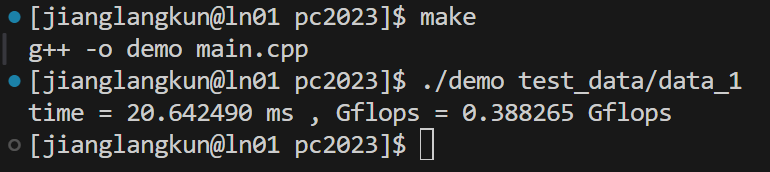
1. **实验准备**

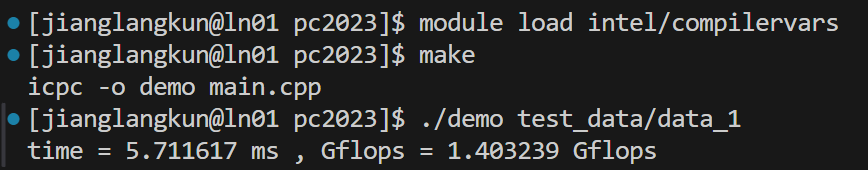
在开始实验之前，需要先学习OpenMP的基本概念，包括指令和指令集，以及如何在代码中使用OpenMP来并行化任务，如：**并行区域、共享数据**、**并行循环和同步等基本概念**。可能的话，尝试不同的编译器选项和OpenMP指令集，以优化程序的性能，GCC编译器支持不同的编译器选项来对程序进行自动地优化。利用**ICC**编译器对于英特尔处理器，可以通过较长的编译时间，换取更快的处理时间，利用**icpc**编译指令以及**-qopenmp**可以达到一个较好的加速比。除此之外，还需要对题目所要求实现的功能和数据存取有深入的了解，了解程序运行的原理，了解在**稀疏矩阵向量乘法**中常见的**CSR**存储格式存在的缺陷，通过**负载均衡**平衡各个CPU线程之间数据的计算量。让每个线程都能够得到平均的数据计算量，减少闲置的的线程，能够最大化利用所有的CPU计算资源，这是最为关键的一步。

1. **实验步骤**

(1). 串行源程序执行时间为：20.642490 ms



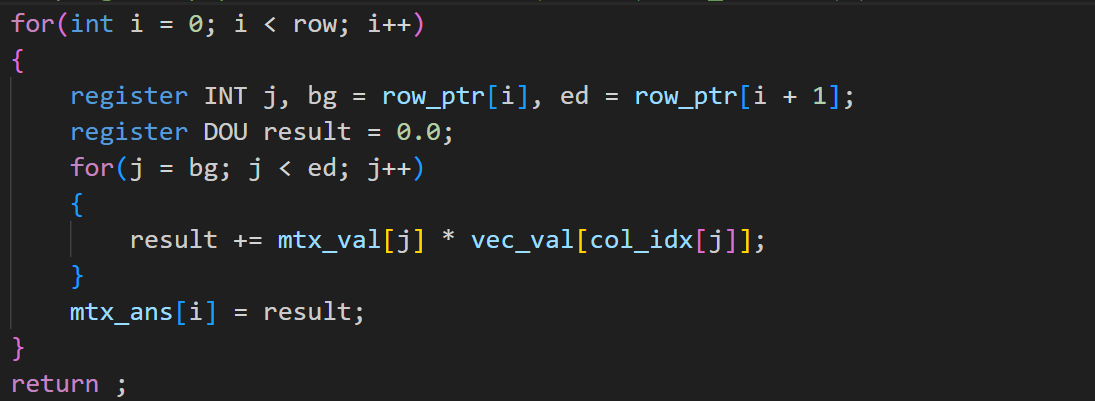
(2). 使用icpc编译器，其充分利用了 Intel 处理器的特性，提供了针对 Intel 架构的高度优化，能够生成性能更好的代码，充分发挥 Intel 处理器的性能潜力。此外，icpc 还支持并行化和向量化优化，能够更好地利用多核和 SIMD (Single Instruction, Multiple Data) 技术，提高并行应用程序的性能.



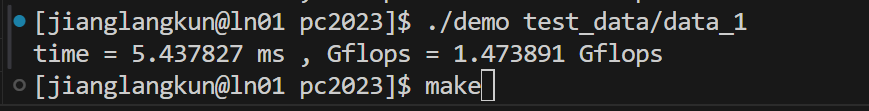
可以得出，加速比为20.642490 / 5.711617 = **3.614124**

由此得出，icc对于英特尔处理的优化程度达到了很好的加速比，非常可观

(3). 变量缓存

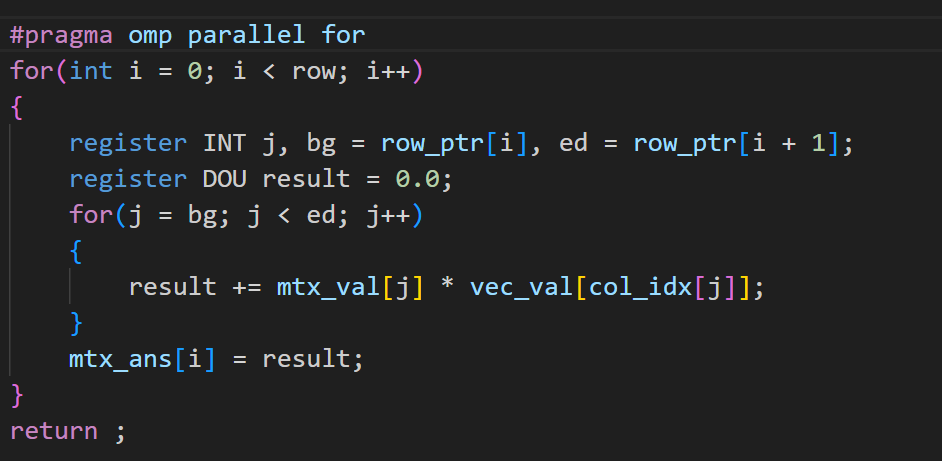


在编写代码时，可以利用bg和ed变量将row\_ptr的值存放下来，并在第二层循环前定义一个j作为循环变量；同时可以利用result作为矩阵向量乘的部分结果，最后再赋值给对应的结果数组，在此过程中可以减少一部分的数据在内存中的存取时间。（register 标识符可以让编译器将变量放在寄存器中，降低访存的时间）



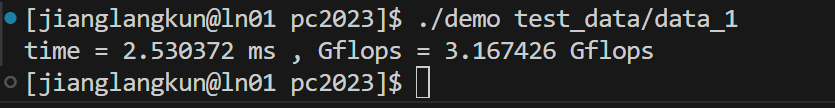
(4). OPENMP优化

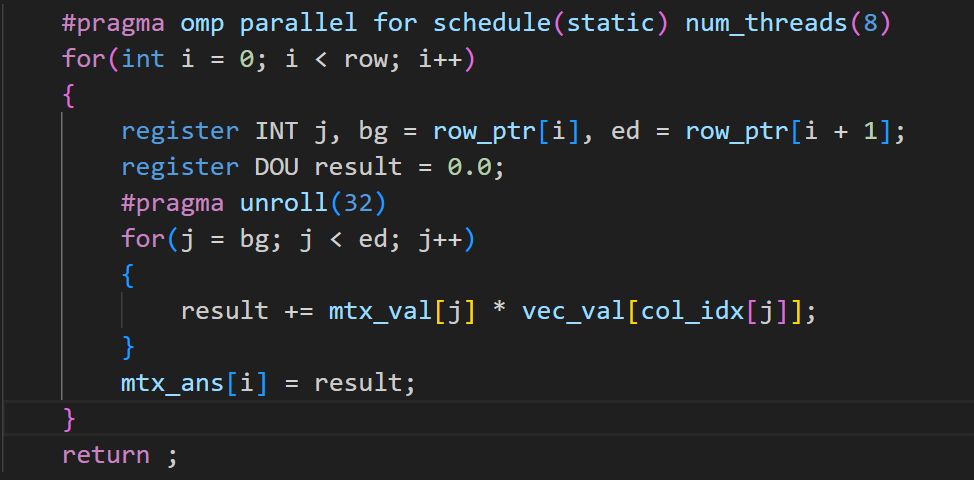
可以利用openmp的并行库的精简操作多核处理器的优势，来将任务以某种方式分配给多个线程，并由多个线程并行执行，来使程序的执行时间极大缩短。



利用#pragma omp parallel for创建并行for循环，openmp将自动创建、执行并销毁线程，并且按照默认规则将任务( 0 – row-1 )分配给不同的线程，并进行并行执行。

运行时间为：

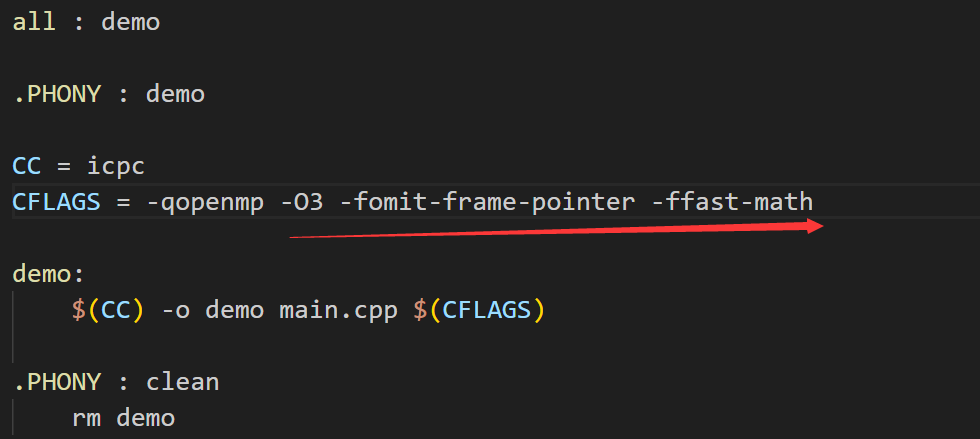




可以通过语句来指定openmp的调度策略，这里指定为static静态调度，然后设置线程数为8，经过反复的尝试，发现这种方式带来的速度提升最为可观。

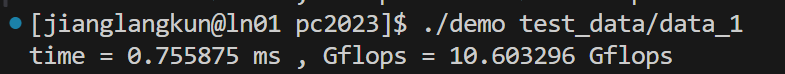
同时还可以在第二层循环使用unroll(32)语句，让编译器自动为我们进行循环展开，减少for循环的条件判断，加快执行速度。

(5). 编译指令优化



使用O3可以尽可能地对代码进行各种优化，以提高程序的执行速度。使用 -fomit-frame-pointer 选项可以告诉编译器在生成机器代码时省略帧指针，从而减小生成的代码的大小。而ffast-math则用于启用一组更激进的数学优化，旨在提高数学表达式的执行速度。

使用这些编译指令可以让程序的执行时间有微弱的提升。



可以得出，加速比为5.711617 / 0.755875 = **7.556298**这里的加速比没有到达预想中的8.0 ①是由于多线程并行本身需要创建线程、分配任务等的开销，会带来额外的时间; ②是由于并没有做到真正的负载均衡，每个线程的任务还有差距。 因此无法达到理想的加速比。

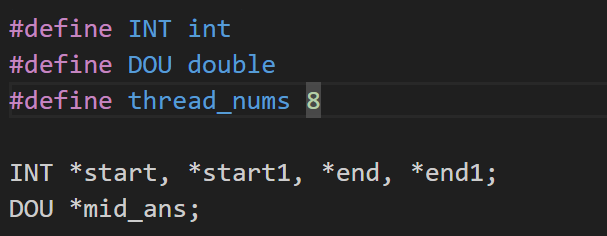
**实验进行时优化如上，以下为借鉴Github上源码的分析和改进**

1. **分析和改进**

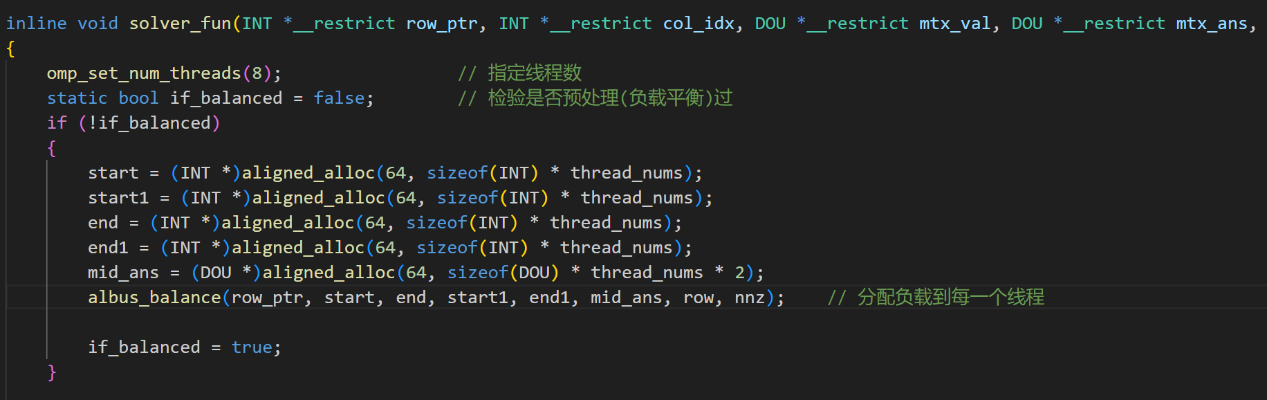
(6). 负载均衡

由于稀疏矩阵每一行可能只有很少的非零元素，也可能每一列都是非零元素，这样openmp在分配时，每个线程的负载差距就会很大，最后会导致线程计算资源的极大浪费，所以需要对线程的负载进行算法的平衡，让每个线程的计算量都不会相差太大。

1. 负载均匀分配

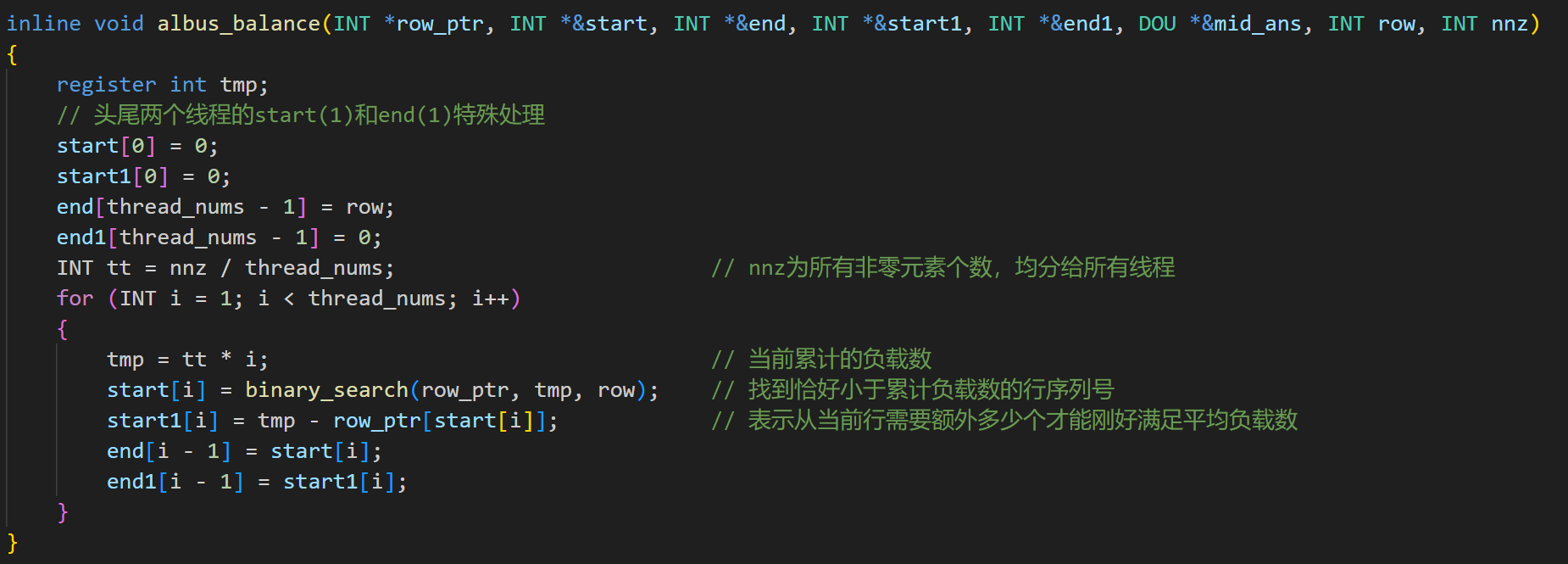


全局数组的申请和宏定义

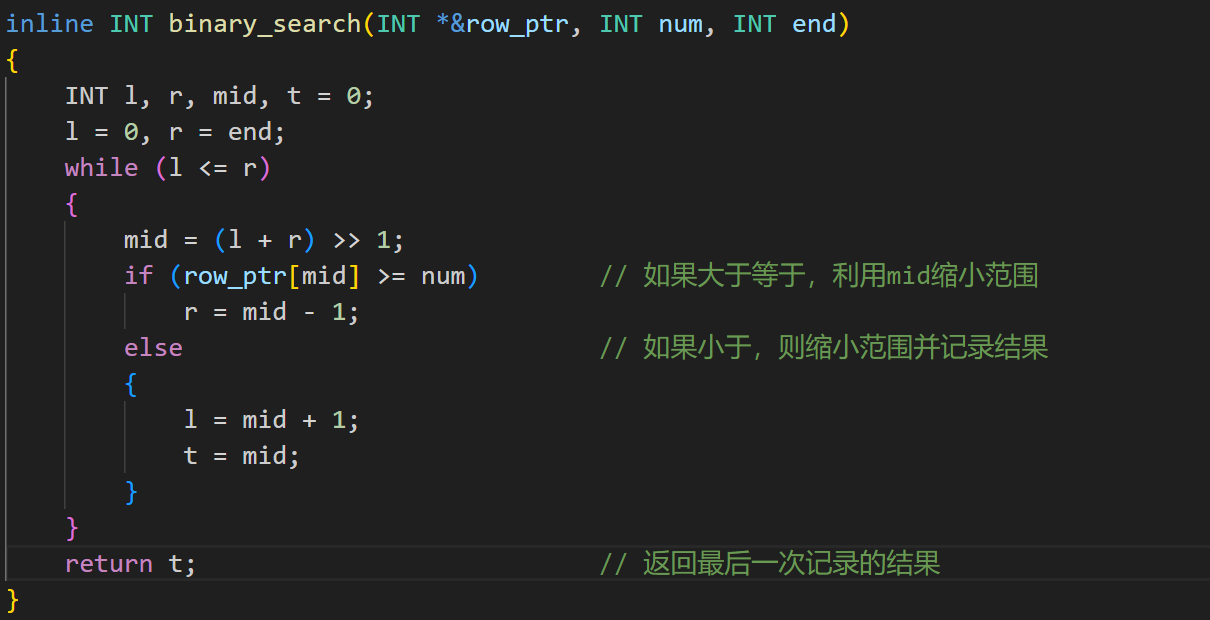


在使用的服务器上，表现最好的线程数为8，因此直接指定线程数。

由于main.cpp函数无法修改，将会在循环中多次调用solver.h函数，因此设置一个static的bool值变量，用来检测是否为数组申请空间以及为所有的稀疏矩阵乘做好了负载的平衡（所有的这些操作只需要进行一次，即预处理）。



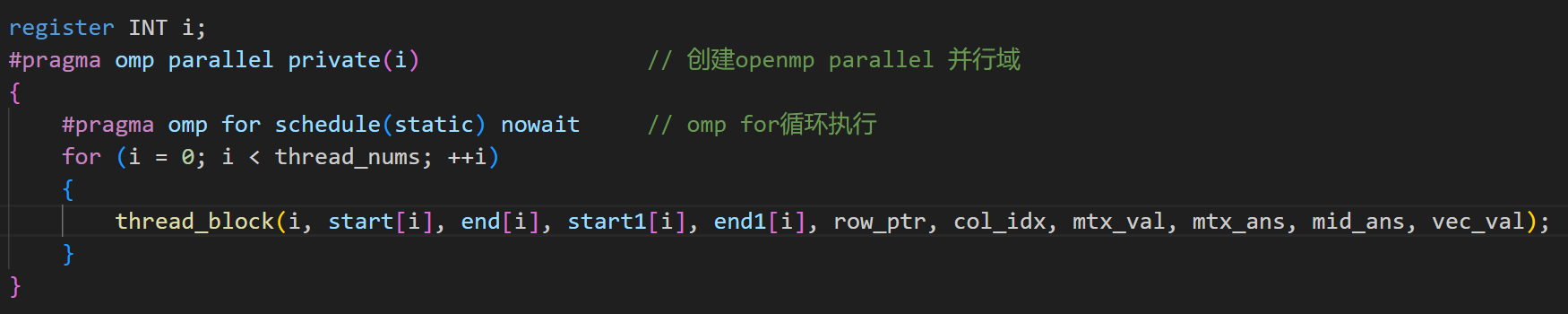
在albus\_balance函数中，start和end存放的是该线程处理的稀疏矩阵乘的开始及结束的行序号，而start1和end1存放的是该线程处理的第一个数据和最后一个数据，在对应行所有非零元素中的顺序。



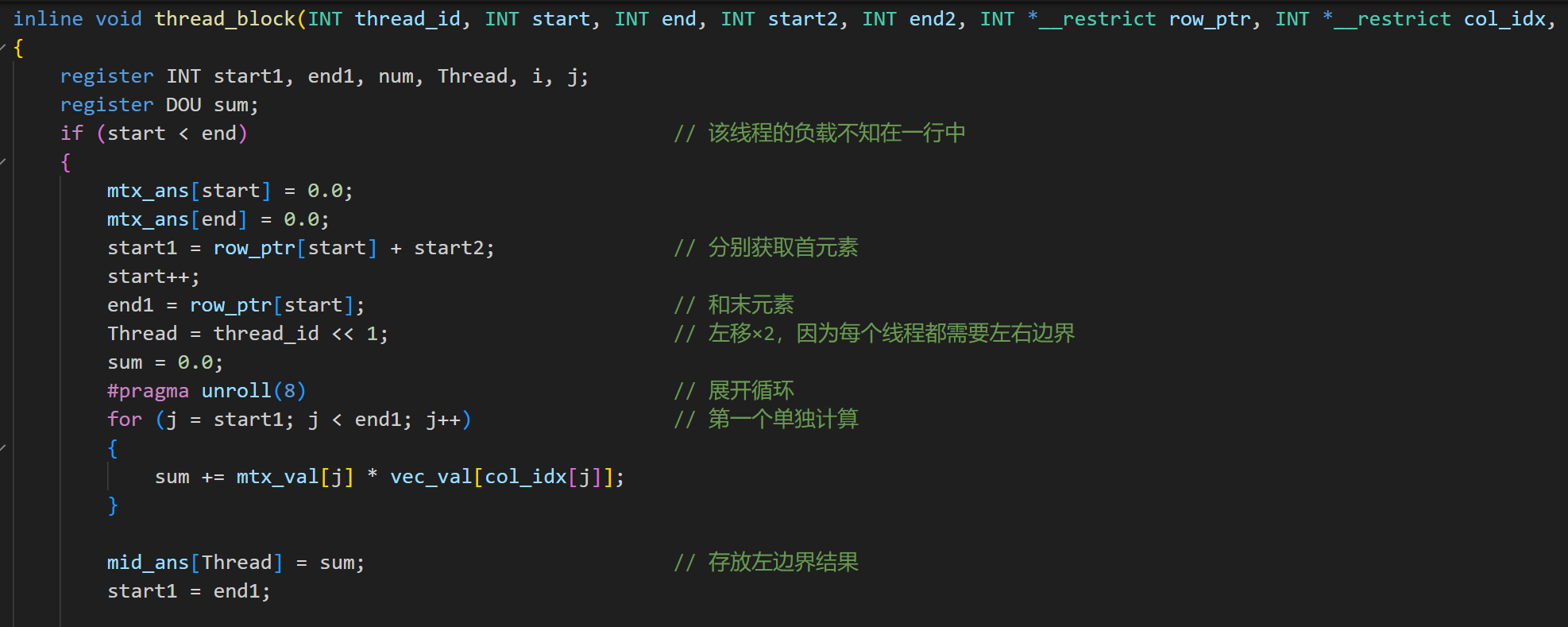
其中，start的实现通过binary\_search函数实现，该函数通过二分查找，找到所有行中，累计之前非零元素个数刚好小于所要求的负载（由所有线程均分）的行序列号。

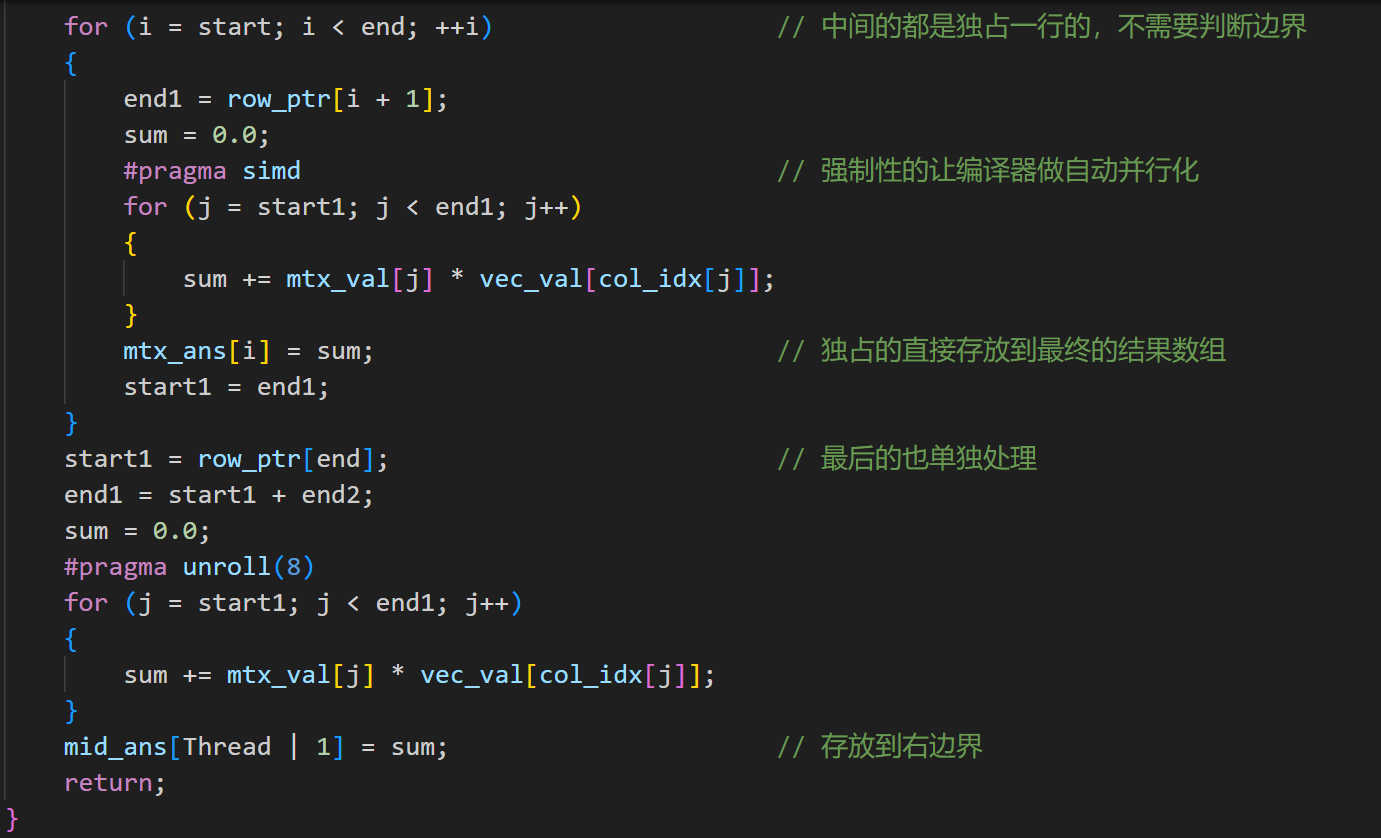
至此，所有的需要处理的负载恰好均分到每一个线程。

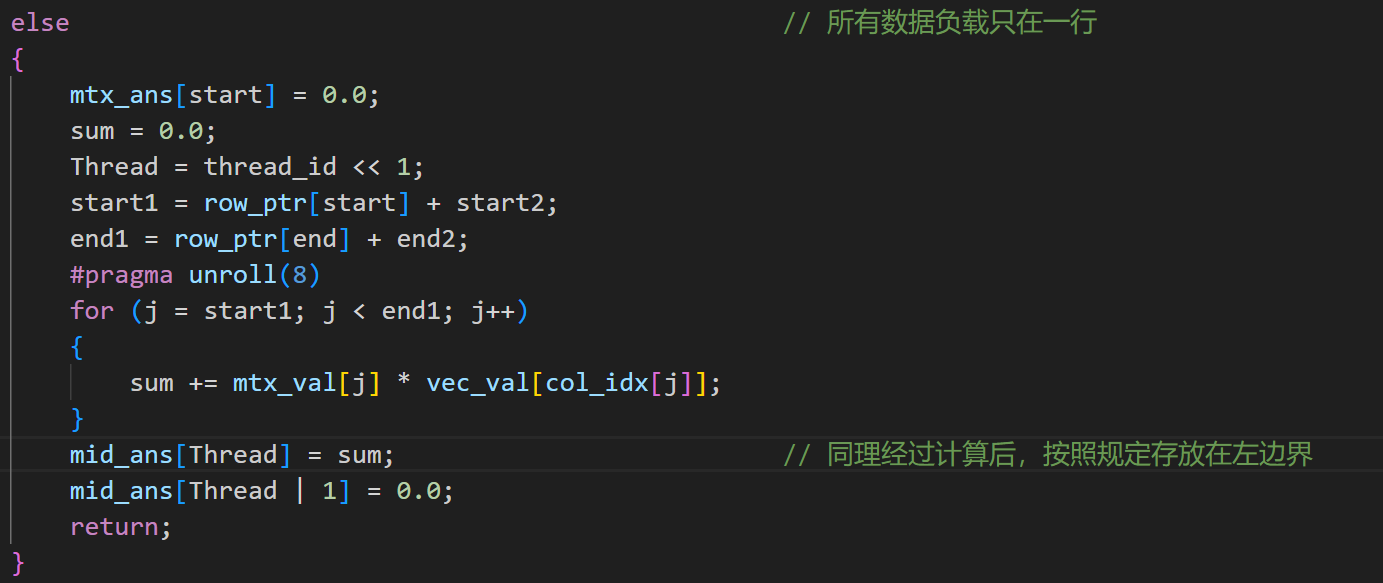
1. 计算并存储



图中第一次#pragma omp 中的 private(i)表示将i转化为让每个线程都建立一个私有变量，第二层表示建立一个并行for循环并把每一个i分配给一个线程执行，schedule(static)表示静态调度（因为我们已经把所有负载都均分了，所以静态调度更适配，不需要编译器帮我们动态分配），而nowait表示消除隐式的 barrier，无需等待，减少不必要的等待时间。

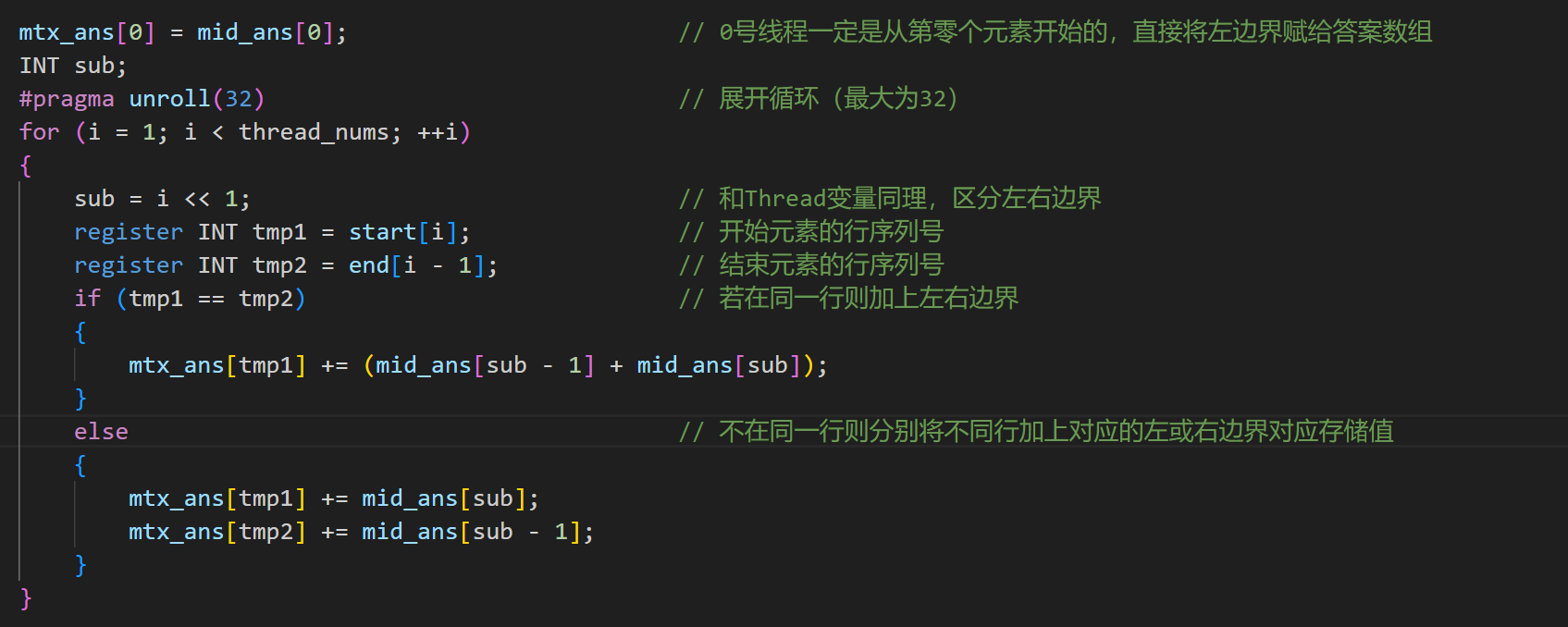






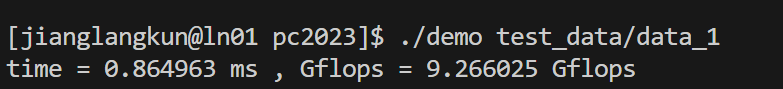
在计算过程中，要先判断所有的负载是否都在同一行，若不在同一行，将第一行和最后一行单独计算，并根据线程号以及是首还是尾将数据存放到mid\_ans数组（表示左边界和有边界的下标）中，中间部分则直接存入结果数组；若在同一行，则计算后按规定存放到左边界中。

1. 边界特殊处理



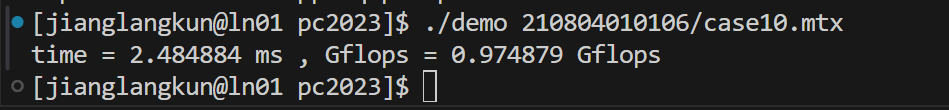
在最后一步中，需要判断每一个线程对应的负载是否只在一行中，根据情况的不同将存储的mid\_ans值，判断左右边界来赋对应的值，将这些在第②步遗漏的部分加回答案数组mtx\_ans。这一步只需要执行O(thread\_nums)时间复杂度的操作。

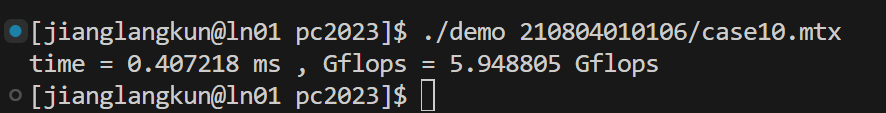
对于test\_data/data\_1而言，使用负载均衡会带来更差的执行性能。



这是由于data\_1本身数据的分布，让它的各个负载已经很高程度地均衡了，因此负载均衡带来的额外的性能损耗让它整体性能降低。

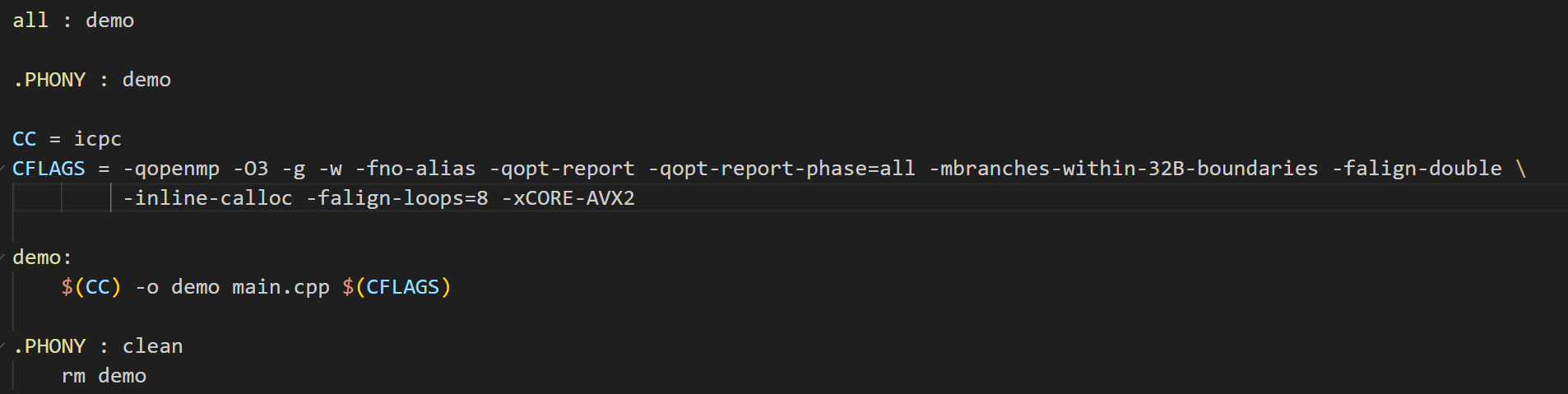
但，比较以下case10.mtx





可以看到，对于一般化的稀疏矩阵向量乘，负载均衡所带来的性能提升还是较为明显的。

(7). 编译指令优化



通过一系列指令可以让编译器通过更多的编译时间，带来更好的执行性能，即运行时间和每秒执行的浮点运算数。

1. **实验总结**

此次试验最为重要的负载均衡，这是稀疏矩阵向量乘法天然自带的一个关键因素，需要我们利用算法对其进行均分和计算，同时还要采取一定的措施来保证答案的正确性。通过负载均衡我们才可以让所有的计算资源得到最大程度的利用，减少cpu线程的闲置时间。但是这种负载均衡有时对于那种负载已经十分均衡的数据，会带来额外的损耗，策略要因不同的数据集而异。

实验结果展示了一系列优化手段的效果，但也揭示了性能优化是一个动态的过程。随着数据规模和计算任务的变化，可能需要不断尝试和调整优化策略。实验的成功并不代表终点，而是为进一步的优化工作奠定了基础。同时，在实验中获得的加速比是鼓舞人心的，但还需要对结果进行更详细的评估。进一步的工作可以包括性能分析工具的使用、更多编译器选项的尝试以及在实际生产环境中的验证。未来的工作也可以探索更先进的并行化技术、算法优化等方面，以提高程序的性能和适用性。