notes on implementing fast exhaustive search

Charles Bouillaguet

January 29, 2021

Algo. 1 shows most generic code for quadratic equations ($d = 2$).

Algorithm 1 Starting Point

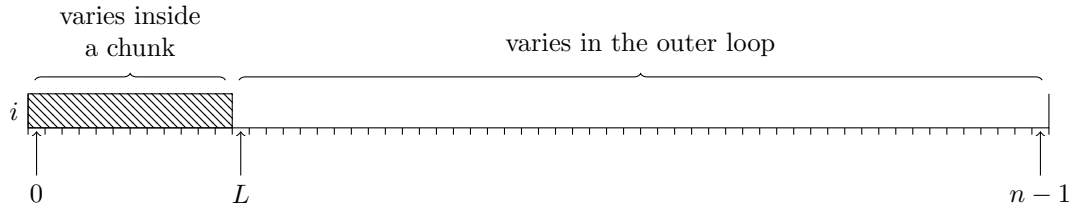
```
1: procedure ZEROES(f)
2:     State ← INIT^[d](f, 0, 0)
3:     if ȳ = 0 then report f(0) = 0
4:     for i from 1 to 2^n − 1 do
5:         let k_1 = b_1(i) in
6:         let k_2 = b_2(i) in
7:         if k_2 ≠ ⊥ then D[k_1] ← D[k_1] ⊕ D[k_1, k_2]
8:         ȳ ← ȳ ⊕ D[k_1]
9:         if ȳ = 0 then report f(GRAYCODE(i)) = 0
```

Here, $b_i(i)$ returns the index of the i-th bit of the integer i set to one. For instance, $b_1(1) = 0$, $b_1(8) = 3$ and $b_1(10) = 1$. When the hamming weight of i is less than k, then $b_k(i)$ is not defined, and returns a special value $\perp$.

To be somewhat efficient, this loop must be unrolled, and the conditional instructions must be removed as much as possible. This is not very difficult.

Removing the "IF". The annoying case $k_2 = \perp$ only happens when i is a power of two. Pour avoid it, we split the main loop into n "macro-steps". This results in Algo. 2.

Unrolling. Now, we must unroll the loop. This means that we will write down a piece of code that implements a fixed number of STEPs, say 2^L, and that we will wrap the required machinery around it. The best value of L should be determined experimentally (larger values reduced the overhead of the loop, but may result in a piece of code that does not fit in the cache. Values around 8,9 or 10 seem reasonable). In the sequel, I will call a block of 2^L STEPs a CHUNK. If we look back to Algo. 1, then the i index will be enumerated as follows :



Note that the first L macro-steps [should not/cannot] be unrolled, because there do not form a complete CHUNK. It makes sense to put a lot of effort into optimizing a single chunk, as for

Algorithm 2 Without testing k_2

```
1: procedure STEP(x)
2:     let k_1 = b_1(x) in
3:     let k_2 = b_2(x) in
4:     D[k_1] ← D[k_1] ⊕ D[k_1, k_2]
5:     y ← y ⊕ D[k_1]
6:     if y = 0 then report f(GRAYCODE(x)) = 0

7: procedure STEPS(x, y)
8:     for i from x to y − 1 do
9:         STEP(i)

10: procedure ZEROES(f)
11:     State ← INIT^[d](f, 0, 0)
12:     if y = 0 then report f(0) = 0
13:     for j from 0 to n − 1 do // j-th macro-step
14:         y ← y ⊕ D[j]
15:         if y = 0 then report f(GRAYCODE(2^j)) = 0
16:         // Now we know that b_2(...) ≠ ⊥
17:         STEPS(2^j + 1, 2^{j+1})
```

large values of n, most of the running time will be spent inside chunks. The k-th chunk of the j-th macro-step looks like this:

```
procedure CHUNK(j, k)
    STEP(2^j + k × 2^L)
    STEP(2^j + k × 2^L + 1)
    STEP(2^j + k × 2^L + 2)
    ⋮
    STEP(2^j + k × 2^L + 2^L − 1)
```

In most steps inside a chunk, the values of k_1 and k_2 can be known in advance. More specifically, inside the ℓ-th step, k_1 depends only on ℓ is $\ell \neq 0$, and k_2 depends only on ℓ if ℓ is not a power of two.

Algorithm 3 is the basis of my plain-C implementation.

1 Parallelism

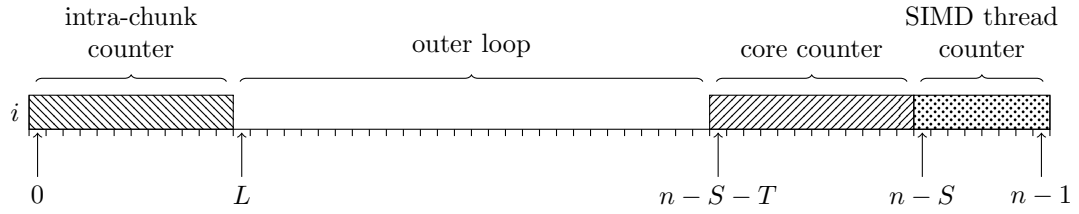
We will try to address the situation where several SIMD units are available. This situation is typical of multi-core CPUs (modern Opterons have 16 cores, each with its own 4-way 32-bit SIMD unit), but also corresponds to GPUs (A GTX 295 has 30 SIMD units, and each one is 16-way).

We must then write "SIMD chunks", i.e. chunks that execute 2^{L+S} steps using a 2^S-way SIMD unit. There are several possible ways to do this. My own guess is that we may have interest run on the same SIMD unit the steps whose number share as many lowest-significant bits as possible :

Algorithm 3 Split in chunks of size 2^L

```
1: procedure CHUNK(j, k)
2:     STEPS(2^j + k × 2^L, 2^j + (k+1) × 2^L)

3: procedure ZEROES(f)
4:     State ← INIT^[d](f, 0, 0)
5:     if ỹ = 0 then report f(0) = 0
6:     // Deal with the small macro-steps
7:     for j from 0 to min(n, L) do
8:         ỹ ← ỹ ⊕ D[j]
9:         if ỹ = 0 then report f(GRAYCODE(2^j)) = 0
10:        STEPS(2^j + 1, 2^{j+1})
11:    // Now split the large macro-steps in chunks
12:    for j from L + 1 to n − 1 do
13:        ỹ ← ỹ ⊕ D[j]
14:        if ỹ = 0 then report f(GRAYCODE(2^j)) = 0
15:        STEPS(2^j + 1, 2^j + 2^L)
16:        for k from 1 to 2^{j−L−1} do
17:            CHUNK(j, k)
```



The interest of doing so is that the pair $(b_1(i), b_2(i))$ has a greater chance of depending only on the lowest-significant bits of i, which means that the 2^S threads of a single SIMD unit have a greater chance of accessing to the same constant $D[k_1, k_2]$. It is then sufficient to fetch it *once* in memory to feed the 2^S threads. This seems particularly suited to NVIDIA GPU's who are capable of coalesced memory accesses.

If several SIMD units are available, it is very likely preferable to parallelize the outer loop, i.e. the enumeration of the *middle bits* of i (paradoxically)...

Initialisation. Implementing this idea will require a slightly more sophisticated initialization procedure, as we must be able to generate the internal state (the $\vec{\mathbf{y}}$ and $D[\cdot]$ values) that are ready to be used even if the loop starts "in the middle" (i.e. with $i > 1$). This is however not difficult. The second-order derivatives (the $D[\cdot, \cdot]$) can be shared by all threads, since they only depend on the equations. With this, we can write algo 4.

Unrolling and avoiding tests again. Unrolling the loop with SIMD instructions is a teeny tiny bit harder than before, because we ought to keep threads doing the same thing as much as possible. The good thing is that u_t only depend on i (since $i > 1$, it follows that $u_t = b_1(i)$. The bad thing is that when i is a power of two, then v_t depends on both i and t. This means that

Algorithm 4 Very basic SIMD version

procedure ZEROES(f)
  $D[\cdot,\cdot] \leftarrow$ INIT_CONSTANTS(f)
  for t **from** 0 **to** $2^S - 1$ **do**
    $(y_t, D_t[\cdot]) \leftarrow$ INIT_STATE $\left(f, t \cdot 2^{n-S}\right)$
    if $\vec{\mathbf{y}}_t = 0$ **then** report $f\left(t \cdot 2^{n-S}\right) = 0$
  for i **from** 1 **to** $2^{n-S} - 1$ **do**
    for t **from** 0 **to** $2^S - 1$ **do**
      let $j_t = i + t \cdot 2^{n-S}$ **in**
      let $u_t = b_1(j_t)$ **in**
      let $v_t = b_2(j_t)$ **in**
      if $v_t \neq \perp$ **then** $D_t\left[u_t\right] \leftarrow D_t\left[u_t\right] \oplus D\left[u_t, v_t\right]$
      $\vec{\mathbf{y}}_t \leftarrow \vec{\mathbf{y}}_t \oplus D_t\left[u_t\right]$
      if $\vec{\mathbf{y}}_t = 0$ **then** report $f\left(\text{GRAYCODE}(j_t)\right) = 0$

inside a single SIMD unit, some threads will fetch different constants $D[\cdot,\cdot]$ than some others. Also, the zero-th thread still has $v_t = \perp$ when i is a power of two.

Long story short: we need to do something special when i is a power of two (in fact this boil down to precompute n vectors of special constants to use in place of the normal ones when i is a power of two).

With unrolling, we obtain Algo. 5, which is the prototype of my SIMD implementation.

Algorithm 5 Improved SIMD version

```
1: procedure ParallelStep(x)
2:     let u = b_1(x) in
3:     let v = b_2(x) in
4:     for t from 0 to 2^S − 1 do
5:         D_t[u] ← D_t[u] ⊕ D[u, v]
6:         y⃗_t ← y⃗_t ⊕ D_t[u]
7:         if y⃗_t = 0 then report f(GrayCode(x + t · 2^{n−S})) = 0

8: procedure SpecialParallelStep(x)
9:     let u = b_1(x) in
10:    for t from 0 to 2^S − 1 do
11:        D_t[u] ← D_t[u] ⊕ SpecialD[u]
12:        y⃗_t ← y⃗_t ⊕ D_t[u]
13:        if y⃗_t = 0 then report f(GrayCode(x + t · 2^{n−S})) = 0

14: procedure Zeroes(f)
15:    D[·,·], SpecialD[·] ← Init_Constants(f)
16:    for t from 0 to 2^S − 1 do
17:        (y_t, D_t[·]) ← Init_State(f, t · 2^{n−S})
18:        if y⃗_t = 0 then report f(t · 2^{n−S}) = 0
19:    // Deal with the small macro-steps
20:    for j from 0 to min(n − S, L) do
21:        Special ParallelStep(2^j)
22:        for i from 2^j + 1 to 2^{j+1} − 1 do
23:            ParallelStep(i)
24:    // Now split the large macro-steps in chunks
25:    for j from L + 1 to n − S − 1 do
26:        SpecialParallelStep(2^j)
27:        for i from 2^j + 1 to 2^{j+1} − 1 do
28:            ParallelStep(i)
```