Введение

При выполнении работы учтите, что программа дает корректные результаты, но очень медленно. Очистка кода и применение некоторых небольших улучшений возможно приведут к существенному росту производительности. Ваша задача использовать навыки параллелизма для ускорения программ.

Не применяйте векторизацию и мультипоточную параллелизацию, или оптимизацию для специфичного аппаратного обеспечения. Используйте стандартный язык C. Более того, не используйте встроенные библиотечные функции. Код должен быть портативным и независимым от типа вычислителя.

Структура кода

Программа которую вам нужно ускорить находятся в папке `everybit`. Отслеживание времени выполнения и тестирование находится в `main.c`, которая вызывает все необходимые функции. Не модифицируйте `main.c`. Также, не меняйте и не удаляйте функции, которые запускаются в `main.c`. В общем, стандартный `main.c` должен всегда компилировать ваш код и выполняться корректно!

Сборка проекта

Вы можете собрать проект запустив

```
make
```

Учтите что для сборки проекта с символами отладки нужно собирать проект с

```
make DEBUG=1
```

После сборки, можете, например, запустить бинарный файл проекта everybit командой

```
srun ./everybit
```

Где srun добавить задачу в очередь на кластере. При выполнении на локальной машине команду `srun` стоит пропустить. Бинарные файлы требуют набор параметров для запуска. Если вы ошиблись с параметром, программа выдаст инструкцию по использованию.

Тестирование

Программа покрыта тестами, но не полностью. Если вы знаете что ваша программа успешно проходит весь набор тестов, но имеет ошибку - это хороший повод добавить ещё один тест в набор.

Вы можете запустить тестирование для проекта запустив команду в директории проекта

```
make test
```

Для добавления дополнительных тестов добавьте соответствующий код в файл `tests.c`.
Если ваши тесты требуют дополнительных файлов для работы, добавьте их в папку `tests/`.

Описание задач

Много программ работают с учетом жестких ограничений, и соответственно требуют высокой производительности заложенных в них алгоритмов. Предположим, что рассматриваемая программа выполняется на носимых устройствах - смартфонах. Программа производит операции с битовыми строками в большом буфере данных. Так как под буфер сознательно выделены все возможные ресурсы, только маленькая часть памяти доступна для работы

служебных функций. В частности, ваша реализация должна использовать только константный размер памяти для словаря, вне зависимости от размера буфера данных и других параметров, таких как количество операций.

Рассмотрим `bitarray.h` и `bitarray.c`. В этом коде описаны функции нужные для размещения, доступа и обработки больших строк битов с использованием минимальной памяти. В реализации, биты запакованы по 8 шт в байте памяти, но к ним может быть получен индивидуальный доступ через публичные функции `bitarray_get()` и `bitarray_set()`.

Ваша задача заключается в ускорении функций `bitarray_rotate()` и `bitarray_count_flips()`. Для решения задачи, не используйте `malloc()` или другие функции управления памятью, и не вызывайте `bitarray_new()` из ваших функций. Вы можете разместить небольшие буферы в стеке или в BSS памяти (то есть глобальные массивы).

Ваша реализация `bitarray_rotate()` и `bitarray_count_flips()` будет верной, если содержимое массива битов, полученное через `bitarray_get_bit_sz()` и `bitarray_get()` не изменится по сравнению с реализацией данной по умолчанию.

Сдвиг битов

Функция `bitarray_rotate()` сдвигает строку битов в массиве определенной длины на определенное расстояние влево либо вправо. Смотрите документацию `bitarray.h`. Реализация по умолчанию медленная, однако, она выполняет множество однобитовых сдвигов снова и снова пока не достигнет нужного результата. Если вы запустите

```
./everybit -r
```

вы увидите что несколько сдвигов даже с небольшим буфером могут потребовать для выполнения несколько минут. Ваша задача сделать более эффективную реализацию `bitarray_rotate()`, согласно правилам указанным выше. Ваше объяснение должно максимально четко описывать как работает ваша реализация.

Самый простой подход - выполнить `k`-битный сдвиг влево по строке длинной `n`. Но так как требуется использовать только константный объем памяти, можно сохранять нулевой бит, потом копировать `k`-тый бит в 0, `2k`-тый в `k`-тый, `3k`-тый в `2k`-тый, и т.д., пока не измените все `n` бит и не вернетесь к 0-му, с

которого начали. Если `k` и `n` относительно простые, потребуется один цикл. Иначе код будет более сложным.

Есть более хитрый способ, однако, он требует сдвигать каждый бит дважды, но его проще запрограммировать. Строка, которую требуется сдвинуть, будет формы `ab`, где `a` и `b` - битовые строки и `a` имеет длину `k`. Мы хотим преобразовать `ab` в `ba`. Заметим равенство `R(R(a)R(B))=ba`, где `R()` - операция которая переворачивает строку. Операция “переворота” может быть выполнена используя память константной длины. Таким образом, строка может быть перевернута 3 переворотами.

Подсчет числа переворотов битов

Функция `bitarray_count_flips()` подсчитывает число изменений соседних битов от 0 к 1 и наоборот в массиве битов. Например, последовательность битов 0000 не имеет переходов (ответ 0), последовательность 0001 имеет один переход, а последовательность 0010 имеет два перехода (смотри документацию в `bitarray.h`). Как вы можете видеть, стандартная реализация очень медленная. Вы можете запустить

```
./everybit -f
```

и увидеть что несколько операций требуют несколько минут для выполнения при большом буфере.

Улучшите производительность `bitarray_count_flips()`, объясните как работает ваш метод.