Project Euler: Problem 2

Brandon Caffie

July 9, 2014

Introduction

The problem under study was taken from a popular math puzzle website *Project Euler*. It involves finding the sum of all even Fibonacci numbers from 1 to n.

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 a Fibonacci AND even number then
        return element;
    else
        return 0;
add even Fibonacci numbers together to achieve sum;
```

Algorithm 1: Finding the sum of even Fibonacci numbers

The most challenging part of designing the algorithm for this problem was to determine whether a number was in the Fibonacci series or not. We can use the following condition to determine whether a given number N was Fibonacci:

$$\sqrt{5N^2 \pm 4}$$

If a natural number is result of the above condition, we can guarantee that N is a Fibonacci number.

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 even Fibonacci numbers, they are set to zero. They are otherwise unaffected.
4. *stapl::accumulate()*
   Adds the even Fibonacci numbers 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 900 to 900,000,000 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.

	900	9000	90000	900000	9000000	90000000	900000000
1	$7.11875E{-5}$	$3.99469E{-4}$	$3.67478E{-3}$	0.036296	0.361639	3.61579	36.4298
2	$1.20344E{-4}$	$3.59531E{-4}$	$2.74881E{-3}$	0.0265592	0.264558	2.64559	26.4449
4	$1.40906E{-4}$	$2.6025E{-4}$	$1.45278E{-3}$	0.0133637	0.132366	1.32238	13.2225
8	$2.01563E{-4}$	$2.67437E{-4}$	$8.91094E{-4}$	$7.2535E{-3}$	0.0697762	0.696138	6.95948
16	$2.29781E{-4}$	$2.79344E{-4}$	$6.04031E{-4}$	$3.87388E{-3}$	0.035219	0.348481	3.48138
32	$3.28938E{-4}$	$3.40969E{-4}$	$5.53625E{-4}$	$2.46E{-3}$	0.0212311	0.210363	2.13763
64	$3.95281E{-4}$	$4.25375E{-4}$	$5.07094E{-4}$	$1.43887E{-3}$	$9.25747E{-3}$	0.0876315	0.870599
128	$4.34219E{-4}$	$4.48125E{-4}$	$5.0625E{-4}$	$8.9775E{-4}$	$4.93769E{-3}$	0.0440985	0.436699
256	$5.84562E{-4}$	$5.78094E{-4}$	$6.10188E{-4}$	$7.72375E{-4}$	$3.04347E{-3}$	0.0224767	0.218105
512	$5.93406E{-4}$	$6.08313E{-4}$	$5.81812E{-4}$	$6.44938E{-4}$	$1.73441E{-3}$	0.0114888	0.109439

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

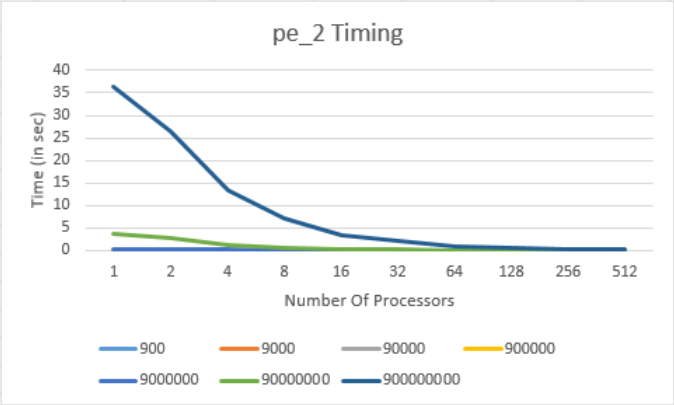
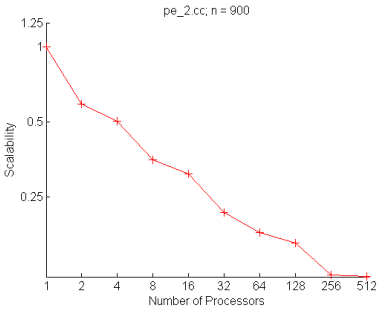


Figure 1: Graph illustrating the execution time based on number of elements.

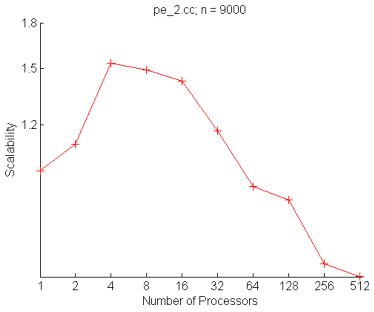
The graphs (a) – (g) represent the scalability for increasing values of n. Scalability was achieved by using the following equation:

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

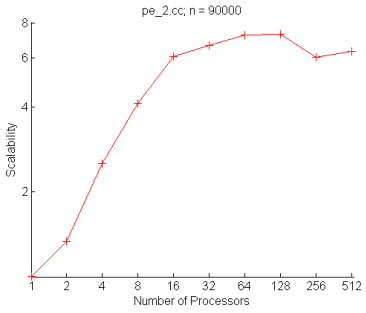
where n is the number of processors used to execute the program.



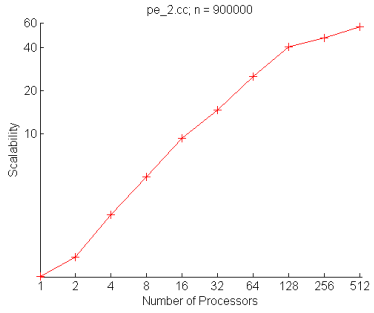
(a) $n = 900$



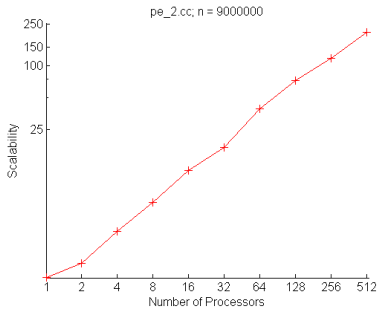
(b) $n = 9000$



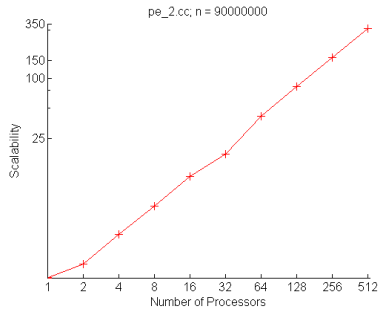
(c) $n = 90000$



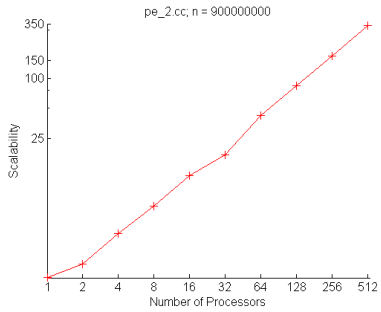
(d) $n = 900000$



(e) $n = 9000000$



(f) $n = 90000000$



(g) $n = 900000000$

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 $n = 900{,}000{,}000$.

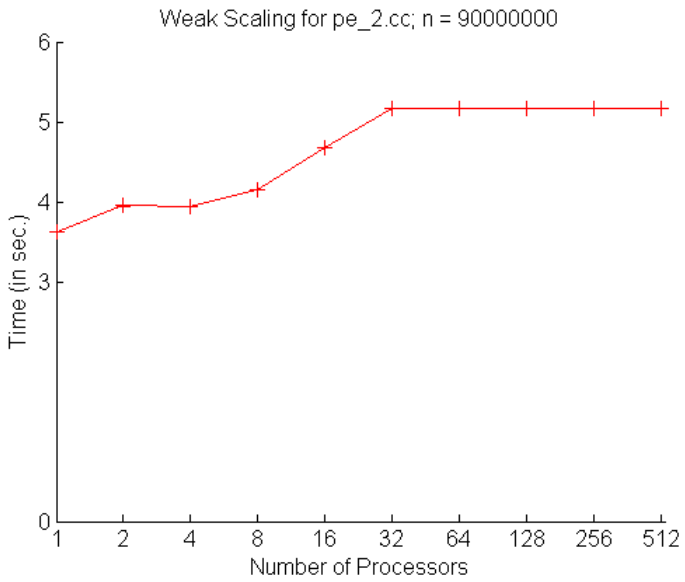


Figure 3: Graph displaying the time of n processors with weak-scale testing.

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 graph to be a constant function, exemplifying the principle of a balanced workload, no matter how huge the data. It is important to note the large increase in time from $1 - 32$ processors in Figure 3 for weak-scale testing. The reason for this is the memory contention of the compute node in which the program was executed.

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.