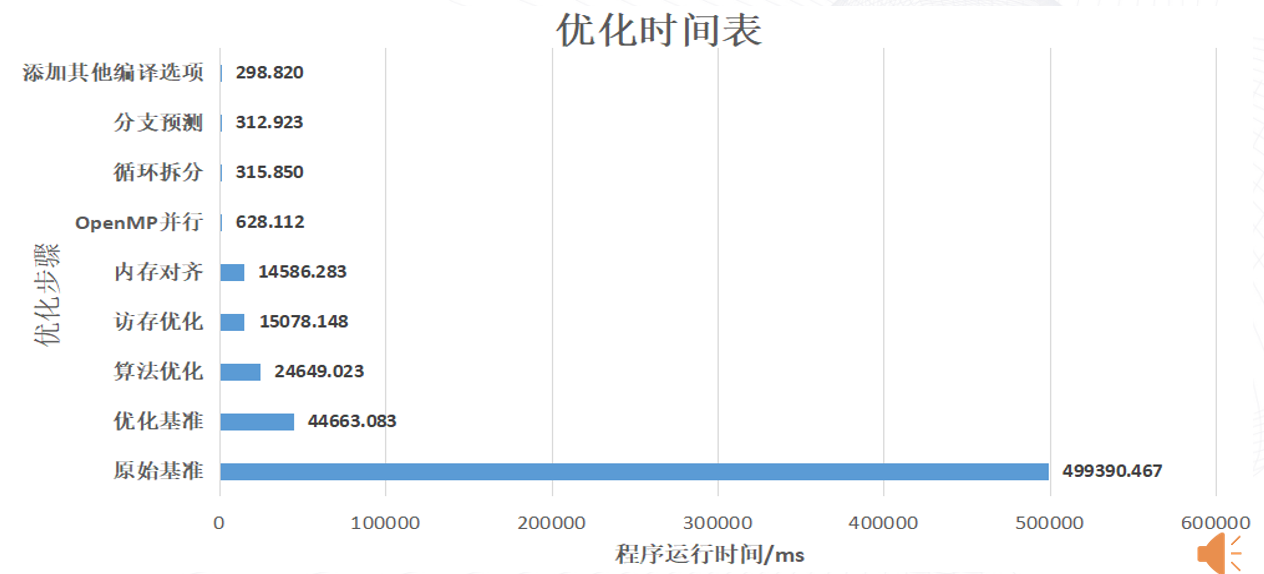
**技术报告**

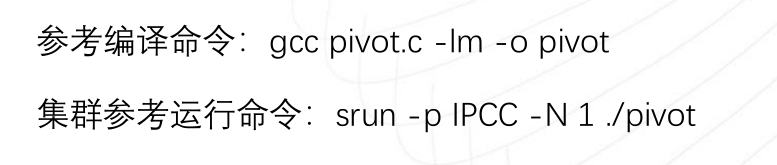
1. **优化时间表**



1. **基准的建立**

我们优化过程主要以初赛赛题所给的第一个算例为优化对象，以下所提及的优化时间均是在所给的第一个算例优化得到的。

我们以平台默认方式为准，编译及运行方法如赛题提供的所示，其中版本为 g++==4.8.5



我们通过多次测试，可以得到在默认编译运行的情况下，源程序的平均耗时为 503600 ms

 （基准1）

之后我们通过对于原代码开启O3编译优化，得到O3优化后程序的平均耗时为 45315 ms

gcc pivot.c -O3 -lm -o pivot

 （基准2）

我们又通过比较gcc和g++编译器对于代码的编译优化程度，以及对不同版本的编译器进行选择测试，最终选择如下编译方式：

g++ pivot.c -O3 -lm -o pivot ( g++==4.8.5 )

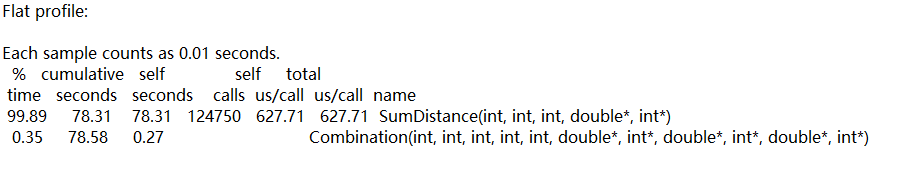
运行程序的平均耗时为：44663 ms

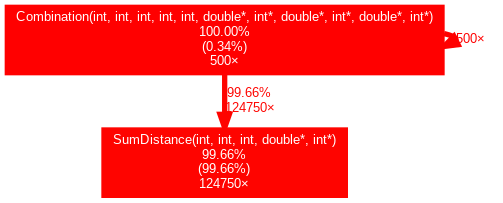
 （基准3）

显然由于源程序性能较差，不利于优化指标参考，所以在优化方法章节，以上述基准3为优化基准作为对比。之后，将以基准1为原始基准，基准3为优化基准。

1. **热点分析**

首先，我们利用vtune和pgrof软件将选定的优化基准代码进行热点分析，来理清代码结构以及明确主要的优化目标。



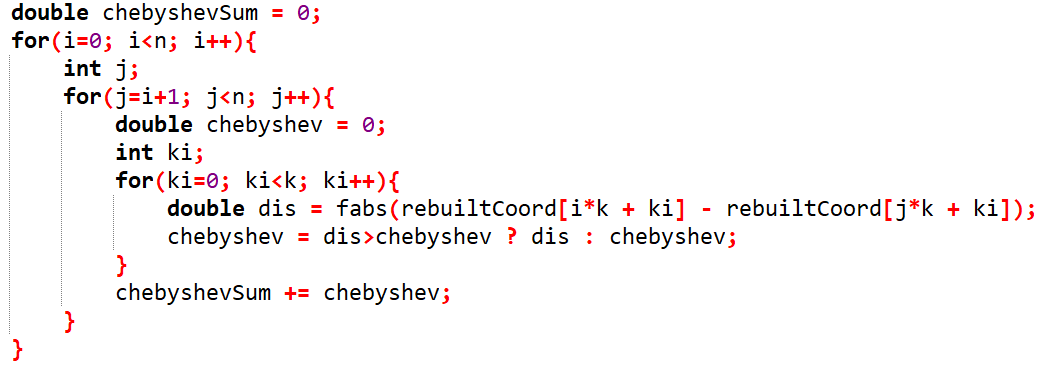


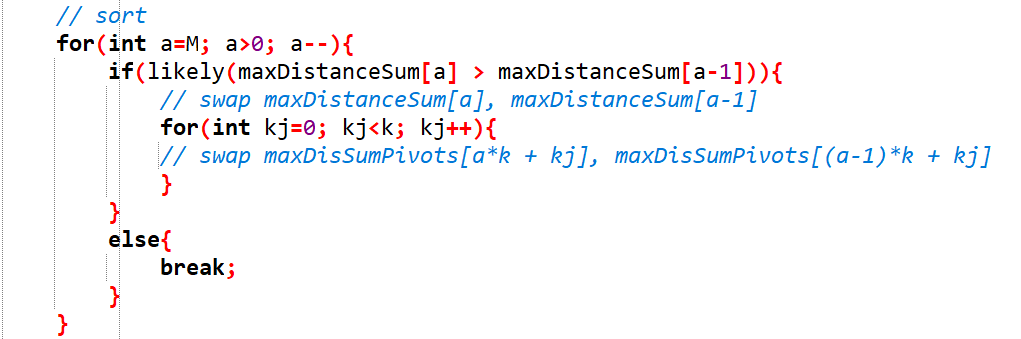
可以发现SumDistance函数耗时占比大，经过分析代码，我们发现SumDistance函数耗时占比大的主要原因在于该函数调用次数多且属于计算、访存密集型程序，为之后的优化提供方向。

1. **具体优化：**

（1）算法优化

我们将赛题代码分析理解之后，发现在原代码计算重建两坐标间切比雪夫距离和以及sort排序的过程中存在算法冗余计算的问题。首先对于计算两坐标间切比雪夫距离的和，只需满足所选的第二个坐标点序号大于第一个坐标点即可。其次，对于sort排序，由于得到的排序数组始终维持着有序的状态，所以在类冒泡排序的过程中，只要一发现不满足冒泡交换条件，说明数组已经有序，整个循环就可以退出。





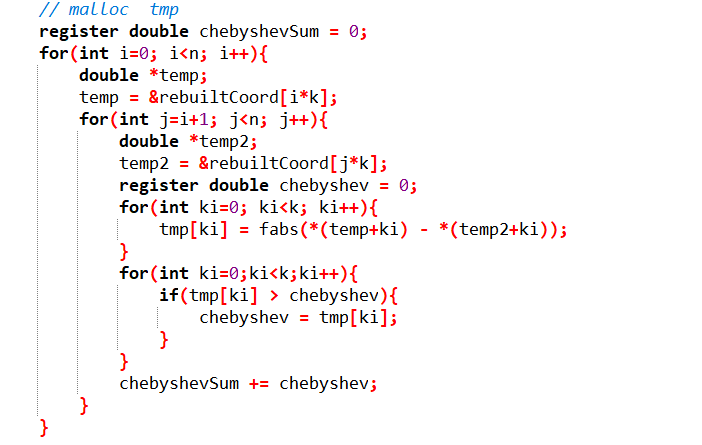
（2）初始化 memset

我们发现在代码的运行过程中，由于后续计算的覆盖，数组的初始化是无用的，所以我们将这部分的初始化过程删去。对于需要赋初值的初始化操作，我们运用memset进行初始化。

1. 访存优化

通过上述的热点分析我们知道SumDistance函数为主要热点函数，且利用计时函数累加计算耗时我们发现运行计算坐标间切比雪夫距离和的过程耗时严重。于是对于这一段访存密集型的代码，我们先进行了访存优化。

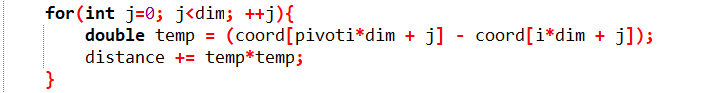
首先，可以发现计算过程中要多次访问rebuiltCoord数组，且访问的过程仅存在部分的联系性，所以我们在循环过程中增加中间变量和索引计算，利用指针指向每次循环开始时rebuiltCoord中的位置，这样可以减少了在每次循环中需要的访存次数，大大提高CPU cache的命中率。其次，我们将三目运算符改成简单条件判断，并将循环拆分，将循环分成可矢量和不可矢量两部分，利于编译器优化，且增加了循环的粒子性，利于之后的并行优化。



不可矢量化

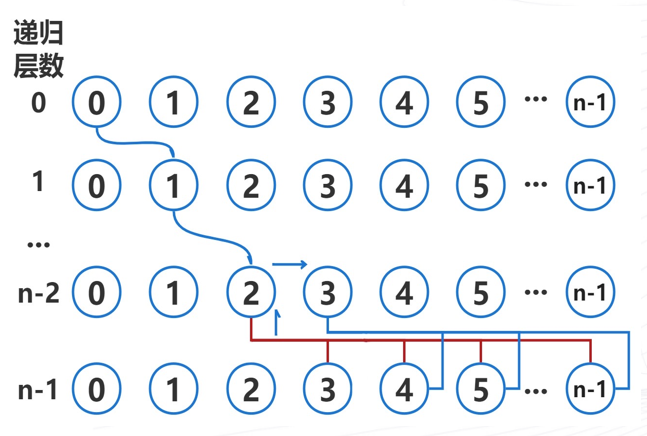
可矢量化

之后，我们对于重建坐标代码部分也进行了增加中间变量等一系列访存优化处理。



1. openmp并行优化

对于Combination函数的递归调用过程，我们发现该算法类似于DFS实现组合数，但在当递归调用到最后一层，该算法采用的是遍历方式实现，过程如图所示。



可以发现最后一层的遍历迭代为我们的并行化提供了基础。

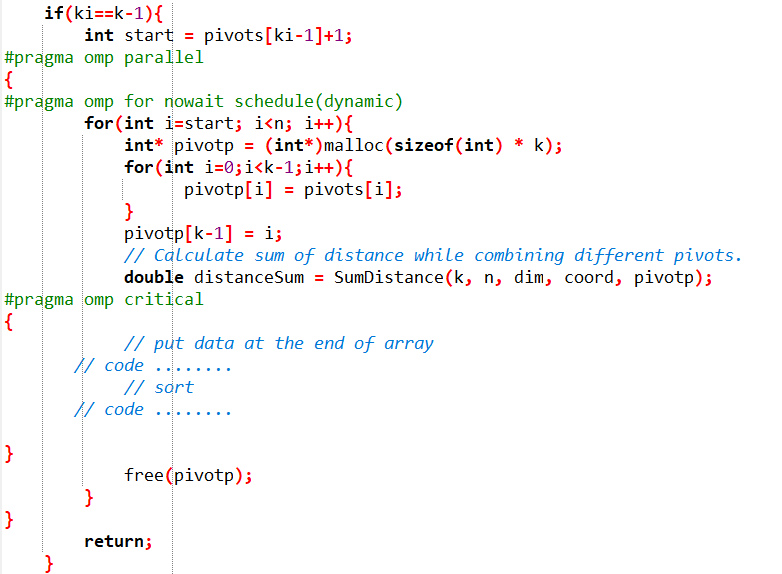
通过阅读相关论文，我们了解到在大数据泛构支撑点穷举选择算法中，支撑点数量k的取值往往很小，所以整个算法递归的层数较少，所以我们将并行的重点放在递归最后一层的遍历环节。

递归最后一层的是利用遍历迭代确定第k个支撑点，并将所选择的支撑点进行求解距离向量，计算切比雪夫距离的和以及sort排序等部分。这些计算的过程均为程序的热点。

我们利用openmp，采用parallel 和omp for指导语句将最后一层的for循环并行化，即将循环中的迭代部分分配给线程组中的个线程运行。对于循环内存在依赖关系只能串行执行的代码，如sort代码等，我们通过构造critical声明临界区，只允许一个线程执行，来满足结果的正确性。

由于pivots数组在每次循环调用过程中可能会被修改，存在依赖关系，于是我们在每次循环的开始部分开配一个新的数组pivotp，用于存储当前pivots的值，并利用新数组代替原数组参与计算，有效解决了上述问题。

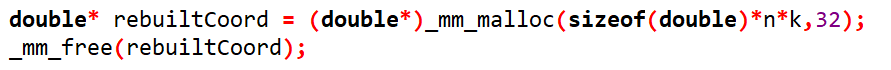
为了提高并行效率，加速并行，我们使用nowait取消for伪指令块结束的隐式同步，利用schedule子句选择dynamic来实现动态负载均衡。



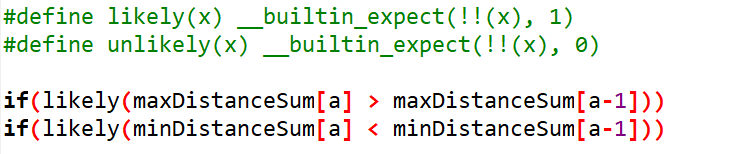
1. 内存对齐 & 分支预测

为了提高运算效率，加快程序的运行。

我们采用\_mm\_malloc 和\_mm\_free替代malloc和free，在分配内存时按照字节对齐。利于数据的读写。



我们采用分支预测的操作，在预判范围内减少判断指令的操作。使用分支预测，显式告知编译器某一分支的概率，避免流水线停顿造成的时间浪费。



1. 编译选项优化

我们整合了以上所述的优化方法，并且优化修改了一些细节。最后将编译选项进行再优化，在编译选项中增加循环展开unroll-all-loops等一系列操作，并且增设OpenMP环境变量OMP\_PROC\_BIND=TRUE, 取消处理器间的迁移线程，优化系统资源的使用。

