Project Euler Problem 3: Largest Prime Factor

Milan Hanus

May 5, 2015

Problem statement

The prime factors of 13195 are 5, 7, 13 and 29. What is the largest prime factor of the number 600851475143? [1]

Algorithm

Let n be an integer. The algorithm for finding largest prime factor of n (LPF(n)), i.e. the largest prime number that divides n with zero remainder, can be expressed by the pseudocode shown below.

Algorithm 1: Largest prime factor of an integer

```
input  : n
output: LPF(n)
container ← sequence from 2 to √n;
foreach element ∈ container do
    x ← element;
    if n mod x = 0 then                          // x is factor of n
        if is_prime(n/x) then
            element ← n/x;
        else
            if not is_prime(x) then element ← 0;
        end
    else
        element ← 0;
    end
end
LPF(n) ← max(container);
```

Since for any $x > \sqrt{n}$,

$$\frac{n}{x} < \frac{n}{\sqrt{n}} = \sqrt{n}, \tag{1}$$

Algorithm 2: `is_prime`

input : x
output: **True** if x is a prime number, **False** otherwise

for $i \leftarrow 2$ **to** $\sqrt{k}$ **do**
  if $x \bmod i = 0$ **then**
    return False;
  end
end
return True;

we only need to test if all integers less than or equal to $\sqrt{n}$ are factors of n. Hence the main algorithm (Algorithm 1) operates on a sequence of integers from 2 only up to $\sqrt{n}$. If x from this sequence is a factor of n, a test is made whether $\frac{n}{x}$ (which is greater than x) is prime and if it is, it replaces the original element x in the sequence. If neither x nor $\frac{n}{x}$ are prime or integer factors of n, x is replaced by zero in the container. Note that property (1) is also being used in the test for primality in Algorithm 2.

STAPL components

The following components of STAPL have been used in `pe_3a.cc` to implement the algorithms described in previous section.

- stapl :: counter<stapl::default_timer>
  Used to measure computation time.
- stapl :: array <size_t>
  An array container.
- stapl :: make_array_view()
  Creates a view into the container; used to access and manipulate values stored in the container.
- stapl :: iota ()
  Populates given container (view) by a sequence of increasing numbers.
- stapl :: transform()
  Transforms values from the container according to given functor and stores the results in another (possibly the same) container.
- stapl :: do_once()
  Ensures that given function will be run only once (on one location); used to print out results.
- stapl :: max_value()
  Computes the largest value in the container.

A more efficient implementation is also possible that doesn't require storage of the testing sequence (using stapl::counting_view() instead of stapl::iota()) and that also replaces calls to stapl::transform() and stapl::max_value() by a single *map-reduce* operation (stapl::map_reduce()). This implementation is provided in file `pe_3b.cc` and has been used for all performance tests that follow.

Performance Tests

Strong Scaling

Table 1 and Figure 1 show the average computation times for 4 different fixed input numbers as the number of processors increases (dashed line represents optimal linear scaling). Note that the size of the problem: $N = \sqrt{n} - 1$ is used in the tables and figures rather than the input number n itself as it is more relevant to studying parallel performance. All experiments were compiled using GNU g++ 4.9.2 with -O3 optimization level, using Boost 1.56, and performed on a Cray XE6m machine with 24 nodes, each with 16 or 32 cores and 2GB of memory per core. The system has a total of 576 cores.

Processor count	Problem size (N)			
	$10^3 - 1$	$10^5 - 1$	$10^7 - 1$	$10^9 - 1$
1	0.0002266	0.002103	0.24227	29.6167
2	0.0004761	0.001389	0.125808	15.0213
4	0.000527	0.001025	0.063322	7.51788
8	0.0006547	0.000888	0.032117	3.75981
16	0.000734	0.000848	0.016431	1.87995
32	0.0009209	0.000972	0.008702	0.9397
64	0.0011362	0.001172	0.004981	0.472608
128	0.0012171	0.001326	0.003069	0.236059
256	0.0015139	0.001344	0.002951	0.119351
512	0.0019396	0.003191	0.002049	0.060777

Table 1: Strong scaling timings (average of 10 runs).

Almost perfect linear scalability up to 128 and 512 processors, respectively, has been achieved with problem sizes of orders 10^7 and 10^9, while communication overhead, increasing as more processors are being employed, is significant for lower problem sizes.

The fraction of linear scalability achieved for given processor count can be quantified by

$$E_p = \frac{T_1}{pT_p},$$

where T_p is the computation time when using p processors. The results are shown in Figure 2.

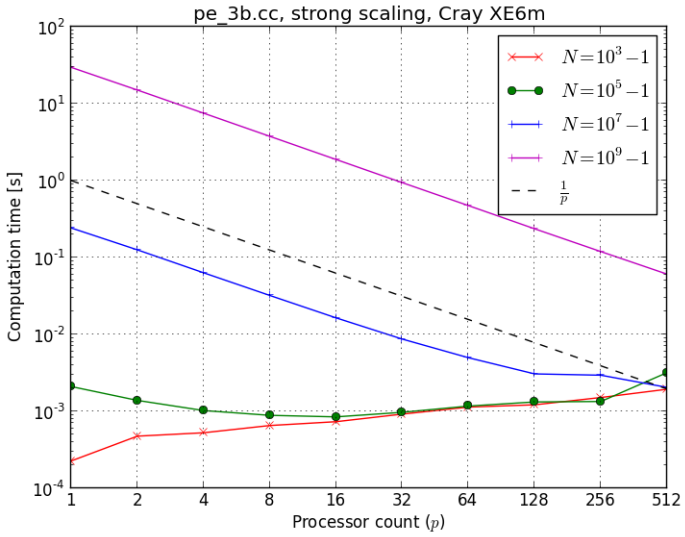

Figure 1: Strong scaling timings (average of 10 runs).

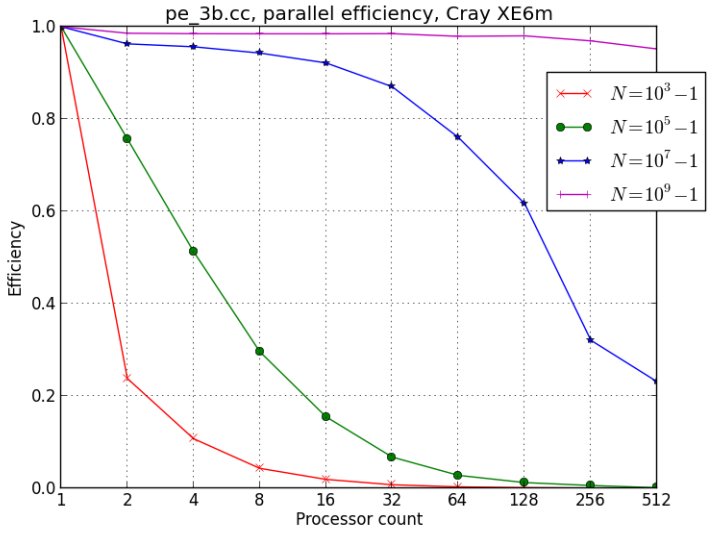

Figure 2: Parallel efficiency.

Weak Scaling

For the weak scaling study, the input number n was multiplied by the square of the number of processors assigned to solve the problem, so that the problem

size per processor was fixed (10^6). The results are given in Table 2 and Figure 3.

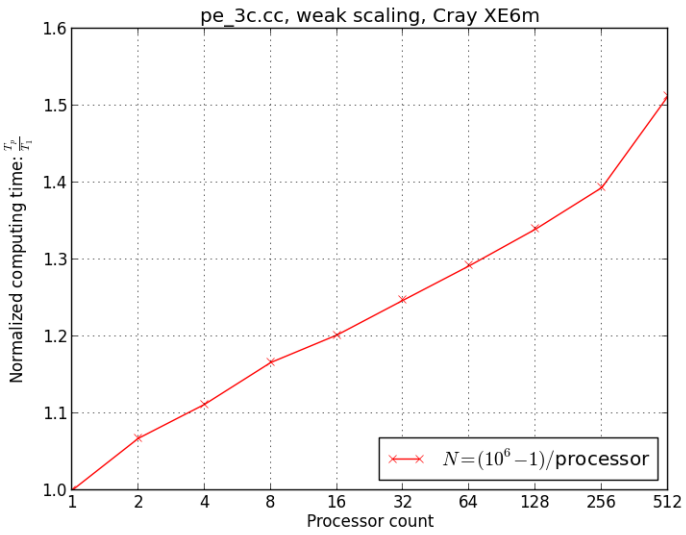

Figure 3: Weak scaling timing

We notice that optimal weak scaling (that would be indicated by a constant function with values near 1.0 in Figure 3) is not achieved by the algorithm. This is caused primarily by the test for primality of each integral factor of n, which is a significant sequential (i.e., not scalable) portion of the algorithm whose size grows with the size of the problem.

Processor count	N/processor 10^6-1
1	0.0215162
2	0.0229693
4	0.0239192
8	0.0250902
16	0.0258590
32	0.0268329
64	0.0277967
128	0.0288213
256	0.0299821
512	0.0325425

Table 2: Weak scaling timings (average of 10 runs).

References

[1] Project Euler website. `https://projecteuler.net/problem=3`