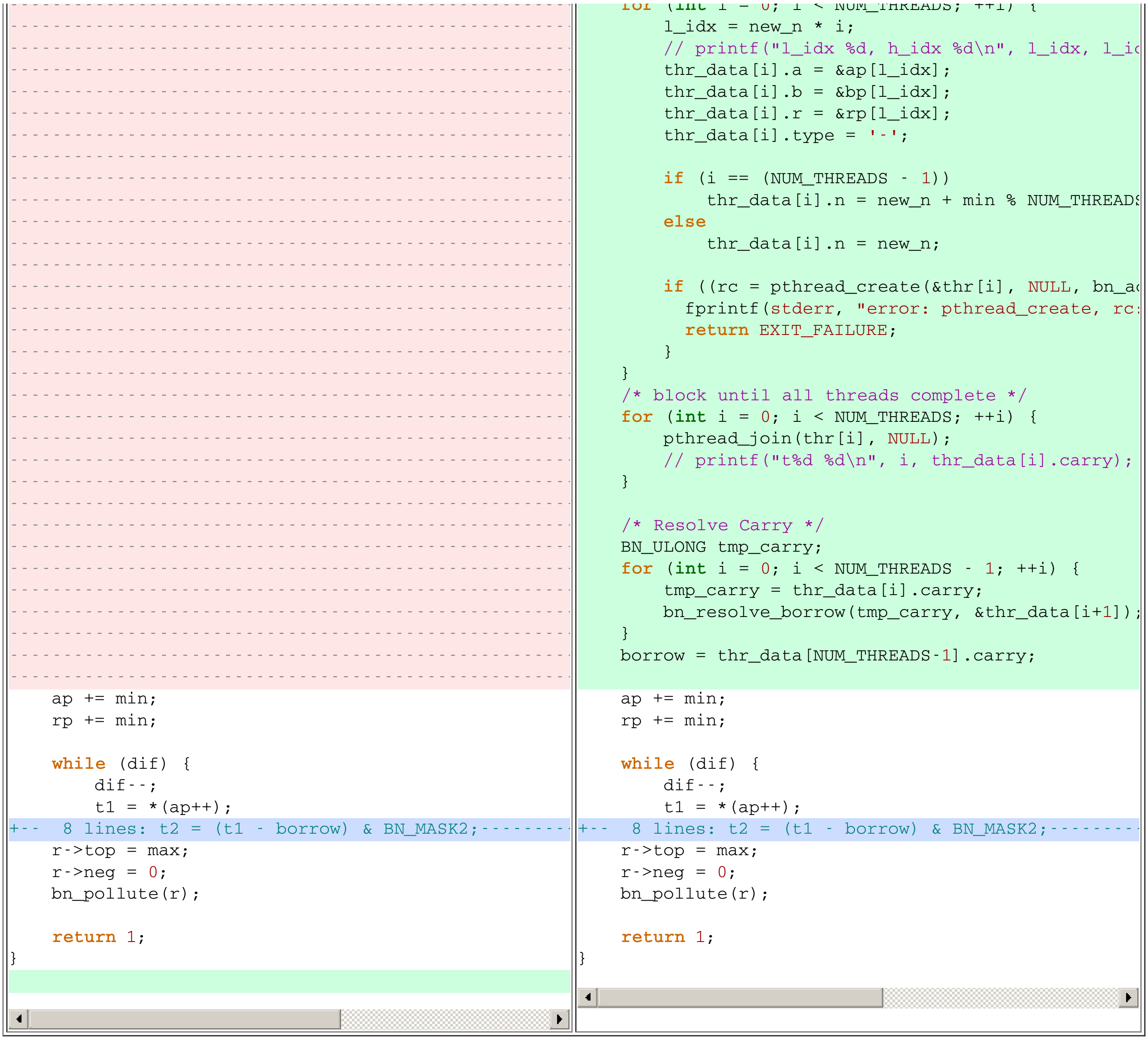
/tmp/8QbJhh_bn_add.c

```
+--  5 lines: Copyright 1995-2018 The OpenSSL Project
 * in the file LICENSE in the source distribution or
 * https://www.openssl.org/source/license.html
 */

#include "internal/cryptlib.h"
#include "bn_local.h"
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------

/* signed add of b to a. */
int BN_add(BIGNUM *r, const BIGNUM *a, const BIGNUM *
{
    int ret, r_neg, cmp_res;

+-- 61 lines: bn_check_top(a);----------------------
    const BN_ULONG *ap, *bp;
    BN_ULONG *rp, carry, t1, t2;

    bn_check_top(a);
    bn_check_top(b);

----------------------------------------------------
    if (a->top < b->top) {
        const BIGNUM *tmp;

        tmp = a;
```

crypto/bn/bn_add.c

```
+--  5 lines: Copyright 1995-2018 The OpenSSL Project
 * in the file LICENSE in the source distribution or
 * https://www.openssl.org/source/license.html
 */

#include "internal/cryptlib.h"
#include "bn_local.h"
#include "bn_par.h"

void *bn_add_sub_words_thread(void *ptr) {
    BN_ULONG c;
    add_sub_args *args = (add_sub_args *) ptr;

    const BN_ULONG* ap = args->a;
    const BN_ULONG* bp = args->b;
    BN_ULONG* rp = args->r;
    BN_ULONG min = args->n;

    if (args->type == '+')
        c = bn_add_words(rp, ap, bp, min);
    else if (args->type == '-')
        c = bn_sub_words(rp, ap, bp, min);

    args->carry = c;
    pthread_exit(NULL);
}

void bn_resolve_carry (BN_ULONG carry, add_sub_args*
    int i = 0;
    BN_ULONG t;
    while (carry && i < arg->n) {
        t = arg->r[i];
        t = (t + carry) & BN_MASK2;
        carry = (t < carry);
        arg->r[i] = t;
        i++;
    }
    if(i == arg->n) {
        arg->carry += carry;
    }
}
void bn_resolve_borrow (BN_ULONG borrow, add_sub_args
    int i = 0;
    BN_ULONG t, t1, c = borrow;
    while (c && i < arg->n) {
        t = arg->r[i];
        t1 = (t - c) & BN_MASK2;
        arg->r[i] = t1;

        //check overflow
        c = (t1 > t);
        i++;
    }
    if(i == arg->n) {
        arg->carry += c;
    }
}

/* signed add of b to a. */
int BN_add(BIGNUM *r, const BIGNUM *a, const BIGNUM *
{
    int ret, r_neg, cmp_res;

+-- 61 lines: bn_check_top(a);----------------------
    const BN_ULONG *ap, *bp;
    BN_ULONG *rp, carry, t1, t2;

    bn_check_top(a);
    bn_check_top(b);

    // a must be longer than b, if otherwise, swap
    if (a->top < b->top) {
        const BIGNUM *tmp;

        tmp = a;
        tmp = a;
        a = b;
        b = tmp;
+--  8 lines: }-----------------------------------------
    r->top = max;

    ap = a->d;
    bp = b->d;
    rp = r->d;

    carry = bn_add_words(rp, ap, bp, min);
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
    rp += min;
    ap += min;

    while (dif) {
        dif--;
        t1 = *(ap++);
+-- 32 lines: t2 = (t1 + carry) & BN_MASK2;---------
        return 0;

    ap = a->d;
    bp = b->d;
    rp = r->d;

    borrow = bn_sub_words(rp, ap, bp, min);
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
```

```
        tmp = a;
        a = b;
        b = tmp;
+--  8 lines: }-----------------------------------------
    r->top = max;

    ap = a->d;
    bp = b->d;
    rp = r->d;

    // thread init
    pthread_t thr[NUM_THREADS];
    int rc;

    /* create a thread_data_t argument array */
    add_sub_args thr_data[NUM_THREADS];

    /* create threads, divide array */
    int new_n = min/NUM_THREADS;
    int l_idx = 0;

    for (int i = 0; i < NUM_THREADS; ++i) {
        l_idx = new_n * i;
        // printf("l_idx %d, h_idx %d\n", l_idx, l_i
        thr_data[i].a = &ap[l_idx];
        thr_data[i].b = &bp[l_idx];
        thr_data[i].r = &rp[l_idx];
        thr_data[i].type = '+';

        if (i == (NUM_THREADS - 1))
            thr_data[i].n = new_n + min % NUM_THREADS
        else
            thr_data[i].n = new_n;

        if ((rc = pthread_create(&thr[i], NULL, bn_a
          fprintf(stderr, "error: pthread_create, rc
          return EXIT_FAILURE;
        }
    }
    /* block until all threads complete */
    for (int i = 0; i < NUM_THREADS; ++i) {
        pthread_join(thr[i], NULL);
        // printf("t%d %d\n", i, thr_data[i].carry);
    }

    /* Resolve Carry */
    BN_ULONG tmp_carry;
    for (int i = 0; i < NUM_THREADS - 1; ++i) {
        tmp_carry = thr_data[i].carry;
        bn_resolve_carry(tmp_carry, &thr_data[i+1]);
    }
    carry = thr_data[NUM_THREADS-1].carry;

    rp += min;
    ap += min;

    while (dif) {
        dif--;
        t1 = *(ap++);
+-- 32 lines: t2 = (t1 + carry) & BN_MASK2;---------
        return 0;

    ap = a->d;
    bp = b->d;
    rp = r->d;

    // create threads
    pthread_t thr[NUM_THREADS];
    int rc;

    /* create a thread_data_t argument array */
    add_sub_args thr_data[NUM_THREADS];

    /* create threads, divide array */
    int new_n = min/NUM_THREADS;
    int l_idx = 0;

    for (int i = 0; i < NUM_THREADS; ++i) {
```



```
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
    ap += min;
    rp += min;

    while (dif) {
        dif--;
        t1 = *(ap++);
+--  8 lines: t2 = (t1 - borrow) & BN_MASK2;---------
    r->top = max;
    r->neg = 0;
    bn_pollute(r);

    return 1;
}
```

```
    for (int i = 0; i < NUM_THREADS; ++i) {
        l_idx = new_n * i;
        // printf("l_idx %d, h_idx %d\n", l_idx, l_id
        thr_data[i].a = &ap[l_idx];
        thr_data[i].b = &bp[l_idx];
        thr_data[i].r = &rp[l_idx];
        thr_data[i].type = '-';

        if (i == (NUM_THREADS - 1))
            thr_data[i].n = new_n + min % NUM_THREADS
        else
            thr_data[i].n = new_n;

        if ((rc = pthread_create(&thr[i], NULL, bn_ad
          fprintf(stderr, "error: pthread_create, rc:
          return EXIT_FAILURE;
        }
    }
    /* block until all threads complete */
    for (int i = 0; i < NUM_THREADS; ++i) {
        pthread_join(thr[i], NULL);
        // printf("t%d %d\n", i, thr_data[i].carry);
    }

    /* Resolve Carry */
    BN_ULONG tmp_carry;
    for (int i = 0; i < NUM_THREADS - 1; ++i) {
        tmp_carry = thr_data[i].carry;
        bn_resolve_borrow(tmp_carry, &thr_data[i+1]);
    }
    borrow = thr_data[NUM_THREADS-1].carry;

    ap += min;
    rp += min;

    while (dif) {
        dif--;
        t1 = *(ap++);
+--  8 lines: t2 = (t1 - borrow) & BN_MASK2;---------
    r->top = max;
    r->neg = 0;
    bn_pollute(r);

    return 1;
}
```

/tmp/KmMjVn_bn_mul.c

```
+--  4 lines: Copyright 1995-2018 The OpenSSL Project
 * this file except in compliance with the License.
 * in the file LICENSE in the source distribution or
 * https://www.openssl.org/source/license.html
 */

#include <assert.h>
-----------------------------------------------------
#include "internal/cryptlib.h"
#include "bn_local.h"
-----------------------------------------------------

#if defined(OPENSSL_NO_ASM) || !defined(OPENSSL_BN_AS
/*
 * Here follows specialised variants of bn_add_words
 * They have the property performing operations on ar
 * The sizes of those arrays is expressed through cl,
+--138 lines: * length ( basically, min(len(a),len(b)
#ifdef BN_RECURSION
/*
 * Karatsuba recursive multiplication algorithm (cf.
 * Computer Programming, Vol. 2)
 */

-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
```

crypto/bn/bn_mul.c

```
+--  4 lines: Copyright 1995-2018 The OpenSSL Project
 * this file except in compliance with the License.
 * in the file LICENSE in the source distribution or
 * https://www.openssl.org/source/license.html
 */

#include <assert.h>
#include <pthread.h>
#include "internal/cryptlib.h"
#include "bn_local.h"
#include "bn_par.h"

#if defined(OPENSSL_NO_ASM) || !defined(OPENSSL_BN_AS
/*
 * Here follows specialised variants of bn_add_words
 * They have the property performing operations on ar
 * The sizes of those arrays is expressed through cl,
+--138 lines: * length ( basically, min(len(a),len(b)
#ifdef BN_RECURSION
/*
 * Karatsuba recursive multiplication algorithm (cf.
 * Computer Programming, Vol. 2)
 */


pthread_mutex_t thr_count_lock;


void *bn_mul_recursive_thread(void *ptr) {
    recursive_args *args = (recursive_args *) ptr;
    BN_ULONG *r = args->r;
    BN_ULONG *a = args->a;
    BN_ULONG *b = args->b;
    int n2 = args->n2;
    int dna = args->dna;
    int dnb = args->dnb;
    BN_ULONG *t = args->t;
    int *used_thr = args->used_thr;

    bn_mul_recursive(r, a, b, n2, dna, dnb, t, used_t
    pthread_exit(NULL);
}

void *bn_mul_part_recursive_thread(void *ptr) {
    recursive_args *args = (recursive_args *) ptr;
    BN_ULONG *r = args->r;
    BN_ULONG *a = args->a;
    BN_ULONG *b = args->b;
    int n = args->n2;
    int tna = args->dna;
    int tnb = args->dnb;
    BN_ULONG *t = args->t;
    int *used_thr = args->used_thr;

    bn_mul_part_recursive(r, a, b, n, tna, tnb, t, us
    pthread_exit(NULL);
}

void start_mul_recursive_thread(pthread_t *thr, recur
    int rc;

    pthread_mutex_lock(&thr_count_lock);
    (*used_thr)++;
    pthread_mutex_unlock(&thr_count_lock);
    set_recursive_arg((*arg), r, a, b, n2, dna, dnb,

    // printf("thread_created %d\n", *(arg->used_thr)
    if ((rc = pthread_create(thr, NULL, bn_mul_recurs
        fprintf(stderr, "error: pthread_create, rc: %
        exit(EXIT_FAILURE);
    } else {
        // printf("create%d success\n", *used_thr);
    }
}
```

```
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
/*-
 * r is 2*n2 words in size,
 * a and b are both n2 words in size.
 * n2 must be a power of 2.
 * We multiply and return the result.
 * t must be 2*n2 words in size
 * We calculate
 * a[0]*b[0]
 * a[0]*b[0]+a[1]*b[1]+(a[0]-a[1])*(b[1]-b[0])
 * a[1]*b[1]
-----------------------------------------------------
 */
/* dnX may not be positive, but n2/2+dnX has to be *
void bn_mul_recursive(BN_ULONG *r, BN_ULONG *a, BN_UL
                      int dna, int dnb, BN_ULONG *t)
{
    int n = n2 / 2, c1, c2;
    int tna = n + dna, tnb = n + dnb;
    unsigned int neg, zero;
    BN_ULONG ln, lo, *p;

+-- 19 lines: # ifdef BN_MUL_COMBA--------------------
        if ((dna + dnb) < 0)
            memset(&r[2 * n2 + dna + dnb], 0,
                   sizeof(BN_ULONG) * -(dna + dnb));
        return;
    }
    /* r=(a[0]-a[1])*(b[1]-b[0]) */
    c1 = bn_cmp_part_words(a, &(a[n]), tna, n - tna);
    c2 = bn_cmp_part_words(&(b[n]), b, tnb, tnb - n);
    zero = neg = 0;
    switch (c1 * 3 + c2) {
    case -4:
        bn_sub_part_words(t, &(a[n]), a, tna, tna - n
        bn_sub_part_words(&(t[n]), b, &(b[n]), tnb, n
        break;
    case -3:
        zero = 1;
        break;
    case -2:
        bn_sub_part_words(t, &(a[n]), a, tna, tna - n
        bn_sub_part_words(&(t[n]), &(b[n]), b, tnb, t
        neg = 1;
        break;
    case -1:
    case 0:
    case 1:
        zero = 1;
        break;
    case 2:
        bn_sub_part_words(t, a, &(a[n]), tna, n - tna
        bn_sub_part_words(&(t[n]), b, &(b[n]), tnb, n
        neg = 1;
```

```
void start_mul_part_recursive_thread(pthread_t *thr,
    int rc;

    pthread_mutex_lock(&thr_count_lock);
    (*used_thr)++;
    pthread_mutex_unlock(&thr_count_lock);
    set_recursive_arg((*arg), r, a, b, n2, dna, dnb,

    // printf("thread_created %d\n", *(arg->used_thr)
    if ((rc = pthread_create(thr, NULL, bn_mul_part_r
        fprintf(stderr, "error: pthread_create, rc: %
        exit(EXIT_FAILURE);
    } else {
        // printf("create%d success\n", *used_thr);
    }
}

int get_used_thread(int* used_thr) {
    pthread_mutex_lock(&thr_count_lock);
    int u = *used_thr;
    pthread_mutex_unlock(&thr_count_lock);
    return u;
}

/*-
 * r is 2*n2 words in size,
 * a and b are both n2 words in size.
 * n2 must be a power of 2.
 * We multiply and return the result.
 * t must be 2*n2 words in size
 * We calculate
 * a[0]*b[0] a_low*b_low
 * a[0]*b[0]+a[1]*b[1]+(a[0]-a[1])*(b[1]-b[0])
 *       a_low*b_low + a_high*b_high + (a_low-a_high)
 * a[1]*b[1] a_high*b_high
 */
/* dnX may not be positive, but n2/2+dnX has to be */
void bn_mul_recursive(BN_ULONG *r, BN_ULONG *a, BN_UL
                      int dna, int dnb, BN_ULONG *t,
{
    int n = n2 / 2, c1, c2;
    int tna = n + dna, tnb = n + dnb;
    unsigned int neg, zero;
    BN_ULONG ln, lo, *p;

+-- 19 lines: # ifdef BN_MUL_COMBA--------------------
        if ((dna + dnb) < 0)
            memset(&r[2 * n2 + dna + dnb], 0,
                   sizeof(BN_ULONG) * -(dna + dnb));
        return;
    }
    /* r=(a[0]-a[1])*(b[1]-b[0]) */
    c1 = bn_cmp_part_words(a, &(a[n]), tna, n - tna);
    c2 = bn_cmp_part_words(&(b[n]), b, tnb, tnb - n);
    zero = neg = 0;
    switch (c1 * 3 + c2) {
    case -4: // a[0] < a[1], b[1] < b[0]
        bn_sub_part_words(t, &(a[n]), a, tna, tna - n
        bn_sub_part_words(&(t[n]), b, &(b[n]), tnb, n
        break;
    case -3: // a[0] < a[1], b[1] == b[0]
        zero = 1;
        break;
    case -2: // a[0] < a[1], b[1] > b[0]
        bn_sub_part_words(t, &(a[n]), a, tna, tna - n
        bn_sub_part_words(&(t[n]), &(b[n]), b, tnb, t
        neg = 1;
        break;
    case -1: // a[0] == a[1], b[1] < b[0]
    case 0: // a[0] == a[1], b[1] =p b[0]
    case 1: // a[0] == a[1], b[1] > b[0]
        zero = 1;
        break;
    case 2: // a[0] > a[1], b[1] < b[0]
        bn_sub_part_words(t, a, &(a[n]), tna, n - tna
        bn_sub_part_words(&(t[n]), b, &(b[n]), tnb, n
        neg = 1;
        neg = 1;
        break;
    case 3:
        zero = 1;
        break;
    case 4:
        bn_sub_part_words(t, a, &(a[n]), tna, n - tna
        bn_sub_part_words(&(t[n]), &(b[n]), b, tnb, t
        break;
    }

# ifdef BN_MUL_COMBA
+-- 16 lines: if (n == 4 && dna == 0 && dnb == 0) { X

        bn_mul_comba8(r, a, b);
        bn_mul_comba8(&(r[n2]), &(a[n]), &(b[n]));
    } else
# endif                         /* BN_MUL_COMBA */
    {
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
        p = &(t[n2 * 2]);
        if (!zero)
            bn_mul_recursive(&(t[n2]), t, &(t[n]), n,
        else
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
            memset(&t[n2], 0, sizeof(*t) * n2);
        bn_mul_recursive(r, a, b, n, 0, 0, p);
        bn_mul_recursive(&(r[n2]), &(a[n]), &(b[n]),
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
    }

    /*-
     * t[32] holds (a[0]-a[1])*(b[1]-b[0]), c1 is the
     * r[10] holds (a[0]*b[0])
     * r[32] holds (b[1]*b[1])
     */

    c1 = (int)(bn_add_words(t, r, &(r[n2]), n2));

    if (neg) {                  /* if t[32] is negati
        c1 -= (int)(bn_sub_words(&(t[n2]), t, &(t[n2]
    } else {
        /* Might have a carry */
        c1 += (int)(bn_add_words(&(t[n2]), &(t[n2]),
    }
```

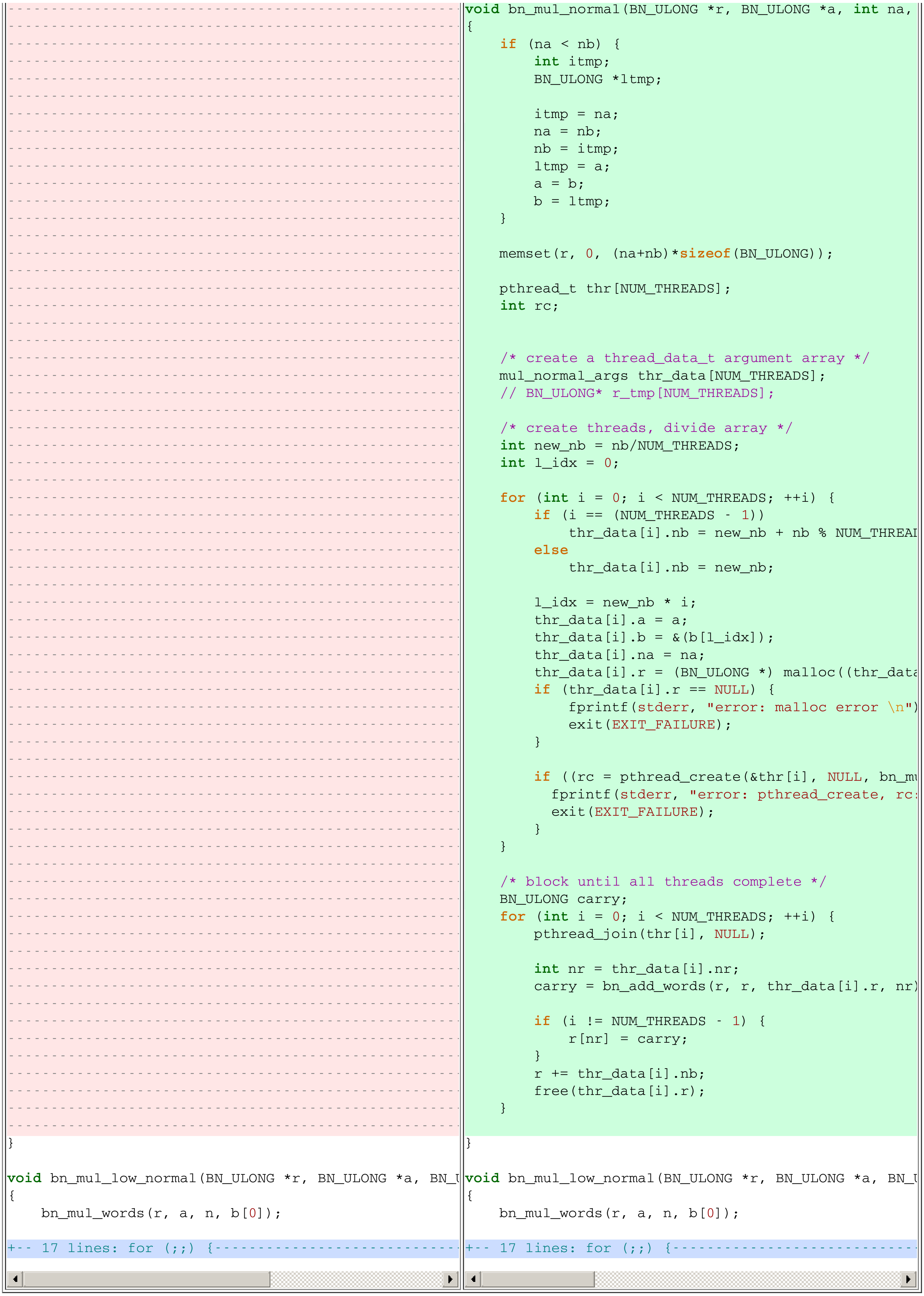
```
        neg = 1;
        break;
    case 3: // a[0] > a[1], b[1] == b[0]
        zero = 1;
        break;
    case 4: // a[0] > a[1], b[1] > b[0]
        bn_sub_part_words(t, a, &(a[n]), tna, n - tna
        bn_sub_part_words(&(t[n]), &(b[n]), b, tnb, t
        break;
    }

# ifdef BN_MUL_COMBA
+-- 16 lines: if (n == 4 && dna == 0 && dnb == 0) { X

        bn_mul_comba8(r, a, b);
        bn_mul_comba8(&(r[n2]), &(a[n]), &(b[n]));
    } else
# endif                         /* BN_MUL_COMBA */
    {
        pthread_t thr[3];
        recursive_args arg[3];
        int running_cnt = 0, rc;
        BN_ULONG* tp[3];
        p = &(t[n2 * 2]);
        if (!zero) {
            if (get_used_thread(used_thr) < NUM_THREA
                tp[0] = (BN_ULONG *) calloc(n2*2, siz
                start_mul_recursive_thread(&(thr[0]),
                running_cnt++;
            } else
                bn_mul_recursive(&(t[n2]), t, &(t[n])
        } else
            memset(&t[n2], 0, sizeof(*t) * n2);

        if (get_used_thread(used_thr) < NUM_THREADS)
            tp[1] = (BN_ULONG *) calloc(n2*2, sizeof(
            start_mul_recursive_thread(&(thr[1]), &(a
            running_cnt++;
        } else
            bn_mul_recursive(r, a, b, n, 0, 0, p, use

        if (get_used_thread(used_thr) < NUM_THREADS)
            tp[2] = (BN_ULONG *) calloc(n2*2, sizeof(
            start_mul_recursive_thread(&(thr[2]), &(a
            running_cnt++;
        } else
            bn_mul_recursive(&(r[n2]), &(a[n]), &(b[n

        /* block until all threads complete */
        // printf("running_cnt %d\n", running_cnt);
        for (int i = 0; i < running_cnt; i++) {
            // printf("i %d\n", i);
            if ((rc = pthread_join(thr[i], NULL))) {
                fprintf(stderr, "error: pthread_join,
                exit(EXIT_FAILURE);
            } else {
                // printf("join%d success\n", i);
            }
            // printf("t%d %d\n", i, thr_data[i].carr
            free(tp[i]);
        }
    }

    /*-
     * t[n2] holds (a[0]-a[1])*(b[1]-b[0]), c1 is the
     * r[0] holds (a[0]*b[0])
     * r[n2] holds (b[1]*b[1])
     */

    c1 = (int)(bn_add_words(t, r, &(r[n2]), n2));

    if (neg) {                  /* if t[n2] is negati
        c1 -= (int)(bn_sub_words(&(t[n2]), t, &(t[n2]
    } else {
        /* Might have a carry */
        c1 += (int)(bn_add_words(&(t[n2]), &(t[n2]),
    }
    /*-
     * t[32] holds (a[0]-a[1])*(b[1]-b[0])+(a[0]*b[0]
     * r[10] holds (a[0]*b[0])
     * r[32] holds (b[1]*b[1])
     * c1 holds the carry bits
     */
    c1 += (int)(bn_add_words(&(r[n]), &(r[n]), &(t[n2
----------------------------------------------------
----------------------------------------------------
    if (c1) {
        p = &(r[n + n2]);
        lo = *p;
        ln = (lo + c1) & BN_MASK2;
        *p = ln;

+-- 14 lines: The overflow will stop before we over w

/*
 * n+tn is the word length t needs to be n*4 is size,
 */
/* tnX may not be negative but less than n */
void bn_mul_part_recursive(BN_ULONG *r, BN_ULONG *a,
                           int tna, int tnb, BN_ULONG
{
    int i, j, n2 = n * 2;
    int c1, c2, neg;
    BN_ULONG ln, lo, *p;

    if (n < 8) {
+-- 45 lines: bn_mul_normal(r, a, n + tna, b, n + tnb
    if (n == 8) {
        bn_mul_comba8(&(t[n2]), t, &(t[n]));
        bn_mul_comba8(r, a, b);
        bn_mul_normal(&(r[n2]), &(a[n]), tna, &(b[n])
        memset(&r[n2 + tna + tnb], 0, sizeof(*r) * (r
    } else {
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
        p = &(t[n2 * 2]);
        bn_mul_recursive(&(t[n2]), t, &(t[n]), n, 0,
        bn_mul_recursive(r, a, b, n, 0, 0, p);
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
        i = n / 2;
        /*
         * If there is only a bottom half to the numb
         */
        if (tna > tnb)
            j = tna - i;
        else
            j = tnb - i;
        if (j == 0) {
            bn_mul_recursive(&(r[n2]), &(a[n]), &(b[n
                             i, tna - i, tnb - i, p);
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
            memset(&r[n2 + i * 2], 0, sizeof(*r) * (r
        } else if (j > 0) {     /* eg, n == 16, i ==
            bn_mul_part_recursive(&(r[n2]), &(a[n]),
```

```
    /*-
     * t[n2] holds (a[0]-a[1])*(b[1]-b[0])+(a[0]*b[0]
     * r[0] holds (a[0]*b[0])
     * r[n2] holds (b[1]*b[1])
     * c1 holds the carry bits
     */
    c1 += (int)(bn_add_words(&(r[n]), &(r[n]), &(t[n2

    // resolve carry on r[n + n2] to last elmt
    if (c1) {
        p = &(r[n + n2]);
        lo = *p;
        ln = (lo + c1) & BN_MASK2;
        *p = ln;

+-- 14 lines: The overflow will stop before we over w

/*
 * n+tn is the word length t needs to be n*4 is size,
 */
/* tnX may not be negative but less than n */
void bn_mul_part_recursive(BN_ULONG *r, BN_ULONG *a,
                           int tna, int tnb, BN_ULONG
{
    int i, j, n2 = n * 2;
    int c1, c2, neg;
    BN_ULONG ln, lo, *p;

    if (n < 8) {
+-- 45 lines: bn_mul_normal(r, a, n + tna, b, n + tnb
    if (n == 8) {
        bn_mul_comba8(&(t[n2]), t, &(t[n]));
        bn_mul_comba8(r, a, b);
        bn_mul_normal(&(r[n2]), &(a[n]), tna, &(b[n])
        memset(&r[n2 + tna + tnb], 0, sizeof(*r) * (r
    } else {
        pthread_t thr[3];
        recursive_args arg[3];
        int running_cnt = 0, rc;
        BN_ULONG* tp[3];
        p = &(t[n2 * 2]);

        if (get_used_thread(used_thr) < NUM_THREADS)
            tp[0] = (BN_ULONG *) calloc(n2*4, sizeof
            start_mul_recursive_thread(&(thr[0]), &(a
            running_cnt++;
        } else
            bn_mul_recursive(&(t[n2]), t, &(t[n]), n,

        if (get_used_thread(used_thr) < NUM_THREADS)
            tp[1] = (BN_ULONG *) calloc(n2*4, sizeof
            start_mul_recursive_thread(&(thr[1]), &(a
            running_cnt++;
        } else
            bn_mul_recursive(r, a, b, n, 0, 0, p, use

        i = n / 2;
        /*
         * If there is only a bottom half to the numb
         */
        if (tna > tnb)
            j = tna - i;
        else
            j = tnb - i;
        if (j == 0) {
            if (get_used_thread(used_thr) < NUM_THREA
                tp[2] = (BN_ULONG *) calloc(n2*2, siz
                start_mul_recursive_thread(&(thr[2]),
                                    i, tna - i, tnb -
                running_cnt++;
            } else
                bn_mul_recursive(&(r[n2]), &(a[n]), &
                                    i, tna - i, tnb -
            memset(&r[n2 + i * 2], 0, sizeof(*r) * (r
        } else if (j > 0) {     /* eg, n == 16, i ==
            if (get_used_thread(used_thr) < NUM_THREA
                                  i, tna - i, tnb - i
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
            memset(&(r[n2 + tna + tnb]), 0,
                   sizeof(BN_ULONG) * (n2 - tna - tnb
        } else {                /* (j < 0) eg, n == 1

            memset(&r[n2], 0, sizeof(*r) * n2);
            if (tna < BN_MUL_RECURSIVE_SIZE_NORMAL
+--  4 lines: && tnb < BN_MUL_RECURSIVE_SIZE_NORMAL)
                    i /= 2;
                    /*
                     * these simplified conditions wo
                     * difference between tna and tnb
                     */
                    if (i < tna || i < tnb) {
                        bn_mul_part_recursive(&(r[n2]
                                              &(a[n])
                                              i, tna
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
                        break;
                    } else if (i == tna || i == tnb)
                        bn_mul_recursive(&(r[n2]),
                                         &(a[n]), &(b
                                         i, tna - i,
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
                        break;
                    }
                }
            }
        }
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
    }

    /*-
     * t[32] holds (a[0]-a[1])*(b[1]-b[0]), c1 is the
     * r[10] holds (a[0]*b[0])
     * r[32] holds (b[1]*b[1])
+-- 41 lines: ---------------------------------------
 * r needs to be n2 words and t needs to be n2*2
 */
void bn_mul_low_recursive(BN_ULONG *r, BN_ULONG *a, B
                          BN_ULONG *t)
{
    int n = n2 / 2;

    bn_mul_recursive(r, a, b, n, 0, 0, &(t[0]));
    if (n >= BN_MUL_LOW_RECURSIVE_SIZE_NORMAL) {
```

```
                tp[2] = (BN_ULONG *) calloc(n2*2, siz
                start_mul_recursive_thread(&(thr[2]),
                                           i, tna - i, tnb -
                running_cnt++;
            } else
                bn_mul_recursive(&(r[n2]), &(a[n]), &
                                           i, tna - i, tnb -
            memset(&(r[n2 + tna + tnb]), 0,
                   sizeof(BN_ULONG) * (n2 - tna - tnb
        } else {                /* (j < 0) eg, n == 1

            memset(&r[n2], 0, sizeof(*r) * n2);
            if (tna < BN_MUL_RECURSIVE_SIZE_NORMAL
+--  4 lines: && tnb < BN_MUL_RECURSIVE_SIZE_NORMAL)
                    i /= 2;
                    /*
                     * these simplified conditions wo
                     * difference between tna and tnb
                     */
                    if (i < tna || i < tnb) {
                        if (get_used_thread(used_thr)
                            tp[2] = (BN_ULONG *) call
                            start_mul_part_recursive_
                                             &(a[n]),
                                             i, tna -
                            running_cnt++;
                        } else
                            bn_mul_part_recursive(&(r
                                                  &(a
                                                  i,
                        break;
                    } else if (i == tna || i == tnb)
                        if (get_used_thread(used_thr)
                            tp[2] = (BN_ULONG *) call
                            start_mul_recursive_threa
                                             &(a[n]),
                                             i, tna -
                            running_cnt++;
                        } else
                            bn_mul_recursive(&(r[n2])
                                             &(a[n]),
                                             i, tna -
                        break;
                    }
                }
            }
        }

        /* block until all threads complete */
        // printf("running_cnt %d\n", running_cnt);
        for (int i = 0; i < running_cnt; i++) {
            // printf("i %d\n", i);
            if ((rc = pthread_join(thr[i], NULL))) {
                fprintf(stderr, "error: pthread_join,
                exit(EXIT_FAILURE);
            } else {
                // printf("join%d success\n", i);
            }
            // printf("t%d %d\n", i, thr_data[i].car
            free(tp[i]);
        }
    }

    /*-
     * t[32] holds (a[0]-a[1])*(b[1]-b[0]), c1 is the
     * r[10] holds (a[0]*b[0])
     * r[32] holds (b[1]*b[1])
+-- 41 lines: ---------------------------------------
 * r needs to be n2 words and t needs to be n2*2
 */
void bn_mul_low_recursive(BN_ULONG *r, BN_ULONG *a, B
                          BN_ULONG *t)
{
    int n = n2 / 2;
    int u = 99;
    bn_mul_recursive(r, a, b, n, 0, 0, &(t[0]), &u);
    if (n >= BN_MUL_LOW_RECURSIVE_SIZE_NORMAL) {
        bn_mul_low_recursive(&(t[0]), &(a[0]), &(b[n]
        bn_add_words(&(r[n]), &(r[n]), &(t[0]), n);
        bn_mul_low_recursive(&(t[0]), &(a[n]), &(b[0]
        bn_add_words(&(r[n]), &(r[n]), &(t[0]), n);
    } else {
+-- 47 lines: bn_mul_low_normal(&(t[0]), &(a[0]), &(b
            goto err;
    } else
        rr = r;

#if defined(BN_MUL_COMBA) || defined(BN_RECURSION)
    i = al - bl;
-----------------------------------------------------
#endif
#ifdef BN_MUL_COMBA
    if (i == 0) {
# if 0
        if (al == 4) {
            if (bn_wexpand(rr, 8) == NULL)
                goto err;
            rr->top = 8;
            bn_mul_comba4(rr->d, a->d, b->d);
            goto end;
        }
# endif
-----------------------------------------------------
        if (al == 8) {
            if (bn_wexpand(rr, 16) == NULL)
                goto err;
            rr->top = 16;
            bn_mul_comba8(rr->d, a->d, b->d);
            goto end;
        }
    }
#endif                          /* BN_MUL_COMBA */
#ifdef BN_RECURSION
    if ((al >= BN_MULL_SIZE_NORMAL) && (bl >= BN_MULL
        if (i >= -1 && i <= 1) {
-----------------------------------------------------
            /*
             * Find out the power of two lower or equ
             * two numbers
             */
            if (i >= 0) {
                j = BN_num_bits_word((BN_ULONG)al);
            }
            if (i == -1) {
                j = BN_num_bits_word((BN_ULONG)bl);
            }
            j = 1 << (j - 1);
-----------------------------------------------------
            assert(j <= al || j <= bl);
            k = j + j;
            t = BN_CTX_get(ctx);
            if (t == NULL)
                goto err;
            if (al > j || bl > j) {
-----------------------------------------------------
                if (bn_wexpand(t, k * 4) == NULL)
                    goto err;
                if (bn_wexpand(rr, k * 4) == NULL)
                    goto err;
                bn_mul_part_recursive(rr->d, a->d, b-
                                      j, al - j, bl -
            } else {            /* al <= j || bl <= j

-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
                if (bn_wexpand(t, k * 2) == NULL)
                    goto err;
                if (bn_wexpand(rr, k * 2) == NULL)
                    goto err;
                bn_mul_recursive(rr->d, a->d, b->d, j
-----------------------------------------------------
```

```
        bn_mul_low_recursive(&(t[0]), &(a[0]), &(b[n]
        bn_add_words(&(r[n]), &(r[n]), &(t[0]), n);
        bn_mul_low_recursive(&(t[0]), &(a[n]), &(b[0]
        bn_add_words(&(r[n]), &(r[n]), &(t[0]), n);
    } else {
+-- 47 lines: bn_mul_low_normal(&(t[0]), &(a[0]), &(b
            goto err;
    } else
        rr = r;

#if defined(BN_MUL_COMBA) || defined(BN_RECURSION)
    i = al - bl;
    // printf("i %d, al %d, bl %d\n", i, al, bl);
#endif
#ifdef BN_MUL_COMBA
    if (i == 0) {
# if 0
        if (al == 4) {
            if (bn_wexpand(rr, 8) == NULL)
                goto err;
            rr->top = 8;
            bn_mul_comba4(rr->d, a->d, b->d);
            goto end;
        }
# endif
    // printf("comba\n");
        if (al == 8) {
            if (bn_wexpand(rr, 16) == NULL)
                goto err;
            rr->top = 16;
            bn_mul_comba8(rr->d, a->d, b->d);
            goto end;
        }
    }
#endif                          /* BN_MUL_COMBA */
#ifdef BN_RECURSION
    if ((al >= BN_MULL_SIZE_NORMAL) && (bl >= BN_MULL
        if (i >= -1 && i <= 1) {
            // printf("recursion\n");
            /*
             * Find out the power of two lower or equ
             * two numbers
             */
            if (i >= 0) {
                j = BN_num_bits_word((BN_ULONG)al);
            }
            if (i == -1) {
                j = BN_num_bits_word((BN_ULONG)bl);
            }
            j = 1 << (j - 1);
            // printf("j %d\n", j);
            assert(j <= al || j <= bl);
            k = j + j;
            t = BN_CTX_get(ctx);
            if (t == NULL)
                goto err;
            if (al > j || bl > j) {
                // printf("mul-part-rec\n");
                if (bn_wexpand(t, k * 4) == NULL)
                    goto err;
                if (bn_wexpand(rr, k * 4) == NULL)
                    goto err;
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------

                int used_thread = 1;
                bn_mul_part_recursive(rr->d, a->d, b-
                                      j, al - j, bl -
            } else {            /* al <= j && bl <= j
                // al or bl is exacly the power of tw
                if (bn_wexpand(t, k * 2) == NULL)
                    goto err;
                if (bn_wexpand(rr, k * 2) == NULL)
                    goto err;
                int used_thread = 1;
                bn_mul_recursive(rr->d, a->d, b->d, j
            }
            rr->top = top;
            goto end;
        }
    }
#endif                          /* BN_RECURSION */
    if (bn_wexpand(rr, top) == NULL)
        goto err;
    rr->top = top;
-----------------------------------------------------
    bn_mul_normal(rr->d, a->d, al, b->d, bl);

#if defined(BN_MUL_COMBA) || defined(BN_RECURSION)
 end:
#endif
    rr->neg = a->neg ^ b->neg;
+--  5 lines: rr->flags |= BN_FLG_FIXED_TOP;---------
 err:
    bn_check_top(r);
    BN_CTX_end(ctx);
    return ret;
}

void bn_mul_normal(BN_ULONG *r, BN_ULONG *a, int na,
{
    BN_ULONG *rr;

    if (na < nb) {
        int itmp;
        BN_ULONG *ltmp;
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------

        itmp = na;
        na = nb;
        nb = itmp;
        ltmp = a;
        a = b;
        b = ltmp;

    }
    rr = &(r[na]);
    if (nb <= 0) {
        (void)bn_mul_words(r, a, na, 0);
        return;
    } else
        rr[0] = bn_mul_words(r, a, na, b[0]);

    for (;;) {
        if (--nb <= 0)
            return;
        rr[1] = bn_mul_add_words(&(r[1]), a, na, b[1]
        if (--nb <= 0)
            return;
        rr[2] = bn_mul_add_words(&(r[2]), a, na, b[2]
        if (--nb <= 0)
            return;
        rr[3] = bn_mul_add_words(&(r[3]), a, na, b[3]
        if (--nb <= 0)
            return;
        rr[4] = bn_mul_add_words(&(r[4]), a, na, b[4]
        rr += 4;
        r += 4;
        b += 4;
    }
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
```

```
            }
            rr->top = top;
            goto end;
        }
    }
#endif                          /* BN_RECURSION */
    if (bn_wexpand(rr, top) == NULL)
        goto err;
    rr->top = top;
    // printf("normal\n");
    bn_mul_normal(rr->d, a->d, al, b->d, bl);

#if defined(BN_MUL_COMBA) || defined(BN_RECURSION)
 end:
#endif
    rr->neg = a->neg ^ b->neg;
+--  5 lines: rr->flags |= BN_FLG_FIXED_TOP;---------
 err:
    bn_check_top(r);
    BN_CTX_end(ctx);
    return ret;
}

void *bn_mul_normal_thread(void *ptr) {
    mul_normal_args *args = (mul_normal_args *) ptr;
-----------------------------------------------------

    const BN_ULONG* a = args->a;
    const BN_ULONG* b = args->b;
    BN_ULONG* r = args->r;
    int na = args->na;
    int nb = args->nb;
    args->nr = na + nb;
    BN_ULONG* rr;

-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------
-----------------------------------------------------

-----------------------------------------------------
    rr = &(r[na]);
    if (nb <= 0) {
        (void)bn_mul_words(r, a, na, 0);
        pthread_exit(NULL);
    } else
        rr[0] = bn_mul_words(r, a, na, b[0]);

    for (;;) {
        if (--nb <= 0)
            pthread_exit(NULL);
        rr[1] = bn_mul_add_words(&(r[1]), a, na, b[1]
        if (--nb <= 0)
            pthread_exit(NULL);
        rr[2] = bn_mul_add_words(&(r[2]), a, na, b[2]
        if (--nb <= 0)
            pthread_exit(NULL);
        rr[3] = bn_mul_add_words(&(r[3]), a, na, b[3]
        if (--nb <= 0)
            pthread_exit(NULL);
        rr[4] = bn_mul_add_words(&(r[4]), a, na, b[4]
        rr += 4;
        r += 4;
        b += 4;
    }

    pthread_exit(NULL);
}

void print_arr(BN_ULONG *a, int n) {
    for (int i = 0; i < n; i++) {
        printf("%lx\n", a[i]);
    }
}
```



```
--------------------------------------------------------------------
--------------------------------------------------------------------
--------------------------------------------------------------------
--------------------------------------------------------------------
--------------------------------------------------------------------
--------------------------------------------------------------------
--------------------------------------------------------------------
--------------------------------------------------------------------
--------------------------------------------------------------------
--------------------------------------------------------------------
--------------------------------------------------------------------
--------------------------------------------------------------------
--------------------------------------------------------------------
--------------------------------------------------------------------
--------------------------------------------------------------------
--------------------------------------------------------------------
--------------------------------------------------------------------
--------------------------------------------------------------------
--------------------------------------------------------------------
--------------------------------------------------------------------
--------------------------------------------------------------------
--------------------------------------------------------------------
--------------------------------------------------------------------
--------------------------------------------------------------------
--------------------------------------------------------------------
--------------------------------------------------------------------
--------------------------------------------------------------------
--------------------------------------------------------------------
--------------------------------------------------------------------
--------------------------------------------------------------------
--------------------------------------------------------------------
--------------------------------------------------------------------
--------------------------------------------------------------------
--------------------------------------------------------------------
--------------------------------------------------------------------
--------------------------------------------------------------------
--------------------------------------------------------------------
--------------------------------------------------------------------
--------------------------------------------------------------------
--------------------------------------------------------------------
--------------------------------------------------------------------
--------------------------------------------------------------------
--------------------------------------------------------------------
--------------------------------------------------------------------
--------------------------------------------------------------------
--------------------------------------------------------------------
--------------------------------------------------------------------
--------------------------------------------------------------------
--------------------------------------------------------------------
--------------------------------------------------------------------
--------------------------------------------------------------------
--------------------------------------------------------------------
--------------------------------------------------------------------
--------------------------------------------------------------------
--------------------------------------------------------------------
--------------------------------------------------------------------
--------------------------------------------------------------------
--------------------------------------------------------------------
--------------------------------------------------------------------
--------------------------------------------------------------------
--------------------------------------------------------------------
--------------------------------------------------------------------
--------------------------------------------------------------------
--------------------------------------------------------------------
--------------------------------------------------------------------
}

void bn_mul_low_normal(BN_ULONG *r, BN_ULONG *a, BN_U
{
    bn_mul_words(r, a, n, b[0]);

+-- 17 lines: for (;;) {----------------------------
```

```
void bn_mul_normal(BN_ULONG *r, BN_ULONG *a, int na,
{
    if (na < nb) {
        int itmp;
        BN_ULONG *ltmp;

        itmp = na;
        na = nb;
        nb = itmp;
        ltmp = a;
        a = b;
        b = ltmp;
    }

    memset(r, 0, (na+nb)*sizeof(BN_ULONG));

    pthread_t thr[NUM_THREADS];
    int rc;


    /* create a thread_data_t argument array */
    mul_normal_args thr_data[NUM_THREADS];
    // BN_ULONG* r_tmp[NUM_THREADS];

    /* create threads, divide array */
    int new_nb = nb/NUM_THREADS;
    int l_idx = 0;

    for (int i = 0; i < NUM_THREADS; ++i) {
        if (i == (NUM_THREADS - 1))
            thr_data[i].nb = new_nb + nb % NUM_THREAI
        else
            thr_data[i].nb = new_nb;

        l_idx = new_nb * i;
        thr_data[i].a = a;
        thr_data[i].b = &(b[l_idx]);
        thr_data[i].na = na;
        thr_data[i].r = (BN_ULONG *) malloc((thr_data
        if (thr_data[i].r == NULL) {
            fprintf(stderr, "error: malloc error \n")
            exit(EXIT_FAILURE);
        }

        if ((rc = pthread_create(&thr[i], NULL, bn_mu
          fprintf(stderr, "error: pthread_create, rc:
          exit(EXIT_FAILURE);
        }
    }

    /* block until all threads complete */
    BN_ULONG carry;
    for (int i = 0; i < NUM_THREADS; ++i) {
        pthread_join(thr[i], NULL);

        int nr = thr_data[i].nr;
        carry = bn_add_words(r, r, thr_data[i].r, nr)

        if (i != NUM_THREADS - 1) {
            r[nr] = carry;
        }
        r += thr_data[i].nb;
        free(thr_data[i].r);
    }

}

void bn_mul_low_normal(BN_ULONG *r, BN_ULONG *a, BN_U
{
    bn_mul_words(r, a, n, b[0]);

+-- 17 lines: for (;;) {----------------------------
```

/tmp/XoKxya_bn_sqr.c

```
+-- 75 lines: Copyright 1995-2018 The OpenSSL Project
            k = j + j;
            if (al == j) {
                if (bn_wexpand(tmp, k * 2) == NULL)
                    goto err;
                bn_sqr_recursive(rr->d, a->d, al, tmp
            } else {
                if (bn_wexpand(tmp, max) == NULL)
                    goto err;
                bn_sqr_normal(rr->d, a->d, al, tmp->c
            }
        }
#else
        if (bn_wexpand(tmp, max) == NULL)
            goto err;
        bn_sqr_normal(rr->d, a->d, al, tmp->d);
+--105 lines: #endif----------------------------------
    else
        memset(&t[n2], 0, sizeof(*t) * n2);
    bn_sqr_recursive(r, a, n, p);
    bn_sqr_recursive(&(r[n2]), &(a[n]), n, p);

    /*-
     * t[32] holds (a[0]-a[1])*(a[1]-a[0]), it is neg
     * r[10] holds (a[0]*b[0])
     * r[32] holds (b[1]*b[1])
     */

    c1 = (int)(bn_add_words(t, r, &(r[n2]), n2));

    /* t[32] is negative */
    c1 -= (int)(bn_sub_words(&(t[n2]), t, &(t[n2]), r

    /*-
     * t[32] holds (a[0]-a[1])*(a[1]-a[0])+(a[0]*a[0]
     * r[10] holds (a[0]*a[0])
     * r[32] holds (a[1]*a[1])
     * c1 holds the carry bits
     */
    c1 += (int)(bn_add_words(&(r[n]), &(r[n]), &(t[n2
    if (c1) {
        p = &(r[n + n2]);
        lo = *p;
+-- 18 lines: ln = (lo + c1) & BN_MASK2;-------------
```

crypto/bn/bn_sqr.c

```
+-- 75 lines: Copyright 1995-2018 The OpenSSL Project
            k = j + j;
            if (al == j) {
                if (bn_wexpand(tmp, k * 2) == NULL)
                    goto err;
                bn_sqr_recursive(rr->d, a->d, al, tmp
            } else {
                if (!bn_mul_fixed_top(r, a, a, ctx))
                    goto err;
-----------------------------------------------------
            }
        }
#else
        if (bn_wexpand(tmp, max) == NULL)
            goto err;
        bn_sqr_normal(rr->d, a->d, al, tmp->d);
+--105 lines: #endif----------------------------------
    else
        memset(&t[n2], 0, sizeof(*t) * n2);
    bn_sqr_recursive(r, a, n, p);
    bn_sqr_recursive(&(r[n2]), &(a[n]), n, p);

    /*-
     * t[n2] holds (a[0]-a[1])*(a[1]-a[0]), it is neg
     * r[0] holds (a[0]*b[0])
     * r[n2] holds (b[1]*b[1])
     */

    c1 = (int)(bn_add_words(t, r, &(r[n2]), n2));

    /* t[n2] is negative */
    c1 -= (int)(bn_sub_words(&(t[n2]), t, &(t[n2]), r

    /*-
     * t[n2] holds (a[0]-a[1])*(a[1]-a[0])+(a[0]*a[0]
     * r[0] holds (a[0]*a[0])
     * r[n] holds (a[1]*a[1])
     * c1 holds the carry bits
     */
    c1 += (int)(bn_add_words(&(r[n]), &(r[n]), &(t[n2
    if (c1) {
        p = &(r[n + n2]);
        lo = *p;
+-- 18 lines: ln = (lo + c1) & BN_MASK2;-------------
```