Как запустить проект Code::Blocks с примером в Windows

1. Скачать и установить Code::Blocks: `https://sourceforge.net/projects/codeblocks/files/Binaries/17.12/Windows/codeblocks-17.12mingw-setup.exe/download`
2. Создать копию проекта с примером, т. е. скопировать себе папку `ParallelTreeExample`. Она находится в `example/parallel_CodeBlocks_Windows`. Данный пример подробно рассмотрен ниже.
3. Запустить Code::Blocks, выбрать «Open an existing project» и выбрать файл `ParallelTreeExample.cbp` в созданной копии папки `ParallelTreeExample`.
4. Дополнительно ничего настраивать не нужно. Библиотеки ParallelTree и OpenMP уже подключены в настройках проекта.

Как запустить проект Code::Blocks с примером в Linux

1. Установить gcc версии 5 или выше и установить Code::Blocks. Как это сделать — зависит от дистрибутива Linux.
2. Создать копию проекта с примером, т. е. скопировать себе папку `ParallelTreeExample`. Она находится в `example/parallel_CodeBlocks_Linux_x86_64` для 64-битного Linux, либо в `example/parallel_CodeBlocks_Linux_x86` для 32-битного Linux. Данный пример подробно рассмотрен ниже.
3. Запустить Code::Blocks, выбрать «Open an existing project» и выбрать файл `ParallelTreeExample.cbp` в созданной копии папки `ParallelTreeExample`.
4. Дополнительно ничего настраивать не нужно. Библиотеки ParallelTree и OpenMP уже подключены в настройках проекта.

Как собрать и запустить пример с помощью make

1. Установить gcc версии 5 или выше и утилиты GNU для разработки. Как это сделать — зависит от операционной системы.
2. Открыть папку `example/parallel_Makefile` в консоли и набрать команду `make`
3. Запустить `./example`

Сборка вручную или в другой среде разработки

Для сборки проектов, использующих библиотеку ParallelTree, компилятору gcc обязательно нужно указать следующие флаги:

1. `-std=c++11` для использования стандарта C++11 (можно и `-std=c++14` для использования стандарта C++14, но не обязательно).

2. `-fopenmp` для включения поддержки OpenMP.

3. `-I <путь к ParallelTree.hpp>` для указания пути к заголовочному файлу библиотеки. Он находится в папке `include`.

4. `-L<путь к библиотеке> -lParallelTree -Wl,-rpath=<путь к библиотеке>` для подключения библиотеки ParallelTree. Версии этой библиотеки для 32-битных и 64-битных Windows и Linux находятся в папке `lib`. Обратите внимание, что `<путь к библиотеке>` пишется после `-L` и после `-Wl,-rpath=` **без пробела**.

Например: для сборки примера в `example/parallel_Makefile` в 64-битном Linux нужно использовать следующую команду:

```
g++ main.cpp -o example -std=c++11 -fopenmp -I ../../include -L../../lib/Linux_x86_64
 ↪  -lParallelTree -Wl,-rpath=../../lib/Linux_x86_64
```

Флаг `-o example` указывает, что имя выходного файла должно быть `example`.

Как пользоваться библиотекой ParallelTree

Библиотека ParallelTree предназначена для распараллеливания древовидных алгоритмов для задач минимизации и максимизации.

Чтобы воспользоваться библиотекой, нужно включить соответствующий заголовочный файл (`ParallelTree.hpp`) и вызвать функцию `parallelTree`:

```
void parallelTree(
    std::vector<std::unique_ptr<Node>> (*processNode)(std::unique_ptr<Node>,
                                                      Result&, const Params*),
    std::unique_ptr<Node> node,
    Result& result,
    const Params* params = nullptr,
    bool (*higherPriority)(const std::unique_ptr<Node>& n1,
                           const std::unique_ptr<Node>& n2,
                           const Params* params) = nullptr);
```

Она принимает следующие параметры:

1. `processNode` — функция обработки узлов дерева вариантов. Принимает текущий узел (`node`), ссылку на рекорд (`result`) и указатель на дополнительные параметры (`params`) и возвращает потомков данного узла (или пустой вектор, если потомков нет). Рекорд — наилучший найденный результат. В примере ниже `params` не используется (класс

`TreeParams` приведен просто для демонстрации). В данную функцию всегда передается копия `result` (создается с помощью метода `clone`), причем создается по копии для каждого потока, поэтому беспокоиться о многопоточном доступе к нему не нужно. По завершении работы функции `processNode` измененная копия `result` автоматически сравнивается с самим `result` (с помощью метода `betterThan`) и `result` заменяется на копию, если копия лучше.

2. `node` — корень дерева вариантов.

3. `result` — класс, который содержит рекорд.

4. `params` — класс, который хранит константные параметры алгоритма, если они нужны. Вместо него можно указать `nullptr`.

5. `higherPriority` — функция приоритетов. Возвращает `true`, если левый узел (`n1`) нужно обрабатывать до правого (`n2`), и `false`, если правый нужно обрабатывать раньше, чем левый. В некоторых ситуациях порядок их обработки может быть нарушен (хотя большинство узлов будут обработаны в правильном порядке), поэтому писать программу с расчетом на определенный порядок обработки узлов нельзя. Вместо `higherPriority` можно передать `nullptr`. Тогда узлы будут обрабатываться в том порядке, в котором получится.

Функция `parallelTree` занимается параллельной обработкой узлов дерева вариантов с помощью функции `processNode`. Для начала ей нужен только корень дерева (`node`). После обработки корня она занимается обработкой потомоков, которых вернет функция `processNode` (потомки также передаются в функцию `processNode`). Алгоритм считается завершенным, если все узлы были обработаны. Вместо `params` и `higherPriority` можно указать `nullptr`, если они не нужны.

Классы `Node`, `Result` и `Params` имеют следующий вид:

```
1  class Node {};
2
3  class Result
4  {
5  public:
6      /**
7       * @brief Должен возвращать true, если данный рекорд лучше (меньше в задачах
8       *        минимизации и больше в задачах максимизации), чем @p other
9       */
10     virtual bool betterThan(const Result& other) = 0;
11
12     /**
13      * @brief Должен возвращать копию данного объекта.
14      */
15     virtual std::unique_ptr<Result> clone() = 0;
16
```

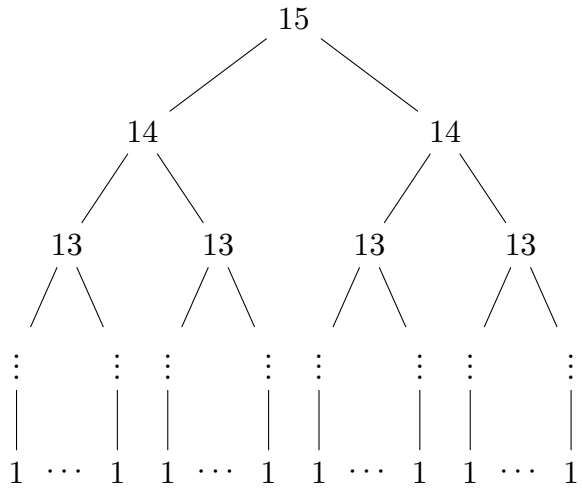
```
    virtual Result& operator= (const Result& other) = 0;
};

class Params {};
```

От них нужно унаследоваться. Три метода класса `Result` предназначены для внутреннего использования в библиотеке. При наследовании их обязательно нужно переопределить, но вызывать их самому не обязательно.

Пример использования библиотеки

Далее рассмотрим пример решения следующей задачи с помощью библиотеки ParallelTree (весь код примера можно найти в папке `example`):

Дано двоичное дерево вариантов, в каждом узле которого находится целое число. Найти минимум этих чисел. На вход подадим дерево следующего вида (ясно, что минимум в нем будет равен 1):



Примечание: чтобы перейти от последовательной версии алгоритма к параллельной с наименьшим количеством изменений кода рекомендуется писать последовательную версию как рекурсивную функцию, или заменить рекурсию на стек (данный код можно найти в папке `example/sequential_Makefile`):

```
#include <iostream>
#include <limits>
#include <stack>

using namespace std;

// Рекурсивный обход дерева
void minTreeRecursive(int value, int& record)
{
    if(value < record)
```

```
        record = value;
    if(value > 1)
    {
        // Потомки данного узла
        minTreeRecursive(value-1, record);
        minTreeRecursive(value-1, record);
    }
}

// Замена рекурсии на стек
void minTreeStack(int rootValue, int& record)
{
    // Узлы дерева, которые нужно обработать
    stack<int> nodes;
    nodes.push(rootValue);

    while(!nodes.empty())
    {
        int value = nodes.top();
        nodes.pop();
        if(value < record)
            record = value;
        if(value > 1)
        {
            // Потомки данного узла
            nodes.push(value-1);
            nodes.push(value-1);
        }
    }
}


int main()
{
    int recordRecursive = numeric_limits<int>::max();
    minTreeRecursive(15, recordRecursive);

    int recordStack = numeric_limits<int>::max();
    minTreeStack(15, recordStack);

    cout << ''Result (recursive): '' << recordRecursive << ''\n'';
    cout << ''Result (stack):     '' << recordStack     << ''\n'';

    return 0;
}
```

Далее напишем параллельную версию алгоритма. Для этого унаследуемся от классов `Node`, `Result` и `Params`:

```cpp
// Узел дерева вариантов. В данном случае каждый узел содержит одно число.
class TreeNode : public Node
{
public:
    TreeNode(int number) :
        number(number) {}

    int number;
};

// Класс, который содержит рекорд.
class NumericResult : public Result
{
public:
    NumericResult(int value) :
        value(value) {}

    /* Эти три метода нужны для внутреннего использования в библиотеке.
     * Пользоваться ими самому не обязательно.
     */

    /* Возвращает true, если данный рекорд лучше other. Поскольку данная задача
     * - задача минимизации, то здесь стоит оператор <. В задачах максимизации
     * нужно поставить >.
     */
    bool betterThan(const Result& other) override
    {
        const NumericResult o = (NumericResult&) other;
        return value < o.value;
    }

    // Создает копию данного рекорда.
    std::unique_ptr<Result> clone() override
    {
        return std::unique_ptr<Result>(new NumericResult(*this));
    }

    // Присваивает данному рекорду значение other.
    Result& operator= (const Result& other) override
    {
        const NumericResult o = (NumericResult&) other;
        value = o.value;
        return *this;
    }

    int value;
};

```

```
/* Если алгоритму нужны какие-либо константные параметры, то их нужно хранить
 * здесь. В данном примере они не нужны. Этот класс приведен просто для
 * демонстрации.
 */
class TreeParams : public Params
{
public:
    const int param1 = 10;
    const double param2 = 2.5;
};
```

Напишем функцию обработки узлов `processNode`:

```
vector<unique_ptr<Node>> processNode(std::unique_ptr<Node> node,
                                     Result& result, const Params* params)
{
    TreeNode* treeNode = (TreeNode*) node.get();
    NumericResult& numericResult = (NumericResult&) result;

    // Если число в данном узле меньше рекорда, то обновляем рекорд.
    if(treeNode->number < numericResult.value)
        numericResult.value = treeNode->number;

    // Если число в данном узле больше 1, то в его потомках находятся числа на 1 меньше.
    if(treeNode->number > 1)
    {
        vector<unique_ptr<Node>> children;
        children.emplace_back(new TreeNode(treeNode->number - 1));
        children.emplace_back(new TreeNode(treeNode->number - 1));

        return children;
    }
    // Если число в данном узле равно 1, то у него нет потомков.
    else
        return vector<unique_ptr<Node>>();
}
```

И напишем функцию приоритетов `higherPriority`, которая будет отдавать предпочтение узлам с меньшим числом в них:

```
bool higherPriority(const std::unique_ptr<Node>& n1,
                    const std::unique_ptr<Node>& n2,
                    const Params* params)
{
    // Узлы с меньшим числом в них обрабатываем раньше
    TreeNode* treeNode1 = (TreeNode*) n1.get();
    TreeNode* treeNode2 = (TreeNode*) n2.get();
```

```
    return treeNode1->number < treeNode2->number;
}
```

И, наконец, напишем `main`:

```
int main()
{
    // Корень дерева
    unique_ptr<Node> root(new TreeNode(15));

    /* Параметры алгоритма. В данном примере они не нужны. Этот класс приведен
     * просто для демонстрации.
     */
    TreeParams params;

    /* Рекорд.
     * Мы не можем присвоить int бесконечность, поэтому присваиваем максимальное
     * возможное значение.
     */
    NumericResult result(numeric_limits<int>::max());

    parallelTree(processNode, move(root), result, &params, higherPriority);

    // По окончании работы функции parallelTree результат будет храниться в result.
    cout << "Result: " << result.value << "\n";

    return 0;
}
```

В результате работы функции `parallelTree` будет найдено минимальное число в дереве. Оно будет сохранено в `result`.