EGOI 2026 Editorial - Census

Task Author: Simon Lindholm, Joakim Blikstad

In this somewhat unusual problem, $N \leq 100$ instances of your program run at once, need to figure out the value of N. Each instance has a distinct ID $I \in \{0, \ldots, M-1\}$ with $M \leq 10^5$. There are 10^{18} locations (memory cells) that can be used for communication, all of which initially have the value 0. In each round, each still-active instance either *reads* from a location $P \in [0, 10^{18})$ or *writes* a value V to one. Reads happen before writes, and no two instances may write to the same location in the same round. Each instance must eventually output N. We aim to minimize the number of rounds used. For full score we must use at most 61 rounds, and all written values V must be either 0 or 1.

Subtask 1: consecutive IDs, $M \leq 100$ (11 points)

A natural start in this subtask is to have each instance write a 1 to the position given by its ID. Then N will equal the number of ones in the range $0, \ldots, M-1$, or equivalently, the position of the leftmost location with the value 0.

For 8 points, you can count the number of ones using a loop from 0 from $M-1$. For the full 11 points, you can use a binary search for the boundary.

Subtask 2: $N \leq 2$ (12 points)

With non-consecutive IDs and M up to 10^5, both solution ideas from subtask 1 fail.

One thing you can do to revive the idea of a counting loop is to use randomization to reduce the value of M. Every instance discards its I value, and generates a new random one from scratch in the range $0 \ldots 60$ (e.g. using `random_device()() % 61` in C++, or `random.randrange(61)` in Python). With high probability these new I values will still be distinct. The instances can then write a 1 to position I, and loop from 0 to 60 to count the number of written values. Because of randomness, this solution may get judged as Incorrect because of multiple writes to the same position. (The judge test data had about 100 test cases with two instances, so this occurs with probability $1 - (60/61)^{100} \approx 0.8$.) Thus, it may require a few resubmits to get it to pass.

Another thing you can do which generalizes better to the full problem is to use a tree-based construction. We start by having each instance write a 1 to its own instance ID, and then pair up IDs 0 and 1, 2 and 3, 4 and 5, and so on. Each instance reads the value at its paired ID. If this value is 1, it must be the case that $N = 2$, and we can exit early. Otherwise, it can change its ID to the

lower of the two IDs in the pair. Since only one of the IDs corresponded to an instance, this can not result in an ID collision.

We can now repeat this with IDs being paired up as 0-2, 4-6, 8-10, ..., then 0-4, 8-12, ..., then 0-8, ..., etc., until we reach the point where ID 0 would be paired up with an ID $\geq M$. At that point, we can be sure that only one instance is running, and output $N = 1$ and exit.

Subtask 3: $M \leq 8000$, large writes allowed ($V \leq 10^9$) (22 points)

In this subtask, it is natural to reuse the tree-based construction from the previous subtask. However, now instead of exiting early we must compute the full count of program instances. Instead of writing 1s, let us write the count of instances in the current subtree.

We again start by having each instance write a 1 to its own instance ID, pair up IDs 0-1, 2-3, etc., and have each instance read the value at its paired ID. Now in principle, we want to add our own sum and the paired one, and write the total to the lower of the IDs. If both sums are non-zero, however, this may cause a double write, which is invalid. Thus, in this case, we make sure that only the instance with the lower of the two IDs performs the write, and the other one is transitioned into a "dead" state where it does not perform any further writes. In this state, in the rounds when other instances would write, it instead performs a dummy read, ignoring the response.

Then, we continue by pairing 0-2, 4-6, ..., then 0-4, ..., etc., and in the end we will be left with the sum written in location 0.

Subtask 4: full problem (up to 55 points)

For the full problem, there are multiple approaches one can use, scoring various amounts of points. Here we outline a couple of them.

Binary adder: $O(\log M \log N)$

One idea is to make use of the same construction as in subtask 3, but writing numbers in binary, using multiple locations instead of just one. All numbers written will be at most $N \leq 100$, and therefore only need $\lceil \log(N+1) \rceil = 7$ bits of memory. We can split each round in $7 + 7$ subrounds, with the first 7 spent reading the paired ID's value, and the second 7 spent writing the sum. The number of rounds used will be $2 \times 7 \times \lceil \log M \rceil + 1 = 239$.

This number can be optimized slightly. One basic idea is to use $1, 2, \ldots, 6$-bit numbers for the first 6 summations, instead of 7-bit ones. This reduces the number of rounds to 203.

Another idea is optimize writes by making use of dead instances. Each number that counts the number of instances in a subtree will, when written in binary,

require a number of bits which is less than or equal to the number of instances in that subtree. Thus, we can parallelize the write of the sum by distributing the responsibility of writing each binary digit to separate instances. Over the course of the tree summation process, each instance is able to learn the number of nodes within their subtree with an ID lower than their own, and can use this as an index into the binary number and know which digit to write. (Similar to before, some instances will not required to write a digit and need to perform a dummy read instead.) We get a number of needed rounds of $(1+7) \times \lceil \log M \rceil + 1 = 137$, or 116 if combined with the previous idea.

It is tempting to try to also optimize reads in a similar fashion. Unfortunately, this quickly gets tricky. Output bits for the addition depend on *all* earlier input bits, and even if we parallelize reading input bits, we somehow need to combine the knowledge of all these 7 bits to compute one output bit. This takes at least $\lceil \log 7 \rceil$ rounds. It may be possible to use a construction like a Kogge-Stone adder, which would give a solution with $O(\log M \log \log N)$ rounds, but the constant factor would probably not be an improvement.

A fourth idea is to use randomness. We could use the same kind of regeneration of I as in subtask 2 to reduce M slightly; however, we can only reduce M to around N^2 before we start getting collision (based on the Birthday Problem). Since $N = 100$ and $M = 100\,000$, this only helps very little, even with many resubmits. However, we can make use of a slightly more advanced version of this idea. Start by mapping I to another value in the same range $0, \ldots, M-1$ using a random permutation (e.g. multiply by a randomly chosen constant mod M). Then, we probably do not need $1, 2, \ldots, 6, 7, \ldots,$ 7-bit number for the summations, but can almost certainly get away with something like $1, 2, \ldots, 2, 3, 4, 5, 6, 7, 7$. Asymptotically, this gives a $O(\log^2 N + \log M)$ solution, which almost (but not quite) achieves full score.

Unary adder: $O(\log M \log N)$

Instead of using binary numbers, the previous solution can also be implemented using unary numbers, i.e. with streaks of consecutive 1s being written to indicate the count of number of instances. While this sounds slower, it actually uses exactly the same number of rounds: reading a number can be done using a binary search, and writing can be done in one round by parallelizing the write as described above.

One way of viewing this solution is that after each instance writes a 1 to the location corresponding to its ID, we perform a merge sort of the values at locations $0, \ldots, M-1$.

Sorting networks: $O(\log M)$

Instead of using a merge sort, we can do sorting using a sorting network. Sorting networks are based on the idea of performing multiple parallel rounds of sorting, where in each round the items to be sorted are partitioned into disjoint pairs,

and each pair (a, b) gets sorted in-place, by swapping a and b if $a > b$. We can simulate a sorting network in our setup by having each instance be responsible for a single 1, and when a pair of locations i, j (with $i < j$) is to be sorted, the instance responsible for for location j reads the value at location i, and if it is zero, writes 0 to location j, 1 to location i, and shifts its responsibility to location i. In other cases, instances perform dummy reads, to ensure they use exactly three queries per round of the sorting network.

At the end, each instance performs a binary search to count the number of 1 in the sorted array, which will all be placed consecutively at one end.

The efficiency of this depends on the depth (i.e. round count) of the sorting network. With a bitonic sorting network, we get a solution that uses $O(\log^2 M)$ rounds (in practice around 450). More interestingly, there are results in theoretical computer science showing the existence of sorting networks that use only $O(\log M)$ rounds. In practice, however, these have terrible constants: the best published value lies around $1800 \log M$ or so, making this solution impractical.

Lazy ternary adder: $O(\log M)$

The main issue that prevents the binary adder solution from reaching full score is that while writing can be parallelized to take $O(1)$ rounds at each level, reading can not. In an ideal world, we would have one instance responsible for each digit of the binary number saved for each subtree. That instance would read the corresponding digit of the other subtree, compute the digit in that location for the sum, and either write that back or shift its responsibility to the digit one step above. The reason this does not work is carries: in the summation of two binary numbers, a carry from the least significant bit sum can ripple all the way to the most significant bit.

While there are ways to optimize latency of binary addition slightly (see e.g. the mention of Kogge-Stone adder above), one way to get down to $O(\log M)$ is to switch to a ternary (base 3) number representation, and make it *lazy*, with digits being allowed to not just take on values in $\{0, 1, 2\}$, but also slightly larger ones.

Let's say that we allow digit values to be one of $\{0, 1, 2, 3, 4\}$. Adding two such digits $0 \le x_i, y_i \le 4$ results in a value $0 \le z_i = x_i + y_i \le 8$, which we can decompose into a digit sum of $a_i = 0 \le z_i \% 3 \le 2$ and carry $c_i = 0 \le \lfloor z_i/3 \rfloor \le 2$. Now, summing the digit sum and the incoming carry, $s_i = a_i + z_{i-1} \le 4$, giving us a sum which again have all digits in $\{0, 1, 2, 3, 4\}$.

Actually turning this solution idea into code requires some care – we have to come up with systems for which instances should be responsible for which digits, how to encode the base-3 digits (e.g., using binary), and carefully time when and by whom bits are read/written. The constant factor is also not *that* much better than that of the $O(\log N \log M)$ binary adder solution; the reference solution that implements this requires 171 rounds.

Groups of powers of two, reduced with half/full adders: $O(\log M)$

Another (both simpler and more round-efficient) way of getting rid of the $O(\log N)$ factor from addition carry chains is to give up on keeping full binary(/ternary) numbers altogether. Instead, we keep sets of powers of two, that get successively combined and reduced into bigger powers of two until only a handful remain.

Each power of two is tied to a location, and can be either active (if the value 1 is written in the location), or inactive (if 0 is). The sum of the active powers of two is always N, and each active power of two has an instance that is responsible for it.

We start with M copies of 2^0 tied to locations $0, \ldots, M-1$, and have each instance write a 1 to the location give by its ID (which it is set as responsible for). Note that the sum of active powers of two is $2^0 + 2^0 + \ldots + 2^0 = N$, as expected.

In the following rounds, we pair equal powers of two up with each other, potentially with some left over. For each pair $(2^k, 2^k)$ at locations (a, b), the sum of active powers of two in that pair may be either 0, 2^k or 2^{k+1}. We pick out two new unused locations (c, d) and assign them the values 2^k and 2^{k+1} respectively. Each instance that is responsible for an active power of two in the pair reads the other's location. If both of the powers of two are active, we have the first instance write a 1 to d and take responsibility of it, while the second instance goes into a "dead" state. If only one of them are active, we have that instance write a 1 to c and take responsibility of it. If none of them are active, c and d will both remain 0. For the next round, the sets of powers of two we consider will be those corresponding to all the new locations (c, d), plus the ones that did not get paired with anything.

If we perform this process repeatedly starting with $M = 10^5$, we will see a number of powers of two that changes as following:

- $\{100000 \times 2^0\}$
- $\{50000 \times 2^0, 50000 \times 2^1\}$
- $\{25000 \times 2^0, 50000 \times 2^1, 25000 \times 2^2\}$
- $\{12500 \times 2^0, 37500 \times 2^1, 37500 \times 2^2, 12500 \times 2^3\}$
- ...

and eventually, after 37 steps (in general, $O(\log M)$ many), we will reach a stage where there is only one copy each of all of $2^0, 2^1, \ldots, 2^6$. (2^7 and above can be ignored, since they are bigger than N and therefore never active.) At this point, all instances can read the locations corresponding to those copies and sum the powers of two that are marked as active (whose locations have a value of 1) to determine N.

This method uses 82 rounds: two for each step (one read + one write), plus 7 for the final reads and 1 for outputting the answer.

A slight improvement over this can be achieved by switching from combining

pairs to powers of two, to triplets. Each triplet $(2^k, 2^k, 2^k)$ can sum to either 0, 2^k, 2^{k+1} or $2^k + 2^{k+1}$, meaning we can replace these three powers of two by just two new ones (2^k and 2^{k+1}). In more technical language, this means that instead of using half adders (mapping two bits a, b to $a + b = 2x + y$), we use full adders (mapping three bits a, b, c to $a + b + c = 2x + y$).

This method uses just 69 rounds.

There are some improvements you can make on top of this to get down to 61 rounds and get full score, although they are all rather finicky. For example, one thing you can do is circulate which nodes get to be in the dead state, and have those do productive work like eagerly reading off some powers of two and adding them to their sum. Another thing is to optimize the exact schedule of half/full adders to use towards the end of the process. The exact details here are left as an exercise to the reader.