексных чисел (**PrintSignal**).

```
void PrintSignal(complex<double> *signal, int size)
{
  cout << "Result signal" << endl;
  for (int i = 0; i < size; i++)
    cout << signal[i] << endl;
}
```

Далее, выведем значения результирующего сигнала и сравним их со значениями, посчитанными вручную.

```
int main()
{
  complex<double> *inputSignal = NULL;
  complex<double> *outputSignal = NULL;

  int size = 0;

  cout << "Fast Fourier Transform. Serial version." << endl;

  // Memory allocation and data inialization
  ProcessInitialization(inputSignal, outputSignal, size);

  // FFT computation
  SerialFFT(inputSignal, outputSignal, size);

  // Result signal output
  PrintSignal(outputSignal, size);

  // Computational process termination
  ProcessTermination(inputSignal, outputSignal);

  cin.get();

  return 0;
}
```

Выполните сборку и запуск приложения. Проведите эксперименты с различными значениями размера входного сигнала. Убедитесь, что выводимые значения совпадают со значениями, посчитанными вручную. При вычислении БПФ могут возникнуть погрешности машинной арифметики, поэтому значения, посчитанные компьютером, могут не совпадать в точности с результатами, посчитанными теоретически.

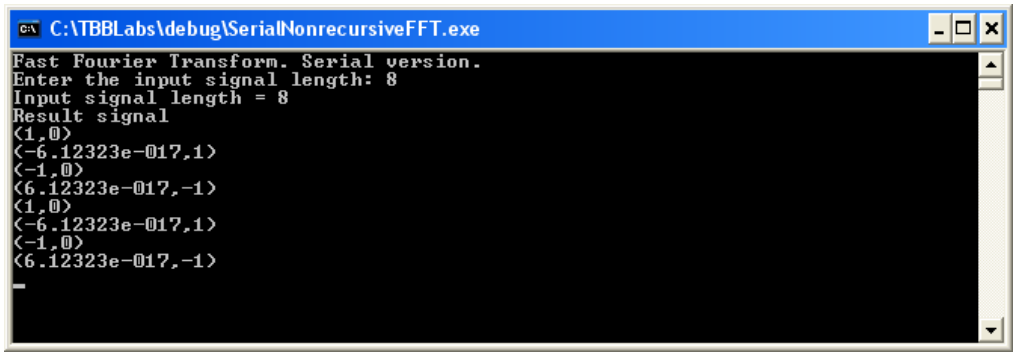

Рис. 9. Проверка корректности работы последовательной версии

3.6. Задание 6 – Проведение вычислительных экспериментов

После реализации параллельной версии алгоритма нам потребуется оценивать ее ускорение. Для этого сейчас необходимо провести вычислительные эксперименты и замерить времена работы последовательной версии. Анализировать время выполнения последовательной реализации разумно для достаточно больших объектов. Задавать элементы больших матриц и векторов будем при помощи датчика случайных чисел. Для этого реализуем еще одну функцию задания элементов **`RandomDataInitialization`** (датчик случайных чисел инициализируется текущим значением времени):

```
// Function for random initialization of objects' elements
void RandomDataInitialization(complex<double>* mas, int size)
{
  srand(unsigned(clock()));
  for(int i = 0; i < size; i++)
    mas[i] = complex<double>(rand() / 1000.0, rand() / 1000.0) ;
}
```

Заметим, что в отладочных целях следует инициализировать датчик случайных чисел при помощи функции **srand()** одним и тем же числом, что обеспечит нам одинаковые последовательности случайных чисел и позволит обнаружить ошибки, если они есть.

Будем вызывать эту функцию вместо ранее разработанной функции **DummyDataInitialization**, которая генерировала данные так, чтобы можно было легко проверить правильность работы алгоритма:

```
// Function for memory allocation and data initialization
void ProcessInitialization(complex<double>* &inputSignal,
  complex<double>* &outputSignal, int &size)
{
  // Setting the size of signals
  do
  {
    <...>
  }
  while(size < 4);
  cout << "Input signal length = " << size << endl;

  inputSignal = new complex<double>[size];
  outputSignal = new complex<double>[size];

  // Initialization of input signal elements - tests
  //DummyDataInitialization(inputSignal, size);

  // Initialization of input signal elements – computational experiments
  RandomDataInitialization(inputSignal, size);
}
```

Скомпилируйте и запустите приложение. Убедитесь в том, что данные генерируются случайным образом.

Замеры времени вычислений будем проводить при помощи класса **tick_count**. Чтобы оценки времени были корректны, будем проводить несколько экспериментов и выбирать наименьшее из времен, аналогично тому, как это делалось в лабораторной работе «Распараллеливание циклов с использованием библиотеки Intel Threading Building Blocks на примере задачи матрично-векторного умножения».

```
int main()
{
  complex<double> *inputSignal = NULL;
  complex<double> *outputSignal = NULL;

  int size = 0;

  const int repeatCount = 16;
  tick_count startTime;
  double duration;
  double minDuration = DBL_MAX;

  cout << "Fast Fourier Transform. Serial version." << endl;

  // Memory allocation and data inialization
  ProcessInitialization(inputSignal, outputSignal, size);

  for(int i = 0; i < repeatCount; i++)
  {
    startTime = tick_count::now();
    // FFT computation
    SerialFFT(inputSignal, outputSignal, size);
    duration  = (tick_count::now() - startTime).seconds();

    if(duration < minDuration)
      minDuration = duration;
  }
```

```
  cout << setprecision(6);
  cout << "Execution time is " << minDuration << " s. " << endl;

  // Result signal output
  PrintSignal(outputSignal, size);

  // Computational process termination
  ProcessTermination(inputSignal, outputSignal);

  cin.get();

  return 0;
}
```

Скомпилируйте и запустите приложение. Для проведения вычислительных экспериментов с большими входными сигналами отключите печать векторов (закомментируйте соответствующие строки кода).

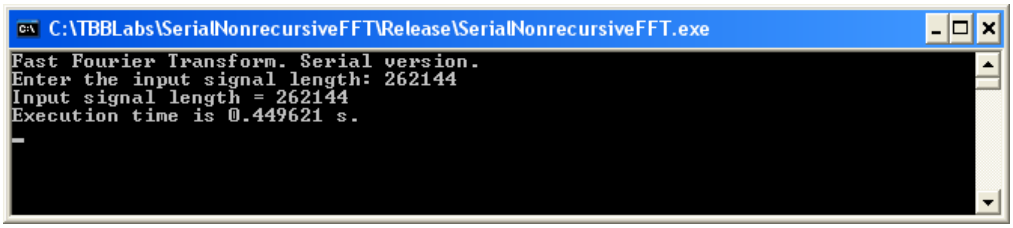

Рис. 10. Замеры времени работы последовательной версии

3.7. Задание 7 – Оценка эффективности

Выполните сборку приложения в конфигурации **Release**. Проведите вычислительные эксперименты, результаты занесите в таблицу.

Таблица 1. Результаты экспериментов вычисления БПФ

Номер теста	Размер входного сигнала	Время работы последовательного приложения (сек.)
1	32768	
2	65536	
3	131072	
4	262144	
5	524288	

4. Разработка параллельного приложения

В данном разделе будет рассмотрен процесс разработки параллельного приложения с использованием библиотеки TBB на основе последовательного приложения:

- создание параллельной версии с использованием алгоритма **tbb::parallel_for**;
- создание параллельной версии с использованием логических задач (класса **tbb::task**).

4.1. Задание 1 – Открытие проекта ParallelNonrecursiveFFT

За основу разработки параллельного приложения возьмем последовательную версию. Функции инициализации, завершения, генерации данных, вывода на экран результирующего сигнала, бит-реверсирования и вычисления «бабочки» останутся без изменений.

Откройте проект **ParallelNonrecursiveFFT**, последовательно выполняя следующие шаги:

- запустите приложение **Microsoft Visual Studio 2005**;
- в меню **File** выполните команду **Open→Project/Solution…**,

- в диалоговом окне **Open Project** выберите папку **C:\TBBLab\ParallelNonrecursiveFFT**,
- дважды щелкните на файле **ParallelNonrecursiveFFT.sln** или, выбрав файл, выполните команду **Open**.

После открытия проекта в окне **Solution Explorer** дважды щелкните на файле исходного кода **ParallelNonrecursiveFFT.cpp**, как это показано на рис. 11. После этих действий программный код, который предстоит в дальнейшем расширить, будет открыт в редакторе кода **Microsoft Visual Studio 2005**.

Рис. 11. Структура проекта **ParallelNonrecursiveFFT**

Соберите исполняемый модуль. Убедитесь, что компиляция прошла успешно. После запуска исполняемого модуля в командной консоли появится сообщение: `"Fast Fourier Transform. Parallel version.",` после чего программа будет ожидать нажатия клавиши **Enter**.

4.2. Задание 2 – Настройка проекта для использования TBB

Первое, что нам предстоит изменить, это подсказать **Microsoft Visual Studio 2005** место расположения заголовочных файлов библиотеки **TBB**. Эта задача может быть решена двумя способами:

- Запустить файл **C:\Program Files\Intel\<TBB Directory>\<arch>\vc8\bin\tbbvars.bat**, где <arch> принимает значения ia32 (Intel® IA-32 processors) или em64t (Intel® EM64T processors).
- В меню **Tools** выбрать пункт **Options**. В открывшемся окне выбрать пункт **Projects and Solutions\VC++ Directories**. В выпадающем списке **Show directories for** выбрать пункт **Include files**. Нажать левой кнопкой мыши на изображении папки и указать путь к папке **Include** библиотеки **TBB**: **C:\Program Files\Intel\<TBB Directory>\include**. Нажать **OK**.

Следующий шаг – указание пути к статической библиотеке **TBB**, с которой будет собираться наше приложение. Таких библиотек две: **tbb_debug.lib** и **tbb.lib**. Первая библиотека выполняет различные проверки во время выполнения приложения и полностью поддерживается профилировщиком **Intel Thread Profiler**, предназначена для компиляции и сборки отладочных версий программ. Вторая библиотека имеет гораздо более эффективную реализацию функций и методов и предназначена для компиляции и сборки финальных версий. Разумеется, имеет смысл подключать одну из двух библиотек в зависимости от ситуации (отладка, финальная сборка).

Для подключения библиотеки необходимо выполнить следующую последовательность действий:

- В меню **Tools** выбрать пункт **Options**. В открывшемся окне выбрать **Projects and Solutions\VC++ Directories**. В выпадающем списке **Show directories for** выбрать пункт **Library files**. Нажать левой кнопкой мыши на изображении папки и указать путь к папке **lib** библиотеки **TBB**: **C:\Program Files\Intel\<TBB Directory>\vc8\lib**. Нажать **OK**.
- В окне **Solution Explorer** нажать правой кнопкой мыши на названии проекта (**ParallelNonrecursiveFFT**) и выбрать пункт **Properties**.
- Выбрать пункт **Linker\Input** и в поле **Additional Dependencies** ввести название библиотеки: **tbb_debug.lib** (для **Debug** сборки), или **tbb.lib** (для **Release** сборки).

4.3. Задание 3 – Инициализация библиотеки TBB

Аналогично тому, как это делалось в лабораторной работе «Распараллеливание циклов с использованием библиотеки Intel Threading Building Blocks на примере задачи матрично-векторного умножения», для начала работы с библиотекой TBB необходимо создать хотя бы один экземпляр класса

task_scheduler_init. Создадим объект этого класса без параметров (число потоков будет определено автоматически).

```
#include "tbb/task_scheduler_init.h"
using namespace tbb;
<...>

int main()
{
  complex<double> *inputSignal = NULL;
  complex<double> *outputSignal = NULL;

  int size = 0;

  task_scheduler_init init;

  cout << "Fast Fourier Transform. Parallel version." << endl;

  // Memory allocation and data inialization
  ProcessInitialization(inputSignal, outputSignal, size);

  // Result signal output
  PrintSignal(outputSignal, size);

  // Computational process termination
  ProcessTermination(inputSignal, outputSignal);

  cin.get();

  return 0;
}
```

4.4. Задание 4 – Разработка параллельной версии с использованием алгоритма tbb::parallel_for

Выполнив эксперименты, нетрудно определить, что бит-реверсирование занимает мало времени по сравнению с остальными вычислениями, поэтому распараллеливать имеет смысл функцию **SerialFFTCalculation**.

```
// FFT computation
void SerialFFTCalculation(complex<double> *signal, int size)
{
  <...>
  for (int p = 0; p < m; p++)
  {
    <...>

    for (int i = 0; i < size / butterflyOffset; i++)
      for (int j = 0; j < butterflySize; j++)
        Butterfly(signal, complex<double>(cos(-j * coeff),
          sin(-j * coeff)), j + i * butterflyOffset, butterflySize);
  }
}
```

Выделенный цикл, в приведенном выше коде, не имеет зависимостей по данным. Каждая итерация этого цикла может быть выполнена независимо, поэтому его можно распараллелить. На основе этого цикла реализуем функтор. Полями функтора будут все общие данные, использующиеся в вычислениях: **signal**, **butterflySize**, **coeff**. Поскольку **butterflyOffset = 2 * butterflySize**, переменную **butterflyOffset** не будем делать общей. Конструктор содержит только два аргумента, т.к. переменную **coeff** можно считать программно в теле конструктора.

```
class ButterflyCalculator
{
  complex<double> *const signal;
```

```
  int const butterflySize;
  double coeff;

public:
   void operator()( const blocked_range<int>& r ) const
   {
     int begin = r.begin(),
         end = r.end(),
         butterflyOffset = 2 * butterflySize;

     for (int i = begin; i != end; i++)
       for (int j = 0; j < butterflySize; j++)
         Butterfly(signal, complex<double>(cos(-j * coeff),
           sin(-j * coeff)), j + i * butterflyOffset, butterflySize);
  }

  ButterflyCalculator(complex<double> *tsignal, int tbutterflySize):
    signal(tsignal), butterflySize(tbutterflySize)
  {
    coeff = PI / butterflySize;
  }
};
```

Функтор содержит поля, в которых хранятся значения переменных, участвующих в вычислениях в методе **operator()**. Инициализация этих полей осуществляется с помощью конструктора. Конструктор копирования и деструктор, создаваемые компилятором по умолчанию, в данном случае корректны, поэтому их реализация в явном виде не приводится. Значения начальной и конечной итераций (**r.begin()**, **r.end()**) считаются вне тела цикла. Это сделано для того, чтобы эти функции не вызывались на каждой итерации, т. к. это может привести к замедлению работы приложения.

Реализуем функции параллельного вычисления БПФ при помощи функции **parallel_for** из библиотеки TBB:

```
// Parallel FFT computation
void ParallelFFTCalculation(complex<double> *signal, int size,
  int grainsize)
{
  int m = 0;
  for(int tmp_size = size; tmp_size > 1; tmp_size /= 2, m++) //size = 2^m
    ;

  for (int p = 0; p < m; p++)
  {
    int butterflyOffset = 1 << (p + 1);
    int butterflySize = butterflyOffset >> 1;
    parallel_for(blocked_range<int>(0, size / butterflyOffset, grainsize),
      ButterflyCalculator(signal, butterflySize));
  }
}

void ParallelFFT(complex<double> *inputSignal,
  complex<double> *outputSignal, int size, int grainsize)
{
  BitReversing(inputSignal, outputSignal, size);
  ParallelFFTCalculation(outputSignal, size, grainsize);
}
```

Параметр **grainsize** у разработанной функции предусмотрен для будущих экспериментов с размером порции итераций.

Добавим вызов **ParallelFFT** в функцию **main**, также добавим код, необходимый для проведения экспериментов с различными значениями **grainsize**.

```
int main()
{
  complex<double> *inputSignal = NULL;
```

```
  complex<double> *outputSignal = NULL;

  int size = 0;

  task_scheduler_init init;

  cout << "Fast Fourier Transform. Parallel version." << endl;

  // Memory allocation and data inialization
  ProcessInitialization(inputSignal, outputSignal, size);

  const int testCount = 17;
  const int repeatCount = 16;
  tick_count startTime;
  double duration;

  cout << setprecision(6);
  for(int j = 0; j < testCount; j++)
  {
    int grainsize = size >> j;
    if (grainsize < 1)
      break;

    double minDuration = DBL_MAX;

    for(int i = 0; i < repeatCount; i++)
    {
      startTime = tick_count::now();
      ParallelFFT(inputSignal, outputSignal, size, grainsize);
      duration = (tick_count::now() - startTime).seconds();

      if (duration < minDuration)
        minDuration = duration;
    }

    cout << "Grain size is " << grainsize << ", execution time is ";
    cout << minDuration << " s. " << endl;
  }

  // Result signal output
  //PrintSignal(outputSignal, size);

  // Computational process termination
  ProcessTermination(inputSignal, outputSignal);

  cin.get();

  return 0;
}
```

Соберите исполняемый модуль. Убедитесь, что компиляция прошла успешно. После завершения работы приложения в командной консоли должно быть следующее[2]:

[2] Времена выполнения могут отличаться от указанных на рисунке.

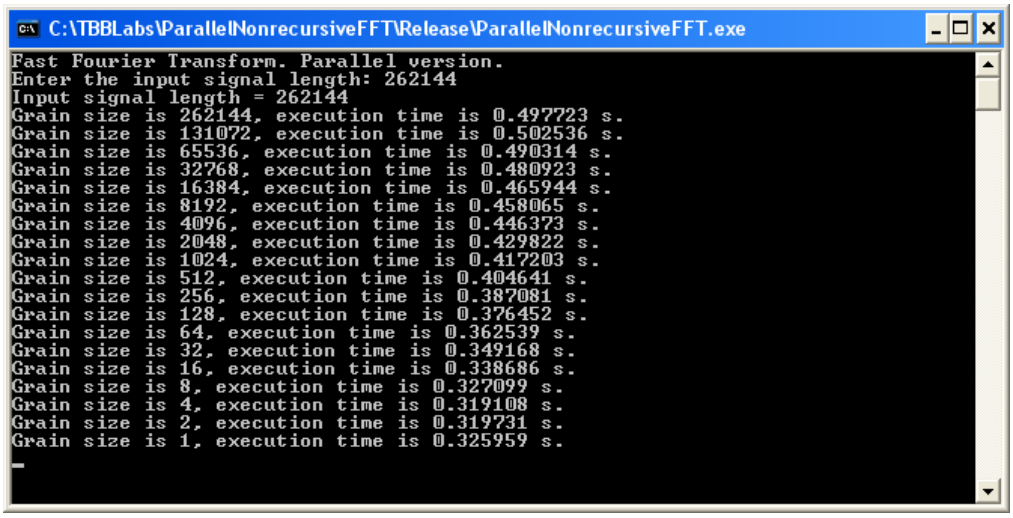

Рис. 12. Замеры времени параллельной версии при различных значениях **`grainsize`**

Выполните сборку приложения в конфигурации **Release**. Проведите эксперименты, задавая разные значения параметра grainsize, чтобы определить минимальное время, и заполните таблицу 2.

Таблица 2. Результаты экспериментов вычисления БПФ

Номер теста	Размер входного сигнала	Минимальное время работы параллельного приложения (сек.)	Выбранное значение **`grainsize`**
1	32768		
2	65536		
3	131072		
4	262144		
5	524288		

Проведя эксперименты и заполнив таблицу, можно обнаружить, что для каждого размера входного сигнала оптимальные значения **`grainsize`** очень малы. Это связано с тем, что в функции **`ParallelFFTCalculation`** итерационное пространство алгоритма **`parallel_for`** меняется в зависимости от номера итерации внешнего цикла. Таким образом, объем вычислений на каждой итерации экспоненциально уменьшается, а **`grainsize`** остается постоянным. Это означает, что с некоторой итерации внешнего цикла размер итерационного пространства станет меньше или равен, чем значение **`grainsize`**, поэтому приложение, начиная с этого шага, будет работать последовательно. Оптимальное значения **`grainsize`** с этой точки зрения равно 1, но оно не будет оптимальным с точки зрения производительности. Поэтому **`grainsize`** нужно пересчитывать на каждой итерации внешнего цикла.

```
// Parallel FFT computation
void ParallelFFTCalculation(complex<double> *signal, int size,
  int grainsize)
{
  int m = 0;
  for(int tmp_size = size; tmp_size > 1; tmp_size /= 2, m++) //size = 2^m
    ;

  for (int p = 0; p < m; p++)
  {
    int butterflyOffset = 1 << (p + 1);
    int butterflySize = butterflyOffset >> 1 ;

    int grain = grainsize / butterflyOffset;
    if (grain < 1)
      grain = 1;

    parallel_for(blocked_range<int>(0, size/butterflyOffset, grain),
      ButterflyCalculator(signal, butterflySize));
```

```
  }
}
```

Выполните сборку приложения в конфигурации **Release**. Проведите эксперименты и заполните таблицу 3.

Таблица 3. Результаты экспериментов вычисления БПФ

Номер теста	Размер входного сигнала	Минимальное время работы параллельного приложения (сек.)	Выбранное значение **grainsize**
1	32768		
2	65536		
3	131072		
4	262144		
5	524288		

4.5. Задание 5 – Разработка параллельной версии с использование логических задач

До сих пор мы рассматривали нерекурсивную реализацию вычисления БПФ. Основным соображением при этом было то, что рекурсия довольно плохо поддается распараллеливанию, например, реализовать параллельную рекурсивную программу на OpenMP – задача совершенно нетривиальная. Однако нетрудно заметить, что сам алгоритм вычисления БПФ целиком построен на рекурсии и, конечно, обычно именно через нее и реализуется. Вот как будет выглядеть типичная его реализация:

```
// FFT computation
void SerialFFTCalculation(complex<double> *const signal, int size)
{
  if (size == 1)
    return;

  double const coeff = 2.0 * PI / size;

  SerialFFTCalculation(signal, size / 2);
  SerialFFTCalculation(&(signal[size / 2]), size / 2);

  for (int j = 0; j < size / 2; j++)
    Butterfly(signal, complex<double>(cos(-j * coeff), sin(-j * coeff)),
      j , size / 2);
}
```

Как видим, код существенно упростился по сравнению с нерекурсивной версией. Вычисление БПФ от входного сигнала есть вычисление БПФ от каждой из его половин и т.д.

Библиотека TBB содержит средства, позволяющие эффективно, а главное достаточно просто распараллеливать рекурсивные алгоритмы. Это средство – *логические задачи*. Подробное описание логических задач и их использования приведено в документе «Библиотека Intel Threading Building Blocks – краткое описание». Здесь же мы отметим минимально-необходимые моменты.

Логическая задача представлена в TBB в виде класса **tbb::task.** Он является базовым при реализации задач, т.е. этот класс должен быть унаследован всеми пользовательскими задачами.

Класс **tbb::task** содержит виртуальный метод **execute**, в котором и происходят все вычисления. Его прототип:

```
virtual task* execute();
```

В этом методе производятся необходимые вычисления, и после этого возвращается указатель на следующую задачу, которую необходимо выполнить. Если возвращается **NULL**, то вычисления прекращаются.

Создание задачи осуществляется с помощью оператора **new**. Некоторые его виды представлены ниже:

- **new(task::allocate_root()) T** – создание «главной» задачи типа **T**.

- **`new(this.allocate_child()) T`** – создание подзадачи типа **`T`**.

Может потребоваться синхронизировать выполнение задач между собой. Для этих целей существует метод **`void wait_for_all()`**. Задача, для которой вызван этот метод будет ожидать завершения всех своих подзадач. Перед тем как вызвать этот метод необходимо с помощью метода **`void set_ref_count(int count)`** указать число подзадач для данной задачи + 1 (**`set_ref_count(n + 1)`**, где **`n`** – число подзадач для данной задачи).

После создания задач их необходимо запустить. Для этого существуют следующие методы:

- **`void spawn(task& child)`** – помещает задачу **`child`** в пул готовых к выполнению. Не блокирующая функция.
- **`void spawn_and_wait_for_all(task& child)`** – помещает задачу **`child`** в пул готовых к выполнению и ожидает завершения всех подзадач. Алгоритм работы аналогичен последовательному вызову методов **`spawn`** и **`wait_for_all`**.
- **`static void spawn_root_and_wait(task& root)`** – запускает задачу **`root`** на выполнение. Память для этой задачи должна быть выделена с помощью **`task::allocate_root()`**.

Реализуем класс логической задачи для вычисления БПФ. Как и в случае функтора, полями класса будут все общие данные, использующиеся в вычислениях: **`signal`**, **`butterflySize`**, **`coeff.`**

Вычисления последовательной версии осуществляются рекурсивно. При этом на каждом шаге происходит разбиение задачи на две равные подзадачи вычисления БПФ. Используем этот факт. Вместо рекурсивного вызова будем создавать две подзадачи, и выполнять их параллельно. При этом после завершения этих задач необходимо выполнить один этап БПФ последовательно. В том случае, когда размер подзадач станет не велик, будем вычислять их без дальнейшего подразбиения на задачи (последовательно).

```
class FFTTask: public task
{
  complex<double> *const signal;
  int const size;
  double coeff;

public:
  FFTTask(complex<double> *const tSignal, int tSize):
    signal(tSignal), size(tSize)
  {
    coeff = 2.0 * PI / size;
  }

  task* execute()
  {
    if (size < 1024)
      SerialFFTCalculation(signal, size);
    else
    {
      FFTTask& a = *new (allocate_child()) FFTTask(signal, size / 2);
      FFTTask& b = *new (allocate_child()) FFTTask(&(signal[size / 2]),
        size/2);

      set_ref_count(3);

      spawn(b);
      spawn_and_wait_for_all(a);

      for (int j = 0; j < size / 2; j++)
        Butterfly(signal, complex<double>(cos(-j * coeff),
          sin(-j * coeff)), j , size / 2);
    }

    return NULL;
  }
};
```

Как видим, рекурсия естественным образом преобразовалась в создание подзадач для половинок пришедшего в текущую задачу входного сигнала.

Создание и запуск «главной» задачи поместим в функцию **ParallelFFTCalculation.**

```
// Parallel FFT computation
void ParallelFFTCalculation(complex<double> *signal, int size)
{
  FFTTask& a = *new(task::allocate_root()) FFTTask(signal, size);
  task::spawn_root_and_wait(a);
}
```

Отметим, что код функции **ParallelFFT** не изменится по сравнению с предыдущей реализацией.

Осталось внести необходимые изменения в функцию **main**.

```
int main()
{
  complex<double> *inputSignal = NULL;
  complex<double> *outputSignal = NULL;

  int size = 0;

  task_scheduler_init init;

  cout << "Fast Fourier Transform. Parallel version." << endl;

  // Memory allocation and data inialization
  ProcessInitialization(inputSignal, outputSignal, size);

  const int repeatCount = 16;
  tick_count startTime;
  double duration;
  double minDuration = DBL_MAX;

  for (int i = 0; i < repeatCount; i++)
  {
    startTime = tick_count::now();
    ParallelFFT(inputSignal, outputSignal, size);
    duration  = (tick_count::now() - startTime).seconds();

    if (duration < minDuration)
      minDuration = duration;
  }

  cout << setprecision(6);
  cout << "Execution time is " << minDuration << " s. " << endl;

  // Result signal output
  //PrintSignal(outputSignal, size);

  // Computational process termination
  ProcessTermination(inputSignal, outputSignal);

  cin.get();

  return 0;
}
```

4.6. Задание 6 – Проведение вычислительных экспериментов

Выполните сборку приложения в конфигурации **Release**. Проведите вычислительные эксперименты с тремя параллельными версиями:

- с простым подбором значения **grainsize** при использовании функции **parallel_for**;

- с адаптивным подбором значения **`grainsize`** при использовании функции **`parallel_for`**;
- с использованием логических задач.

Заполните таблицу 4. На основе значений, записанных в таблицу 4, посчитайте ускорения и заполните таблицу 5. Убедитесь в том, что наилучшие результаты показывает версия, реализованная на основе механизма логических задач библиотеки TBB.

Таблица 4. Результаты вычислительных экспериментов для вычисления БПФ

Номер теста	Размер входного сигнала	Время работы последовательного приложения (сек.)	Время работы параллельного приложения 1 (сек.)	Время работы параллельного приложения 2 (сек.)	Время работы параллельного приложения 3 (сек.)
1	32768				
2	65536				
3	131072				
4	262144				
5	524288				

Таблица 5. Ускорение вычислений, получаемое для параллельного вычисления БПФ

Номер теста	Размер входного сигнала	Ускорение параллельного алгоритма 1	Ускорение параллельного алгоритма 2	Ускорение параллельного алгоритма 3
1	32768			
2	65536			
3	131072			
4	262144			
5	524288			

5. Контрольные вопросы

- Как собрать приложение, использующее библиотеку TBB?
- Как инициализировать библиотеку TBB?
- Как завершить работу библиотеки TBB?
- Как инициализировать библиотеку TBB с определенным числом программных потоков?
- Как оценить полученные результаты ускорения приложения?
- В каких случаях нужно программировать с использованием логических задач?
- Что такое функтор?
- Каков смысл параметра **`grainsize`**?
- Как работает конструкция **`parallel_for`** в библиотеке TBB?
- Что такое логическая задача в библиотеке TBB?
- В чем состоят особенности в разработке параллельных версий алгоритмов с использованием логических задач?

6. Задания для самостоятельной работы

- Реализуйте параллельную версию бит-реверсированния. Оцените вклад в ускорение, который внесет такая реализация.
- Параллельная версия, разработанная с использованием логических задач, реализована таким образом, что последняя итерация вычислений выполняется последовательно. Реализуйте вычисление БПФ так, чтобы последняя итерация тоже выполнялась параллельно.

7. Литература

7.1. Использованные источники информации

1. Сиднев А.А., Сысоев А.В., Мееров И.Б. Библиотека Intel Threading Building Blocks – краткое описание // Материалы образовательного комплекса «Технологии разработки параллельных программ». Нижний Новгород, 2007.
2. Intel® Threading Building Blocks. Reference Manual. Version 1.6. – Intel Corporation, 2007.
3. Intel® Threading Building Blocks. Tutorial. Version 1.6. – Intel Corporation, 2007.
4. Гонсалес Р., Вудс Р. Цифровая обработка сигналов. – М.: Техносфера, 2005.

7.2. Рекомендуемая литература

5. Andrews, G. R. (2000). Foundations of Multithreaded, Parallel, and Distributed Programming.. – Reading, MA: Addison-Wesley (русский перевод Эндрюс Г. Р. Основы многопоточного, параллельного и распределенного программирования. – М.: Издательский дом «Вильямс», 2003).
6. Quinn, M. J. (2004). Parallel Programming in C with MPI and OpenMP. – New York, NY: McGraw-Hill.
7. Воеводин В.В., Воеводин Вл.В. Параллельные вычисления. – СПб.: БХВ-Петербург, 2002.
8. Гергель В.П. Теория и практика параллельных вычислений. – М.: Интернет-Университет Информационных технологий; БИНОМ. Лаборатория знаний, 2007.
9. Немнюгин С.А, Стесик О.Л. Параллельное программирование для многопроцессорных вычислительных систем. – СПб.:БХВ-Петербург, 2002.

7.3. Дополнительная литература

10. Юнаковский А.Д. Начала вычислительных методов для физиков. – Н. Новгород: ИПФ РАН, 2007
11. Березин И.С., Жидков И.П. Методы вычислений. – М.: Наука, 1966.
12. Майерс С. Эффективное использование C++. 35 новых способов улучшить стиль программирования. – СПб: Питер, 2006.
13. Майерс С. Эффективное использование C++. 55 верных способов улучшить структуру и код ваших программ. – М.: ДМК Пресс, 2006.
14. Павловская Т.А. C/C++. Программирование на языке высокого уровня. – СПб: Питер, 2003.
15. Рихтер Дж. Windows для профессионалов: создание эффективных Win32 приложений с учетом специфики 64-разрядной версии Windows/Пер. с англ. – 4-е изд. – СПб: Питер; М.: Издательско-торговый дом «Русская редакция», 2001.

7.4. Информационные ресурсы сети Интернет

16. Страница библиотеки TBB на сайте корпорации Intel – [http://www.intel.com/cd/software/products/asmo-na/eng/294797.htm].
17. Сайт сообщества пользователей TBB – [http://threadingbuildingblocks.org].
18. Сайт Лаборатории Параллельных информационных технологий НИВЦ МГУ – [http://www.parallel.ru].
19. Официальный сайт OpenMP – [www.openmp.org].

8. Приложения

8.1. Программный код последовательного вычисления БПФ

```
#include <iomanip>
#include <iostream>
#include <limits>
#include <complex>
#include <time.h>
#include "tbb/tick_count.h"

using namespace tbb;
using namespace std;

#define PI (3.14159265358979323846)

// Function for simple initialization of input signal elements
void DummyDataInitialization(complex<double>* mas, int size)
{
  for(int i = 0; i < size; i++)
    mas[i] = 0 ;

  mas[ size - size/4 ]=1;
}

// Function for random initialization of objects' elements
void RandomDataInitialization(complex<double>* mas, int size)
{
  srand(unsigned(clock()));
  for(int i = 0; i < size; i++)
    mas[i] = complex<double>(rand() / 1000.0, rand() / 1000.0) ;
}

// Function for memory allocation and data initialization
void ProcessInitialization(complex<double>* &inputSignal,
  complex<double>* &outputSignal, int &size)
{
  // Setting the size of signals
  do
  {
    cout << "Enter the input signal length: ";
    cin >> size;

    if(size < 4)
      cout << "Input signal length should be greater or equal than four"
        << endl;
    else
    {
      int tmpSize = size;

      while (tmpSize != 1)
      {
        if (tmpSize % 2 != 0)
        {
          cout << "Input signal length should be powers of two" << endl;
          size = -1;
          break;
        }
        tmpSize /= 2;
      }
    }
  }
```

```
  while(size < 4);
  cout << "Input signal length = " << size << endl;

  inputSignal = new complex<double>[size];
  outputSignal = new complex<double>[size];

  //Computing experiments
  //RandomDataInitialization(inputSignal, size);
  //-------------------------------------------------

  // Initialization of input signal elements
  DummyDataInitialization(inputSignal, size);
}

void ProcessTermination(complex<double>* &inputSignal,
  complex<double>* &outputSignal)
{
  delete [] inputSignal;
  inputSignal = NULL;

  delete [] outputSignal;
  outputSignal = NULL;
}

void BitReversing(complex<double> *inputSignal,
  complex<double> *outputSignal, int size)
{
  int j = 0, i = 0;

  while (i < size)
  {
    if (j > i)
    {
      outputSignal[i] = inputSignal[j];
      outputSignal[j] = inputSignal[i];
    }
    else
      if (j == i)
        outputSignal[i] = inputSignal[i];

    int m = size >> 1;
    while ((m >= 1) && (j >= m))
    {
      j -= m ;
      m = m >> 1;
    }
    j += m;
    i ++;
  }
}

__inline void Butterfly(complex<double> *signal, complex<double> &u,
  int offset, int butterflySize)
{
  complex<double> tem = signal[offset +  butterflySize] * u;

  signal[offset + butterflySize] = signal[offset] - tem;
  signal[offset] += tem;
}

void SerialFFTCalculation(complex<double> *signal, int size)
{
  int m = 0;
```

```
  for (int tmp_size = size; tmp_size > 1; tmp_size /= 2, m++) //size = 2^m
    ;
  for (int p = 0; p < m; p++)
  {
    int butterflyOffset = 1 << (p + 1);
    int butterflySize = butterflyOffset >> 1;
    double coeff = PI / butterflySize;

    for (int i = 0; i < size / butterflyOffset; i++)
      for (int j = 0; j < butterflySize; j++)
        Butterfly(signal, complex<double>(cos(-j * coeff),
          sin(-j * coeff)), j + i * butterflyOffset, butterflySize);
  }
}

// FFT computation
void SerialFFT(complex<double> *inputSignal, complex<double> *outputSignal,
  int size)
{
  BitReversing(inputSignal, outputSignal, size);
  SerialFFTCalculation(outputSignal, size);
}

void PrintSignal(complex<double> *signal, int size)
{
  cout << "Result signal" << endl;
  for (int i = 0; i < size; i++)
    cout << signal[i] << endl;
}

void PrintAmplitude(complex<double> *signal, int size)
{
  for(int i = 0; i < size; i++)
    cout << abs(signal[i]) << endl;
}

int main()
{
  complex<double> *inputSignal = NULL;
  complex<double> *outputSignal = NULL;

  int size = 0;

  const int repeatCount = 16;
  tick_count startTime;
  double duration;
  double minDuration = DBL_MAX;

  cout << "Fast Fourier Transform. Serial version." << endl;

  // Memory allocation and data inialization
  ProcessInitialization(inputSignal, outputSignal, size);

  for(int i = 0; i < repeatCount; i++)
  {
    startTime = tick_count::now();
    // FFT computation
    SerialFFT(inputSignal, outputSignal, size);
    duration  = (tick_count::now() - startTime).seconds();

    if(duration < minDuration)
      minDuration = duration;
  }
```

```
  cout << setprecision(6);
  cout << "Execution time is " << minDuration << " s. " << endl;

  // Result signal output
  PrintSignal(outputSignal, size);

  // Computational process termination
  ProcessTermination(inputSignal, outputSignal);

  cin.get();

  return 0;
}
```

8.1. Программный код параллельного вычисления БПФ с использованием функции tbb::parallel_for

```
#include <iomanip>
#include <iostream>
#include <limits>
#include <complex>
#include <time.h>
#include "tbb/tick_count.h"
#include "tbb/parallel_for.h"
#include "tbb/task_scheduler_init.h"
#include "tbb/blocked_range.h"

using namespace tbb;
using namespace std;

#define PI (3.14159265358979323846)

// Function for simple initialization of input signal elements
void DummyDataInitialization(complex<double>* mas, int size)
{
  for(int i = 0; i < size; i++)
    mas[i] = 0 ;

  mas[size – size / 4] = 1;
}

// Function for random initialization of objects' elements
void RandomDataInitialization(complex<double>* mas, int size)
{
  srand(unsigned(clock()));
  for(int i = 0; i < size; i++)
    mas[i] = complex<double>(rand() / 1000.0, rand() / 1000.0) ;
}

// Function for memory allocation and data initialization
void ProcessInitialization(complex<double>* &inputSignal,
  complex<double>* &outputSignal, int &size)
{
  // Setting the size of signals
  do
  {
    cout << "Enter the input signal length: ";
    cin >> size;

    if (size < 4)
      cout << "Input signal length should be greater or equal than four"
        << endl;
```

```
    else
    {
      int tmpSize = size;

      while (tmpSize != 1)
      {
        if (tmpSize % 2 != 0)
        {
          cout << "Input signal length should be powers of two" << endl;
          size = -1;
          break;
        }
        tmpSize /= 2;
      }
    }
  }
  while (size < 4);
  cout << "Input signal length = " << size << endl;

  inputSignal = new complex<double>[size];
  outputSignal = new complex<double>[size];

  //Computing experiments
  //RandomDataInitialization(inputSignal, size);
  //-------------------------------------------------

  // Initialization of input signal elements
  DummyDataInitialization(inputSignal, size);
}

void ProcessTermination(complex<double>* &inputSignal,
  complex<double>* &outputSignal)
{
  delete [] inputSignal;
  inputSignal = NULL;

  delete [] outputSignal;
  outputSignal = NULL;
}

void BitReversing(complex<double> *inputSignal,
  complex<double> *outputSignal, int size)
{
  int j = 0, i = 0;

  while (i < size)
  {
    if (j > i)
    {
      outputSignal[i] = inputSignal[j];
      outputSignal[j] = inputSignal[i];
    }
    else
      if (j == i)
        outputSignal[i] = inputSignal[i];

    int m = size >> 1;
    while ((m >= 1) && (j >= m))
    {
      j -= m ;
      m = m >> 1;
    }
    j += m;
```

```
    i ++;
  }
}

__inline void Butterfly(complex<double> *signal, complex<double> &u,
  int offset, int butterflySize)
{
  complex<double> tem = signal[offset +  butterflySize] * u;

  signal[offset + butterflySize] = signal[offset] - tem;
  signal[offset] += tem;
}

class ButterflyCalculator
{
  complex<double> *const signal;
  int const butterflySize;
  double coeff;

public:
   void operator()( const blocked_range<int>& r ) const
   {
     int begin = r.begin(),
         end = r.end(),
         butterflyOffset = 2 * butterflySize;

     for (int i = begin; i != end; i++)
       for (int j = 0; j < butterflySize; j++)
         Butterfly(signal, complex<double>(cos(-j * coeff),
           sin(-j * coeff)), j + i * butterflyOffset, butterflySize);
  }

  ButterflyCalculator(complex<double> *tsignal, int tbutterflySize):
    signal(tsignal), butterflySize(tbutterflySize)
  {
    coeff = PI / butterflySize;
  }
};

void ParallelFFTCalculation(complex<double> *signal, int size,
  int grainsize)
{
  int m = 0;
  for(int tmp_size = size; tmp_size > 1; tmp_size /= 2, m++) //size = 2^m
    ;

  for (int p = 0; p < m; p++)
  {
    int butterflyOffset = 1 << (p + 1);
    int butterflySize = butterflyOffset >> 1;
    int grain = grainsize;

    //Adaptive grainsize
    //grain = grainsize/ butterflyOffset;
    //if (grain < 1)
    //  grain = 1;

    parallel_for(blocked_range<int>(0, size / butterflyOffset, grain),
      ButterflyCalculator(signal, butterflySize));
  }
}

void ParallelFFT(complex<double> *inputSignal,
```

```
  complex<double> *outputSignal, int size, int grainsize)
{
  BitReversing(inputSignal, outputSignal, size);
  ParallelFFTCalculation(outputSignal, size, grainsize);
}

void PrintSignal(complex<double> *signal, int size)
{
  cout << "Result signal" << endl;
  for (int i = 0; i < size; i++)
    cout << signal[i] << endl;
}

void PrintAmplitude(complex<double> *signal, int size)
{
  for(int i=0; i<size; i++)
    cout << abs(signal[i]) << endl;
}

int main()
{
  complex<double> *inputSignal = NULL;
  complex<double> *outputSignal = NULL;

  int size = 0;

  task_scheduler_init init;

  cout << "Fast Fourier Transform. Parallel version." << endl;

  // Memory allocation and data inialization
  ProcessInitialization(inputSignal, outputSignal, size);

  const int testCount = 17;
  const int repeatCount = 16;
  tick_count startTime;
  double duration;

  cout << setprecision(6);
  for(int j = 0; j < testCount; j++)
  {
    int grainsize = size >> j;
    if (grainsize < 1)
      break;

    double minDuration = DBL_MAX;

    for(int i = 0; i < repeatCount; i++)
    {
      startTime = tick_count::now();
      ParallelFFT(inputSignal, outputSignal, size, grainsize);
      duration = (tick_count::now() - startTime).seconds();

      if (duration < minDuration)
        minDuration = duration;
    }

    cout << "Grain size is " << grainsize << ", execution time is ";
    cout << minDuration << " s. " << endl;
  }

  // Result signal output
  //PrintSignal(outputSignal, size);
```

```
  // Computational process termination
  ProcessTermination(inputSignal, outputSignal);

  cin.get();

  return 0;
}
```

8.2. Программный код параллельного вычисления БПФ с использованием логических задач

```
#include <iomanip>
#include <iostream>
#include <limits>
#include <complex>
#include <time.h>
#include "tbb/tick_count.h"
#include "tbb/task.h"
#include "tbb/task_scheduler_init.h"

using namespace tbb;
using namespace std;

#define PI (3.14159265358979323846)

// Function for simple initialization of input signal elements
void DummyDataInitialization(complex<double>* mas, int size)
{
  for(int i = 0; i < size; i++)
    mas[i] = 0 ;

  mas[size – size / 4] = 1;
}

// Function for random initialization of objects' elements
void RandomDataInitialization(complex<double>* mas, int size)
{
  srand(unsigned(clock()));
  for(int i = 0; i < size; i++)
    mas[i] = complex<double>(rand() / 1000.0, rand() / 1000.0) ;
}

// Function for memory allocation and data initialization
void ProcessInitialization(complex<double>* &inputSignal,
  complex<double>* &outputSignal, int &size)
{
  // Setting the size of signals
  do
  {
    cout << "Enter the input signal length: ";
    cin >> size;

    if(size < 4)
      cout << "Input signal length should be greater or equal than four"
        << endl;
    else
    {
      int tmpSize = size;

      while (tmpSize != 1)
      {
        if (tmpSize % 2 != 0)
```

```
        {
          cout << "Input signal length should be powers of two" << endl;
          size = -1;
          break;
        }
        tmpSize /= 2;
      }
    }
  }
  while(size < 4);
  cout << "Input signal length = " << size << endl;

  inputSignal = new complex<double>[size];
  outputSignal = new complex<double>[size];

  //Computing experiments
  //RandomDataInitialization(inputSignal, size);
  //-------------------------------------------------

  // Initialization of input signal elements
  DummyDataInitialization(inputSignal, size);
}

void ProcessTermination(complex<double>* &inputSignal,
  complex<double>* &outputSignal)
{
  delete [] inputSignal;
  inputSignal = NULL;

  delete [] outputSignal;
  outputSignal = NULL;
}

void BitReversing(complex<double> *inputSignal,
  complex<double> *outputSignal, int size)
{
  int j = 0, i = 0;

  while (i < size)
  {
    if (j > i)
    {
      outputSignal[i] = inputSignal[j];
      outputSignal[j] = inputSignal[i];
    }
    else
      if (j == i)
        outputSignal[i] = inputSignal[i];

    int m = size >> 1;
    while ((m >= 1) && (j >= m))
    {
      j -= m ;
      m = m >> 1;
    }
    j += m;
    i ++;
  }
}

__inline void Butterfly(complex<double> *signal, complex<double> &u,
  int offset, int butterflySize)
{
```

```
  complex<double> tem = signal[offset +  butterflySize] * u;

  signal[offset + butterflySize] = signal[offset] - tem;
  signal[offset] += tem;
}

void SerialFFTCalculation(complex<double> *const signal, int size)
{
  if (size == 1)
    return;

  double const coeff = 2.0 * PI / size;

  SerialFFTCalculation(signal, size / 2);
  SerialFFTCalculation(&(signal[size / 2]), size / 2);

  for (int j = 0; j < size / 2; j++)
    Butterfly(signal, complex<double>(cos(-j * coeff), sin(-j * coeff)),
      j , size / 2);
}

class FFTTask: public task
{
  complex<double> *const signal;
  int const size;
  double coeff;

public:
  FFTTask(complex<double> *const tSignal, int tSize):
    signal(tSignal), size(tSize)
  {
    coeff = 2.0 * PI / size;
  }

  task* execute()
  {
    if (size < 1024)
      SerialFFTCalculation(signal, size);
    else
    {
      FFTTask& a = *new (allocate_child()) FFTTask(signal, size / 2);
      FFTTask& b = *new (allocate_child()) FFTTask(&(signal[size / 2]),
        size/2);

      set_ref_count(3);

      spawn(b);
      spawn_and_wait_for_all(a);

      for (int j = 0; j < size / 2; j++)
        Butterfly(signal, complex<double>(cos(-j * coeff),
          sin(-j * coeff)), j , size / 2);
    }

    return NULL;
  }
};

void ParallelFFTCalculation(complex<double> *signal, int size)
{
  FFTTask& a = *new(task::allocate_root()) FFTTask(signal, size);
  task::spawn_root_and_wait(a);
}
```

```
void ParallelFFT(complex<double> *inputSignal,
  complex<double> *outputSignal, int size)
{
  BitReversing(inputSignal, outputSignal, size);
  ParallelFFTCalculation(outputSignal, size);
}

void PrintSignal(complex<double> *signal, int size)
{
  cout << "Result signal" << endl;
  for (int i = 0; i < size; i++)
    cout << signal[i] << endl;
}

void PrintAmplitude(complex<double> *signal, int size)
{
  for(int i = 0; i < size; i++)
    cout << abs(signal[i]) << endl;
}

int main()
{
  complex<double> *inputSignal = NULL;
  complex<double> *outputSignal = NULL;

  int size = 0;

  task_scheduler_init init;

  cout << "Fast Fourier Transform. Parallel version." << endl;

  // Memory allocation and data inialization
  ProcessInitialization(inputSignal, outputSignal, size);

  const int repeatCount = 16;
  tick_count startTime;
  double duration;
  double minDuration = DBL_MAX;

  for (int i = 0; i < repeatCount; i++)
  {
    startTime = tick_count::now();
    ParallelFFT(inputSignal, outputSignal, size);
    duration = (tick_count::now() - startTime).seconds();

    if (duration < minDuration)
      minDuration = duration;
  }

  cout << setprecision(6);
  cout << "Execution time is " << minDuration << " s. " << endl;

  // Result signal output
  //PrintSignal(outputSignal, size);

  // Computational process termination
  ProcessTermination(inputSignal, outputSignal);

  cin.get();

  return 0;
}
```