

操作系统课程试验

高级进程间通信问题——快速排序问题

BobAnkh

June 2021

编程平台: Ubuntu 18.04.5 LTS (GNU/Linux 4.15.0-142-generic x86_64)

编程语言: C/C++, 采用了部分 C++11 特性

一. 实验题目

1. 问题描述

对于有 1,000,000 个乱序数据的数据文件执行快速排序。

2. 实验步骤

- (1) 首先产生包含 1,000,000 个随机数（数据类型可选整型或者浮点型）的数据文件；
- (2) 每次数据分割后产生两个新的进程（或线程）处理分割后的数据，每个进程（线程）处理的数据小于 1000 以后不再分割（控制产生的进程在 20 个左右）；
- (3) 线程（或进程）之间的通信可以选择下述机制之一进行：
 - 管道（无名管道或命名管道）
 - 消息队列
 - 共享内存
- (4) 通过适当的函数调用创建上述 IPC 对象，通过调用适当的函数调用实现数据的读出与写入；
- (5) 需要考虑线程（或进程）间的同步；
- (6) 线程（或进程）运行结束，通过适当的系统调用结束线程（或进程）。

二. 设计思路

为了方便传递快速排序的左右界，定义了一个结构体 `Qparam` 来存放这两个界，它也是指示每一个子排序任务（即每一个子排序任务只需要知道左右界即可进行）。定义了一个任务队列 `queue<Qparam> job_queue` 用于存放目前需要线程来进行执行的任务，通过信号量 `jobs` 来标识队列中有多少个任务待处理；主进程中在完成相关的初始化之后，会创建第一个工作线程并将总任务加入到任务队列中，从而开始整个排序任务。每次排序或分划之后，会累加已经排好序的数量，当这个数量与总数相等时，认为排序完成。

每个工作线程都运行一个函数，该函数主要反复运行以下过程：

- 检查是否已经排序完毕，如果是则提升排序完毕信号量 `finished`，从而通知主进程取消所有子线程并输出结果；
- 从任务队列中取出一个新的任务（如果有则取，否则等待）；
- 如果任务数据量小于下阈值则直接调用 `qsort` 进行排序，否则进行划分，将划分之后的两个任务依次放入任务队列中，并逐次提升信号量 `jobs`，同时每放入一个任务则检查当前线程数是否达到上限，若没有则创建新线程。

对于本程序各工作线程而言，整体结构图如下 1 所示，伪码算法如下 1 所示：

Algorithm 1: IPC 主要排序函数伪码算法 (`mutex` 是互斥锁, `jobs` 和 `finished` 是信号量, `queue` 是任务队列)

```
1 Function quicksort():
2   while TRUE do
3     if sort_finish = True then
4       | up(&finished)
5     lock(&mutex)
6     param = queue.front()
7     queue.pop()
8     unlock(&mutex)
9     if param.right - param.left < LIMIT then
10      | qsort(param)
11    else
12      mid = partition(param)
13      lock(&mutex)
14      queue.push(Qparam(param.left, mid-1))
15      up(jobs)
16      if now_threads_num ≤ max_thread_num then
17        | create_new_thread()
18      queue.push(Qparam(mid-1, param.right))
19      up(jobs)
20      if now_threads_num ≤ max_thread_num then
21        | create_new_thread()
22      unlock(&mutex)
```

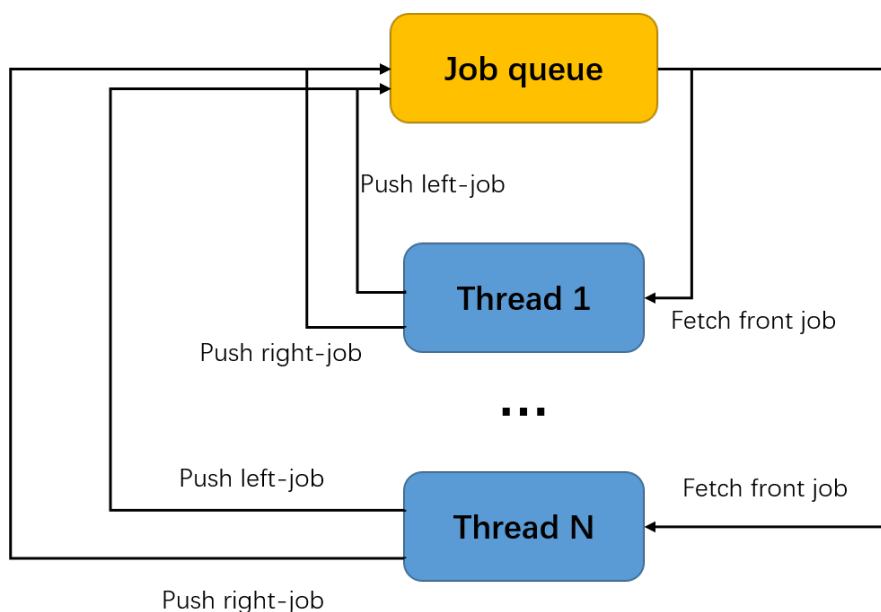


图 1: 工作线程整体结构图

三. 程序结构

本程序在具体实现上, 主要采用了 `pthread`, `semaphore` 以及 `sys/mman` 这三个库进行实现, 直接使用了 `algorithm` 库中的 `qsort` 函数直接排序数据量小于下阈值时的情况, 另外还主要在队列 `queue` 上采用了 C++11 的相关特性, 从而方便编程和数据结构的实现, 同时这样也利于跨平台。

为了告知工作线程快速排序的左右界, 定义了一个结构体 `Qparam` 来存放这两个界, 它也是指示每一个子排序任务 (即每一个子排序任务只需要知道左右界即可进行), 线程之间的控制信息交换主要依赖它和存放它的一个队列 `queue<Qparam> job_queue`。该结构体定义如下:

```

struct Qparam
{
    int left;
    int right;
    Qparam(int l, int r)
    {
        left = l;
        right = r;
    }
};

```

利用信号量 `sem_t jobs` 来标识队列中存在的任务数量。每当一个线程在一次循环中产生一个新任务时, 都会在互斥锁加锁的情况下, 将任务推入该队列, 并提升该信号量。

主函数在初始化相关变量后, 会想创建共享内存, 将数据文件 `input.dat` 映射进来, 通过这个共享内存区让所有工作线程进行数据通信, 此后会创建第一个工作线程并将总任务加入到任务队列中, 从而开始整个排序任务。每个工作线程将执行快速排序函数, 快速排序函数内部的循环每次执行时将检查是否所有的数都已经排序完毕——这一点是依赖已经排序的数量是否与总共需要排序的数量相等来做出判断的, 如果已经排序完毕则报告给主进程, 从而结束所有工作线程并将结果输出到文件中, 关闭共享内存区。

下介绍最主要的、工作线程所运行的 quicksort 函数。

1. quicksort 函数

这一函数的代码如下：

```
void *quicksort(void *args)
{
    int *buffer = (int *)args;
    while (true)
    {
        // 检查是否完成排序
        if (sortedNum == SORT_NUM && sorted == false)
        {
            pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);
            pthread_mutex_lock(&mutex);
            if (sorted == false)
            {
                sorted = true;
                sem_post(&finished);
            }
            pthread_mutex_unlock(&mutex);
            pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
        }
        // 抓取新任务
        sem_wait(&jobs);
        pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);
        pthread_mutex_lock(&mutex);
        Qparam param = job_queue.front();
        cout << "[FETCH JOB] left:" << param.left << " right:" << param.right
              << endl;
        job_queue.pop();
        pthread_mutex_unlock(&mutex);
        if (param.left <= param.right)
        {
            // 直接进行排序
            if (param.right - param.left < DIRECT_SORT)
            {
                qsort(buffer + param.left, param.right - param.left + 1,
                      sizeof(int), cmp);
                pthread_mutex_lock(&mutex);
                sortedNum += param.right - param.left + 1;
                cout << "[DIRECT SORT] left:" << param.left << " right:" <<
                     param.right << " sorted:" << sortedNum << endl;
                pthread_mutex_unlock(&mutex);
            }
            else
            {
                int pos = partition(buffer, param);
                pthread_mutex_lock(&mutex);
```

```

        sortedNum++;
        // 向队列中推入左作业
        job_queue.push(Qparam(param.left, pos - 1));
        sem_post(&jobs);
        cout << "[PUSH JOB-LEFT] left:" << param.left << " rightL" <<
            pos - 1 << " sorted:" << sortedNum << endl;
        if (created_thread < MAX_THREAD)
        {
            int res = pthread_create(&pID[created_thread], NULL,
                quicksort, (void *) (buffer));
            if (res)
            {
                cout << "Fail to create thread " << created_thread <<
                    " [ERROR]: " << strerror(res) << endl;
                exit(-1);
            }
            created_thread++;
        }

        // 向队列中推入右作业
        job_queue.push(Qparam(pos + 1, param.right));
        sem_post(&jobs);
        cout << "[PUSH JOB-RIGHT] left:" << pos + 1 << " right:" <<
            param.right << " sorted:" << sortedNum << endl;
        if (created_thread < MAX_THREAD)
        {
            int res = pthread_create(&pID[created_thread], NULL,
                quicksort, (void *) (buffer));
            if (res)
            {
                cout << "Fail to create thread " << created_thread <<
                    " [ERROR]: " << strerror(res) << endl;
                exit(-1);
            }
            created_thread++;
        }
        pthread_mutex_unlock(&mutex);
    }
}

pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
pthread_testcancel();
}
}

```

这一函数是会在 while 循环中反复执行的，直到排序完毕主进程取消为止。

循环开始会先检查是否已经完成了全部数据的排序。如果是则提升 finished 信号量通知主进程。否则就等待信号量 jobs，即等待任务队列里的任务，如果有则去取任务队列中的第一个任务。

再取到任务之后，如果其长度小于我们设置的下阈值，则直接调用 `algorithm` 库中的 `qsort` 函数进行排序，并且在加锁的情况下更新已经排序过的数量。此时该循环执行完毕，会执行下一个循环。

如果长度大于我们设置的下阈值，则使用 `partition` 函数进行划分，划分完后更新已排序过的总数量，将划分出来的左半部分作为左作业（`job-left`）加入到任务队列中并提升信号量 `jobs`，然后看此时已经被创建的线程数量，如果小于我们设置的线程数量限制，则创建一个新工作线程；对划分出来的右半部分作为右作业（`job-right`）同样操作。

如上，则会使得整块需要排序的数据被不停地分划为一个个小块的任务，交给各个工作线程来处理，并且会直接排序较小的块，从而使得所有数据最终完成排序。

四. 运行结果

使用 `make` 或者 `make build` 来构建快速排序程序的主体，显示如下：

```
> make
g++ quicksort.cpp -o quicksort.exe -std=c++11 -lpthread
```

编写 `generate.cpp` 程序，用于生成随机乱序数到输入文件中，其中采用了梅森旋转的方法生成了随机数，并且以二进制的格式输出到输入文件 `input.dat` 中供快速排序程序进行使用并排序，同样的内容输出到 `input.txt` 以人类可读的形式供查阅比较。具体主要代码如下：

```
int main()
{
    int d;
    ofstream output("input.txt", ios::out);
    FILE *fp = fopen("input.dat", "wb");
    random_device rd;
    mt19937_64 eng(rd());
    uniform_int_distribution<int> distr(0, LARGE);
    for (int n = 0; n < MAX_NUM; n++)
    {
        d = distr(eng);
        // printf("%d ", d);
        fwrite(&d, sizeof(int), 1, fp);
        data[n] = d;
        output << d << " ";
    }
    fclose(fp);
    return 0;
}
```

使用 `make generate` 来构建 `generate.exe` 并运行它，显示如下：

```
> make generate
g++ generate.cpp -o generate.exe -std=c++11; ./generate.exe
```

运行 `quicksort.exe`，由于输出内容非常多，所以仅展示开始和结尾的部分：

```

> ./quicksort.exe
[MAIN JOB] 0 999999
[FETCH JOB] left:0 right:999999
[PUSH JOB-LEFT] left:0 rightL735292 sorted:1
[PUSH JOB-RIGHT] left:735294 right:999999 sorted:1
[FETCH JOB] left:0 right:735292
[FETCH JOB] left:735294 right:999999
[PUSH JOB-LEFT] left:735294 rightL893753 sorted:2
[PUSH JOB-RIGHT] left:893755 right:999999 sorted:2
[FETCH JOB] left:735294 right:893753
[FETCH JOB] left:893755 right:999999
[PUSH JOB-LEFT] left:893755 rightL950071 sorted:3
[PUSH JOB-RIGHT] left:950073 right:999999 sorted:3
[FETCH JOB] left:893755 right:950071
[PUSH JOB-LEFT] left:735294 rightL854602 sorted:4
[PUSH JOB-RIGHT] left:854604 right:893753 sorted:4
[FETCH JOB] left:950073 right:999999
[FETCH JOB] left:735294 right:854602
[FETCH JOB] left:854604 right:893753
[PUSH JOB-LEFT] left:0 rightL338111 sorted:5
[PUSH JOB-RIGHT] left:338113 right:735292 sorted:5
[FETCH JOB] left:0 right:338111
[FETCH JOB] left:338113 right:735292
[PUSH JOB-LEFT] left:893755 rightL945507 sorted:6
[PUSH JOB-RIGHT] left:945509 right:950071 sorted:6
[FETCH JOB] left:893755 right:945507
[FETCH JOB] left:945509 right:950071
[PUSH JOB-LEFT] left:950073 rightL988926 sorted:7
[PUSH JOB-RIGHT] left:988928 right:999999 sorted:7
[PUSH JOB-LEFT] left:854604 rightL877617 sorted:8
[PUSH JOB-RIGHT] left:877619 right:893753 sorted:8
[PUSH JOB-LEFT] left:945509 rightL949469 sorted:9
[PUSH JOB-RIGHT] left:949471 right:950071 sorted:9
[FETCH JOB] left:950073 right:988926
[FETCH JOB] left:988928 right:999999
[FETCH JOB] left:854604 right:877617
[FETCH JOB] left:877619 right:893753
[FETCH JOB] left:945509 right:949469

```

```

[DIRECT SORT] left:392212 right:393076 sorted:992825
[DIRECT SORT] left:331963 right:332708 sorted:993571
[DIRECT SORT] left:299279 right:300218 sorted:994511
[DIRECT SORT] left:321637 right:321905 sorted:994780
[FETCH JOB] left:321907 right:323024
[PUSH JOB-LEFT] left:323039 rightL323241 sorted:994781
[PUSH JOB-RIGHT] left:323243 right:324076 sorted:994781
[PUSH JOB-LEFT] left:359963 rightL360763 sorted:994782
[PUSH JOB-RIGHT] left:360765 right:361093 sorted:994782
[FETCH JOB] left:377511 right:377647
[DIRECT SORT] left:361095 right:361386 sorted:995074
[FETCH JOB] left:377649 right:379152
[FETCH JOB] left:323039 right:323241
[PUSH JOB-LEFT] left:321907 rightL322721 sorted:995075
[PUSH JOB-RIGHT] left:322723 right:323024 sorted:995075
[FETCH JOB] left:323243 right:324076
[FETCH JOB] left:359963 right:360763
[FETCH JOB] left:360765 right:361093
[DIRECT SORT] left:377511 right:377647 sorted:995212
[FETCH JOB] left:321907 right:322721
[PUSH JOB-LEFT] left:377649 rightL378463 sorted:995213
[PUSH JOB-RIGHT] left:378465 right:379152 sorted:995213
[FETCH JOB] left:322723 right:323024
[DIRECT SORT] left:323039 right:323241 sorted:995416
[FETCH JOB] left:377649 right:378463
[DIRECT SORT] left:360765 right:361093 sorted:995745
[FETCH JOB] left:378465 right:379152
[DIRECT SORT] left:322723 right:323024 sorted:996047
[DIRECT SORT] left:359963 right:360763 sorted:996848
[DIRECT SORT] left:323243 right:324076 sorted:997682
[DIRECT SORT] left:321907 right:322721 sorted:998497
[DIRECT SORT] left:377649 right:378463 sorted:999312
[DIRECT SORT] left:378465 right:379152 sorted:1000000
All thread finish!
Write sorted data into output.txt.
Program finished!
You can run `make check` to see if the sort is correct.

```

可以看到其较好的完成了排序任务。

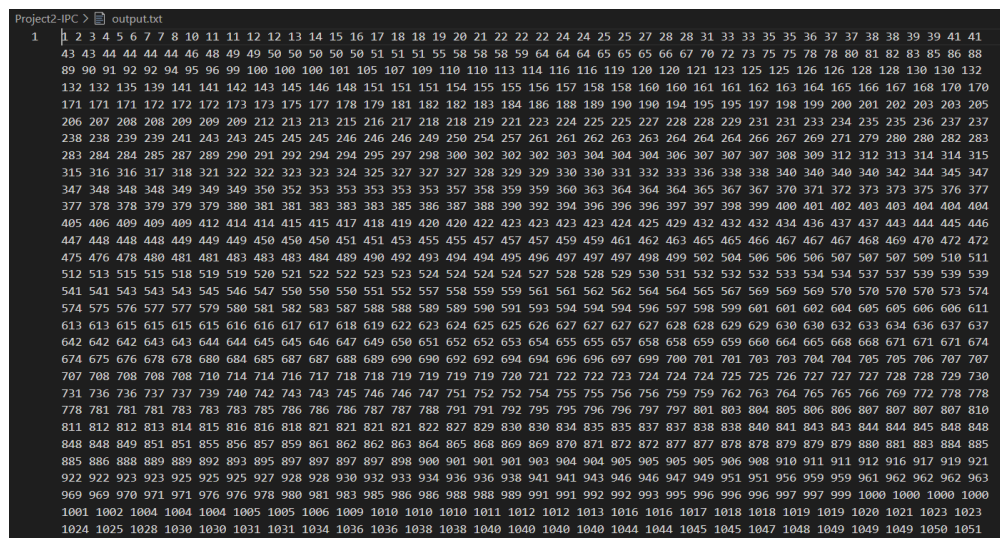
由于数据量比较大，所以我写了一个程序 `check.cpp` 用于检查快速排序程序的输出结果是否正确，具体主要部分的代码如下：

```
int main()
{
    read_input("input.txt", ground_truth);
    read_input("output.txt", result);
    sort(ground_truth.begin(), ground_truth.end());
    if (ground_truth.size() != result.size())
    {
        cout << "[ERROR]: sorted data size(" << result.size() << ") doesn't  
match ground truth data size(" << ground_truth.size() << ")!\n";
        return 1;
    }
    if (ground_truth != result)
    {
        cout << "[ERROR]: sorted data is different from ground truth data!\n";
        ;
        return 2;
    }
    cout << "Sorted data(size: " << result.size() << ") is the same as ground  
truth data(size: " << ground_truth.size() << ")! Check passed!\n";
    return 0;
}
```

使用 `make check` 来构建 `check.exe` 并运行它，显示如下：

```
> make check
g++ check.cpp -o check.exe -std=c++11; ./check.exe
Sorted data(size: 1000000) is the same as ground truth data(size: 1000000)! Check passed!
```

可以看到其结果都是正确的。我们也可以打开 `output.txt`，其显示如下：



```
Project2-IPC > output.txt
1 1 2 3 4 5 6 7 8 10 11 11 12 12 13 14 15 16 17 18 18 19 20 21 22 22 22 24 24 25 25 27 28 28 31 33 33 35 35 36 37 37 38 38 39 39 41 41
43 43 44 44 44 44 46 48 49 49 50 50 50 50 51 51 51 55 58 58 58 59 64 64 65 65 65 66 67 70 72 73 75 75 78 78 80 81 82 83 85 86 88
89 90 91 92 92 94 95 96 99 100 100 100 101 105 107 109 110 110 113 114 116 116 119 120 120 121 123 125 125 126 126 128 128 130 130 132
132 132 135 139 141 141 142 143 145 146 148 151 151 151 154 155 155 156 157 158 158 160 160 161 161 162 163 164 165 166 167 168 170 170
171 171 171 172 172 172 173 173 175 177 178 179 181 182 182 183 184 186 188 189 190 190 194 195 195 197 198 199 200 201 202 203 205
206 207 208 208 209 209 209 212 213 213 215 216 217 218 218 219 221 223 224 225 225 227 228 228 229 231 231 233 234 235 235 236 237 237
238 238 239 239 241 243 243 245 245 245 246 246 246 249 250 254 257 261 261 262 263 263 264 264 264 266 267 269 271 279 280 280 282 283
283 284 284 285 287 289 290 291 292 294 294 295 297 298 300 302 302 302 303 304 304 304 306 307 307 307 308 309 312 312 313 314 314 315
315 316 316 317 318 321 322 322 323 323 324 325 327 327 327 328 329 329 330 330 331 332 333 336 338 338 340 340 340 340 342 344 345 347
347 348 348 348 349 349 349 350 352 353 353 353 353 353 357 358 359 359 360 363 364 364 364 365 367 367 370 371 372 373 373 375 376 377
377 378 378 379 379 379 380 381 381 383 383 383 385 386 387 388 390 392 394 396 396 396 397 397 398 399 400 401 402 403 403 404 404 404
405 406 409 409 409 412 414 414 415 415 417 418 419 420 420 422 423 423 423 423 424 425 429 432 432 432 434 436 437 437 443 444 445 446
447 448 448 448 449 449 449 450 450 450 451 451 451 455 455 455 457 457 457 459 459 461 462 463 465 465 466 467 467 467 468 469 470 472 472
475 476 478 480 481 481 483 483 483 484 489 490 492 493 494 494 495 496 497 497 497 498 499 502 504 506 506 506 507 507 507 509 510 511
512 513 515 515 518 519 519 520 521 522 522 523 523 524 524 524 524 527 528 528 529 530 531 532 532 532 533 534 534 537 537 539 539 539
541 541 543 543 543 545 546 547 550 550 550 551 552 557 558 559 559 561 561 562 562 564 564 565 567 569 569 569 570 570 570 570 573 574
574 575 576 577 577 579 580 581 582 583 587 588 588 589 589 590 591 593 594 594 594 596 597 598 599 601 601 602 604 605 605 606 606 611
613 613 615 615 615 615 616 616 617 617 618 619 622 623 624 625 625 626 627 627 627 627 628 628 629 629 630 630 632 633 634 636 637 637
642 642 642 643 643 644 644 645 645 646 647 649 650 651 652 652 653 654 655 655 657 658 658 659 659 660 664 665 668 668 671 671 674
674 675 676 678 680 684 685 687 687 688 689 690 692 692 694 694 696 696 697 699 700 701 701 703 703 704 704 705 705 706 706 707 707
707 708 708 708 708 710 714 714 716 717 718 718 719 719 719 720 721 722 722 723 724 724 724 725 725 726 727 727 727 728 728 729 730
731 736 736 737 737 739 740 742 743 743 745 746 746 747 751 752 752 754 755 755 756 756 759 759 762 763 764 765 765 766 769 772 778 778
778 781 781 781 783 783 783 785 786 786 786 787 787 788 791 791 792 795 795 796 796 797 797 801 803 804 805 806 806 807 807 807 807 810
811 812 812 813 814 815 816 816 818 821 821 821 821 822 827 829 830 830 834 835 835 837 837 838 838 840 841 843 843 844 844 845 848 848
848 848 849 851 851 855 856 857 859 861 862 862 863 864 865 868 869 870 871 872 872 877 877 878 878 879 879 879 880 881 883 884 885
885 886 888 889 889 892 893 895 897 897 897 897 898 900 901 901 901 903 904 904 905 905 905 905 906 908 910 911 911 912 916 917 919 921
922 922 923 923 925 925 925 927 928 928 930 932 933 934 936 936 938 941 941 943 946 946 947 949 951 951 956 959 959 961 962 962 962 963
969 969 970 971 971 976 976 978 980 981 983 985 986 986 988 988 989 991 991 992 992 993 995 996 996 996 997 997 999 1000 1000 1000 1000
1001 1002 1004 1004 1004 1005 1005 1006 1009 1010 1010 1010 1011 1012 1012 1013 1016 1016 1017 1018 1018 1019 1019 1020 1021 1023 1023
1024 1025 1028 1030 1030 1031 1031 1034 1036 1036 1038 1040 1040 1040 1044 1044 1045 1045 1047 1048 1049 1049 1049 1050 1051
```

可以看到其都是排序好了的结果，手动验证了排序的正确性。

五. 思考题

1. 你采用了你选择的机制而不是另外的两种机制解决该问题，请解释你做出这种选择的理由。

我采用了共享内存来进行高级进程间通信。选择这个方法的理由主要是：使用共享内存的机制时，数据并不需要被来回复制传送，而是直接在内存中进行操作，通信的速度快，效率高，而且基于本题划分式的思路，所有工作线程都是都不会使得同时处理区域发生交叠，这使得对于数据的访问不需要加锁从而很大程度上提高了效率，而且一旦一个线程对某一个区域发生改变之后，对下一个访问这个区域的线程来说，非常自然地得到的就是改变之后的结果，从而对于实时性高效而言是性能优良的。

而使用管道则需要开辟多个管道，通信复杂度较高，容易出错且效率不高；同样的，使用消息队列在大量数据通信的场景下的工作效率也不如共享内存高。

2. 你认为另外的两种机制是否同样可以解决该问题？如果可以请给出你的思路；如果不能，请解释理由。

另外两种机制都同样可以解决该问题。对于管道方法而言，可以采用一个主进程（线程），对所有子进程（线程）都开辟管道，然后通过主子进程（线程）之间的管道传送数据，从而实现任务分划与处理结果回传；对于消息队列方法而言，也是类似可以在主子进程（线程）之间将数据放入消息队列中进行传输，从而实现任务的传递和结果的回传。

六. 实验感受和总结收获

本次大作业大大增进了我对于高级进程间通信这个问题的理解和认识，对于进程间通信的各种机制也有了更加深刻清晰的认知，对课程中教授的理论知识有了更加深刻的巩固和印证，同时也对于共享内存这个机制在 Linux 下的使用和相关细节操作有了更多的掌握，更加熟悉了多线程编程的一些具体操作和细节实现，提升了这一方面的编程能力。此外也是复习了快速排序的递归实现，对其有了深入理解之后才设计实现了多线程排序。另外对于限制并发线程数这一要求，也是在考虑了多种程序设计之后选定了目前这种扩展性比较好，逻辑思路比较明晰的方式。

七. 附录

随报告附主程序 `quicksort.cpp`，本测试所用的生成随机乱序数输入文件的程序 `generate.cpp` 和检查输出是否排序正确的程序 `check.cpp`，以及可以用来进行方便编译的 `Makefile`。