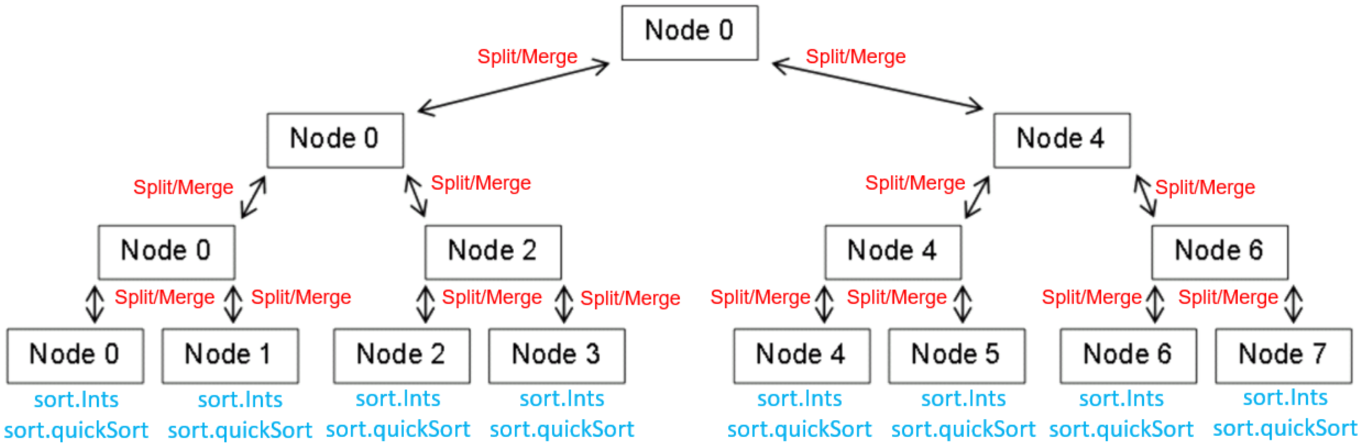
1. Idea

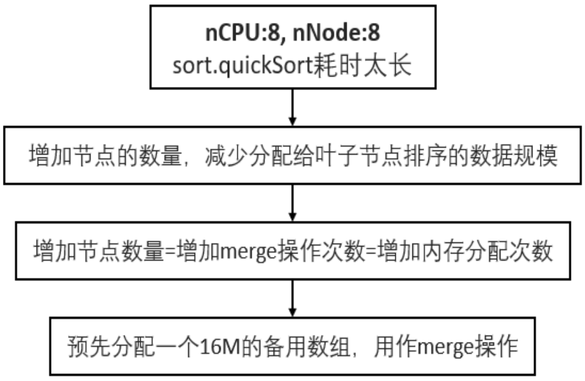
基本思路如下图所示：

1. 中间节点从父节点收到自己负责的数据块后，将数据块一分为二，自己负责左半部分数据块的排序，将右半部分分给儿子节点；
2. 如此递归下去，直到叶子节点，叶子节点采用golang sort包提供的sort函数对数据块进行排序；
3. 当儿子节点完成自己负责的数据块排序后，父节点再对左右两个数据块进行merge操作。merge行为的协同由wg.waitGroup来实现；



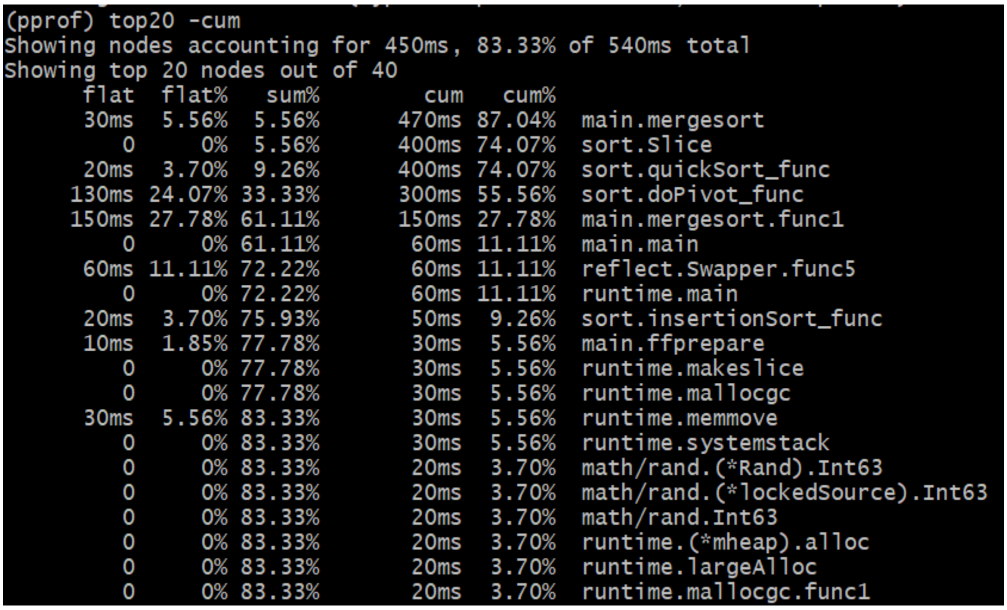
2. PProf

基本优化思路如下图所示。因内存占用情况和routine race的情况都正常，所以下面只分析了CPU Profile的结果。


nCPU:8, nNode:8
sort.quickSort耗时太长
增加节点的数量，减少分配给叶子节点排序的数据规模
增加节点数量=增加merge操作次数=增加内存分配次数
预先分配一个16M的备用数组，用作merge操作

> **Phase 1**: node的数量=CPU的数量，每次merge操作临时开辟内存

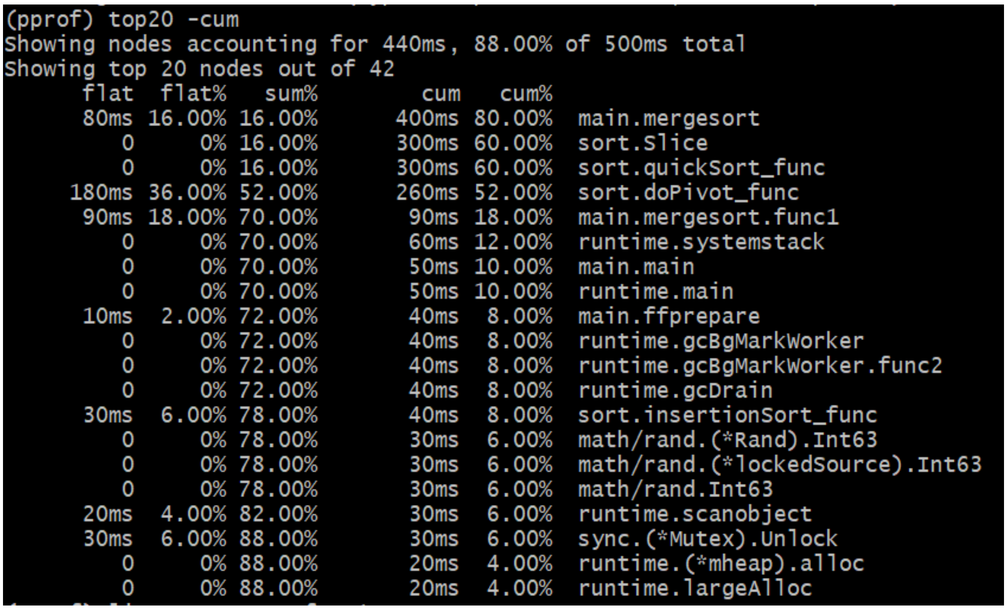
我的电脑是8线程的，有8个node参加了底层的sort.quickSort排序，相当于每个sort.quickSort的数据规模是2M， 2048个整数。CPU Profile的结果如下图所示，可以看到在quickSort的doPivot阶段花了相当一部分时间，于是想增加叶子节点，减小每个叶子节点排序的数据规模，以减少doPivot花费的时间。



```
(pprof) top20 -cum
Showing nodes accounting for 450ms, 83.33% of 540ms total
Showing top 20 nodes out of 40
      flat  flat%   sum%        cum   cum%
      30ms  5.56%  5.56%      470ms 87.04%  main.mergesort
         0     0%  5.56%      400ms 74.07%  sort.Slice
      20ms  3.70%  9.26%      400ms 74.07%  sort.quickSort_func
     130ms 24.07% 33.33%      300ms 55.56%  sort.doPivot_func
     150ms 27.78% 61.11%      150ms 27.78%  main.mergesort.func1
         0     0% 61.11%       60ms 11.11%  main.main
      60ms 11.11% 72.22%       60ms 11.11%  reflect.Swapper.func5
         0     0% 72.22%       60ms 11.11%  runtime.main
      20ms  3.70% 75.93%       50ms  9.26%  sort.insertionSort_func
      10ms  1.85% 77.78%       30ms  5.56%  main.ffprepare
         0     0% 77.78%       30ms  5.56%  runtime.makeslice
         0     0% 77.78%       30ms  5.56%  runtime.mallocgc
      30ms  5.56% 83.33%       30ms  5.56%  runtime.memmove
         0     0% 83.33%       30ms  5.56%  runtime.systemstack
         0     0% 83.33%       20ms  3.70%  math/rand.(*Rand).Int63
         0     0% 83.33%       20ms  3.70%  math/rand.(*lockedSource).Int63
         0     0% 83.33%       20ms  3.70%  math/rand.Int63
         0     0% 83.33%       20ms  3.70%  runtime.(*mheap).alloc
         0     0% 83.33%       20ms  3.70%  runtime.largeAlloc
         0     0% 83.33%       20ms  3.70%  runtime.mallocgc.func1
```

> **Phase 2**：node的数量=n*CPU的数量

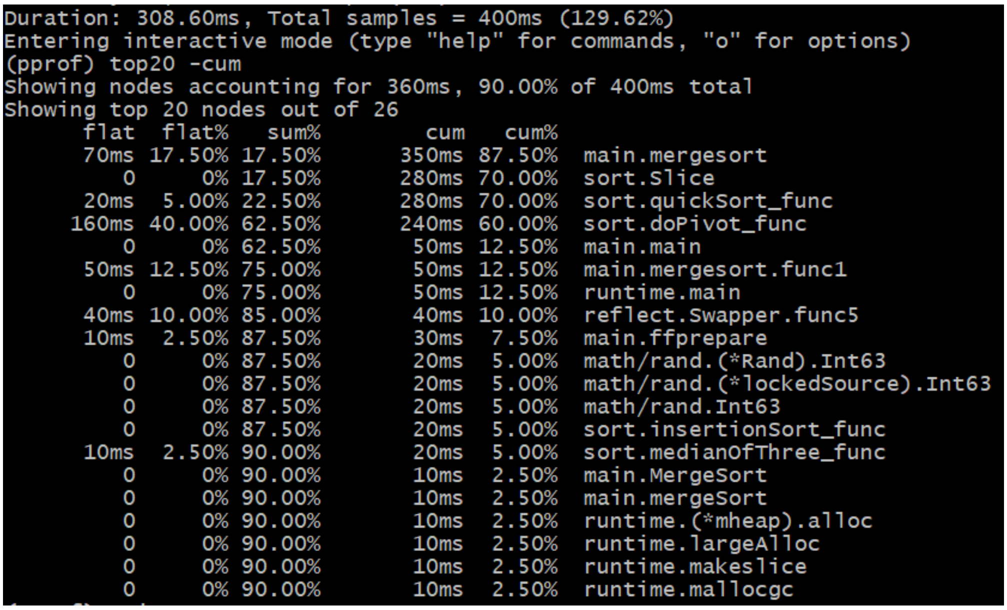
在8线程，16GB RAM的电脑上，分别尝试node数量为8, 16, 32, 64, 128的情况，发现64是一个凹点（此时每个node要排256个数据）, CPU Profile输出的结果如下图所示。可以看出quickSort的时间减少了100ms, 但在runtime.systemstack上的时间却增多了，多出来的这些时间又大多在内存分配上。故想办法减少内存的分配次数。



```
(pprof) top20 -cum
Showing nodes accounting for 440ms, 88.00% of 500ms total
Showing top 20 nodes out of 42
      flat  flat%   sum%        cum   cum%
      80ms 16.00% 16.00%      400ms 80.00%  main.mergesort
         0     0% 16.00%      300ms 60.00%  sort.Slice
         0     0% 16.00%      300ms 60.00%  sort.quickSort_func
     180ms 36.00% 52.00%      260ms 52.00%  sort.doPivot_func
      90ms 18.00% 70.00%       90ms 18.00%  main.mergesort.func1
         0     0% 70.00%       60ms 12.00%  runtime.systemstack
         0     0% 70.00%       50ms 10.00%  main.main
         0     0% 70.00%       50ms 10.00%  runtime.main
      10ms  2.00% 72.00%       40ms  8.00%  main.ffprepare
         0     0% 72.00%       40ms  8.00%  runtime.gcBgMarkWorker
         0     0% 72.00%       40ms  8.00%  runtime.gcBgMarkWorker.func2
         0     0% 72.00%       40ms  8.00%  runtime.gcDrain
      30ms  6.00% 78.00%       40ms  8.00%  sort.insertionSort_func
         0     0% 78.00%       30ms  6.00%  math/rand.(*Rand).Int63
         0     0% 78.00%       30ms  6.00%  math/rand.(*lockedSource).Int63
         0     0% 78.00%       30ms  6.00%  math/rand.Int63
      20ms  4.00% 82.00%       30ms  6.00%  runtime.scanobject
      30ms  6.00% 88.00%       30ms  6.00%  sync.(*Mutex).Unlock
         0     0% 88.00%       20ms  4.00%  runtime.(*mheap).alloc
         0     0% 88.00%       20ms  4.00%  runtime.largeAlloc
```

> **Phase 3**: node的数量=n*CPU的数量，预分配用于merge的备用数组

为了避免merge结果的拷贝，将merge数组和存放源数据的数组src交替使用，最终得到的CPU Profile的输出结果如下，可以看到runtime.systemstack的开销已大大降低。



```
Duration: 308.60ms, Total samples = 400ms (129.62%)
Entering interactive mode (type "help" for commands, "o" for options)
(pprof) top20 -cum
Showing nodes accounting for 360ms, 90.00% of 400ms total
Showing top 20 nodes out of 26
      flat  flat%   sum%        cum   cum%
      70ms 17.50% 17.50%      350ms 87.50%  main.mergesort
         0     0% 17.50%      280ms 70.00%  sort.Slice
      20ms  5.00% 22.50%      280ms 70.00%  sort.quickSort_func
     160ms 40.00% 62.50%      240ms 60.00%  sort.doPivot_func
         0     0% 62.50%       50ms 12.50%  main.main
      50ms 12.50% 75.00%       50ms 12.50%  main.mergesort.func1
         0     0% 75.00%       50ms 12.50%  runtime.main
      40ms 10.00% 85.00%       40ms 10.00%  reflect.Swapper.func5
      10ms  2.50% 87.50%       30ms  7.50%  main.ffprepare
         0     0% 87.50%       20ms  5.00%  math/rand.(*Rand).Int63
         0     0% 87.50%       20ms  5.00%  math/rand.(*lockedSource).Int63
         0     0% 87.50%       20ms  5.00%  math/rand.Int63
         0     0% 87.50%       20ms  5.00%  sort.insertionSort_func
      10ms  2.50% 90.00%       20ms  5.00%  sort.medianOfThree_func
         0     0% 90.00%       10ms  2.50%  main.MergeSort
         0     0% 90.00%       10ms  2.50%  main.mergeSort
         0     0% 90.00%       10ms  2.50%  runtime.(*mheap).alloc
         0     0% 90.00%       10ms  2.50%  runtime.largeAlloc
         0     0% 90.00%       10ms  2.50%  runtime.makeslice
         0     0% 90.00%       10ms  2.50%  runtime.mallocgc
```