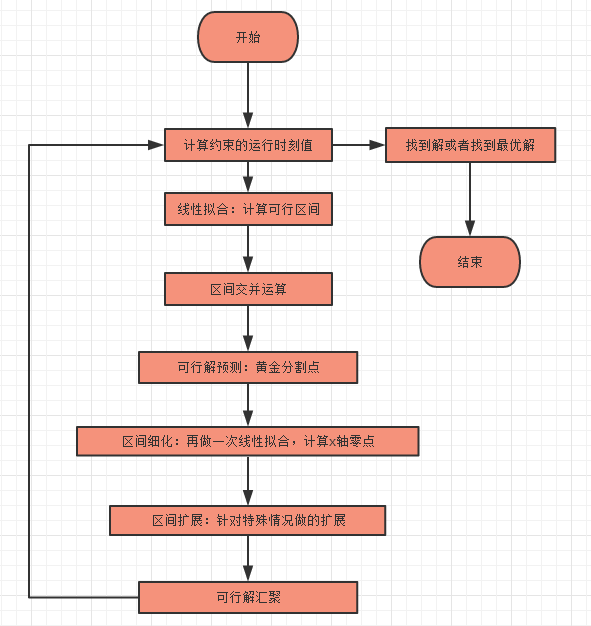
**LFF Parallelizing Solver 并行化算法设计方案**

首先先谈一下ZY（包括LFF Solver）的计算过程，以及存在的问题，ZY提出了任务层面的并行化方案，也即**多变量交叉搜索，**这个没有问题。在搜索树上的某一个变量方向的计算迭代过程如下图：



上述存在一个问题：重复计算，这个包含两个方面：

1）上一轮**已经计算过的当前变量**的运行时刻值重复被计算；

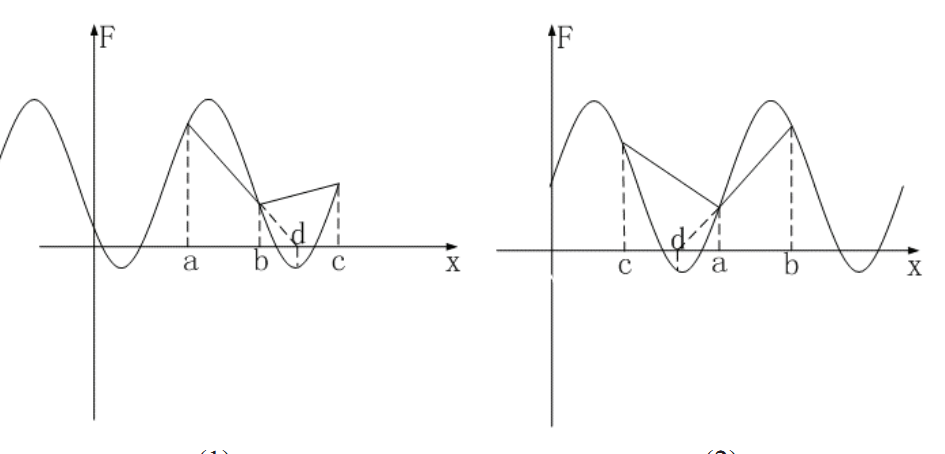
2）每一轮在做区间细化的时候寻找x轴零点（对于无解的线性拟合部分），又**重新的把所有的线性拟合做了一次**，不过这次是为了求零点，不需要做区间运算。

第一个方面的重复计算是因为在拿到预测解得时候，做按照当前自变量从小到大做插入排序，这里是不区分上一轮的预测解还是已经计算过的值，所以这里做了重复计算。

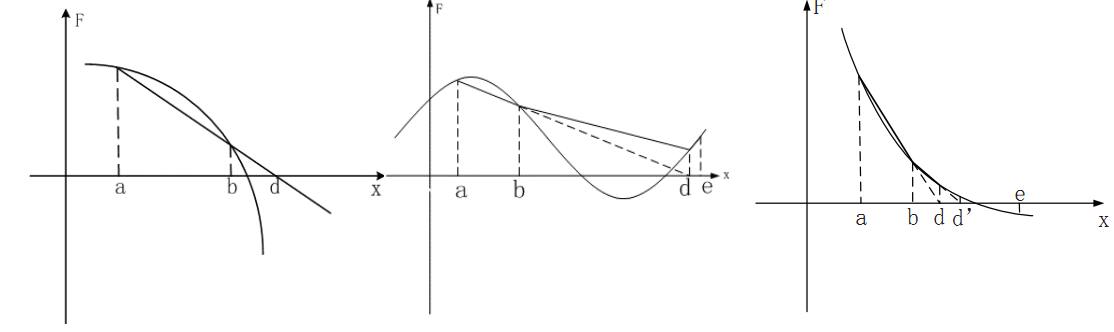
对于第二方面，是因为在第一次做线性拟合求解每一个子约束的可行区间的时候，没有把无解的对应的线性拟合的x轴零点计算出来，区间细化的目的主要就是收集所有的零点，所以这里又重新计算了一次。

下面是ZY做区间细化和区间扩展的示意：

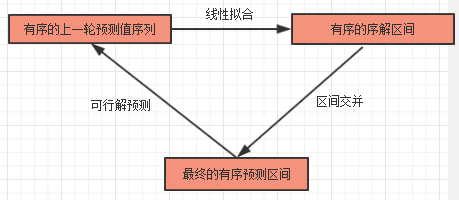
区间细化，当前拟合区间无解的时候，把拟合的零点加入到预测解，如下图：



区间扩展，针对特殊情况的处理，比如下图的图1和图3，主要是为了防止陷入无限的迭代



我观察了整个的计算过程，其实迭代过程的有序性是可以保证的，如下图：



为什么这是一个有序的环呢？原因如下：

假设这里有一个自变量序列{1,2,3}，那么我们按照次序两两做线性拟合，在这里只有{1,2}和{2,3}做拟合，假设分别得到的解区间是[1,2] 和[2,2,5]，也就是说**有序的预测序列经过拟合可以得到有序的解区间，经过区间交并运算依然是有序的；**然后针对最终的解区间[1,2] 和[2,2,5]预测可行解，假设是得到{1.35,2.25}，就是说有**序的解区间经过区间预测可以得到有序的预测解**，那么我们可以对本轮的预测解{1.35,2.25}和已经计算过的解{1,2,3}，如果要合并，二者做**一次归并即可得到最终的有序，**最后得到有序的区间。

这里还要考虑到不可行区间的零点的位置顺序，这个和前后区间的预测点的顺序也是容易确定的。

经过上述的分析，ZY以及LFF Solver还存在另一个问题，就是针对预测解的插入问题，要是选择插入，应该从尾部插入以减少遍历时间，但是ZY都是从**头部插入，其实这一部分时间很费时**。

**关于CUDA的存储结构，主要分为一下6种，说明如下：**

1、寄存器Register：线程私有；

2、局部存储器 local memory：线程私有；

4、全局存储器 global memory：全局共享；

3、共享存储器 shared memory：block中的线程共有；

5、常数存储器constant memory：特点：只读，全局共享；

6、纹理存储器 texture memory：只读，其适合实现图像处理和查找。

对于寄存器和局部存储：在我们做并行计算的时候，线程内部的临时变量肯定要用到这些空间；

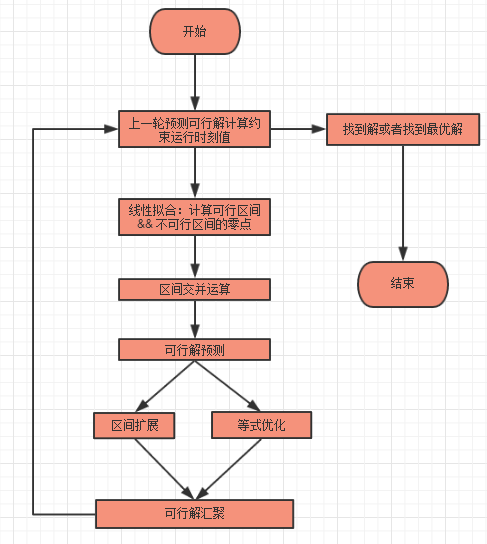
对于全局存储器：大部分使用的都是全局存储

对于共享：这个实在block内部共享，在计算max、min等等时候可能用到；

对于常数存储器：相当于const，迭代参数可以使用这块

对于纹理存储器：基本用不到，

下面是我的想法，主要流程图如下：



之前分析的有序性可以保证上述的过程均可以并行化实现，算法思路如下：

设ArrayA表示已经执行过运行时刻值的向量，ArrayB表示上一轮预测的可行解，设置N为迭代次数，ArrayB内部记录着父节点的index（父节点可能在ArrayB核ArrayA中），最终的解区间是final，预测是ArrayP，零点是ArrayZero

初始状态：ArrayA为null，ArrayB为随机值，其余的都是null

1: for i : 0->N

2: 设置线程池Thread Pool大小sizeB，并做init

3： 启动核函数并行计算运行时刻值

4: if arrayB中存在满足复合约束的向量v

5 return v；

6: else

7: 设置线程池Thread Pool大小sizeB，并做init

8: 评估arrayB中的每一个解向量的优先级

9: endif

10: 设置线程池Thread Pool大小为sizeA+sizeB，并做init

11: 并行执行ArrayA和ArrayB，做线性拟合，并计算解区间

12: 设置线程池Thread Pool大小为sizeA+sizeB-1，并做init

13: 解区间并行交并运算得到最终的解区间final

14: 设置线程池Thread Pool大小为sizeFinal，并做init

15: 并行化的获取每一个区间的预测解ArrayP

16: 设置线程池Thread Pool大小为sizeA+sizeB，并做init

17: 收集线性拟合过程计算好的零点ArrayZero

18: 归并ArrarP和ArrayZero为ArrayP

19: 归并ArrayA和ArrayB为ArrayA，ArrayB=ArrayP

20： 设置线程池Thread Pool大小为sizeA+sizeB，并做init

21： 获取ArrayA和ArrayB的最大值最小值，完成区间扩展

20: end for

21：设置线程池Thread Pool大小为sizeA+sizeB，并做init

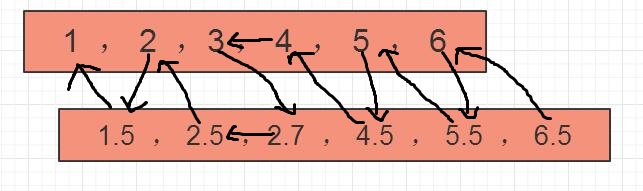
22：并行化的搜索已经计算的优先级最大的值并返回（参数迭代结束）

上述关于使用线程池来完成并行化的详细细节以及部分数据结构如下：

对于当前向量v，假设当前的自变量是vi，那么除了vi之外的其余向量在当前搜索方向vi都是不变的，所以vi作为一个全局的向量存储，在整个迭代过程，真正变化的是当前的自变量vi，比如ArrayA就是关于vi的一个有序序列。

那么v和ArrayA和ArrayB都可以放到全局变量上存储，那么ArrayA和ArrayB做线性拟合的时候，怎么区分两个节点是否是相邻节点呢？

假设对于某一个路径约束constraint，ArrayA是{1,3}，ArrayB是{2}，那么并行化执行完ArrayB后就可以执行线性拟合，对于ArrayA和ArrayB是一个有序的序列，那么每一个元素的前驱节点是很容易确定的，比如ArrayB中的2的前驱节点是ArrayA中的1，ArrayA中的3的前驱节点是ArrayB中的2，就可以根据前驱关系，并行化得计算约束，前驱关系类似如下图



这个前驱节点的确定是在上一轮的归并ArrayA和ArrayB为ArrayA，ArrayB=ArrayP的时候做的，所以这个可以保证所有的节点可以找到比自己小的元素，然后做线性拟合。

**针对每一次的线性拟合，要做如下的事情：**

If 存在解区间

保存解区间

Else

计算拟合曲线零点，并判断是否有效

若有效保存该零点

**因为这里已经把所有的信息计算并保存下来了，**所以可以避免ZY的第二次线性拟合计算

**针对区间交运算：**

采用类似折半搜索的思路，来完成区间合并。

思路如下：

对于区间的序列a，

While（sizeA>=2）

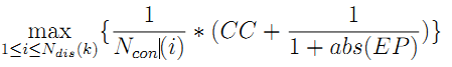
并行化合并a[0]&a[sizea-1],a[1]&a[sizea-2]...

sizeA=sizeA/2;

**针对并行评估每一个解得优先级：**

就是根据已经计算好的每一个析取范式的合区子式的每一个简单自约束的满足程度，计算公式如下

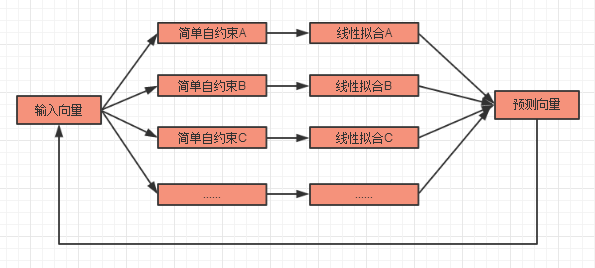




**对于可行解并行预测：**

根据收集到的解区间集合A，针对每一个区间求黄金分割点。

上述我是按照一个只有一个简单子约束来考虑的，所以数据存储就是一个vector，但是假如对于多个约束呢，大致流程如下



那么所以数据存储就是一个矩阵，然后操作就是矩阵的每一行的之间的相互操作和行内的操作。

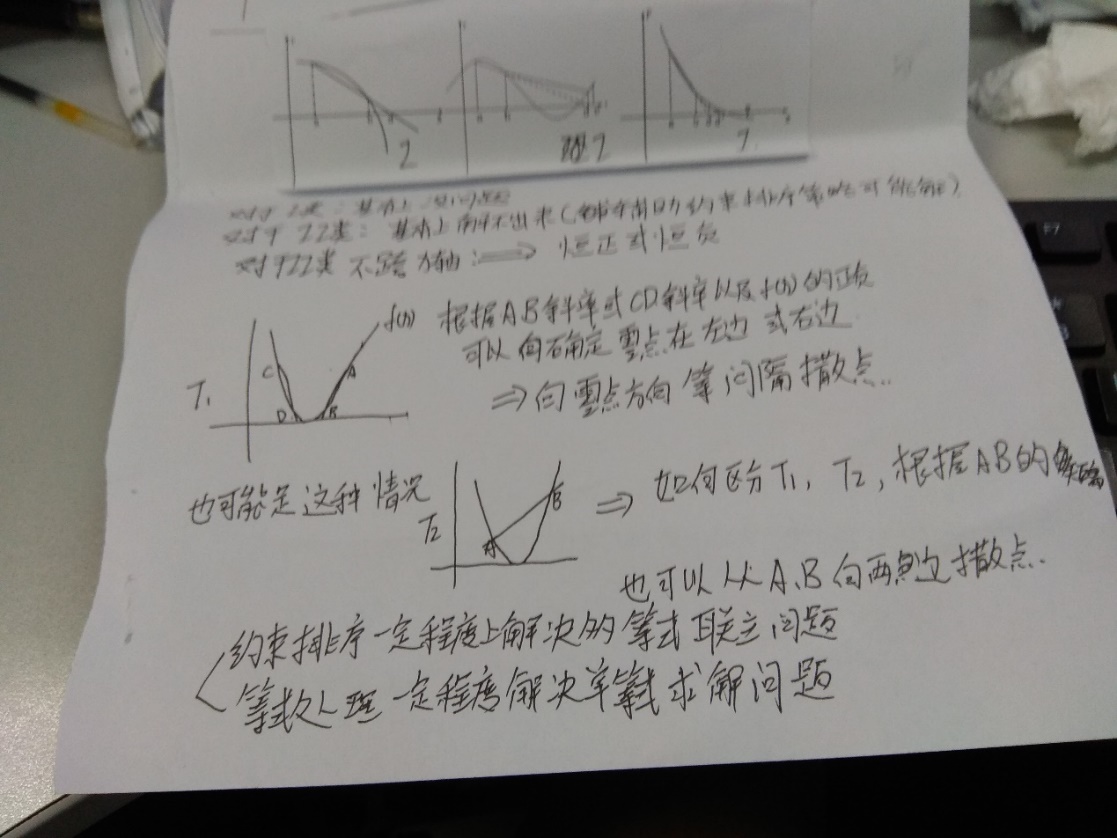
上面的还仅仅是是在某一个变量方向上的计算的并行，根据之前提出的**多变量交叉搜索**和**基于约束的依赖分析搜索空间递减**来完成上层的任务层面的并行化，这个可以使用DFS或者BFS遍历搜索树，或者辅助一个优先级队列来完成。

与此同时，我还在考虑其他的优化:

**1,增加计算量以提高拟合信息量**。我认为除了黄金分割点之外，可以在尝试其他的点，是不是可以考虑针对解区间做三等分，取其三分点，或者黄金分割点+随机取点，这个的目的是增加一定的随机性和拟合信息的量。

**2，针对等式的优化。**在ZY之前的工作都没有考虑对于等式的优化，因为对于线性拟合很难拟合到一个点上，这个对于某些benchmark十分明显，比如 y=x\*x的是基本解不出来的。同时根据之前的实验经验，等式之间的约束次序明显影响求解速度和概率（主要针对多等式联立的情况）。

初步的想法是对等式约束做线性拟合的时候，根据拟合曲线的走向，在区间均匀取点，想法是如下图：



通过撒点可以一定程度的提高某一个等式约束的求解出来的概率，还有通过简单自约束重排序，来提高多等式联立的求解出来的概率，通过这两方面提高对于涉及等式约束的求解速度。