Subquadratic Algorithms for 3SUM

Ilya Baran, Erik D. Demaine, and Mihai Pătraşcu

MIT Computer Science and Artificial Intelligence Laboratory,
{ibaran, edemaine, mip}@mit.edu

Abstract. We obtain subquadratic algorithms for 3SUM on integers and rationals in several models. On a standard word RAM with w-bit words, we obtain a running time of $O(n^2/\max\{\frac{w}{\lg^2 w}, \frac{\lg^2 n}{(\lg\lg n)^2}\})$. In the circuit RAM with one nonstandard AC^0 operation, we obtain $O(n^2/\frac{w^2}{\lg^2 w})$. In external memory, we achieve $O(n^2/(MB))$, even under the standard assumption of data indivisibility. Cache-obliviously, we obtain a running time of $O(n^2/\frac{MB}{\lg^2 M})$. In all cases, our speedup is almost quadratic in the parallelism the model can afford, which may be the best possible. Our algorithms are Las Vegas randomized; time bounds hold in expectation, and in most cases, with high probability.

1 Introduction

The 3SUM problem can be formulated as follows: given three sets A, B, C of cardinalities at most n, determine whether there exists a triplet $(a, b, c) \in A \times B \times C$, with $a + b = c$. This problem has a simple $O(n^2)$-time algorithm, which is generally believed to be the best possible. Erickson [7] proved an $\Omega(n^2)$ lower bound in the restricted linear decision tree model. Many problems have been shown to be 3SUM-hard (reducible from 3SUM), suggesting that they too require $\Omega(n^2)$ time; see, for example, the seminal work of [8]. This body of work is perhaps the most successful attempt at understanding complexity inside **P**.

In this paper, we consider the 3SUM problem on integers and rationals. We consider several models of computation, and achieve $o(n^2)$ running times in all of them. Our algorithms use Las Vegas randomization. The only previously known subquadratic bound is an FFT-based algorithm which can solve 3SUM on integers in the range $[0, u]$ in $O(u \lg u)$ time [4, Ex. 30.1-7]. However, this bound is subquadratic in n only when $u = o(n^2/\lg n)$, whereas our algorithm improves on the simple $O(n^2)$ solution for all values of n and u.

It is perhaps not surprising that 3SUM should admit slightly subquadratic algorithms. However, our solution requires ideas beyond just bit tricks, and we believe that our techniques are interesting in their own right. In addition, observe that in all cases our speedups are quadratic or nearly quadratic in the parallelism that the model can afford. If the current intuition about the quadratic nature of the 3SUM problem turns out to be correct, such improvements may be the best possible.

The Circuit RAM. The transdichotomous RAM (Random Access Machine) assumes memory cells have a size of w bits, where w grows with n. Input integers must fit in a machine word, and so must the problem size n (so that the entire memory can be addressed by word-size pointers); as a consequence, $w = \Omega(\lg n)$. Operations are allowed to touch a constant number of machine words at a time.

It remains to specify the operations and their costs. The circuit RAM allows any operation which has a polynomial-size (in w) circuit, with unbounded fan-in gates. The cost (time) of the operation is its depth; thus, unit-cost operations are those with an AC^0 implementation. The model tries to address the thorny issue of what is a "good" set of unit-cost operations. Allowing any operation from a complexity-theoretic class eliminates the sensitivity of the model to arbitrary choices in the instruction set. The depth of the circuit is a restrictive and realistic definition for the time it should take to execute the operation, making the model theoretically interesting. Previously, the circuit RAM was used to study the predecessor and dictionary problems, as well as sorting (see, e.g. [2, 3, 9]).

There are AC^0 implementations for all common arithmetic and boolean operations, except multiplication and division. Multiplication requires depth (in this model, time) $\Theta(\frac{\lg w}{\lg\lg w})$. However, the model allows other arbitrary operations, and we shall use this power by considering one nonstandard AC^0 operation. Note that the previous investigations of the model also considered nonstandard operations. Our operation, WORD-3SUM solves the 3SUM problem on small sets of small integers, in the sense that they can be packed in a word. For a fixed $s = \Theta(\lg w)$, WORD-3SUM takes three input words which are viewed as sets of $O(w/s)$ s-bit integers. The output is a bit which specifies whether there is a triplet satisfying $a + b = c \pmod{2}^s$ with a, b, c in these sets, respectively.

It is easy to see that WORD-3SUM is in AC^0, because addition is in AC^0, and all $O((\frac{w}{\lg w})^3)$ additions can be performed in parallel. On a RAM augmented with this operation, we achieve a running time of $O(n^2 \cdot \frac{\lg^2 w}{w^2} + n \cdot \frac{\lg w}{\lg\lg w} + \text{sort}(n))$. The first term dominates for any reasonable w and we essentially get a speedup of $\frac{w^2}{\lg^2 w}$. The term $O(n\frac{\lg w}{\lg\lg w})$ comes from evaluating $O(n)$ multiplicative hash functions, and sort(n) is the time it takes to sort n values (sort$(n) = O(n \lg\lg n)$ by [9]; note that this is also $O(n \lg w)$).

The Word RAM. Perhaps the most natural model for our problem is the common word RAM. This model allows as unit-time operations the usual arithmetic and bitwise operations, including multiplication. In this model, we achieve a running time of $O(n^2/\max\{\frac{\lg^2 n}{(\lg\lg n)^2}, \frac{w}{\lg^2 w}\} + \text{sort}(n))$.

To achieve the speedup by a roughly $\lg^2 n$ factor, we use the algorithm for the circuit RAM. We restrict WORD-3SUM to inputs of $\varepsilon \lg n$ bits, so that it can be implemented using a lookup table of size $n^{3\varepsilon} = o(n)$, which is initialized in negligible time. One could hope to implement WORD-3SUM using bit tricks, not a lookup table, and achieve a speedup of roughly w^2. However, if the current conjecture about the hardness of 3SUM is true, any circuit for WORD-3SUM should require $\Omega(w^2/\operatorname{poly}(\lg w))$ gates (regardless of depth). On the other hand, the standard operations for the word RAM have implementations with

$O(w \operatorname{poly}(\lg w))$ gates (for instance, using FFT for multiplication), so we cannot hope to implement WORD-3SUM efficiently using bit tricks.

We can still take advantage of a higher word size, although our speedup is just linear in w. This result uses bit tricks, but, to optimize the running time, it uses a slightly different strategy than implementing WORD-3SUM.

External Memory. Finally, we consider the external-memory model, with pages of size B, and a cache of size M. Both these quantities are measured in data items (cells); alternatively, the cache has M/B pages. As is standard in this model, we assume that data items are indivisible, so we do not try to use bit blasting inside the cells inside each page. If desired, our algorithms can be adapted easily to obtain a speedup both in terms of w and in terms of M and B.

In this model, we achieve a running time of $O(\frac{n^2}{MB} + \operatorname{sort}(n))$, where $\operatorname{sort}(n)$ is known to be $\Theta(\frac{n}{B} \lg_{M/B} \frac{n}{B})$. Note that even though the external-memory model allows us to consider M data items at a time, reloading $\Omega(M)$ items is not a unit-cost operation, but requires $\Omega(M/B)$ page reads. Thus, it is reasonable that the speedup we achieve is only MB, and not, say, M^2.

The Cache-Oblivious Model. This model is identical to the external-memory model, except that the algorithm does not know M or B. For a survey of results in this model, see [5]. Under the standard tall-cache assumption ($M = B^{1+\Omega(1)}$), we achieve a running time of $O(n^2 / \frac{MB}{\lg^2 M} + \operatorname{sort}(n) \lg n)$. This bound is almost as good as the bound for external memory.

Organization. In Section 2, we discuss a first main idea of our work, which is needed as a subroutine of the final algorithm. In Section 3, we discuss a second important idea, leading to the final algorithm. These sections only consider the integer problem, and the discussion of all models is interwoven, since the basic ideas are unitary. Section 4 presents some extensions: the rational case, testing approximate satisfiability, and obtaining time bounds with high probability.

2 Searching for a Given Sum

We first consider the problem of searching for a pair with a given sum. We are given sets A and B, which we must preprocess efficiently. Later we are given a query σ, and we must determine if there exists $(a, b) \in A \times B : a + b = \sigma$. The 3SUM problem can be solved by making n queries, for all $\sigma \in C$.

Our solution is based on the following linear time algorithm for answering a query. In the preprocessing phase, we just sort A and B in increasing order. In the query phase, we simultaneously scan A upwards and B downwards. Let i be the current index in A, and j the current index in B. At every step, the algorithm compares $A[i] + B[j]$ to σ. If they are equal, we are done. If $A[i] + B[j] < \sigma$, we know that $A[t] + B[j] < \sigma$ for all $t \leq i$ (because the numbers are sorted). Therefore, we can increment i and we do not lose a solution. Similarly, if $A[i] + B[j] > \sigma$, j can advance to $j - 1$.

Lemma 1. *In external memory, searching for a given sum requires $O(n/B)$ time, given preprocessing time $O(\mathrm{sort}(n))$.*

Proof. Immediate, since the query algorithm relies on scanning. $\square$

Our goal now is to achieve a similar speedup, which is roughly linear in w, for the RAM models. The idea behind this is to replace each element with a hash code of $\Theta(\lg w)$ bits, which enables us to pack $P = O(\min\{\frac{w}{\lg w}, n\})$ elements in a word. A query maintains indices i and j as above. The difference is that at every step we advance either i or j by P positions (one word at a time). We explain below how to take two chunks $A[i\mathinner{..}(i+P-1)]$ and $B[(j-P+1)\mathinner{..}j]$, both packed in a word, and efficiently determine whether there is a pair summing to σ. If no such pair exists, we compare $A[i+P-1]+B[j-P+1]$ to σ (using the actual values, not the hash codes). If the first quantity is larger, we know that $A[q]+B[r] > \sigma$ for all $q \geq i+P-1$ and $r \geq j-P+1$. Also, we know that $A[q]+B[r] \neq \sigma$ for all $q \leq i+P-1$ and $r \geq j-P+1$. It follows that no value $B[r]$ with $r \geq j-P+1$ can be part of a good pair, so we advance j to $j-P$. Similarly, if $A[i+P-1]+B[j-P+1] < \sigma$, we advance i to $i+P$.

2.1 Linear Hashing

We now describe how to test whether any pair from two chunks of P values sums to σ. We hash each value into $s = \Theta(\lg w)$ bits (then, P is chosen so that $P \cdot s \leq w$). In order to maintain the 3SUM constraint through hashing, the hash function must be linear. We use a very interesting family of hash functions, which originates in [6]. Pick a random *odd* integer a on w bits; the hash function maps x to `(a*x)>>(w-s)`. This should be understood as C notation, with the shift done on unsigned integers. In other words, we multiply x by a on w bits, and keep the high order s bits of the result. This function is almost linear. In particular $h(x) \oplus h(y) \oplus h(z) \in h(x+y+z) \ominus \{0,1,2\}$, where circled operators are modulo 2^s. This is because multiplying by a is linear (even in the ring modulo 2^w), and ignoring the low order $w-s$ bits can only influence the result by losing the carry from the low order bits. When adding three values, the carry across any bit boundary is at most 2.

Reformulate the test $x+y=\sigma$ as testing whether $z = x+y-\sigma$ is zero. If $z=0$, we have $h(x) \oplus h(y) \oplus h(-\sigma) \in \{0,-1,-2\}$, because $h(0)=0$ for any a. If, on the other hand, $z \neq 0$, we want to argue that $h(x) \oplus h(y) \oplus h(-\sigma) \in \{0,-1,-2\}$ is only true with small probability. We know that $h(x) \oplus h(y) \oplus h(-\sigma)$ is at most 2 away from $h(z)$. So if $h(z) \notin \{0, \pm 1, \pm 2\}$, we are fine. Since $z \neq 0$, it is equal to $b \cdot 2^c$, for odd b. Multiplying $b \cdot 2^c$ with a random odd number a will yield a uniformly random odd value on the high order $w-c$ bits, followed by c zeros. The hash function retains the high order s bits. We now have the following cases:

$w-c > s$: $h(z)$ is uniformly random, so it hits $\{0, \pm 1, \pm 2\}$ with probability $\frac{5}{2^s}$.
$w-c = s$: the low order bit of $h(z)$ is one, and the rest are uniformly random. Then $h(z)$ can only hit ± 1, and this happens with probability $\frac{2}{2^{s-1}}$.

$w - c = s - 1$**:** the low order bit $h(z)$ is zero, the second lowest is one, and the rest are random. Then $h(z)$ can only hit ± 2, and this happens with probability $\frac{2}{2^{s-2}}$.

$w - c \leq s - 2$**:** the two low order bits of $h(z)$ are zero, and $h(z)$ is never zero, so the bad values are never attained.

We have thus shown that testing whether $h(x) \oplus h(y) \oplus h(-\sigma) \in \{0, -1, -2\}$ is a good filter for the condition $x + y = \sigma$. If the condition is true, the filter is always true. Otherwise, the filter is true with probability $\frac{O(1)}{2^s}$. When considering two word-packed arrays of P values, the probability that we see a false positive is at most $P^2 \cdot \frac{O(1)}{2^s}$, by a union bound over all P^2 pairs which could generate a false positive through the hash function. For $s = \Theta(\lg P) = \Theta(\lg w)$, this can be made $1/\operatorname{poly}(P)$, for any desired polynomial.

2.2 Implementation in the RAM Models

We now return to testing if any pair from two chunks of P values sums to σ. If for all pairs (x, y), we have $h(x) \oplus h(y) \oplus h(-\sigma) \notin \{0, -1, -2\}$, then we know for sure that no pair sums to σ. On the circuit RAM, this test takes constant time: this is the WORD-3SUM operation, where the third set is $h(-\sigma) \ominus \{0, 1, 2\} \pmod{2)^s}$ (so the third set has size 3).

If the filter is passed, we have two possible cases. If there is a false positive, we can afford time $\operatorname{poly}(P)$, because a false positive happens with $1/\operatorname{poly}(P)$ probability. In particular, we can run the simple linear-time algorithm on the two chunks, taking time $O(P)$. The second case is when we actually have a good pair, and the algorithm will stop upon detecting the pair. In this case, we cannot afford $O(P)$ time, because this could end up dominating the running time of a query for large w. To avoid this, we find *one* pair which looks good after hashing in $O(\lg P) = O(\lg n)$ time, which is vanishing. We binary search for an element in the first chunk which is part of a good pair. Throw away half of the first word, and ask again if a good pair exists. If so, continue searching in this half; otherwise, continue searching in the other half. After we are down to one element in the first chunk, we binary search in the second. When we find a pair that looks good after hashing, we test whether the actual values sum to σ. If so, we stop, and we have spent negligible time. Otherwise, we have a false positive, and we run the $O(P)$ algorithm (this is necessary because there might be both a false positive, and a match in the same word). The expected running time of this step is $o(1)$ because false positives happen with small probability.

Lemma 2. *On a circuit RAM, searching for a given sum takes* $O(n \cdot \frac{\lg w}{w})$ *expected time, given preprocessing time* $O(n \cdot \frac{\lg w}{\lg \lg w} + \operatorname{sort}(n))$.

Proof. Follows from the above. Note that the preprocessing phase needs to compute $O(n)$ hash values using multiplication. □

Lemma 3. *On a word RAM, searching for a sum takes* $O(n \cdot \min\{\frac{\lg^2 w}{w}, \frac{\lg \lg n}{\lg n}\})$ *expected time, given preprocessing time* $O(\operatorname{sort}(n))$.

Proof. All we have to do is efficiently implement the test for $h(x) \oplus h(y) \oplus h(-\sigma) \in \{0, -1, -2\}$ for all pairs (x, y) from two word-packed sets (a special case of WORD-3SUM). We use a series of bit tricks to do this in $O(\lg w)$ time. Of course, if we only fill words up to $\varepsilon \lg n$ bits, we can perform this test in constant time using table lookup, which gives the second term of the running time.

Hash codes are packed in a word, with a spacing bit of zero between values. We begin by replacing each $h(A[q])$ from the first word with $-h(-\sigma) \ominus h(A[q])$. Consider a $(s+1)$-bit quantity z with the low s bits being $-h(-\sigma)$, and the high bit being one. We multiply z by a constant pattern with a one bit for every logical position in the packed arrays, generating P replicas of z. The set high order bits of z overlap with the spacing zero bits in the word-packed array. Now, we subtract the word containing elements of A from the word containing copies of z. This accomplished a parallel subtraction modulo 2^s. Because each $(s+1)$-st bit was set in z, we don't get carries between the elements. Some of these bits may remain set, if subtractions do not wrap around zero. We can AND the result with a constant cleaning pattern, to force these spacing bits to zero.

Now we have one word with $h(B[r])$ and one with $-h(-\sigma) \ominus h(A[q])$. We must test whether there is a value $-h(-\sigma) \ominus h(A[q])$ at most 2 greater than some $h(B[r])$. To do this, we concatenate the two words (this is possible if we only fill half of each word in the beginning). Then, we sort the $2P$ values in this word-packed array (see below). Now, in principle, we only have to test whether two consecutive elements in the sorted order are at distance ≤ 2. We shift the word by $s+1$ bits to the left, and subtract it from itself. This subtracts each element from the next one in sorted order. A parallel comparison with 2 can be achieved by subtracting the word from a word with $2P$ copies of $2^s + 2$, and testing which high order bits are reset. There are two more things that we need to worry about. First, we also need to check the first and the last elements in sorted order (because we are in a cyclic group), which is easily done in $O(1)$ time. Then, we need to consider only consecutive elements such that the first comes from $h(B[r])$ and the second from $-h(-\sigma) \ominus h(A[q])$. Before sorting, we attach an additional bit to each value, identifying its origin. This should be zero for $h(B[r])$ and it should be treated as the low order bit for sorting purposes (this insures that equal pairs are sorted with $h(B[r])$ first). Then, we can easily mask away the results from consecutive pairs which don't have the marker bits in a zero-one sequence.

To sort a word-packed array, we simulate a bitonic sorting network as in [1, Sec. 3]. It is known (and easy to see) that one step of a comparison network can be simulated in $O(1)$ time using word-level parallelism. Because a bitonic sorting network on $2P$ elements has depth $O(\lg P)$, this algorithm sorts a bitonic sequence in $O(\lg P) = O(\lg w)$ time. However, we must ensure that the original sequence is bitonic. Observe that we are free to pack the hash codes in a word in arbitrary order (we don't care that positions of the hash values in a word correspond to indices in the array). Hence, we can pack each word of B in increasing order of hash codes, ensuring that the values $h(B[r])$ appear in increasing order. We can also pack $h(A[q])$ values in decreasing order. When we apply a subtraction modulo 2^s, the resulting sequence is always a cyclic shift of a monotonic sequence, which is bitonic (the definition of bitonicity allows cyclic shifts). Thus we can first sort the values coming from A using the bitonic sorting network, then concatenate with the second word, and sort again using the bitonic sorting network. There is one small problem that this scheme introduces: when we find a pair which looks good after hashing, we do not know which real elements generated this pair, because hash values are resorted. But we can attach $O(\lg P) = O(\lg w)$ bits of additional information to each value, which is carried along through sorting and identifies the original element. $\square$

3 The General Algorithm

By running the query algorithm from the previous section n times, for every element in C, we obtain a speedup which is roughly linear in the parallelism in the machine. The subproblem that is being solved in parallel is a small instance of 3SUM with $|C| = O(1)$. We can make better use of the parallelism if we instead solve 3SUM subproblems with A, B, and C roughly the same size. The linear scan from the previous section does not work because it may proceed differently for different elements of C. Instead, we use another level of linear hashing to break up the problem instance into small subproblems. We hash all elements to $o(n)$ buckets and the linearity of the hash function ensures that for every pair of buckets, all of the sums of two elements from them are contained in $O(1)$ buckets. The algorithm then solves the subproblems for every pair of buckets.

3.1 A Hashing Lemma

We intend to map n elements into n/m buckets, where m is specified below. Without loss of generality, assume n/m is a power of two. The algorithm picks a random hash function, from the same almost linear family as before, with an output of $\lg(n/m)$ bits. A bucket contains elements with the same hash value. In any fixed bucket, we expect m elements. Later, we will need to bound the expected number of elements which are in buckets with more than $O(m)$ elements. By a Markov bound, we could easily conclude that the expected number of elements in buckets of size $\geq mt$ decreases linearly in t. A sharper bound is usually obtained with k-wise independent hashing, but a family of (almost) linear hash functions cannot even achieve strong universality (2-independence): consider hashing x and $2x$. Fortunately, we are saved by the following slightly unusual lemma:

Lemma 4. *Consider a family of universal hash functions $\{h : U \to [\frac{n}{m}]\}$, which guarantees that $(\forall)x \neq y : \Pr_h[h(x) = h(y)] \leq \frac{m}{n}$. For a given set S, $|S| = n$, let $\mathcal{B}(x) = \{y \in S \mid h(y) = h(x)\}$. Then the expected number of elements $x \in S$ with $|\mathcal{B}(x)| \geq t$ is at most $\frac{2n}{t-2m+2}$.*

Proof. Pick $x \in S$ randomly. It suffices to show that $p = \Pr_{h,x}[|\mathcal{B}(x)| \geq s] \leq \frac{2}{s-2m+1}$. Now pick $y \in S \setminus \{x\}$ randomly, and consider the collision probability $q = \Pr_{x,y,h}[h(x) = h(y)]$. By universality, $q \leq \frac{m}{n}$. Let $q_h = \Pr_{x,y}[h(x) = h(y)]$ and $p_h = \Pr_x[|\mathcal{B}(x)| \geq s]$.

We want to evaluate q_h as a function of p_h. Clearly, $\Pr[h(x) = h(y) \mid |\mathcal{B}(x)| \geq s] \geq \frac{s-1}{n}$. Now consider $S' = \{x \mid |\mathcal{B}(x)| < s\}$; we have $|S'| = (1 - p_h)n$. If $x \in S'$ and we have a collision, then $y \in S'$ too. By convexity of the square function, the collision probability of two random elements from S' is minimized when the same number of elements from S' hash to any hash code. In this case, $|\mathcal{B}(x)| \geq \lfloor \frac{|S'|}{n/m} \rfloor \geq (1 - p_h)m - 1$. So $\Pr[h(x) = h(y) \mid x \in S'] \geq \frac{(1-p_h)m-2}{n}$. Now $q_h \geq p_h \frac{s-1}{n} + (1-p_h)\frac{(1-p_h)m-2}{n} \geq \frac{1}{n}(p_h(s-1) + (1-2p_h)m - 2(1-p_h)) = \frac{1}{n}(p_h(s - 2m + 2) + m - 2)$. But we have $q = E[q_h] \leq \frac{m}{n}$, so $p_h(s - 2m + 2) + m - 2 \leq m$, which implies $p_h \leq \frac{2}{s-2m+2}$. □

The lemma implies that in expectation, $O(\frac{n}{m})$ elements are in buckets of size greater than $3m$. It is easy to construct a universal family showing that the linear dependence on t is optimal, so the analysis is sharp. Note that the lemma is highly sensitive on the constant in the definition of universality. If we only know that $\Pr_h[h(x) = h(y)] \leq \frac{O(1) \cdot m}{n}$, the probability that an element is in a bucket larger than t decreases just as $\frac{m}{t}$ (by the Markov bound). Again, we can construct families showing that this result is optimal, so the constant is unusually important in our application. Fortunately, the universal family that we considered achieves a constant of 1.

3.2 The Algorithm

The idea of the algorithm is now simple. We hash the three sets A, B, C separately, each into n/m buckets. Consider a pair of buckets $(\mathcal{B}^A, \mathcal{B}^B)$ from A and B, respectively. Then, there exist just two buckets $\mathcal{B}_1^C, \mathcal{B}_2^C$ of C, such $(\forall)(x, y) \in \mathcal{B}^A \times \mathcal{B}^B$, if $x+y \in C$, then $x+y \in \mathcal{B}_1^C \cup \mathcal{B}_2^C$. This follows by the almost linearity of our hash functions: if $x + y = z$, then $h(z) \in h(x) \oplus h(y) \oplus \{0, 1\}$.

The algorithm iterates through all pairs of buckets from A and B, and for each one looks for the sum in two buckets of C. This corresponds to solving n^2/m^2 independent 3SUM subproblems, where the size of each subproblem is the total size of the four buckets involved. The expected size of each bucket is m, but this does not suffice to get a good bound, because the running times are quadratic in the size of the buckets. Here, we use the Lemma 4, which states that the expected number of elements which are in buckets of size more than $3m$ is $O(n/m)$. The algorithm caps all buckets to $3m$ elements, and applies the above reasoning on these capped buckets. The elements that overflow are handled by the algorithms from the previous section: for each overflowing element, we run a query looking for a pair of values from the other two sets, which satisfy the linear constraint. Thus, we have n^2/m^2 subproblems of worst-case size $O(m)$, and $O(n/m)$ additional queries in expectation.

Theorem 5. *The 3SUM problem can be solved in* $O(n^2 \cdot \frac{\lg^2 w}{w^2} + n \cdot \frac{\lg w}{\lg \lg w} + \mathrm{sort}(n))$ *expected time on a circuit RAM.*

Proof. Choose $m = O(\min\{\frac{w}{\lg w}, \sqrt{n}\})$. We use a second level of hashing inside each bucket: replace elements by an $O(\lg w)$-bit hash value, and pack each bucket into a word. For buckets of C, we pack two values per element: $h(z)$ and $h(z) \ominus 1$. Now we can test whether there is a solution in a triplet of buckets in constant time, using WORD-3SUM. If we see a solution, we verify the actual elements of the buckets using $O(m^2)$ time. When we find a solution, we have an additive cost of $O(m^2) = O(n)$. The time spent on useless verification is $o(1)$ since the probability of a false positive is $1/\operatorname{poly}(w)$. We also need to run $O(n/\frac{w}{\lg w})$ queries in the structure of Lemma 2. $\square$

Theorem 6. *The 3SUM problem can be solved in* $O(n^2/\max\{\frac{\lg^2 n}{(\lg\lg n)^2}, \frac{w}{\lg^2 w}\} + \operatorname{sort}(n))$ *expected time on a word RAM.*

Proof. As above, but we only choose $m = O(\frac{\lg n}{\lg\lg n})$ and we implement WORD-3SUM using a lookup table. $\square$

Theorem 7. *In external memory, 3SUM can be solved in expected time* $O(\frac{n^2}{MB} + \operatorname{sort}(n))$.

Proof. Choose $m = O(M)$, so that each subproblem fits in cache, and it can be solved without memory transfers. Loading a bucket into cache requires $\Theta(\frac{M}{B})$ page transfers, so the running time for all subproblems is $O(\frac{n^2}{MB})$. We must also run $O(\frac{n}{M})$ additional queries, taking $O(\frac{n}{B})$ time each, by Lemma 1. The startup phases only require time to sort. $\square$

Theorem 8. *There is a cache-oblivious algorithm for the 3SUM problem running in expected time* $O(n^2 \cdot \frac{\lg^2 M}{MB} + \operatorname{sort}(n)\lg n)$.

Proof. This requires some variations on the previous approach. First, we hash all values using our almost linear hash functions into the universe $\{0, \ldots, n-1\}$. We store a redundant representation of each of the 3 sets; consider A for concreteness. We construct a perfect binary tree $\mathcal{T}^A$ over n leaves (we can assume n is a power of two). The hash code $h(x)$ of an element $x \in A$ is associated with leaf $h(x)$. For some node u of $\mathcal{T}^A$, let $s(u)$ be the number of elements of A with a hash code that lies below u. Also, let $\ell(u)$ be the number of leaves of $\mathcal{T}^A$ under u. Each node u stores a set $S(u)$ with elements having hash codes under u, such that $|S(u)| \leq \min\{s(u), 3\ell(u)\}$. Let v, w be the children of u. Then $S(v) \subseteq S(u), S(w) \subseteq S(u)$. If exactly one child has $s(\cdot) > 3\ell(\cdot)$, u has a chance to store more elements than $S(v) \cup S(w)$. The additional elements are chosen arbitrarily. Now, the representation of A consists of all nodes, with elements stored in them, listed in a preorder traversal. The set $S(\cdot)$ of each node is sorted by element values. Elements which appear for the first time in $S(u)$, i.e. they are in $S(u) \setminus (S(v) \cup S(w))$, are specially marked. Note that the representation of A has size $O(n \lg n)$, and it is easy to construct in time $O(\operatorname{sort}(n) \cdot \lg n)$.

We now give a recursive procedure, which is given three vertices v^A, v^B, v^C and checks whether there is a 3SUM solution in $S(v^A) \times S(v^B) \times S(v^C)$. It is easy to calculate the allowable interval of hash codes that are under a vertex. If, looking at the three intervals, it is mathematically impossible to find a solution, the procedure returns immediately. Otherwise, it calls itself recursively for all 8 combinations of children. Finally, it must take care of the elements which appear for the first time in one of $S(v^A), S(v^B), S(v^C)$. Say we have such an $x \in S(v^A)$ which does not appear in the sets of the children of v^A. Then, we can run the linear-scan algorithm to test for a sum of x in $S(v^B) \times S(v^C)$. These sets are conveniently sorted.

As usual in the cache-oblivious model, we analyze the algorithm by constructing an ideal paging strategy (because real paging strategies are $O(1)$-competitive, under constant-factor resource augmentation). Consider the depth threshold where $\ell(v) \le \varepsilon \frac{M}{\lg M}$. Observe that the representation for v and all nodes below it has size $O(\ell(v) \lg \ell(v)) = O(M)$ in the worst-case. Furthermore, these items appear in consecutive order. When the algorithm considers a triplet of vertices under the depth threshold, the pager loads all data for the vertices and their descendents into cache. Until the recursion under those vertices finishes, there need not be any more page transfers. Thus, the number of page transfers required by vertices below the threshold is $O((n/\frac{M}{\lg M})^2 \cdot \frac{M}{B}) = O(n^2 \frac{\lg^2 M}{MB})$. This is because the triples that we consider are only those which could contain a sum (due to the recursion pruning), and there are $O((n/\frac{M}{\lg M})^2)$ such triplets.

For a vertex u above the threshold, we only need to worry about the elements which appear for the first time in $S(u)$. If we define each vertex at the threshold depth to be a bucket, in the sense of Lemma 4, these are elements which overflow their buckets. Thus, we expect $O(n/\frac{M}{\lg M})$ such elements. An element which is first represented at level i is considered at most twice in conjunction with any vertex on level i. Each time, a linear scan is made through the set of such a vertex; the cost is one plus the number of full pages read. Then, handling such an element requires $O(\frac{n}{B} + n\frac{M}{\lg M})$. The first term comes from summing the full pages, and the second comes from the startup cost of one for each vertex on level i. In total for all elements that appear above the threshold, we spend time $O(n^2 \cdot \frac{\lg M}{M}(\frac{\lg M}{M} + \frac{1}{B}))$. Under the standard tall-cache assumption, $\frac{\lg M}{M} = o(\frac{1}{B})$, and we obtain the stated time bound. $\square$

4 Further Results

The Rational Case. Up to now, we have only discussed the integer case, but the rational case can be solved by reduction. Let d be the least common denominator of all fractions. We replace each fraction $\frac{a}{b}$ by adb^{-1}, which is an integer. The new problem is equivalent to the old one, but the integers can be rather large (up to 2^{wn}). To avoid this, we take all integers modulo a random prime between 3 and $O(wn^c \lg(wn))$, for some constant c. Consider a triplet of integers x, y, z. If $x+y=z$, certainly $x+y=z \pmod{p}$. If $x+y-c \neq 0$, the Chinese remainder theorem guarantees that $x+y-z=0 \pmod{p}$ for less than wn primes. By the density of primes, there are $\Theta(wn^c)$ primes in our range, so with high probability a non-solution does not become a solution modulo p. By a union bound over all triples, the prime does not introduce any false positive with high probability.

Note that our primes have $O(\lg w + \lg n) = O(w)$ bits, so the problem becomes solvable by the integer algorithm. We also need to check that a reported solution really is a solution, but the time spent verifying false positives is $o(1)$. Also note that in $a_1 d b_1^{-1} + a_2 d b_2^{-1} = a_3 d b_3^{-1} \pmod{p}$ we can eliminate d, because $d \neq 0$, so we don't actually need to compute the least common denominator. We simply replace $\frac{a}{b}$ with $ab^{-1} \pmod{p}$. Inverses modulo p can be computed in $O(\mathrm{poly}(\lg n, \lg w))$ time.

Bounds with High Probability. We describe a general paradigm for making our bounds hold with high probability, for reasonable w, M, B (when the overall time bounds are not too close to linear). The higher level of hashing is easy to handle: repeat choosing hash functions until the total number of elements overflowing their buckets is only twice the expectation. Rehashing takes roughly linear time, so repeating this step $O(\lg n)$ times (which is sufficient with high probability) is not significant. For the RAM, we have two more places where randomness is used. First, all $\frac{n^2}{m^2}$ pairs of buckets are checked by packing hash values in a word. The trouble is that these pairs are not independent, because the same hash function is used. However, after going through n^ε buckets in the outer loop, we choose a different hash function and reconstruct the word-packed representation of all buckets. Reconstruction takes linear time, so this is usually a lower order term. However, the total running time is now the sum of $n^{1-\varepsilon}/m$ independent components, and we achieve a with high probability guarantee by a Chernoff bound. Note that each term has a strict upper bound of $\mathrm{poly}(m)$ times its mean, so we need $n^{1-\varepsilon}/m$ to be bigger than a polynomial in m to cover the possible variance of the terms. We also have expected time bounds in searching for a given sum. Since we perform $O(\frac{n}{m})$ queries, we can reuse the same trick: reconstruct the word-packed representations every n^ε queries, giving n^ε independent terms.

Approximate Linear Satisfaction. We can also approximate the minimum triplet sum in absolute value (i.e., $\min_{a,b,c} |a+b+c|$), within a $1+\varepsilon$ factor. The running time increases by $O(\lg w)$ for any constant ε (also, the dependence on $\frac{1}{\varepsilon}$ can be made logarithmic). We consider a threshold 3SUM problem: for a given T, the problem is to find a triplet with sum in the interval $[-T, T]$, or report that none exists. It is straightforward to verify that all algorithms from above can solve this more general problem in the same time bounds, when $T = O(1)$.

To approximate the minimum sum, we first test whether the sum is zero (classic 3SUM). If not, we want to find some $e \in [0, w/\lg(1+\varepsilon)]$ such that there is a sum with absolute value less than $(1+\varepsilon)^e$, and none with absolute value less than $(1+\varepsilon)^{e-1}$. We show how to perform a binary search for e, which adds an $O(\lg w)$ factor to the running time. We must be able to discern the case when the minimum sum is at least $(1+\varepsilon)L$ (for some L) versus when it is at most L. We divide all numbers by $(\varepsilon L)/6$ and take the floor. Then we use threshold 3SUM with $T = 6/\varepsilon + 3$. Suppose a triplet x, y, z has sum in $[-L, L]$. Then the modified triplet has a sum whose absolute value is bounded by $L(6/(\varepsilon L)) + 3$; the term of 3 comes from a possible deviation by at most 1 for every floor operation. But this bound is exactly T, so we always find a good triplet. Suppose on the other hand that all triplets have a sum whose absolute value is at least $(1+\varepsilon)L$. Then the modified triplets have sums with absolute values larger than $(1+\varepsilon)L(6/(\varepsilon L))-3>T$, so we never report a bad triplet. Note that this algorithm also works in the rational case, because we take a floor at an early stage; the only difference is that we must also allow e to be negative, up to $-w/\lg(1+\varepsilon)$.

5 Conclusions

An interesting open problem is whether it is possible to obtain a subquadratic algorithm for finding collinear triples of points in the plane (the original motivation for studying 3SUM). This problem seems hard for a fundamental reason: there are no results on using bit tricks in relation to slopes. For example, the best bound for static planar point location is still the comparison-based $O(\lg n)$, despite attempts by several researchers.

Another interesting question is what improvements are possible for the r-SUM problem. There, the classic bounds are $\widetilde{O}(n^{\lceil r/2\rceil})$ by a meet-in-the-middle attack. Unfortunately, our techniques don't mix well with this strategy, and we cannot achieve a speedup that grows with r.

Acknowledgments. We thank Jeff Erickson and Rasmus Pagh for helpful discussions.

References

1. Albers, S., Hagerup, T.: Improved parallel integer sorting without concurrent writing. In: Proc. 3rd ACM/SIAM Symposium on Discrete Algorithms (SODA). (1992) 463–472.
2. Andersson, A., Miltersen, P.B., Riis, S., Thorup, M.: Static dictionaries on AC^0 RAMs: Query time $\Theta(\sqrt{\log n/\log\log n})$ is necessary and sufficient. In: Proc. 37th IEEE Symposium on Foundations of Computer Science (FOCS). (1996) 441–450.
3. Andersson, A., Miltersen, P.B., Thorup, M.: Fusion trees can be implemented with AC^0 instructions only. Theoretical Computer Science **215** (1999) 337–344.
4. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. 2nd edn. MIT Press and McGraw-Hill (2001).
5. Demaine, E.D.: Cache-oblivious algorithms and data structures. In: Lecture Notes from the EEF Summer School on Massive Data Sets, BRICS, University of Aarhus, Denmark. LNCS (2002) 39–46.
6. Dietzfelbinger, M.: Universal hashing and k-wise independent random variables via integer arithmetic without primes. In: Proc. 13th Symposium on Theoretical Aspects of Computer Science (STACS). (1996) 569–580.
7. Erickson, J.: Bounds for linear satisfiability problems. Chicago Journal of Theoretical Computer Science, 1999(8).
8. Gajentaan, A., Overmars, M.H.: On a class of $O(n^2)$ problems in computational geometry. Computational Geometry: Theory and Applications **5** (1995) 165–185.
9. Thorup, M.: Randomized sorting in $O(n\log\log n)$ time and linear space using addition, shift, and bit-wise boolean operations. Journal of Algorithms **42** (2002) 205–230. See also SODA'97.