Project Euler: Problem 1

Brandon Caffie

July 3, 2014

Introduction

The problem under study was taken from a popular math puzzle website *Project Euler*. It involves finding the sum of all natural numbers from 1 to n that is divisible by 3 or 5.

Design

Designing the algorithm involves using multiple *STAPL* algorithms to transform the data and compute the final sum. The code shown below illustrates how to solve this problem.

```
Create array container from 1 to n;
for element ∈ container do
    if element is divisible by 3 or 5 then
        return element;
    else
        return 0;
add non-zero numbers together to achieve sum;
```

Algorithm 1: Finding the sum of numbers divisible by 3 or 5

The following *STAPL* functions are used in the program:

1. *stapl::array <unsigned long long>*
   Create an array to hold all numbers from 1 to n.
2. *stapl::iota()*
   Produces the numbers from 1 to n in counting order and stores them in the array.

3. *stapl::replace_if()*
   For numbers that aren't divisible by 3 or 5, they are set to zero. They are otherwise unaffected.
4. *stapl::accumulate()*
   Adds the numbers that are divisible by 3 and 5 together.
5. *stapl::do_once({})*
   Prints one statement displaying the sum.
6. **stapl::counting_view<unsigned long long>()*
   Creates a non-storage counting view from 1 to n. This can replace *stapl::array* and *stapl::iota* so storage won't have to be created.
7. **stapl::map_reduce()*
   Can replace *stapl::replace_if* and *stapl::accumulate* by doing both at the same time, assuming functor is the same.

Two parallel versions of this program have been developed. One is using the first 4 (1 – 4) components shown above, and the other uses only the last 2 components (6 – 7), which is a combination of the first four components.

Experimental Results

What will determine the success of executing this program in parallel is both strong scale and weak scale testing. The machine that will produce the results for this experiment is a 576 core system, locally named *RAIN*. With this many cores available, it allows the implementation of *STAPL* containers, views, and algorithms to create the partitioning and distribution of the data and work.

Strong Scale Testing

For strong testing, we will test the scalability of the program. *Scalability* is a crucial component to understand as a parallel programmer, because it ultimately determines whether or not a program is computed faster as more processors are used to execute it.

Both Table 1 and Figure 1 illustrate the program execution time based on the size of the container, from 99 to 99,999,999 elements. Each of the time outputs shown in Table 1 is the average time of 32 execution instances. Figure 1 gives a visual of Table 1.

	99	999	9999	99999	999999	9999999	99999999
1	$3.40998E{-}5$	$3.775E{-}5$	$7.44687E{-}5$	$4.11656E{-}4$	$3.80756E{-}3$	0.0375361	0.374931
2	$9.54687E{-}5$	$9.6E{-}5$	$1.14E{-}4$	$2.76719E{-}4$	$1.97178E{-}3$	0.0188286	0.187169
4	$1.31E{-}4$	$1.31219E{-}4$	$1.4E{-}4$	$2.22562E{-}4$	$1.06191E{-}3$	$9.52656E{-}3$	0.0939813
8	$1.86344E{-}4$	$1.78969E{-}4$	$1.85125E{-}4$	$2.36312E{-}4$	$7.07437E{-}4$	$5.41678E{-}3$	0.0519532
16	$2.26656E{-}4$	$2.51469E{-}4$	$2.41875E{-}4$	$2.48469E{-}4$	$4.97031E{-}4$	$2.85972E{-}3$	0.0264027
32	$3.24656E{-}4$	$3.26094E{-}4$	$3.08062E{-}4$	$3.14563E{-}4$	$4.94437E{-}4$	$1.79591E{-}3$	0.0137933
64	$4.15625E{-}4$	$3.71437E{-}4$	$3.9925E{-}4$	$4.33219E{-}4$	$4.69312E{-}4$	$1.19366E{-}3$	$7.44138E{-}3$
128	$5.5225E{-}4$	$4.97562E{-}4$	$5.16563E{-}4$	$4.47406E{-}4$	$5.815E{-}4$	$8.56031E{-}4$	$3.97394E{-}3$
256	$5.97125E{-}4$	$4.88062E{-}4$	$4.52687E{-}4$	$5.54156E{-}4$	$5.43281E{-}4$	$6.66906E{-}4$	$2.312E{-}3$
512	$6.54469E{-}4$	$5.78375E{-}4$	$5.77625E{-}4$	$5.72156E{-}4$	$5.73406E{-}4$	$6.90312E{-}4$	$1.49475E{-}3$

Table 1: Execution Time (in seconds) for n elements with p processors.

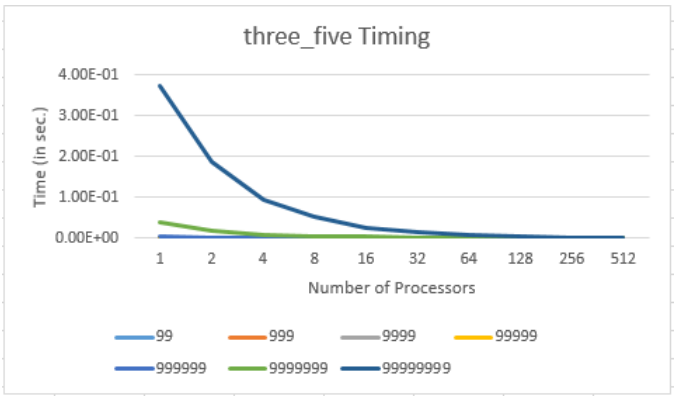


Figure 1: Graph illustrating the execution time based on number of elements.

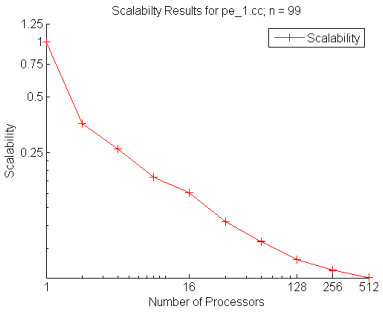
It is important to note from Table 1 that an increase of execution times as a result of processor growth is possible. This is because the amount of work being done on smaller data sizes takes less time than the communication between processors to compute the final result.

The graphs (a) – (g) represent the scalability for increasing values of n. Scalability was achieved by using the following equation:

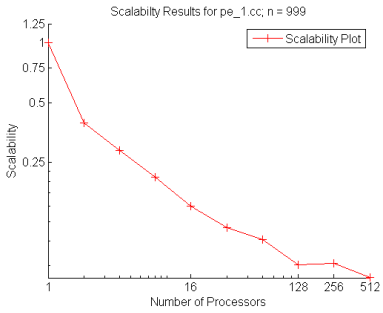
$$Scalability = \frac{time\ of\ P_1}{time\ of\ P_n}$$

where n is the number of processors used to execute the program.

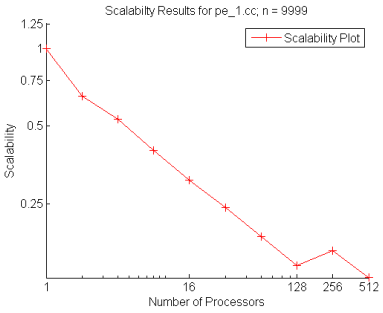
A slope of negative scalability shows that time increases as the number of processors increase. A slope of positive scalability shows that time decreases as the number of processor increases.



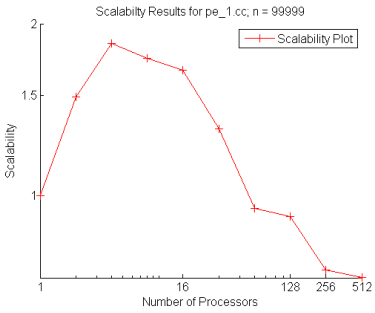
(a) $n = 99$



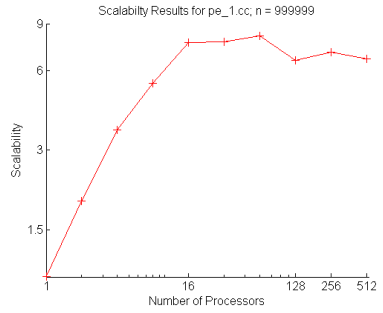
(b) $n = 999$



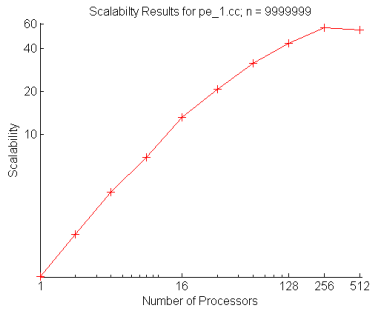
(c) $n = 9999$



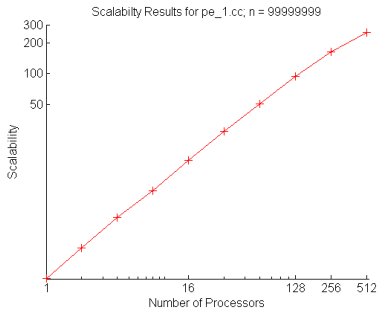
(d) $n = 99999$



(e) $n = 999999$



(f) $n = 9999999$



(g) $n = 99999999$

Weak Scale Testing

The importance of weak scale testing is to make sure that the workload for each processor used is the same as the data increases by a factor of the number of processors used. Figure 3 shows the results of weak scale testing of $n = 99{,}999{,}999$.

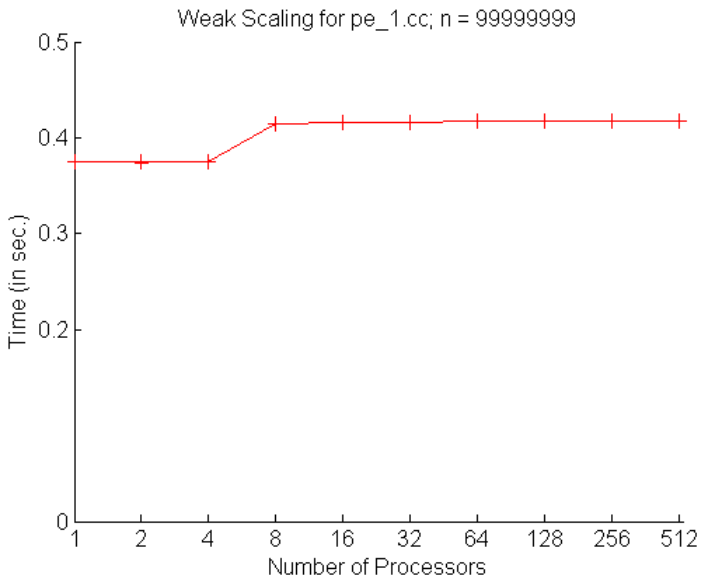


Figure 3: Graph displaying the time of n processors with weak-scale testing.

Analysis

For strong scale testing, we expected the graph to grow as a positive linear function as n increases. This is linear scalability for a parallel program. For weak scale testing, we expected the function the graph to be a constant function, exemplifying the principle of a balanced workload, no matter how huge the data.

In summary, both strong and weak scale testing seem to hold true as expected based on the results, which shows that this program has been successfully implemented in parallel. For sufficiently large data sizes the components from *STAPL* provide good parallel performance for both strong and weak scaling scenarios.