Нижегородский государственный университет им. Н.И.Лобачевского

Факультет Вычислительной математики и кибернетики

Инструменты параллельного программирования для систем с общей памятью

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

Мееров И.Б., Сысоев А.В., Сиднев А.А.
Кафедра математического обеспечения ЭВМ

Содержание

- Введение
- Рекомендации по выполнению работы
- Структура работы
- Задача выполнения быстрого преобразования Фурье
- Краткий обзор работы
- Контрольные вопросы
- Задания для самостоятельной работы
- Литература

Введение…

- В работе рассматривается один из инструментов параллельного программирования, предназначенный для распараллеливания решения задач в системах с общей памятью, – библиотека **Intel Threading Building Blocks (TBB)**.
- **Основная идея TBB**: использование C++ для быстрой разработки *кросс-платформенных*, *хорошо масштабируемых параллельных приложений.*
- TBB предоставляет **механизмы абстрагирования** от парадигм многопоточного программирования, позволяя сосредоточиться непосредственно на решении *прикладной задачи*.

Введение

- Изучение принципов функционирования и вопросов эффективного использования ТВВ проводится в данной лабораторной работе 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. При этом основной упор делается на ознакомление с механизмом *логических задач*.
- 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.

Рекомендации по выполнению работы…

- Целью данной лабораторной работы является приобретение практических навыков распараллеливания циклов для систем с общей памятью при помощи библиотеки TBB.
- Системные требования (для Windows-систем):
- **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.

Рекомендации по выполнению работы…

- Целью данной лабораторной работы является приобретение практических навыков распараллеливания циклов для систем с общей памятью при помощи библиотеки TBB.
- Системные требования (для Windows-систем):
- **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.

Структура работы

- Изучение способов инициализации и завершения работы библиотеки TBB.
- Рассмотрение функциональности библиотеки TBB, связанной с распараллеливанием циклов и рекурсии.
- Разбор одного из вариантов решения учебной задачи – вычисление БПФ, демонстрирующей принципы распараллеливания циклов и рекурсии при помощи библиотеки TBB.
- Выполнение дополнительных заданий, состоящих в самостоятельной разработке и отладке параллельных программ, реализующих распараллеливание циклов и рекурсии при помощи TBB.

Быстрое преобразование Фурье

Быстрое преобразование Фурье

- Дискретное преобразование Фурье (ДПФ) – это разложение дискретной функций на гармонические составляющие.
- ДПФ широко применяется в задачах анализа спектра сигнала, цифровой обработки сигналов и др.
- Вычисление ДПФ чаще всего происходит в виде быстрого преобразования Фурье (БПФ).
- Существуют специализированные процессоры для вычисления ДПФ (DSP).

Комплексные числа

Комплексное число

$z = x + iy;$ x, y - вещественные числа; $i^2 = -1$

Тригонометрическая форма записи

$z = r(\cos\varphi + i\sin\varphi);$ $r = |z|$

$$\cos\varphi = \frac{x}{|z|}, \sin\varphi = \frac{x}{|z|}$$

Показательная форма записи

$z = re^{i\varphi} = r(\cos\varphi + i\sin\varphi)$

Дискретное преобразование Фурье

$f(x)$ - комплексная функция действительного аргумента

$$x_k = f(k), k = \overline{0, n-1}$$

$$y_p = \sum_{k=0}^{n-1} x_k e^{-\frac{kp}{n}2\pi i}, p = \overline{0, n-1}$$ - дискретное преобразование Фурье (ДПФ)

Трудоёмкость: $O(n^2)$

Быстрое преобразование Фурье…

$$y_p = \sum_{k=0}^{n-1} x_k e^{-\frac{kp}{n}2\pi i}, p = \overline{0, n-1}$$

Пусть n – чётное, тогда $y_p = \underbrace{\sum_{k=0}^{n/2-1} x_{2k} e^{-\frac{2kp}{n}2\pi i}}_{\text{Чётные слагаемые}} + \underbrace{\sum_{k=0}^{n/2-1} x_{2k+1} e^{-\frac{(2k+1)p}{n}2\pi i}}_{\text{Нечётные слагаемые}} =$

$$= \underbrace{\sum_{k=0}^{n/2-1} x_{2k} e^{-\frac{kp}{n/2}2\pi i}}_{\text{ДПФ над чётными слагаемыми}} + \underbrace{\sum_{k=0}^{n/2-1} (x_{2k+1} e^{-\frac{kp}{n/2}2\pi i}}_{\text{ДПФ над нечётными слагаемыми}}) \cdot \underbrace{e^{-\frac{p}{n}2\pi i}}_{\text{Коэффициент поворота}}$$

Быстрое преобразование Фурье…

$$t = n/2 + p$$

$$y_t = \sum_{k=0}^{n/2-1} x_{2k} e^{-\frac{kt}{n/2}2\pi i} + \sum_{k=0}^{n/2-1} (x_{2k+1} e^{-\frac{kt}{n/2}2\pi i}) \cdot e^{-\frac{t}{n}2\pi i} =$$

$$= \sum_{k=0}^{n/2-1} x_{2k} e^{-\frac{kp}{n/2}2\pi i} \underbrace{e^{-kp2\pi i}}_{1} + \sum_{k=0}^{n/2-1} (x_{2k+1} e^{-\frac{kp}{n/2}2\pi i} \underbrace{e^{-kp2\pi i}}_{1}) \cdot e^{-\frac{p}{n}2\pi i} \underbrace{\cdot e^{-\pi i}}_{-1} =$$

$$= \underbrace{\sum_{k=0}^{n/2-1} x_{2k} e^{-\frac{kp}{n/2}2\pi i}}_{a_p} - \underbrace{\sum_{k=0}^{n/2-1} (x_{2k+1} e^{-\frac{kp}{n/2}2\pi i})}_{b_p} \cdot e^{-\frac{p}{n}2\pi i}$$

Быстрое преобразование Фурье

$$p = 0,\dots,n/2-1$$

$$t = n/2 + p$$

$$y_p = \underbrace{\sum_{k=0}^{n/2-1} x_{2k} e^{-\frac{kp}{n/2}2\pi i}}_{a_p} + \underbrace{\sum_{k=0}^{n/2-1} (x_{2k+1} e^{-\frac{kp}{n/2}2\pi i})}_{b_p} \cdot e^{-\frac{p}{n}2\pi i}$$

$$y_t = \underbrace{\sum_{k=0}^{n/2-1} x_{2k} e^{-\frac{kp}{n/2}2\pi i}}_{a_p} - \underbrace{\sum_{k=0}^{n/2-1} (x_{2k+1} e^{-\frac{kp}{n/2}2\pi i})}_{b_p} \cdot e^{-\frac{p}{n}2\pi i}$$

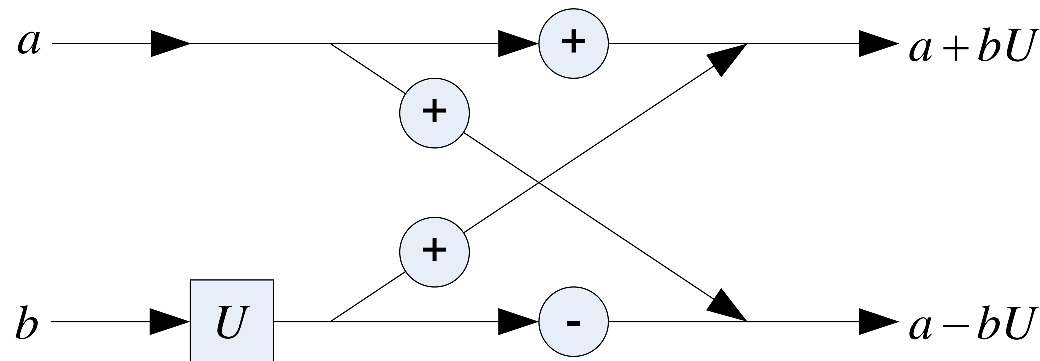
БПФ. «Бабочка»…

Для вычисления ДПФ над **n** элементами по соотношению «бабочки» необходимо вычислить два ДПФ размерности **n/2**

$$y_p = a_p + b_p \cdot e^{-\frac{p}{n}2\pi i}$$

$$y_{n/2+p} = a_p - b_p \cdot e^{-\frac{p}{n}2\pi i}$$

$$0 \le p < \frac{n}{2}$$

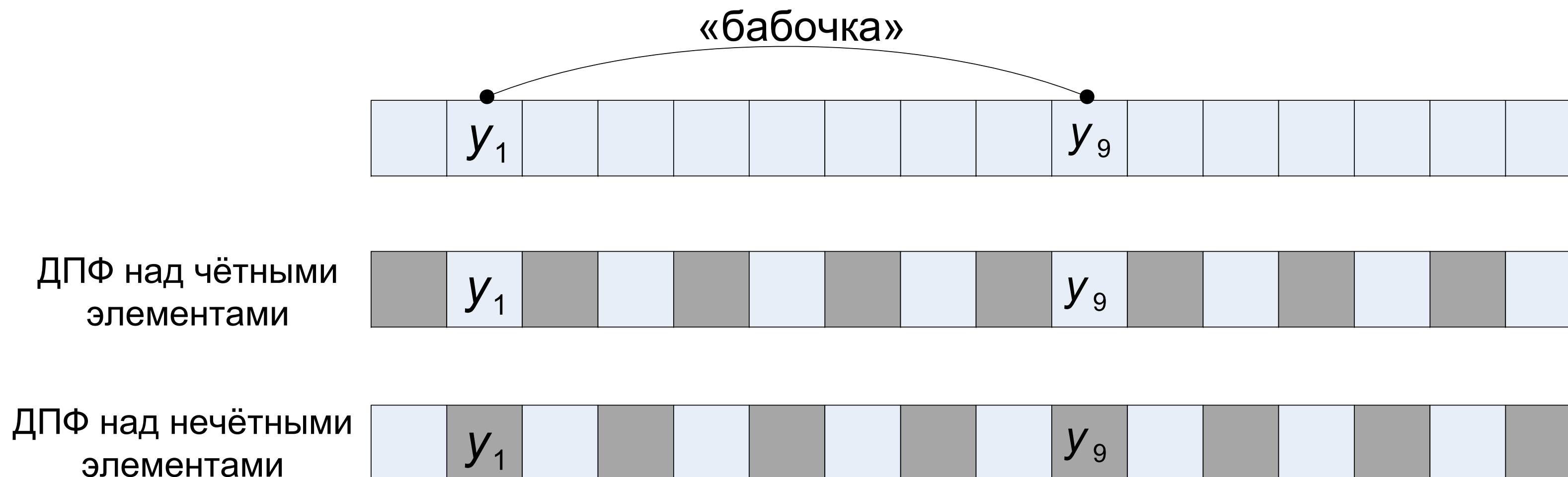


БПФ. «Бабочка»

$$y_p = a_p + b_p \cdot e^{-\frac{p}{n}2\pi i}$$

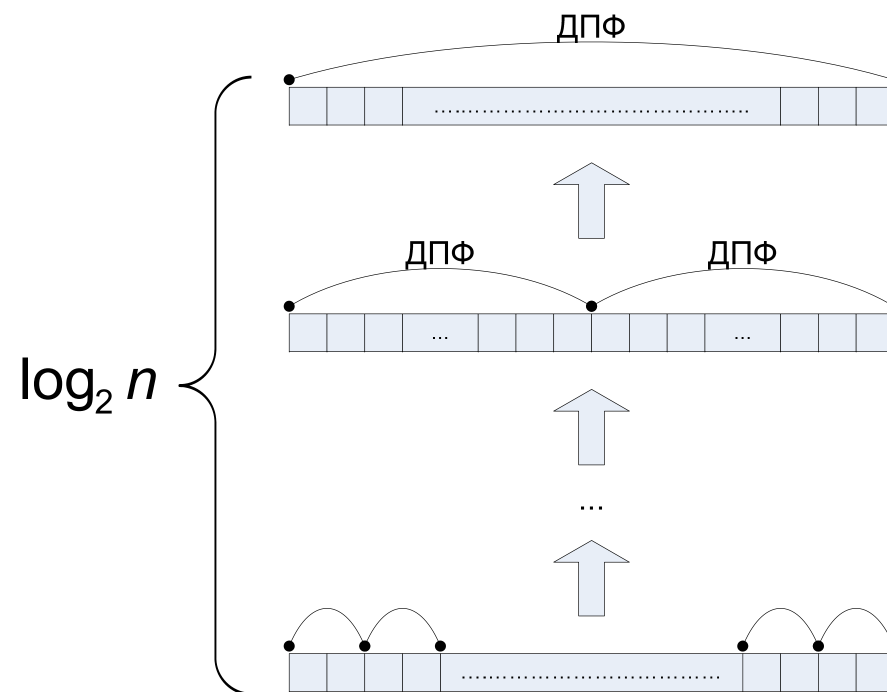
$$y_{n/2+p} = a_p - b_p \cdot e^{-\frac{p}{n}2\pi i}$$

$$a_1 = \sum_{k=0}^{7} x_{2k} e^{-\frac{k}{8}2\pi i} \qquad b_1 = \sum_{k=0}^{7} x_{2k+1} e^{-\frac{k}{8}2\pi i}$$



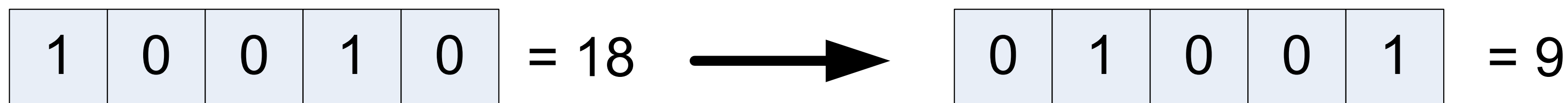
Быстрое преобразование Фурье

- Алгоритм Кули и Таки (Cooley-Tukey)
- Основан на рекуррентном соотношении
- Глубина рекурсии равна $\log_2 n$
- Трудоёмкость: $O(n \log n)$



Бит-реверсирование

- Переупорядочивание элементов массива
- Бит-реверсирование — это преобразование двоичного числа заключающееся в изменении порядка следования бит на противоположный



Бит-реверсирование. Пример

- Пример изменения индексов элементов при выполнении бит-реверсирования над массивом из 16 элементов

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

Бит-реверсирование. Реализация…

- Исходные данные:
  - ind – переменная с обычным порядком следования бит.
- Результат:
  - revInd – переменная с бит-реверсивным порядком следования бит.

```
//size = 1 << bitsCount;

int mask = 1 << (bitsCount - 1);
revInd = 0;

for(int i=0; i<bitsCount; i++)
{
  bool val = ind & mask;
  revInd |= val << i;
  mask = mask >> 1;
}
```

Первая итерация

ind	1	0	0	1	0
mask	1	0	0	0	0
val	0	0	0	0	1
revInd	0	0	0	0	1

Бит-реверсирование. Реализация

```
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 ++;
  }
}
```

БПФ. Алгоритм

- Бит-реверсирование
- Рекурсия (БПФ)
  - Вычисление БПФ над первой половиной массива
  - Вычисление БПФ над второй половиной массива
  - Применение «бабочки» ко всем парам элементов

$$y_p = a_p + b_p \cdot e^{-\frac{p}{n}2\pi i}$$

$$y_{n/2+p} = a_p - b_p \cdot e^{-\frac{p}{n}2\pi i}$$

$$0 \le p < \frac{n}{2}$$

Постановка задачи

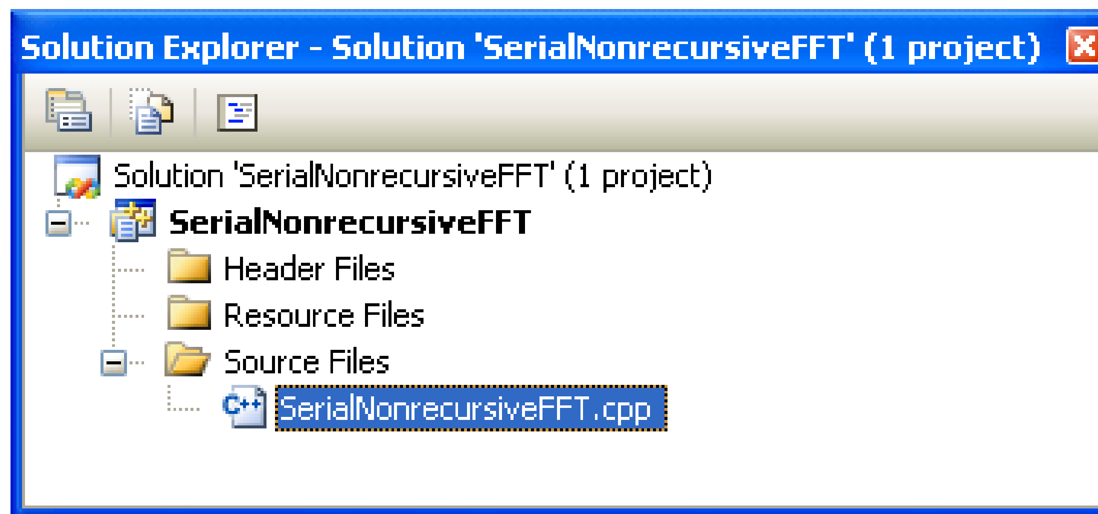
- Реализовать алгоритм БПФ
- Исходные данные:
  - Входной сигнал (массив комплексных чисел);
  - Размер массива (ограничиваемся случаем, когда размер равен степени числа 2).
- Результат:
  - Выходной сигнал (массив комплексных чисел, того же размера что и входной массив).

Последовательный алгоритм

- Некоторые замечания:
  - 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.

Реализация последовательного алгоритма. Открытие проекта…

- **Задание 1 – Открытие проекта SerialNonrecursiveFFT**
- Откройте проект **SerialNonrecursiveFFT**



- После открытия проекта в окне **Solution Explorer** (Ctrl+Alt+L) дважды щелкните на файле исходного кода **SerialNonrecursiveFFT.cpp**. После этих действий программный код, который предстоит в дальнейшем расширить, будет открыт в редакторе кода **Microsoft Visual Studio 2005**.

Реализация последовательного алгоритма. Открытие проекта…

- 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. Для работы с ними будем использовать класс **complex<double>** библиотеки STL, подключив соответствующий заголовочный файл: **#include <complex>**.

Реализация последовательного алгоритма. Открытие проекта…

```
complex<double> *inputSignal = NULL;
complex<double> *outputSignal = NULL;
int size = 0;
```

- 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**.

Реализация последовательного алгоритма. Открытие проекта

- Выполните команду **Rebuild Solution** в меню **Build,** запустите приложение (**F5** или **Debug/Start debugging**).
- После запуска исполняемого модуля в командной консоли появится сообщение: "Fast Fourier Transform. Serial version.", после чего программа будет ожидать нажатия клавиши **Enter**.

Реализация последовательного алгоритма. Ввод и генерация исходных данных…

- **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.

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

Реализация последовательного алгоритма. Ввод и генерация исходных данных…

- 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**.

Реализация последовательного алгоритма. Ввод и генерация исходных данных…

- 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
  cout << "Enter the input signal length: ";
  cin >> size;
  cout << "Input signal length = " << size << endl;
}
```

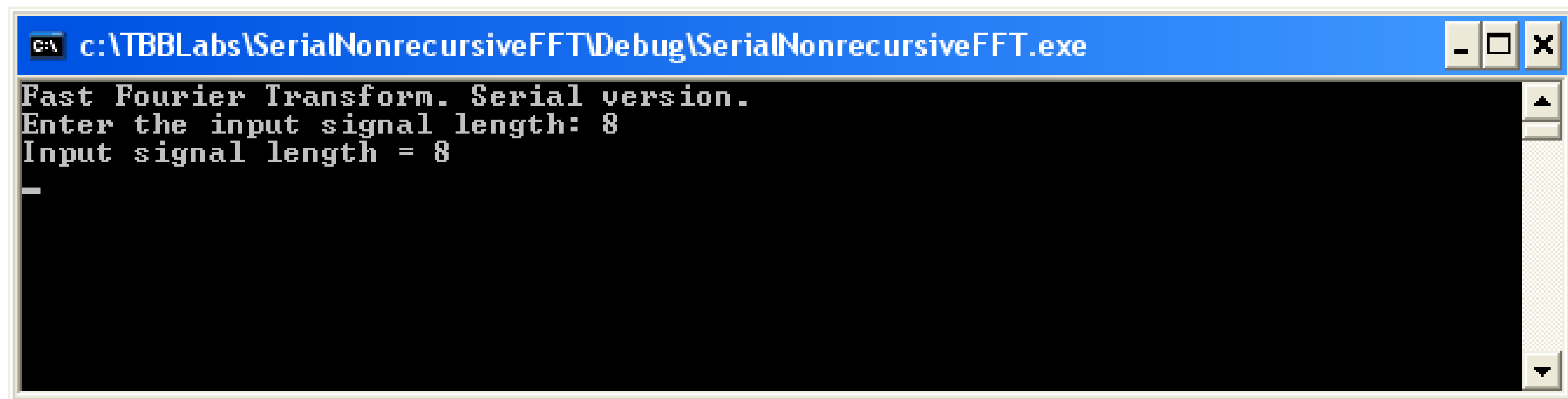
Реализация последовательного алгоритма. Ввод и генерация исходных данных…

- После строки, выводящей на экран приветствие, добавьте вызов функции инициализации процесса вычислений **ProcessInitialization** в тело основной функции последовательного приложения.

```
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;
}
```

Реализация последовательного алгоритма. Ввод и генерация исходных данных…

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



Реализация последовательного алгоритма. Ввод и генерация исходных данных…

- Теперь обратимся к вопросу контроля правильности ввода: фрагмент кода, который производит ввод размера объектов, поместим в цикл с постусловием.

```
// 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;
}
while(size < 4);
```

- Скомпилируйте и запустите приложение, проверьте его работоспособность.

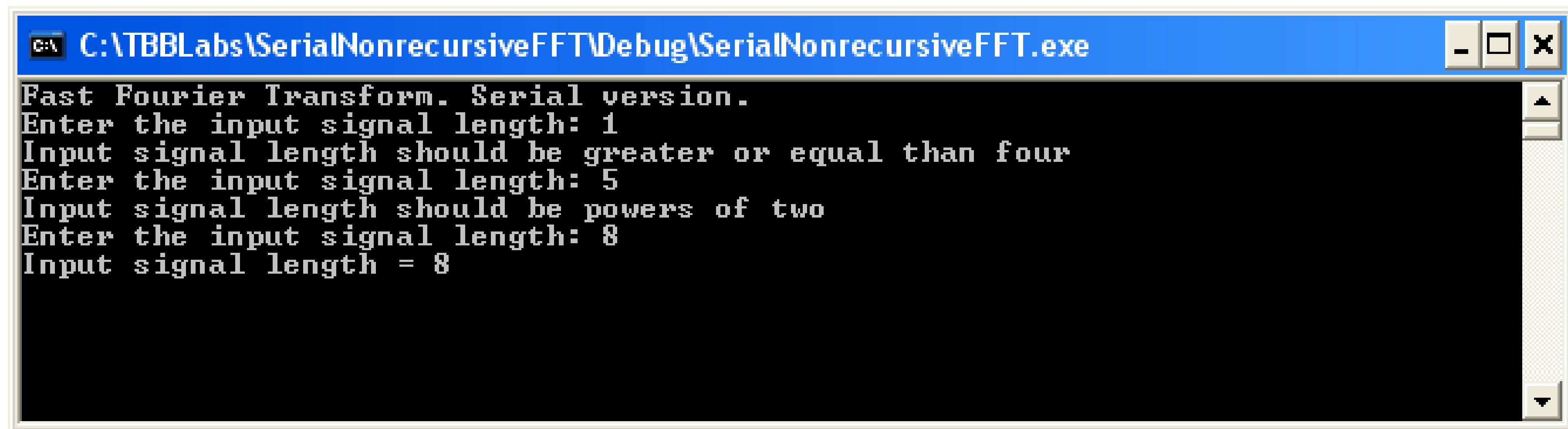
Реализация последовательного алгоритма. Ввод и генерация исходных данных…

- Далее добавим проверку того, что длина сигнала имеет степень двойки:

```
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;
  }
}
```

Реализация последовательного алгоритма. Ввод и генерация исходных данных…

- Скомпилируйте и запустите приложение, проверьте его работоспособность.



Реализация последовательного алгоритма. Ввод и генерация исходных данных…

- **Ввод данных**
- 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
  <...>

  cout << "Input signal length = " << size << endl;

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

Реализация последовательного алгоритма. Ввод и генерация исходных данных…

- 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**. Используя такие начальные данные, легко визуально проверить корректность работы реализованного алгоритма.

```
// Function for simple initialization
void DummyDataInitialization(complex<double>* mas, int
size)
{
  for(int i = 0; i < size; i++)
    mas[i] = 0 ;
  mas[size - size / 4] = 1;
}
```

Реализация последовательного алгоритма. Ввод и генерация исходных данных

- 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
  <...>

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

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

Реализация последовательного алгоритма. Завершение процесса вычислений…

- **<u>Задание 3 – Завершение процесса вычислений</u>**
- Перед вычислением БПФ сначала разработаем функцию для корректного завершения процесса вычислений.

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

Реализация последовательного алгоритма. Завершение процесса вычислений

- Вызов функции **ProcessTermination** необходимо добавить в самый конец главной функции **main**.

```
int main()
{
  <…>
  // 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.

Реализация последовательного алгоритма. Реализация БПФ…

- **Задание 4 – Реализация БПФ**
- Для выполнения БПФ реализуем функцию **SerialFFT**, состоящую из двух последовательных шагов:
  – преобразование входного массива данных (*бит-реверсирование*);
  – вычисление БПФ.

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

Реализация последовательного алгоритма. Реализация БПФ…

- Бит-реверсирование

```
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 ++;
  }
}
```

Реализация последовательного алгоритма. Реализация БПФ…

- Вычисление БПФ

```
void SerialFFTCalculation(complex<double> *signal, int size)
{
  int m = 0;
  for (int tmp_size = size; tmp_size > 1; tmp_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);
  }
}
```

Реализация последовательного алгоритма. Реализация БПФ

- Функция **Butterfly** реализует вычисление базовой операции алгоритма – «бабочки».

```
__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;
}
```

Реализация последовательного алгоритма. Проверка корректности…

- **Задание 5 – Проверка корректности**
- Для проверки корректности выполнения БПФ 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:

$$\cos(\frac{\pi}{2}k) + i\sin(\frac{\pi}{2}k) \quad k = \overline{0,7}$$

Реализация последовательного алгоритма. Проверка корректности…

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

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

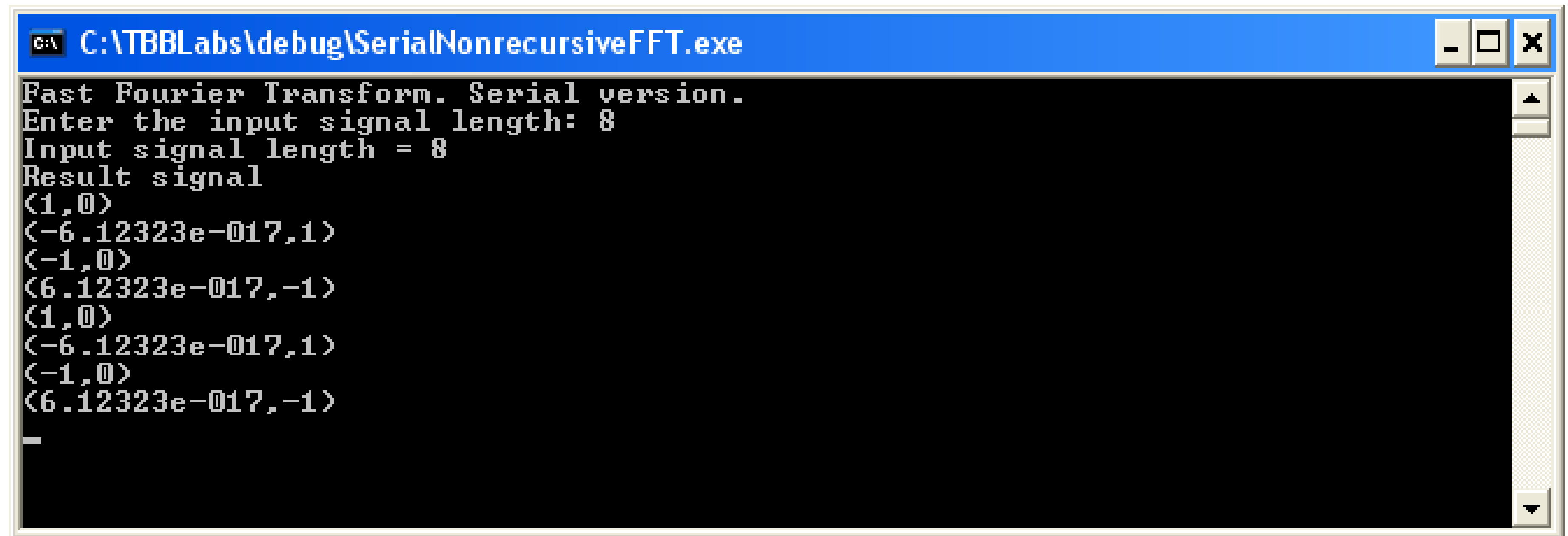
Реализация последовательного алгоритма. Проверка корректности…

- Для контроля правильности распечатаем результирующий сигнал:

```
int main()
{
  <…>
  // Memory allocation and data inialization
  ProcessInitialization(inputSignal, outputSignal, size);

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

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

  cin.get();
  return 0;
}
```

Реализация последовательного алгоритма. Проверка корректности

- Скомпилируйте и запустите приложение. Проанализируйте результат работы алгоритма умножения матрицы на вектор.



Реализация последовательного алгоритма. Проведение вычислительных экспериментов…

- **<u>Задание 6 – Проведение вычислительных экспериментов</u>**
- После реализации параллельной версии алгоритма нам потребуется оценивать ее ускорение.
- Для этого необходимо провести вычислительные эксперименты и замерить времена работы последовательной версии.
- Анализировать время выполнения последовательной реализации разумно для достаточно большого сигнала.
- Задавать элементы входного сигнала будем при помощи датчика случайных чисел.
- Для этого реализуем еще одну функцию задания элементов **RandomDataInitialization** (датчик случайных чисел инициализируется текущим значением времени).

Реализация последовательного алгоритма. Проведение вычислительных экспериментов…

```
// 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);
}
```

- 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.

Реализация последовательного алгоритма. Проведение вычислительных экспериментов…

- 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
  <...>
  // Memory allocation
  <...>
  // Random data initialization of input signal elements
  RandomDataInitialization(inputSignal, size);
}
```

Реализация последовательного алгоритма. Проведение вычислительных экспериментов…

- Реализуем функцию **GetTime()**, которая позволяет достаточно точно замерить время выполнения участка кода с использованием функций библиотеки WinAPI:

```
// Function that converts numbers form LongInt type to double type
double LiToDouble (LARGE_INTEGER x)
{
double result =
  ((double)x.HighPart) * 4.294967296E9 + (double)((x).LowPart);
  return result;
}
// Function that gets the timestamp in seconds
double GetTime()
{
  LARGE_INTEGER lpFrequency, lpPerfomanceCount;
  QueryPerformanceFrequency(&lpFrequency);
  QueryPerformanceCounter(&lpPerfomanceCount);
  return LiToDouble(lpPerfomanceCount) / LiToDouble(lpFrequency);
}
```

Реализация последовательного алгоритма. Проведение вычислительных экспериментов…

- Функция **GetTime** возвращает текущее значение времени, которое отсчитывается от фиксированного момента в прошлом, в секундах. Следовательно, вызвав эту функцию два раза – до и после исследуемого фрагмента можно вычислить время его работы. Например, этот фрагмент вычислит время **duration** работы функции **f()**.

```
double t1, t2;
t1 = GetTime();
f();
t2 = GetTime();
double duration = (t2 - t1);
```

Реализация последовательного алгоритма. Проведение вычислительных экспериментов…

- Библиотека TBB реализует собственный способ замера времени, сочетающий высокую разрешаемую способность с простотой использования. Для этого предусмотрен специальный класс **tick_count**, содержащий содержит статический метод **tick_count::now()**, возвращающий текущее время как объект класса **tick_count**.
- Как и раньше, для измерения времени необходимо вызвать метод **now()** в начале и в конце фрагмента программы с вычислительной частью, после чего вычесть из второго значения первое.
- Полученный результат можно перевести в секунды при помощи метода **tick_count::seconds()**.
- Отметим, что в реализации библиотеки TBB для ОС Windows функция **tick_count::now()** вызывает использованный нами **QueryPerformanceCounter**.

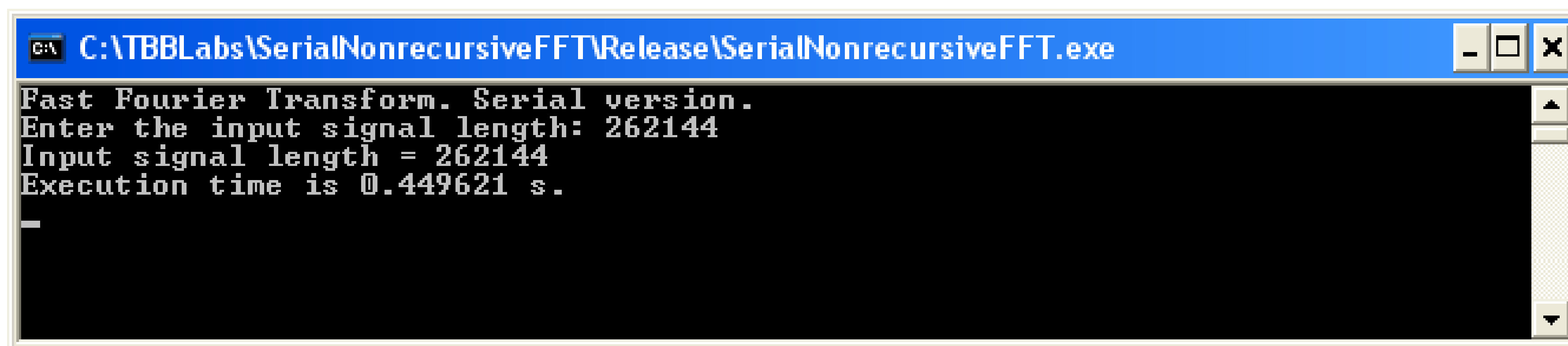
Реализация последовательного алгоритма. Проведение вычислительных экспериментов…

- Чтобы воспользоваться описанным способом замера времени, необходимо подключить заголовочный файл tbb/tick_count.h.

```
tick_count start, finish;
double duration;

// FFT computation
start = tick_count::now();
SerialFFT(inputSignal, outputSignal, size);
finish = tick_count::now();
duration = (finish - start).seconds();
// Print result signal
// PrintSignal(outputSignal, size);
// Printing the time spent by FFT computation
cout << setprecision(6);
cout << "Execution time is " << minDuration << " s. " << endl;
```

Реализация последовательного алгоритма. Проведение вычислительных экспериментов

- Скомпилируйте и запустите приложение.
- Для проведения вычислительных экспериментов с большими сигналами отключите печать (закомментируйте соответствующие строки кода).
- Убедитесь, что выбрана конфигурация **Release**.
- Проведите вычислительные эксперименты.



Реализация последовательного алгоритма. Оценка эффективности…

- **<u>Задание 7 – Оценка эффективности</u>**
- Запустите программу несколько раз, задавая одни и те же размеры исходных данных.
- Нетрудно видеть, что время работы меняется от запуска к запуску. Эффект вызван особенностями работы программ под управлением многозадачной операционной системы и погрешностью механизма измерения.
- В связи с этим существуют несколько методик организации замеров времени.
- Мы в данной лабораторной работе рекомендуем выполнить приложение несколько раз и выбрать минимальное из полученных времен работы при одних и тех же размерах сигнала.

Реализация последовательного алгоритма. Оценка эффективности

- Проведите эксперименты, результаты занесите в таблицу.
- *Таблица 1*:

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

Параллельный алгоритм. Принципы распараллеливания…

- *Действия для определения эффективного способа организации параллельных вычислений могут состоять в следующем*:
  – Выполнить анализ имеющейся вычислительной схемы и осуществить ее разделение (*декомпозицию*) на части (*подзадачи*), которые могут быть реализованы в значительной степени независимо друг от друга.
  – Выделить для сформированного набора подзадач *информационные взаимодействия*, которые должны осуществляться в ходе решения исходной поставленной задачи.
  – Выполнить распределение выделенных *подзадач* по вычислительным элементам (процессорам, ядрам).

Параллельный алгоритм. Принципы распараллеливания

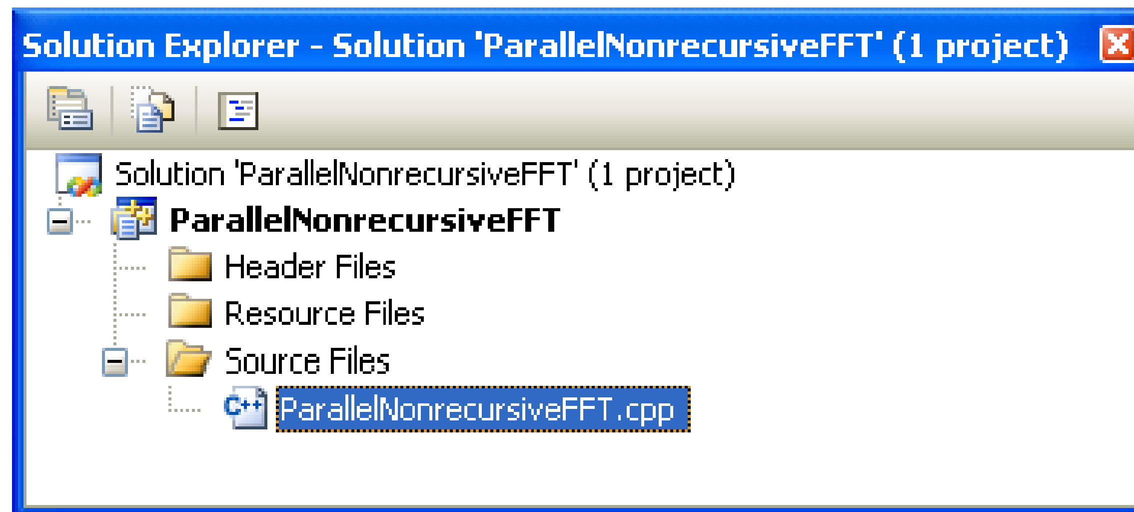
- Желательно обеспечить примерно одинаковый объем вычислений для каждого используемого потока – это позволит обеспечить равномерную вычислительную загрузку (*балансировку*).
- Распределение подзадач между процессорами (ядрами) желательно выполнено таким образом, чтобы число информационных связей (*коммуникационных взаимодействий*) между подзадачами было минимальным.

Реализация параллельного алгоритма…

- При выполнении этого упражнения необходимо реализовать параллельный алгоритм вычисления БПФ на основе имеющейся реализации последовательного алгоритма.
- При этом будут рассмотрены следующие возможности библиотеки TBB:
  - инициализация библиотеки;
  - настройка проекта для работы с библиотекой;
  - использование алгоритма **tbb::parallel_for**;
  - использование логических зада.

Реализация параллельного алгоритма. Открытие проекта…

- <u>**Задание 1 – Открытие проекта ParallelNonrecursiveFFT**</u>
- Откройте проект **ParallelNonrecursiveFFT**:



Реализация параллельного алгоритма. Открытие проекта…

- **<u>Задание 2 – Настройка проекта для использования TBB</u>**

Установка путей к заголовочным файлам (2 способа):

- 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**.

Реализация параллельного алгоритма. Открытие проекта

- **Задание 2 – Настройка проекта для использования TBB**

Установка путей к библиотекам TBB:

- 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).

Реализация параллельного алгоритма. Инициализация библиотеки TBB…

- **Задание 3 – Инициализация библиотеки TBB**
- Началу работы с библиотекой предшествует обязательная процедура инициализации. Для этого необходимо создать экземпляр класса **task_scheduler_init**.

Реализация параллельного алгоритма. Инициализация библиотеки TBB…

- **Задание 3 – Инициализация библиотеки TBB**
- Конструктор класса **Task_scheduler_init** в зависимости от полученных параметров выполняет следующие действия:
  – Указывает библиотеке TBB на необходимость автоматически определить число создаваемых потоков (**task_scheduler_init::automatic** – является значением по умолчанию).
  – Указывает библиотеке создать нужное разработчику число потоков (параметр конструктора – число потоков). Указывает библиотеке на необходимость отложить инициализацию до момента, когда это будет необходимо разработчику (**task_scheduler_init::deferred**). В этом случае сама инициализация происходит только после вызова метода **task_scheduler_init::initialize** с параметром *n* – число потоков, которое необходимо разработчику.

Реализация параллельного алгоритма. Инициализация библиотеки TBB…

- **Рекомендация**:
  – При выполнении данной лабораторной работы мы рекомендуем придерживаться схемы, подразумевающей инициализацию планировщика потоков в начале программы с параметром по умолчанию (автоматическое определение числа потоков), и завершение работы библиотеки в конце программы. Достаточно часто предлагаемая схема обеспечивает наилучшую производительность.

Реализация параллельного алгоритма. Инициализация библиотеки TBB

- Инициализация по умолчанию

```
#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;
  <...>
  return 0;
}
```

Реализация параллельного алгоритма. Использованием алгоритма tbb::parallel_for…

- **<u>Задание 4 – Разработка параллельной версии с использованием алгоритма tbb::parallel_for</u>**
  - Выполнив эксперименты, нетрудно определить, что бит-реверсирование занимает мало времени по сравнению с остальными вычислениями, поэтому распараллеливать имеет смысл функцию **SerialFFTCalculation.**

Реализация параллельного алгоритма. Использованием алгоритма tbb::parallel_for…

```
// 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 не имеет зависимостей по данным. Каждая итерация этого цикла может быть выполнена независимо, поэтому его можно достаточно легко распараллелить. На основе этого цикла реализуем функтор **ButterflyCalculator**.

Реализация параллельного алгоритма. Использованием алгоритма tbb::parallel_for…

- Полями функтора **ButterflyCalculator** будут все общие данные для потоков, использующиеся в вычислениях: **signal**, **butterflySize**, **coeff**. Поскольку **butterflyOffset = 2 * butterflySize**, переменную **butterflyOffset** не будем делать общей.
- Конструктор функтора будет содержать только два аргумента, т.к. переменную **coeff** можно считать программно в теле конструктора.

Реализация параллельного алгоритма. Поля и конструктор функтора

- Конструктор копирования и деструктор, создаваемые компилятором по умолчанию, в данном случае корректны, поэтому их реализация в явном виде не приводится.

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

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

Реализация параллельного алгоритма. Метод operator() функтора

- 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.

```
class ButterflyCalculator
{
  <...>
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);
  }
};
```

Реализация параллельного алгоритма. Использованием алгоритма tbb::parallel_for…

- Реализуем функции параллельного вычисления БПФ при помощи функции **parallel_for.**

```
// 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++)

  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)
    );
  }
}
```

Реализация параллельного алгоритма. Использованием алгоритма tbb::parallel_for…

- Реализуем общую функцию вычисления БПФ.

```
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.

Реализация параллельного алгоритма. Использованием алгоритма tbb::parallel_for…

- Добавим вызов **ParallelFFT** в функцию **main.**

```
int main()
{
  <...>
  // Memory allocation and data inialization
  ProcessInitialization(inputSignal, outputSignal, size);

  tick_count startTime;
  double duration;

  startTime = tick_count::now();
  ParallelFFT(inputSignal, outputSignal, size, grainsize);
  duration = (tick_count::now() - startTime).seconds();
  cout << "Execution time is " << duration << " s. " << endl;

  // Print signal output
  <...>
  return 0;
}
```

Реализация параллельного алгоритма. Использованием алгоритма tbb::parallel_for…

- Добавим код, необходимый для автоматического проведения экспериментов с различными значениями **grainsize.**

```
tick_count startTime;
double duration;
const int testCount = 20;  //Количество экспериментов
const int repeatCount = 8; //Количество повторений эксперимента

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;
}
```

Реализация параллельного алгоритма. Использованием алгоритма tbb::parallel_for…

- Соберите исполняемый модуль. Убедитесь, что компиляция прошла успешно. Запустите исполняемый модуль.

C:\TBBLabs\ParallelNonrecursiveFFT\Release\ParallelNonrecursiveFFT.exe

```
Fast Fourier Transform. Parallel version.
Enter the input signal length: 262144
Input signal length = 262144
Grain size is 262144, execution time is 0.497723 s.
Grain size is 131072, execution time is 0.502536 s.
Grain size is 65536, execution time is 0.490314 s.
Grain size is 32768, execution time is 0.480923 s.
Grain size is 16384, execution time is 0.465944 s.
Grain size is 8192, execution time is 0.458065 s.
Grain size is 4096, execution time is 0.446373 s.
Grain size is 2048, execution time is 0.429822 s.
Grain size is 1024, execution time is 0.417203 s.
Grain size is 512, execution time is 0.404641 s.
Grain size is 256, execution time is 0.387081 s.
Grain size is 128, execution time is 0.376452 s.
Grain size is 64, execution time is 0.362539 s.
Grain size is 32, execution time is 0.349168 s.
Grain size is 16, execution time is 0.338686 s.
Grain size is 8, execution time is 0.327099 s.
Grain size is 4, execution time is 0.319108 s.
Grain size is 2, execution time is 0.319731 s.
Grain size is 1, execution time is 0.325959 s.
_
```

Реализация параллельного алгоритма. Использованием алгоритма tbb::parallel_for…

- Выполните сборку приложения в конфигурации **Release**. Проведите эксперименты, выбирая оптимальное значение параметра grainsize, и заполните таблицу.
- *Таблица 2*:

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

Реализация параллельного алгоритма. Использованием алгоритма tbb::parallel_for…

- 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.

Реализация параллельного алгоритма. Использованием алгоритма tbb::parallel_for…

- Реализация параллельного алгоритма с адаптивным подбором **grainSize.**

```
// Parallel FFT computation
void ParallelFFTCalculation(complex<double> *signal, int size,
  int grainsize)
{
  <...>
  for (int p = 0; p < m; p++)
  {
    <...>
    int grain = grainsize / butterflyOffset;
    if (grain < 1)
      grain = 1;
    parallel_for(
          blocked_range<int>(0, size / butterflyOffset, grain),
          ButterflyCalculator(signal, butterflySize)
    );
  }
}
```

Реализация параллельного алгоритма. Использованием алгоритма tbb::parallel_for

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

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

Реализация параллельного алгоритма. Использованием логических задач…

- **<u>Задание 5 – Разработка параллельной версии с использование логических задач</u>**
- 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.

Реализация параллельного алгоритма. Использованием 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.

Реализация параллельного алгоритма. Использованием логических задач…

- Типичная рекурсивная реализация БПФ.

```
// 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);
}
```

Реализация параллельного алгоритма. Использованием логических задач…

- 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.

Реализация параллельного алгоритма. Использованием логических задач…

- 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).

Реализация параллельного алгоритма. Использованием логических задач…

- 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()**.

Реализация параллельного алгоритма. Использованием логических задач…

- Реализуем класс логической задачи для вычисления БПФ.
- 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).

Реализация параллельного алгоритма. Поля и конструктор логической задачи

- Как и в случае функтора, полями класса будут все общие данные, использующиеся в вычислениях: **signal**, **butterflySize**, **coeff.**

```
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;
  }
};
```

Реализация параллельного алгоритма. Метод execute() логической задачи

```
class FFTTask: public task
{
  <...>
public:
  task* execute()
  {
    if (size < 1024) //Пороговое значение
      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;
  }
};
```

Реализация параллельного алгоритма. Использованием логических задач…

- Добавим создание и запуск «главной» задачи 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);
}
```

Реализация параллельного алгоритма. Использованием логических задач

- Внесём необходимые изменения в функцию **main.**

```
int main()
{
  <...>
  tick_count startTime;
  double duration;
  const int repeatCount = 16;
  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;

  <...>
  return 0;
}
```

Реализация параллельного алгоритма. Проведение вычислительных экспериментов…

- **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.

Реализация параллельного алгоритма. Проведение вычислительных экспериментов…

- Заполните таблицу 4.
- *Таблица 4*:

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				

Реализация параллельного алгоритма. Проведение вычислительных экспериментов

- На основе значений, записанных в таблицу 4, посчитайте ускорения (<время работы последовательного алгоритма> / <время работы параллельного алгоритма>) и заполните таблицу 5.
- *Таблица 5*:

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			

Краткий обзор работы

- Данная лабораторная работа посвящена изучению распараллеливания циклов и рекурсии на общей памяти с использованием библиотеки Intel Threading Building Blocks.
- Изучение материала выполнено на примере задачи для демонстрации возможностей параллельного программирования – вычисление быстрого преобразования Фурье.
- Рассмотрена классическая постановка задачи, приведен последовательный алгоритм решения, разобрана его последовательная реализация.
- Приведено два из возможных алгоритмов распараллеливания, а также его реализация при помощи TBB.

Контрольные вопросы…

- Как собрать приложение, использующее библиотеку ТВВ?
- Как инициализировать библиотеку ТВВ?
- Как завершить работу библиотеки ТВВ?
- Как инициализировать библиотеку ТВВ с определенным числом программных потоков?

Контрольные вопросы

- 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?

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.

Использованные источники информации

- Гергель В.П., Лабутина А.А. Умножение матрицы на вектор // Материалы образовательного комплекса «Параллельное программирование на OpenMP». Нижний Новгород, 2007.
- 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.
- Intel® Threading Building Blocks. Reference Manual. Version 1.6. – Intel Corporation, 2007.
- Intel® Threading Building Blocks. Tutorial. Version 1.6. – Intel Corporation, 2007.

Рекомендуемая литература

- 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).
- Quinn, M.J. (2004). Parallel Programming in C with MPI and OpenMP. – New York, NY: McGraw-Hill.
- Воеводин В.В., Воеводин Вл.В. Параллельные вычисления. – СПб.: БХВ-Петербург, 2002.
- Гергель В.П. Теория и практика параллельных вычислений. – М.: Интернет-Университет Информационных технологий; БИНОМ. Лаборатория знаний, 2007.
- Немнюгин С.А, Стесик О.Л. Параллельное программирование для многопроцессорных вычислительных систем. – СПб.:БХВ-Петербург, 2002.

Дополнительная литература

- Березин И.С., Жидков И.П. Методы вычислений. – М.: Наука, 1966.
- Майерс С. Эффективное использование C++. 35 новых способов улучшить стиль программирования. – СПб: Питер, 2006.
- Майерс С. Эффективное использование C++. 55 верных способов улучшить структуру и код ваших программ. – М.: ДМК Пресс, 2006.
- Павловская Т.А. C/C++. Программирование на языке высокого уровня. – СПб: Питер, 2003.
- 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.

Информационные ресурсы сети Интернет

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

Авторский коллектив

- Мееров Иосиф Борисович,
  к.т.н., доцент кафедры Математического обеспечения ЭВМ факультета ВМК ННГУ.
- Сысоев Александр Владимирович,

  ассистент кафедры Математического обеспечения ЭВМ факультета ВМК ННГУ.
- Сиднев Алексей Александрович,

  ассистент кафедры Математического обеспечения ЭВМ факультета ВМК ННГУ.