Huge Numbers Multiplication

Huge Numbers

Huge numbers are numbers that are significantly larger than those ordinarily used. The term typically refers to large positive integers, or more generally, large positive real numbers, but it may also be used in other contexts.

Requirement

Perform the multiplication of two huge numbers.

Input Data

The input of the program will be taken from a text file. The file will contain two huge numbers on the first two lines of the file, each number on one line. Each number will be followed by a `\n` character and will not contain any characters between it's digits. The huge numbers digits will be in their usual human-readable order, the one that we got used to from mathematics. The file will have at least an empty line after the huge numbers.

Output Data

The output of the program will be the result of the multiplication of the two huge numbers. The result will be written in a text file. The output number's digits will also be in the usual human-readable order.

Sequential Implemetation

The sequantial implementation can be found here.
The program accepts two parameters. The first one is the input file and the second one is the output file (the file in which the result will be stored).
The huge numbers are kept in memory as C-like int arrays. The length of one huge number can be fount on position `0` of the array. On the next positions of the array, the digits of the number can be found, in reverse order.

- Example:For the input number `567`, the in-memory array will look like this:

0	**1**	**2**	**3**
3	7	6	5

The algorithm used to multiply the huge numbers is the classical way. Each digit `i` from the first number is multiplied by each digit `j` from the second number and added to the result on position `i + j - 1`. After this, one more iteration is done to ensure that only single digit numbers are stored on each position.

The overall complexity of the sequential implementation is:

```
O(n * m)
```

Parallel Implementation

The parallel implementation can be found here. The program accepts two parameters. The first one is the input file and the second one is the output file (the file in which the result will be stored). The huge numbers are kept in memory as C-like int arrays. The length of one huge number can be fount on position `0` of the array. On the next positions of the array, the digits of the number can be found, in reverse order.

For computing the result, the program uses the same method as the sequential implementation, exept that it equaly balances the computations that need to be done among all the threads and each thread, after computing it's calculations, adds the result to the final array by syncronizing with a mutex.

Each thread is uniquely identified by an `id` and it get's to compute the multiplications for `n / T` digits from the first number with all the digits from the second number, where `n` is the total number of digits and T is the number of threads. Each threads work looks like this:

```
    for (i = id; i <= A[0]; i += T) {
        for (j = 1; j <= B[0]; j++) {
            C[i + j - 1] += A[i] * B[j];
        }
    }
```

The number of threads is stored in the variable T, but it can be easily changed by setting the `NO_THREADS` enviroment variable to a suitable integer.

Testing

Tests generation

Test cases can be generated through a shell script that is available here. Once runned, the script will generate a bunch of test cases in a folder named tests. The script can be personalized as for how many test cases it generates and how much bigger do they get, by modifing the variables `TESTS_NUMBER` and `GROWTH_FACTOR`. The tests generator will generate for each test two huge numbers of the same digits numbers and for each test, it will also generate a file containing the correct solution.

The shell script looks like this:

```
#!/bin/bash

TESTS_NUMBER=20
```

```
GROWTH_FACTOR=100

TEST_FILENAME='tests/test'
TEST_EXTENSION='in'
ANSWER_EXTENSION='ok'

random_number () {
    num=`cat /dev/urandom | env LC_CTYPE=C tr -dc '0-9' | fold -w  | head -n 1`
    echo $num
}

mkdir -p tests

echo "Generating $TESTS_NUMBER tests..."

for i in `seq 1 $TESTS_NUMBER`;
do
    printf -v no "%02d" $i
    FILENAME=$TEST_FILENAME$no.$TEST_EXTENSION
    ANSWER_FILENAME=$TEST_FILENAME$no.$ANSWER_EXTENSION
    DIGITS_NBR=`echo $(($i * $GROWTH_FACTOR))`
    echo "Generating test: $FILENAME. The numbers have $DIGITS_NBR digits."
    nbr1=$(random_number $DIGITS_NBR)
    nbr2=$(random_number $DIGITS_NBR)
    echo $nbr1 > $FILENAME
    echo $nbr2 >> $FILENAME
    c="$(BC_LINE_LENGTH=0 bc <<< "$nbr1 * $nbr2" )"
    echo $c | tr -d "[:space:]" | tr -d "\\" > $ANSWER_FILENAME
done
```

Tests running

Tests can be run througn a shell script that is available here. By running the script, the sources for both the iterative and parallel implementation will be compiled. After that, the script will run both implementations for each test case present in the `tests/` folder and will print on the screen the time taken for each test case along with an appropriate error message if the test case result was not successful.

The shell script looks like this:

```
#!/bin/bash

TEST_FILENAME='tests/test'
TEST_FOLDER='tests/'
TEST_EXTENSION='in'
ANSWER_EXTENSION='ok'
OUTPUT_EXTENSION='out'

rm iterative
rm parallel

echo "Compiling iterative implementation"
g++ -o iterative iterative.cpp
```

```
echo "Compiling parallel implementation"
g++ -lpthread -o parallel parallel.cpp

tests=`ls $TEST_FOLDER | grep .$TEST_EXTENSION`

echo "Testing iterative implementation...\n"

for file in $tests;
do
    name=`echo "$file" | cut -f 1 -d '.'`
    TIME_S=`TIMEFORMAT=%R bash -c "time ./iterative $TEST_FOLDER$file
$name.$OUTPUT_EXTENSION"`
    DIFF=`diff $TEST_FOLDER$name.$ANSWER_EXTENSION $name.$OUTPUT_EXTENSION`
    echo "Test: $file"
    echo $TIME_S
    if [ $DIFF ]
    then
        echo "ERROR!"
    fi
done

echo "Testing parallel implementation...\n"

for file in $tests;
do
    name=`echo "$file" | cut -f 1 -d '.'`
    TIME_S=`TIMEFORMAT=%R bash -c "time ./parallel $TEST_FOLDER$file
$name.$OUTPUT_EXTENSION"`
    DIFF=`diff $TEST_FOLDER$name.$ANSWER_EXTENSION $name.$OUTPUT_EXTENSION`
    echo "Test: $file"
    echo $TIME_S
    if [ $DIFF ]
    then
        echo "ERROR!"
    fi
done
```

Conclusions

This project has helped me to better understand parallelization along with it's advantages and disadvantages. As it turns out, even though parallelization is usually the best solution for optimizations, some implementations are better off left sequential as the case for this particular implementation of huge numbers multiplication. Using the classical dummy algorithm requires a lot of simple operations and a parallel implementation can not take advantages of the processors computation power because it requires too much time for syncronizations.

Of course that there are some implementations of huge numbers multiplication that will perform better when parallelized (some examples of such implementation are Karatsuba's Algorithm or Fast Fourier transorm), but my purpose of this project was to compare the iterative and parallel implemtations of the classical dummy algorithm.

Tests results

After running both implementations on the same set of data, this was the result:

Test NO	No of digits	Sequential	Parallel with 2 threads	Parallel with 8 threads
1	100	0.003	0.029	0.042
2	200	0.004	0.128	0.152
3	300	0.004	0.292	0.333
4	400	0.004	0.510	0.598
5	500	0.004	0.801	0.936
6	600	0.005	1.127	1.358
7	700	0.005	1.499	1.764
8	800	0.005	1.993	2.363
9	900	0.006	2.581	2.898
10	1000	0.006	3.189	3.617
11	1100	0.007	3.831	4.399
12	1200	0.008	4.573	5.331
13	1300	0.008	5.461	6.172
14	1400	0.009	6.239	7.120
15	1500	0.010	7.138	8.131
16	1600	0.010	8.173	9.440
17	1700	0.011	9.237	10.680
18	1800	0.013	10.124	11.682
19	1900	0.014	11.279	13.311
20	2000	0.015	12.799	14.446

The tests were run on a 2015 15' Macbook Pro with an Inter Core i7 2.8 processor.