

题目描述:

a,b,c 均为大于 0 的自然数:

$$\frac{a}{b+c} + \frac{b}{a+c} + \frac{c}{a+b} = 4$$

1. 编一个程序证明在 $a < 65536, b < 65536, c < 65536$ 情况下无解
2. 编一个程序证明以下值是正确的:

a=4373612677928697257861252602371390152816537558161613618621437993378423467772036

b=36875131794129999827197811565225474825492979968971970996283137471637224634055579

c=154476802108746166441951315019919837485664325669565431700026634898253202035277999

分析:

$$\frac{a}{b+c} + \frac{b}{a+c} + \frac{c}{a+b} = 4$$

该等式是一个具有对称性的分式，初步来看，问题 2 是验证性的问题，只是要处理一下大数的计算，很快就可以得出结论。代码中相关函数为

```
int check_ans_abc0(const char* a, const char* b, const char* c, char(*rs) [256]);
```

对于问题 1 编程实现求解，需要解决一下问题：

- 1) 计算机系统内浮点数运算丢失精度，以及相等如何判定；
- 2) 大数的加减乘除运算；
- 3) 算法的效率，直接暴力求解，时间复杂度 $O(n^3)$,就以 65536 的上限来说，大约需要 $65536 \times 35536 \times 65536 = 2^{48}$ ，需要耗费巨大的计算资源；

下面对以上三个问题逐一解决：

- 1) 计算机系统内浮点数运算丢失精度，以及相等如何判定；

在正整数范围内，

$$\frac{a}{b+c} + \frac{b}{a+c} + \frac{c}{a+b} = 4$$

可等价变形为：

$$a^3 + b^3 + c^3 = 5 \times a \times b \times c + 3(a^2 \times (b + c) + b^2 \times (a + c) + c^2 \times (a + b)) \quad \textcircled{1}$$

这样的做的目的：

- a) 避免浮点数运算;
- b) 使用整型判断相等: 最终只需判断代表左边结果的字符串和代表右边结果的字符串是否相等即可;

代码中相关函数为:

```
int check_abc0(const char* a, const char* b, const char* c, char(*rs) [256])
```

该函数的最后使用的是 `return !strcmp(sumabcCube, tmp[0]);` `sumabcCube` 代表①式左边的值, `tmp[0]`代表①式右边的值, 返回值根据 `strcmp` 比较:二者相等返回 0,否则返回 1。

- c) 只需要大数加法和乘法;

代码中相关函数为:

```
void big_data_multiply( const char* num1, const char* num2, char* sum );
```

```
void big_data_add( const char* num1, const char* num2, char* sum );
```

2) 大数的加减乘除运算;

在 1) 分析的基础上, 只需要实现大数的加法和乘法, 这个有两个思路:

- a. 一种思路是将大整数截成几段存放到一个整型数组中,计算的时候分段计算并乘以相应的权重;

如 $12345678=1234\times10000+5678\times1$ ，因此数组中的值[1234],[5678]

b. 采用字符数组，存放大数的每一位，本方案采取这种方式；

3) 效率问题；

暴力求解，时间复杂度 $O(n^3)$ ，时间复杂度比较高，本方案从两个方面解决这个问题：

a. 缩小搜索范围

$$\frac{a}{b+c} + \frac{b}{a+c} + \frac{c}{a+b} = 4$$

是一个对称等式，因此不妨假设 $a \leq b \leq c$

$$\begin{aligned} \frac{c}{a+b} < 4 & \quad \text{也即} \quad c < 4 \times (a+b) \quad , \quad \text{同理} \\ \frac{a}{b+c} < 4 & \quad \text{也即} \quad c > \left(\frac{a}{4} - b\right) \\ \frac{b}{a+c} < 4 & \quad \text{也即} \quad c > \left(\frac{b}{4} - a\right) \end{aligned}$$

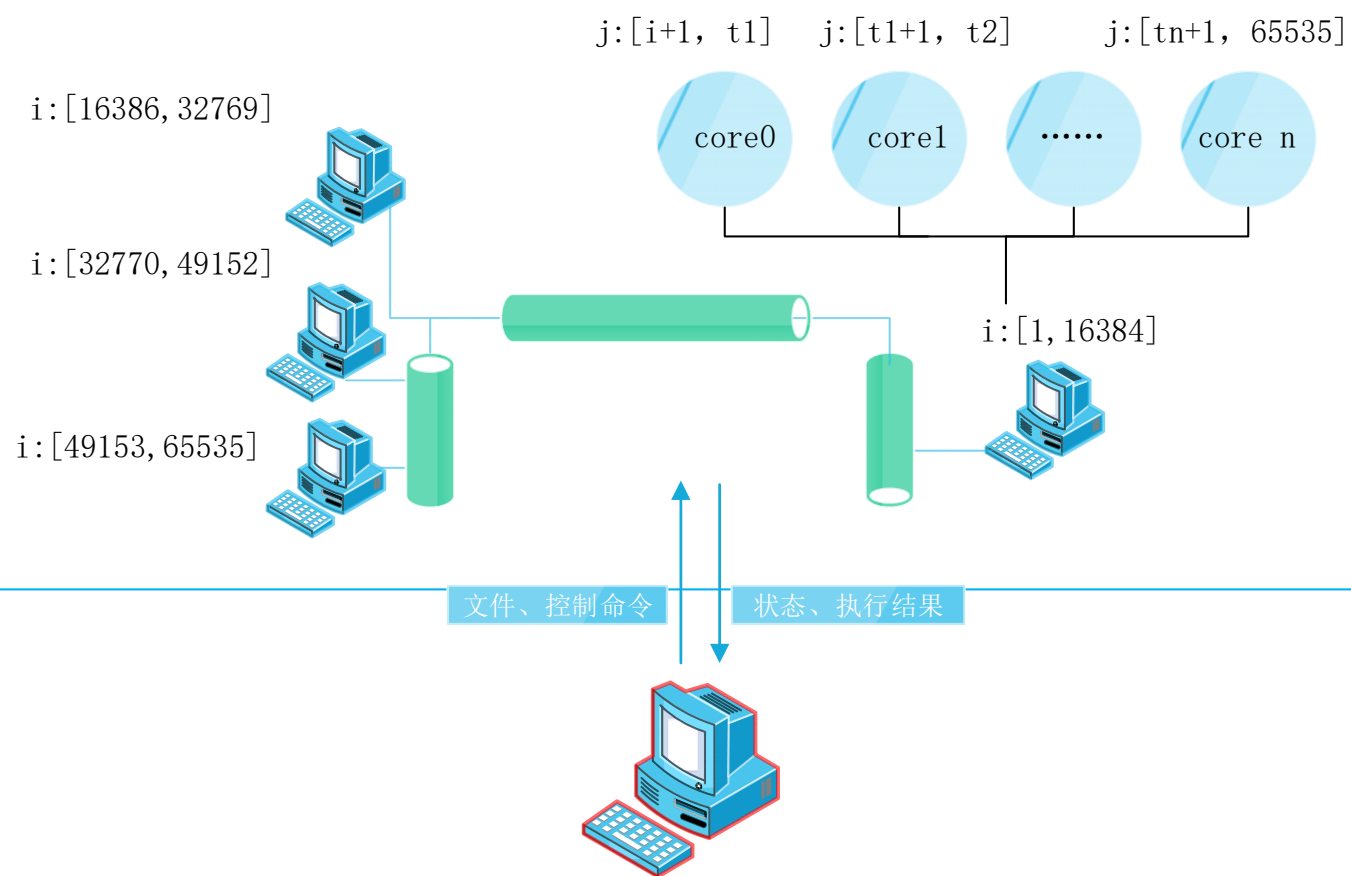
因此 c 的范围是： $\max\left\{\left(\frac{b}{4} - a\right), \left(\frac{a}{4} - b\right)\right\} < c < 4 \times (a+b)$ ，由此可以得到下面的搜索范围框架：

分布式计算及多核计算

通过上面的处理之后，一定程度上缩小了计算规模，但是算法时间复杂度的数量级并没有改变，因此在单核甚至于单个节点上计算，效率是比较低的。

```
for (i = LOW; i < HIGH; i++)
{
    sprintf(a_str, "%d", i);
    for (j = local_val; j < min(local_val + STEP0, LIMIT) ; j++)
    {
        sprintf(b_str, "%d", j);
        for (k = max(max(i / 4 - j, j / 4 - i), local_val); k <= min(4 * (i + j), LIMIT); k++)
        {
            sprintf(c_str, "%d", k);
            local_count += check_ans_abc0(a_str, b_str, c_str, NULL);
        }
    }
}
```

计算模型



如图所示，本方案使用分布式计算和多核计算的思想：对于上面框架中的三个循环，可以将最外层的循环根据节点的数目，分配到不同的节点上去计算，也即对于单个节点来说，最外层循环的参数 i 范围是整个范围[1,65535]的一个子集，各个节点最外层范围不重叠。第二层循环可以根据节点上核心数目，分配到同一个节点的不同核上去计算。整个系统由一个控制节点来保持心跳，检测各个计算节点任务的完成情况，模型如图所示，当然也可以由计算节点主动通知控制节点计算结果，本方案采取前者。

代码实现中第一层循环的划分由 shell 脚本实现,如下图所示。第二层循环在 C 代码中实现，见后文的描述：

```
#分发任务到各节点
while [ ${low} -le ${range_limit} ];
do
    high=$(( ${low}+${step} ))
    #在某个节点上进行[low, high]范围内的最外层循环
    ssh -f "${ip_table_array[${index}]}" "cd ${tencent2017_interview_dir}/ && ./tencent2017_interview Q1 ${low} ${high}"
    sleep 1s
    print_message "MULTEXU_INFO" "${ip_table_array[${index}]} execute range[${low}, ${high}]..."
    wait
    low=${high}
    index=$(((${index}+1)%${nodes_num}))
done
```

当然，这时候带来了一些节点之间通信、线程之间同步互斥的开销，但是与其带来的性能提升来说，显然这个开销是微不足道的。假设有四台计算节点，1 台控制节点，每个节点 24 核（本方面不占用全部资

源，每个节点用十个线程，分别在不同的核上计算）。

具体实现中考虑的问题：

- i. 首先要考虑在单个节点中，对 j 的范围进行划分。对于任何一个线程，要能够清楚的知道该线程负载计算的范围，线程之间不能重叠，因此表示边界的变量的更新应该是原子的。很容易想到使用锁机制，或者 `atomic_t`。考虑到本方案应对的是计算密集型的问题，锁持有的时间比较短，选用自旋锁控制。这段代码中，每个线程启动后，执行到该部分，对维持计算范围位置的下标变量加锁，读取数据后进行更新，并处理边界问题，再开始计算部分。类似问题还有 CPU 核心选择的问题：使用锁机制维持表示核心编号的下标，每一个核心被分配给一个计算线程后，核心编号原子递增 1，尽量保证每个现在都在独立的核心上运行。解的数量，每个线程计算完毕后都要将自己求得的解的数量加到一个全局变量中去（这个值最终被发送给控制节点），因此也要做相关的控制，机制都是

类似的，在此不再赘述。

- ii. 其次是单个节点计算完毕后，主节点如何汇总的问题。本方案是这样做的：每个节点待该节点上所有的线程计算完毕后，会将解的数量写入到一个文件中。代码中相关函数：

```
int write_result(char* name_prefix, int result)
```


```
pthread_spin_lock(&global_start_index_a_lock);  
//global_start_index_a的意义是global_start_index_a 前的范围都有线程在计算了  
local_val = global_start_index_a;  
//更新global_start_index_a的位置  
global_start_index_a += STEP0;  
if (global_start_index_a > LIMIT)  
{  
    global_start_index_a = global_start_index_a;  
}  
pthread_spin_unlock(&global_start_index_a_lock);
```

比较重要的代码段

```

for (i = LOW; i < HIGH; i++)
{
    for (j = local_val; j < min(local_val + STEP0, LIMIT) ; j++)
    {
        for (k = max(max(i / 4 - j, j / 4 - i), local_val); k <= min(4 * (i + j), LIMIT); k++)
        {
            sprintf(a_str, "%d", i);
            sprintf(b_str, "%d", j);
            sprintf(c_str, "%d", k);
            local_count += check_ans_abc0(a_str, b_str, c_str, NULL); //
        }
    }
}
//计算完毕，更新解的数量
pthread_mutex_lock(&solution_num_mutex);
solution_num += local_count;
pthread_mutex_unlock(&solution_num_mutex);

```



主节点会定期的检测每个节点上该文本中的值，一旦发现某个文本中该值大于 0，表示找到方程的解，立刻终止计算。如果所有节点计算完毕，没有任何一个节点返回大于 0 的值，说明无解。本方案采用被动检测的形式，使用了一个启发式的检测方式：任务发布后，等待一个 $\text{time } s$ 的时间后检测一次，如果没有计算完毕或者没有找到解，就等待 $\frac{\text{time}}{2} s$ 后再检测，如果还是没有达到算法终止条件，再等待 $\frac{\text{time}}{4} s$ 当然等待时间不能小于某个设定的下限值（这部分在 shell 脚本中完成）。

```

while [ true ];
do
    print_message "MULTEXU_INFO" "waiting nodes which
    its ip in ${iptable_file} computing,the next check time will be ${sleeptime}s later..."
    sleep ${sleeptime}s
    count=0

    for host_ip in ${ip_table_array[*]}
    do
        retval=`ssh -f ${host_ip} "cat ${signal_file}"`
        #retval=$?
        if [ "x$retval" != "x" ];then
            if [ $retval -gt 0 ];then
                print_message "MULTEXU_INFO" "node ${host_ip} find at least one solution..."
                #终止所有任务
                #send_execute_statu_signal "" ""
                __clear
                exit 0
            elif [ $retval -eq 0 ];then
                (( count += 1 ))
                #全部执行完毕 且没有找到解
                if [ ${count} -eq ${nodes_num} ];then
                    break 2
                fi
            fi
        fi
    done
    if [ $sleeptime -gt $limit ];then
        let sleeptime/=2
    fi
done

```

结果:

问题 1

首先, 选择一个比较小的范围看一下整个程序的运行流程 (设置 65536 需要较长的时间), 示例中设定 range 参数为 32, 也即 i(a)的范围为[1,32], j(b)的范围为[1,65532],k(c)的范围根据 i、j 确定, 主节点在所有计算节点计算完毕后, 汇总各节点的情况, 得出结论。见演示视频。

问题 2

```
root@localhost tmp#  
[root@localhost tmp]# gcc tencent2017_interview.c -o tencent2017_interview -lpthread  
[root@localhost tmp]# ./tencent2017_interview Q2  
a*a*a + b*b*b + c*c*c = 5 * a * b * c + 3 * (aa * (b + c) + bb * (a + c) + cc * (a + b))  
LOG: 3736518211673479436470049110874577380698923709919531439206826840702394047135035516769185657470113743369196601210874419010393735792147176255552929703365234981844243237459155159444532749802492211431663730410572151494952365148599617232539286194  
LOG: 3736518211673479436470049110874577380698923709919531439206826840702394047135035516769185657470113743369196601210874419010393735792147176255552929703365234981844243237459155159444532749802492211431663730410572151494952365148599617232539286194  
[root@localhost tmp]#
```

可见, 当 a、b、c 的值分别为

$$a=4373612677928697257861252602371390152816537558161613618621437993378423467772036$$

$$b=36875131794129999827197811565225474825492979968971970996283137471637224634055579$$

$$c=154476802108746166441951315019919837485664325669565431700026634898253202035277999$$

时, $a^3 + b^3 + c^3$ 和 $5 \times a \times b \times c + 3(a^2 \times (b + c) + b^2 \times (a + c) + c^2 \times (a + b))$ 的值均为

3736518211673479436470049110874577380698923709919531439206826840702394047135035516769185
6574701137433691966012108744190103937357921471762555529297033652349818442432374591551594
44532749802492211431663730410572151494952365148599617232539286194, 因而等式成立。

说明:

- 1) 主节点控制部分是在本人实现的一个小工具基础之上实现的, 源代码:

<https://github.com/ShijunDeng/LustreTools>

<https://github.com/ShijunDeng/multexu>

- 2) 本方案全部源码:

<https://github.com/ShijunDeng/aiocc/tree/master/batch/tmp>

改进与提升

1. 使用字符数组表示一个整数, 每次计算都要 $O(n)$ 时间复杂度加法、和 $O(n*n)$ 时间复杂度的乘法;
2. 对于

$x*x*x + y*y*y + z*z*z == 5 * x * y * z + 3 * (x*x * (y + z) + y*y * (x + z) + z*z * (x + y))$ 的判定,我多次调用的加法函数、乘法函数;

3. 对于 x, y, z 每个操作数还需要转换为字符串,需要一定时间;

所以大大降低了整个算法的效率低。如果,采取分布式计算的思想,单节点计算直接使用 64 位整数,效率会提高不少。但是,使用四个节点计算的时候,观察到:

其它三个节点很快计算完毕了,其中负责最外层循环 ($x:1-16385$) 的那个节点的计算迟迟没有结束,而且使用 top 看每个核的情况,发现只是其中几个核没有计算完毕。

先前没有考虑到的另一个问题:

最内层 z 的循环随着外层的 x, y 递增是【非均匀分布】的。因此,结点间划分任务和节点上线程之间划分任务的时候,需要一个类似负载均衡的机制,事实上,当 $x+y$ 大于 32768 的时候,最内层的循环几乎不用执行了。将任务更细粒度,增加“计算资源池”的思想,尽量保持每个计算节点以及每个计算节点上的任务负载均衡,保持满载状态。对于最外层的循环,设置一个粒度,设节点的数目为 `nodes_num`,则粒度可以设置为

$\text{granu} = \text{nodes_num} * 100$, $\text{step} = 65536 / \text{granu}$, 那么第一个节点一次的计算任务就是 $[1, 65536 / \text{granu}]$, 第二个节点一次的计算任务是 $[1 + 65536 / \text{granu}, 1 + 2 * 65536 / \text{granu}] \dots\dots$, 主节点不断检测个节点任务的执行情况。一旦发现某个节点的任务计算完毕, 而总的计算任务还没结束, 就划分一个新的任务给这个空闲的节点。

```

#分发任务到各节点
while [ ${low} -le ${range_limit} ];
do
    #在某个节点上进行[low, high]范围内的最外层循环
    for host_ip in ${ip_table_array[*]}
    do
        retval1=`ssh -f ${host_ip} "cat ${solution_signal_file}"`
        #retval=?
        #先检测是否找到解了
        if [ "x${retval1}" != "x" ];then
            if [ ${retval1} -gt 0 ];then
                print_message "MULTEXU_INFO" "node ${host_ip} find at least one solution..."
                #终止所有任务
                #send_execute_statu_signal "" ""
                __clear
                exit 0
            fi
        fi
        #若当前节点任务完成则分派一个新的任务
        retval2=`ssh -f ${host_ip} "cat ${execute_signal_file}"`
        if [ "x${retval2}" == "x0" ];then
            high=$(( ${low}+${step} ))
            ssh -f ${host_ip} ":> ${execute_signal_file}"
            print_message "MULTEXU_INFO" "distribute a task to node ${host_ip}, range[${low}, ${high}]..."
            ssh -f "${host_ip}" "cd ${tencent2017_interview_dir}/ && ./tencent2017_interview Q1 ${low} ${high}"
            low=${high}
            sleep 1s
        fi
    done
    sleep 1s
done

```


单个节点上每个线程的任务也做类似的处理：采用线程计算池的思想，每个节点暂时分配一小段任务，线程计算完毕后发现当前节点的任务还没有计算完毕，就再去获取新的计算任务。这样可以避免计算资源浪费和长时间等待的情况，保证各节点基本是差不多同时计算完毕的。

```
pthread_spin_lock(&global_start_index_a_lock);
//global_start_index_a 的意义是 global_start_index_a 前的范围都有线程在计算了
local_val = global_start_index_a;
while (local_val < LIMIT)
{
    //更新 global_start_index_a 的位置
    global_start_index_a += STEP0;
    if (global_start_index_a > LIMIT)
    {
        global_start_index_a = LIMIT;
    }
    pthread_spin_unlock(&global_start_index_a_lock);
    if (CPU_ISSET(local_cpu_core, &get))
    {
        pid = getpid();
        tid = pthread_self();
        printf("[LOG] pid:%u, u_tid:%u (0x%x), k_tid:%u, processor_no:%2d, x_range:[%d, %d), y_range[%d, %d)\n", \
            (unsigned int)pid, (unsigned int)tid, (unsigned int)tid, (unsigned int)gettidv1(), \
            local_cpu_core, LOW, HIGH, local_val, local_val + STEP0);
    }
}
```

```

else
{
    printf("[LOG]k_tid:%u, processor_no:%2d, x_range :[%d, %d), y_range[%d, %d)\n", (unsigned int)gettidv1(),\
        local_cpu_core, LOW, HIGH, local_val, local_val + STEP0);
}
for(x = LOW; x < HIGH && !local_count; x++)
{
    for(y = local_val; y < min(LIMIT, local_val + STEP0) && !local_count; y++)
    {
        for(z = min(4 * (x + y), LIMIT); z > 2 * (x + y) && !local_count; z--)
        {
            if( (y + z) <= x)
            {
                break;
            }
            if(x * (x + z) * (x + y) + y * (y + z) * (x + y) + z * (y + z) * (x + z) == 4 * (x + y) * (y + z) * (x + z))
            {
                local_count ++;
            }
        }
    }
}
pthread_spin_lock(&global_start_index_a_lock);
//global_start_index_a 的意义是 global_start_index_a 前的范围都有线程在计算了
local_val = global_start_index_a;
} //while

```

```
//这里一定要释放锁
pthread_spin_unlock(&global_start_index_a_lock);
```

测试:

机器配置

CPU	Intel(R) Xeon(R) CPU E5-2620 0 @ 2.00GHz 2x
内存容量	16384 MB
内存详情	16GB DDR3 RAM
硬盘容量	1168 GB
硬盘详情	2x SAS Disks (15000RPM, 146GB) configured RAID1,6x SAS Disks (15000RPM, 146GB)configured RAID5
网卡信息	Mellanox InfiniBand QDR 40Gb/s NIC
操作系统	RedHat

单节点代码以及计算时间大概 3 小时左右

File Edit View Search Terminal Help

```
[root@localhost development]# g++ cc.cc -o cc
[root@localhost development]# date && time ./cc && date
Tue Mar 28 23:03:15 CST 2017
not find x,y,z

real    196m17.526s
user    196m17.485s
sys     0m0.004s
Wed Mar 29 02:19:32 CST 2017
[root@localhost development]#
```

```
#include <iostream>
#include <stdint.h>
using namespace std;

int
main ()
{
    uint64_t  x = 0, y = 0, z = 0;
    for (x = 1; x < 65536; x++)
    {
        for (y = x; y < 65536; y++)
        {
            //feature:  $2(x+y) < z < 4(x+y)$ 
            for (z = 4 * (x + y); z > 2 * (x + y) && z < 65536; z--)
            {
```

```

        if ((y + z) <= x)
            break;
        if (x * (x + z) * (x + y) + y * (y + z) * (x + y) +
            z * (y + z) * (x + z) == 4 * (x + y) * (y + z) * (x + z))
        {
            cout << "x=" << x << ",y=" << y << ",z=" << z << endl;
            return 0;
        }
    }
}

cout << "not find x,y,z" << endl;
return 0;
}

```

采用本方案计算时间 2070 秒

```
[LOG] pid:5768, u_tid:3078482800 (0xb77deb70), k_tid:5769, proce
[LOG] pid:5768, u_tid:3067992944 (0xb6dddb70), k_tid:5770, proce
[LOG] pid:5768, u_tid:3078482800 (0xb77deb70), k_tid:5769, proce
[LOG] pid:5768, u_tid:3042966384 (0xb55ffb70), k_tid:5772, proce
[LOG] pid:5768, u_tid:3005213552 (0xb31feb70), k_tid:5775, proce
[LOG] pid:5768, u_tid:3055549296 (0xb61ffb70), k_tid:5771, proce
[LOG] pid:5768, u_tid:3015703408 (0xb3bffb70), k_tid:5774, proce
[LOG] pid:5768, u_tid:3030383472 (0xb49ffb70), k_tid:5773, proce
MULTEXU_INFO:No new tasks, waiting for the remaining tasks to fi
MULTEXU_INFO:waiting nodes which its ip in nodes_compute.out com
MULTEXU_INFO:No solution in range [1, 65536] ...
MULTEXU_INFO:computing process finished...
MULTEXU_INFO:Total time spent:2070.673465 s
Wed Mar 29 10:46:32 CST 2017
```