Паралелен BFS, client - server

Проектът включва няколко файлове и е комплексен, но възможно е да го стартирате, следвайки няколко стъпки:

Инсталиране на компилатор и библиотеката за сокети:

- Уверете се, че имате инсталиран компилатор за C++ като g++.
- Проектът използва Winsock за работа с сокети. На Windows това вече е включено в библиотеката. Ако сте на Linux, може да се наложи да инсталирате "libws2-32-dev".

Генериране на изпълнимите файлове:

- Отворете терминал или команден прозорец в директорията на проекта.
- Изпълнете командите за компилация от makefile:

```
make
```

- Това ще компилира всички файлове и създаде два изпълними файла: `client.exe` и `server.exe`.

Стартиране на сървъра:

- Стартирайте `server.exe`. Този файл служи като сървър, който слуша за връзки.

Стартиране на клиента:

- Стартирайте `client.exe`. Този файл служи като клиент, който изпраща данни към сървъра.

Въвеждане на данни:

- След стартиране на клиента ще бъдете подканени да въведете размера на таблицата и броя на процесите. Въведете желаните стойности.

Резултат:

- След въвеждане на данните клиента ще изпрати данните към сървъра, който ще изпълни BFS (breadth-first search) алгоритъм и ще върне резултата.

Забележки:

- Проектът изглежда да използва Windows сокети и библиотеката `winsock2.h`. Уверете се, че сте на Windows и че компилацията и стартирането протича без проблеми.
- Моля, обърнете внимание, че сървърът и клиентът трябва да бъдат стартирани на различни компютри или поне на различни портове, за да не се конфликтоват.
- Проверете дали фаеруолът ви позволява комуникацията през използвания порт (8080 в този случай).

Файл "thread_pool.h"

Клас `ThreadPool`

- `ThreadPool(int number_of_workers)`: Конструктор на класа, инициализира броя на работници.
- `void start()`: Стартира работата на работниците.
- `void stop()`: Спира работата на работниците.
- `bool push_job(const BFSTask& job)`: Добавя задача в опашката за работниците.
- `bool is_busy()`: Проверява дали опашката с задачи е празна.
- `void wait_work()`: Изчаква приключването на работата на работниците.
- `void thread_loop(const int index)`: Цикъл за работните нишки.

Файл "thread_pool.cpp"

Функции в `ThreadPool`

- `bool ThreadPool::is_busy()`: Проверява дали опашката с задачи е празна.
- `void ThreadPool::wait_work()`: Изчаква приключването на работата на работниците.
- `ThreadPool::ThreadPool(int number_of_workers)`: Конструктор на класа, инициализира броя на работници.
- `void ThreadPool::start()`: Стартира работата на работниците.
- `void ThreadPool::stop()`: Спира работата на работниците.
- `void ThreadPool::thread_loop(int thread_number)`: Цикъл за работните нишки.
- `bool ThreadPool::push_job(const BFSTask& job)`: Добавя задача в опашката за работниците.

Файл "bfs_task.h"

Клас `BFSTask`

- `BFSTask(std::vector<std::vector<int>>& _table)`: Конструктор, инициализира задачата с таблица.
- `void fill(int number)`: Изпълнява BFS върху таблицата.

Файл "bfs_task.cpp"

Функции в `BFSTask`

- `BFSTask::BFSTask(std::vector<std::vector<int>>& _table)`: Конструктор, инициализира задачата с таблица.

- `void BFSTask::fill(int number)`: Изпълнява BFS върху таблицата.

Файл "client.cpp"

Функции в `client.cpp`

- `int read_bytes_from_socket(SOCKET& s, char *buf, int len, int flags)`: Чете байтове от сокет.
- `int main()`: Основната функция на клиента.

Файл "server.cpp"

Функции в `server.cpp`

- `pair<long long, long long> do_bfs(const long long table_size, const long long thread_count)`: Изпълнява BFS на сървъра.
- `int main()`: Основната функция на сървъра.