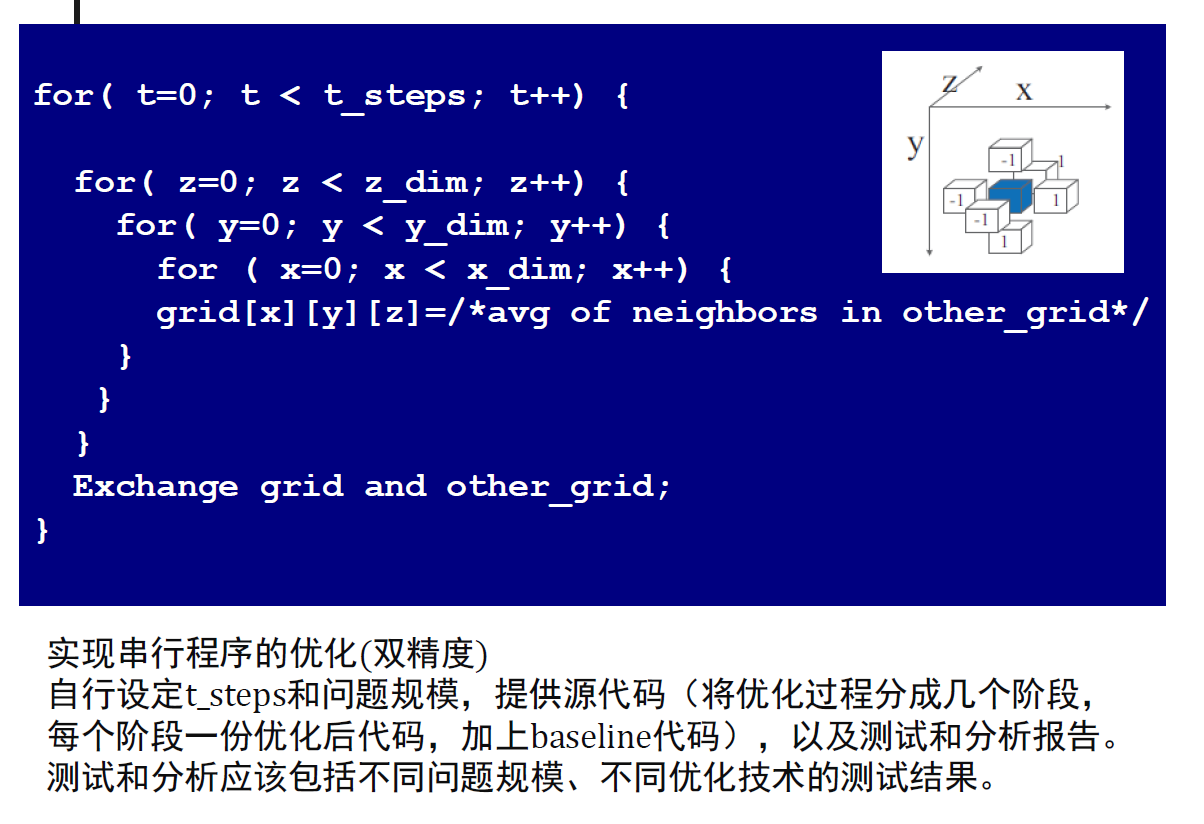
并行计算第一次作业

杨柳 2012011626 航院力2

作业内容：



1. 算法的初步实现

为简单起见，我们将边界设为刚性边界(保持始终为1)，在物理上可以认为是一个热源。在内部，则用0-1的随机数作为初始值。每次迭代的时候，仅仅更新内部的网格。

我们尝试着用三种方式实现迭代，分别是：

1. Dim1Exchange采用一维静态数组，其大小为总的网格数xsize\*ysize\*zsize。数据的存储顺序就是简单的列-行-面，或者说，三维网格中[i][j][k]的数据在一维数组中被存储在[i\*ysize\*zsize+j\*zsize+k]的位置。在每次迭代的时候，都要计算周围六个网格的索引。我们分配两个一维数组，ThisGrid和NextGrid。在迭代的时候，用ThisGrid中的数据计算NextGrid，一次迭代结束后，交换ThisGrid与NextGrid的首地址，完成更新。
2. Dim1DoubleIteration依然是一维静态数组。我们分配两个一维数组，ThisGrid和NextGrid。我们先用ThisGrid中的数据计算NextGrid，在所有网格都更新之后，我们再用NextGrid中更新过的数据计算反过来再更新ThisGrid。这样，我们就避免了交换数据的麻烦。我们在下文中将这种迭代方式成为交换迭代。
3. Dim3DoubleIteration采用三维静态数组。依然分配两个数组，类似2中进行交换迭代。之所以不像1中那样中那样交换首地址，是因为我对C/C++语言还不是很熟悉，尝试了好久没有成功，仅仅可以实现交换迭代。

在首次实现中，我们不做任何优化，代码附在附件中的“初步实现”文件夹。编译指令为默认指令。

理论上，一维数组和三维数组在内存上其实是一样的，所以我们可以预计，2和3在实际操作中性能应该少有差别。理论上，由于交换首地址需要的时间开销很小，所以1和2应该差别也不大。但是，实际情况并非如此。

迭代时间结果：（注：空间尺寸是指迭代的空间尺寸，而不包括边界。计算时间只取迭代部分，单位为s）

空间尺寸：100\*100\*100，时间步数：2000

1、Dim1Exchange：8.21; 8.23; 8.11; 8.3; 8.29;

2、Dim1DoubleIteration：4.24; 4.13; 4.2; 4.11; 4.12;

3、Dim3DoubleIteration：4.51; 4.23; 4.16; 4.14; 4.14;

空间尺寸：200\*200\*200，时间步数：2000

1、Dim1Exchange：67.82；68.65；66.87；67.9；66.41

2、Dim1DoubleIteration：43.62；42.51；44.39；42.91；45.94

3、Dim3DoubleIteration：44.42；43.96；37.41；43.97；39.36

可见，2和3的时间确实和我们预想的一样，差别不大，但是，1和2却相差很大。显然，简单的交换首地址不能解释这样的差别。

为了试图解释这样的结果，我调出了编译报告。我发现，Intel编译器在2中迭代过程中执行了很好的向量化 ，在在1中并没有出现。我猜想，是1中最后的指针操作让编译器不敢随意优化，最终导致了时间上的巨大差距。

此外，值得注意的是，在扩大了空间规模之后，2和3并没有如理想中那样按照比例扩大时间耗费，而是比预计要久一些。我猜测，这是由于扩大规模之后，Cache的大小却没有相应增加，在迭代中Cache miss 不按照比例地增加，进而使得时间耗费多出额外的部分。

考虑到我们在初步实现得到的结果，也就是三维数组和一维数组在交换迭代中表现接近，而采用交换首地址这样的指针操作则会导致编译器无法很好地优化，我们在下面的优化中，采取第三种方案作为Baseline。

1. 编译选项优化

在这个环节中，我们对上面提到的Baseline，改变不同的编译选项，尝试比较各个编译选项的影响，最后挑出一种理想的选择，作为下一个环节代码优化的编译选项。我们固定空间尺寸为100\*100\*100，时间步数为2000。

默认选项：4.51；4.23；4.16；4.14；4.14；

-O3：4.34；4.25；4.19；4.32；4.28；

-O3-ftz：4.21；4.27；4.22；4.25；4.29；

-O3-ftz-ip：4.28；4.26；4.27；4.23；4.27；

-O3-ftz-ip-fp-model fast=2：4.26；4.29；4.22；4.31；4.21；

很不幸的是，我并没有观察到各个编译选项明显的区别，也许是因为这个算法本身比较简单，优化潜力有限。不过，我决定还是选用最后一种选项来作为下一环节的编译选项。

1. 代码优化

在这个环节中，我们采用不同的代码优化技术，在不同的空间尺寸上进行测试。代码保存在附件中“代码优化”文件夹中。

1. 乘法代替除法

在之前的计算中，我们一直采用/6.0来实现取平均，这显然会拖慢计算。因此，我们采用\*0.166666666666667来替代除法。

空间尺寸：100\*100\*100 时间步数：2000

除法：4.28；4.26；4.23；4.21；4.34；

乘法：2.46；2.46；2.55；2.53；2.47；

空间尺寸：200\*200\*200 时间步数：2000

除法：38.6；39；38.45；35.14；35.89；

乘法：38.15；33.98；30.6；29.94；30.67

可以发现，引入乘法的优化后，速度提升不少。这对我未来的科学计算是很有指导作用的。

1. 修改迭代顺序

我现在的迭代顺序是和数据的存储顺序一致的。我认为，如果不考虑block，这样的迭代顺序已经是最优的了，因为我们在计算了一个格子后，由于Cacheblock的缘故，会同时取出后面的数据，正好方便下一个格子的计算。为了检验我的猜想，我修改了迭代顺序，进行对比。

空间尺寸：100\*100\*100 时间步数：2000

基线（ijk迭代）：2.46；2.46；2.55；2.53；2.47；

Ikj迭代：2.6；2.51；2.45；2.47；2.47；

Kij迭代：2.51；2.82；2.61；2.45；2.54；

Kji迭代：2.43；2.44；2.41；2.41；2.42；

Jik迭代：2.46；2.58；2.42；2.41；2.42；

Jki迭代：2.41；2.4；2.41；2.42；2.41；

跟我的预测不相符。我觉得之所以会这样，是因为GridSize和CacheBlock不吻合的缘故。如果我们让GridSize和CacheBlock是整数倍的关系，那ijk的计算速度优势会明显很多。可惜的是，我现在时间来不及了。

1. 内存对齐

针对乘法优化后的结果，我们强调内存对齐。不过虽然我尝试着声明内存对齐，但是最后编译的报告显示，其实并没有完全对齐。事实上，报告的结果与没有声明内存对齐的结果基本是一样的。这样，计算速度类似也就是自然而然的。我还没有弄明白为什么会出现这样的结果。

1. 分块

如果我没有理解错，分块的原理就是通过调整计算的顺序、增加缓存中的数据的利用率，使得缓存中的数据在被更新前能尽量多地被访问，进而减少总的更新缓存的次数。

我觉得在之前的ijk迭代中，一大问题是，在访问周围6个数据时，k方向的数据利用率较好，但是ij两个方向的不是很好。所以，我在ij两个方向进行分块处理。在每个块中先ijk迭代，然后逐块ij迭代。结果如下：（BlockNum是指Block的个数，BlockSize是每个Block的大小。ij方向是一样的.时间步数都是2000）

BlockNum：1 BlockSize：100（相当于不分块）

2.53；2.55；2.42；2.58；2.55

BlockNum：10 BlockSize：10

2.12；2.12；2.13；2.2；2.17

BlockNum：20 BlockSize：5

2.15；2.05；2.21；2.06；2.04

可见，分块确实可以有效地优化计算！而且，不同大小的分块对效果也是有区别的。这是因为，分块是缓存大小和数据利用范围的博弈。块太大了，那么缓存无法一下子存储这样多的数据，在每个块中计算都不得不更新缓存，使得分块的优势丧失；块太小了，那么缓存中的数据还来不及在一个块的计算中被反复利用就被迫进入下一个块中，很可能就立即被更新了。我个人感觉是，分块的大小最好和缓存大小差不多。这也许可以解释为什么最后一种分块的结果最好。