1 Пул потоков

1.1 Простой пул потоков

Реализовать пул потоков, позволяющий выполнять заданную функцию параллельно в нескольких потоках. Номер потока и общее количество потоков передается в функцию в качестве аргументов. Класс имеет следующий прототип.

```
struct Thread_pool {

    /// Creates a pull with @nthreads number of threads.
    explicit Thread_pool(int nthreads) {}

    /// Calls function @f in each thread passing
    /// thread number and total number of threads
    /// as arguments: f(thread_no, nthreads).
    template<class Func>
    void run(Func f) {}

    /// Waits for all threads to finish.
    void wait() {}
};
```

Шаблон кода находится в папке `/mnt/root/hpc/01_thread_pool`. Доступ к ant.apmath.spbu.ru осуществляется по протоколу SSH (см. пункт 4.1), а сборка проекта – с помощью инструментов Autotools (см. пункт 4.2).

1.2 Базовые параллельные операции над массивами

Реализовать параллельные версии операций *map* и *reduce*, используя созданный пул потоков. Каждый поток должен обрабатывать отдельную часть массива.

```
template<class Container, class Func>
void map(Container& cnt, Func f) {
        Thread_pool pool(2);
        pool.run([&cnt,f] (int rank, int nthreads) {
            // parallel map code
        });
        pool.wait();
}

template<class Container, class Func>
void reduce(Container& cnt, Func f) {
        Thread_pool pool(2);
        pool.run([&cnt,f] (int rank, int nthreads) {
            // reduce code
        });
        pool.wait();
}
```

Для синхронизации потоков внутри последовательного цикла можно воспользоваться следующей функцией.

```
void sync_threads(int rank, int nthreads) {
    static std::atomic<int> counter(0);
    static volatile bool sense = false;
    bool old_sense = sense;
    int old = counter++;
    if (old + 1 == nthreads) {
        counter = 0;
        sense = !old_sense;
    } else while (sense == old_sense);
}
```

1.3 Среднее и дисперсия выборки

С помощью полученных функций реализуйте алгоритм вычисления среднего и дисперсии выборки. Найдите такой размер массива, при котором параллельный алгоритм дает ускорение (хотя бы на 2-х потоках) по сравнению с последовательным алгоритмом. Проверьте, сохраняется ли это ускорение на большем количестве потоков. Измерить время можно с помощью команды *time <program>*. Для получения более точных показателей вызов функций можно формально произвести в цикле. Для запуска на отдельном узле можно воспользоваться командой *hpc-shell*, которая выделяет 4 свободных ядра на одном из узлов кластера и заходит на этот узел.

```
template<class Container>
float mean(Container& cnt) {
    // calls to map/reduce
}

template<class Container>
float variance(Container& cnt) {
    // calls to map/reduce
}
```

2 Технология OpenMP

2.1 Параллельные циклы

Перепишите функции для вычисления среднего и дисперсии выборки с помощью директив OpenMP и протестируйте его производительность (на 1 и 2 потоках). Шаблон кода находится в папке `/mnt/root/hpc/02_openmp`.

3 Пул потоков с очередями

3.1 Пул потоков с одной очередью

Реализуйте пул потоков с общей для всех потоков очередью. В очередь на исполнение ставятся актеры – объекты, управляющие вычислениями. Каждый поток в цикле обрабатывает актеров по мере их поступления в очередь, переходя в режим ожидания при их отсутствии. Добавление актеров в очередь может осуществляться из *любого* потока. Прототип класса имеет следующий вид.

```
struct Actor {
    virtual void act() = 0;
};

struct Thread_pool {

    /// Creates a pool with @nthreads number of threads.
    explicit Thread_pool(int nthreads
        = std::thread::hardware_concurrency()) {}

    /// Places actor @a to execution queue.
    void submit(Actor* a) {}

    /// Waits for all threads to finish.
    void wait() {}

    /// Stops processing of actors.
    void stop() {}

private:
    std::queue<Actor*> actors;
};
```

3.2 Пул потоков с несколькими очередями

Реализуйте пул потоков с отдельной очередью для каждого потока. Перед отправкой актера на исполнение необходимо определить, в какую очередь лучше всего его направить. Для этого нужно реализовать *функцию распределения*, которая ставит в соответсвие каждому актеру номер очереди. Чтобы оправдать использование функции, она должна быть достаточно простой, поэтому для взаимного исключения эффективнее всего использовать атомарные операции или циклическую блокировку (*Spin_mutex*). Распределение актеров по очередям должно быть близким к равномерному. Пул с несколькими очередями проще всего реализовать через уже имеющийся класс для одной очереди, как показано ниже.

```
struct Super_thread_pool {

    /// Creates a pool with @nthreads number of threads.
    explicit Super_thread_pool(int nthreads
        = std::thread::hardware_concurrency()) {}

    /// Places actor @a to execution queue.
    void submit(Actor* a) {}

    /// Waits for all threads to finish.
    void wait() {}

    /// Stops processing of actors.
    void stop() {}

private:
    std::vector<Thread_pool*> pools;
};
```

3.3 Иерархия актеров

Для того чтобы получившуюся систему было удобно использовать, необходимо добавить возможность создания дочерних актеров и возможность сбора результатов их работы. Реализуйте эти две возможности с помощью следующего прототипа.

```
enum struct Result { UNDEFINED, DONE };

struct Actor {

    Actor(): parent(nullptr) {}

    /// Performs some computation.
    virtual void act() = 0;

    /// Collects results from @child actor.
    virtual void react(Actor* child) = 0;

private:

    void done() { result = Result::DONE; }

    Actor* parent;
    Result result = Result::UNDEFINED;
};
```

Поле *parent* должно инициализироваться при создании каждого актера указателем на актера, в методе которого он создается. Таким образом, все актеры, кроме самого первого имеют родителей. Поле *result* должно устанавливаться в значение *DONE* только после того, как все дочерние актеры завершили свою работу. Таким образом актеры, не имеющие дочерних завершают свою работу после выполнения метода *act()*. После завершения работы у актера-родителя вызывается метод *react()*, где в качестве аргумента передается дочерний актер, завершивший работу; затем этот актер удаляется. Если актера-родителя не существует (или, что то же самое, самый первый актер завершил свою работу), то программа завершается. Вызов метода *react()* должен происходить в одном потоке для дочерних актеров одного и того жеродителя.

4 Дополнительные сведения

4.1 Вход по SSH

Подсоединиться к машине можно либо с помощью командной строки, введя *ssh <имя пользователя>@ant.apmath.spbu.ru*, либо с помощью программы *Putty*, которую можно бесплатно скачать в интернете.

4.2 Сборка проекта

Для того чтобы начать работу над программой, необходимо скопировать исходный код в домашнюю директорию и скофигурировать проект с помощью следующих команд.

```
cp -r /mnt/root/hpc/01_thread_pool ~/
cd 01_thread_pool
autoreconf --install
./configure       # конфигурация
make              # сборка
./src/thread_pool # запуск программы
```

Далее для повторной сборки проекта достаточно набрать команду *make*.

4.3 Справочные сведения

Полная информация о прототипах функций, классах и методах стандартной библиотеки C++ доступна на множестве интернет-сайтов. Наиболее полная информация содержится на `http://www.open-std.org` и в более доступном виде на `http://www.cplusplus.com`.