Fastest way to calculate primes in C#?

Asked 16 years, 3 months ago Modified 9 years, 5 months ago Viewed 11k times

14

I actually have an answer to my question but it is not parallelized so I am interested in ways to improve the algorithm. Anyway it might be useful as-is for some people.

```
int Until = 20000000;
BitArray PrimeBits = new BitArray(Until, true);

/*
 * Sieve of Eratosthenes
 * PrimeBits is a simple BitArray where all bit is an integer
 * and we mark composite numbers as false
 */

PrimeBits.Set(0, false); // You don't actually need this, just
PrimeBits.Set(1, false); // remindig you that 2 is the smallest prime

for (int P = 2; P < (int)Math.Sqrt(Until) + 1; P++)
    if (PrimeBits.Get(P))
        // These are going to be the multiples of P if it is a prime
        for (int PMultiply = P * 2; PMultiply < Until; PMultiply += P)
            PrimeBits.Set(PMultiply, false);

// We use this to store the actual prime numbers
List<int> Primes = new List<int>();

for (int i = 2; i < Until; i++)
    if (PrimeBits.Get(i))
        Primes.Add(i);
```

Maybe I could use multiple `BitArray`s and BitArray.And() them together?

c# .net performance algorithm bitarray

Share

Improve this question

Follow

edited Dec 29, 2008 at 22:26

Mark Cidade

99.8k ● 33 ● 229 ● 237

asked Aug 27, 2008 at 18:54

palotasb

4,648 ● 3 ● 29 ● 35

1 The fastest way I know of using multiprocessing in C# is the code I submitted as an answer to another question at: stackoverflow.com/a/18885065/549617. It con find the total number of primes to one billion in about 0.32 seconds, the number of primes in the 32-bit number range in about 1.29 seconds, and the number of primes to ten billion in about 3 seconds **not** using enumeration on an Intel i7-2700K (3.5 GHz with four cores/eight threads including Hyper Threading). To give results faster than this, one would have to use C code as in code.google.com/p/primesieve. – GordonBGood Dec 13, 2013 at 9:58

I tried the solution above and I get an exception: 'arithmetic operation resulted in an overflow'. The List<int> should be List<Int64>. – Devid May 10, 2014 at 11:40

There is no need for the "+ 1" following your square root. If it happens to round down, rounding up instead would produce a result that is higher than your test number. – Tyler Pantuso Dec 18, 2015 at 2:26

8 Answers

Sorted by: Highest score (default)

8

You might save some time by cross-referencing your bit array with a doubly-linked list, so you can more quickly advance to the next prime.

Also, in eliminating later composites once you hit a new prime p for the first time - the first composite multiple of p remaining will be p*p, since everything before that has already been eliminated. In fact, you only need to multiply p by all the remaining potential primes that are left after it in the list, stopping as soon as your product is out of range (larger than Until).

There are also some good probabilistic algorithms out there, such as the Miller-Rabin test. The wikipedia page is a good introduction.

Share Improve this answer Follow

answered Aug 29, 2008 at 9:08

Tyler

28.9k ● 12 ● 93 ● 108

4

Parallelisation aside, you don't want to be calculating sqrt(Until) on every iteration. You also can assume multiples of 2, 3 and 5 and only calculate for N%6 in {1,5} or N%30 in {1,7,11,13,17,19,23,29}.

You should be able to parallelize the factoring algorithm quite easily, since the Nth stage only depends on the sqrt(n)th result, so after a while there won't be any conflicts. But that's not a good algorithm, since it requires lots of division.

You should also be able to parallelize the sieve algorithms, if you have writer work packets which are guaranteed to complete before a read. Mostly the writers shouldn't conflict with the reader - at least once you've done a few entries, they should be working at least N above the reader, so you only need a synchronized read fairly occasionally (when N exceeds the last synchronized read value). You shouldn't need to synchronize the bool array across any number of writer threads, since write conflicts don't arise (at worst, more than one thread will write a true to the same place).

The main issue would be to ensure that any worker being waited on to write has completed. In C++ you'd use a compare-and-set to switch to the worker which is being waited for at any point. I'm not a C# wonk so don't know how to do it that language, but the Win32 InterlockedCompareExchange function should be available.

You also might try an actor based approach, since that way you can schedule the actors working with the lowest values, which may be easier to guarantee that you're reading valid parts of the the sieve without having to lock the bus on each increment of N.

Either way, you have to ensure that all workers have got above entry N before you read it, and the cost of doing that is where the trade-off between parallel and serial is made.

Share Improve this answer Follow

answered Aug 27, 2008 at 20:52

Pete Kirkham

49.3k ● 5 ● 94 ● 173

1

> Without profiling we cannot tell which bit of the program needs optimizing.

If you were in a large system, then one would use a profiler to find that the prime number generator is the part that needs optimizing.

Profiling a loop with a dozen or so instructions in it is not usually worth while - the overhead of the profiler is significant compared to the loop body, and about the only ways to improve a loop that small is to change the algorithm to do fewer iterations. So IME, once you've eliminated any expensive functions and have a known target of a few lines of simple code, you're better off changing the algorithm and timing an end-to-end run than trying to improve the code by instruction level profiling.

Share Improve this answer Follow

answered Sep 4, 2008 at 20:38

Pete Kirkham

49.3k ● 5 ● 94 ● 173

0

@DrPizza Profiling only really helps improve an implementation, it doesn't reveal opportunities for parallel execution, or suggest better algorithms (unless you've experience to the otherwise, in which case I'd really like to see your profiler).

I've only single core machines at home, but ran a Java equivalent of your BitArray sieve, and a single threaded version of the inversion of the sieve - holding the marking primes in an array, and using a wheel to reduce the search space by a factor of five, then marking a bit array in increments of the wheel using each marking prime. It also

reduces storage to O(sqrt(N)) instead of O(N), which helps both in terms of the largest N, paging, and bandwidth.

For medium values of N (1e8 to 1e12), the primes up to sqrt(N) can be found quite quickly, and after that you should be able to parallelise the subsequent search on the CPU quite easily. On my single core machine, the wheel approach finds primes up to 1e9 in 28s, whereas your sieve (after moving the sqrt out of the loop) takes 86s - the improvement is due to the wheel; the inversion means you can handle N larger than 2^32 but makes it slower. Code can be found here. You could parallelise the output of the results from the naive sieve after you go past sqrt(N) too, as the bit array is not modified after that point; but once you are dealing with N large enough for it to matter the array size is too big for ints.

Share Improve this answer Follow

answered Sep 1, 2008 at 0:06

Pete Kirkham
49.3k ● 5 ● 94 ● 173

0

You also should consider a possible change of algorithms.

Consider that it may be cheaper to simply add the elements to your list, as you find them.

Perhaps preallocating space for your list, will make it cheaper to build/populate.

Share Improve this answer Follow

answered Sep 17, 2008 at 1:43

EvilTeach
28.8k ● 21 ● 88 ● 144

0

Are you trying to find new primes? This may sound stupid, but you might be able to load up some sort of a data structure with known primes. I am sure someone out there has a list. It might be a much easier problem to find existing numbers that calculate new ones.

You might also look at Microsofts Parallel FX Library for making your existing code multi-threaded to take advantage of multi-core systems. With minimal code changes you can make you for loops multi-threaded.

Share Improve this answer Follow

answered Sep 17, 2008 at 1:57

Jason Jackson
17.2k ● 8 ● 51 ● 75

0

There's a very good article about the Sieve of Eratosthenes: [The Genuine Sieve of Eratosthenes]

It's in a functional setting, but most of the opimization do also apply to a procedural implementation in C#.

The two most important optimizations are to start crossing out at P^2 instead of 2*P and to use a wheel for the next prime numbers.

For concurrency, you can process all numbers till P^2 in parallel to P without doing any unnecessary work.

Share Improve this answer Follow

answered Sep 17, 2008 at 7:15

Thomas Danecker

4,685 ● 4 ● 34 ● 31

-1

```
void PrimeNumber(long number)
{
    bool IsprimeNumber = true;
    long  value = Convert.ToInt32(Math.Sqrt(number));
    if (number % 2 == 0)
    {
        IsprimeNumber = false;
        MessageBox.Show("No It is not a Prime NUmber");
        return;
    }
    for (long i = 3; i <= value; i=i+2)
    {
       if (number % i == 0)
        {

            MessageBox.Show("It is divisible by" + i);
            IsprimeNumber = false;
            break;
        }

    }
    if (IsprimeNumber)
    {
        MessageBox.Show("Yes Prime NUmber");
    }
    else
    {
        MessageBox.Show("No It is not a Prime NUmber");
    }
}
```

Share Improve this answer Follow

answered Jul 24, 2015 at 7:16

Raj

1 • 3