摘 要

为了适应日益增长的宽带信号和非线性系统的工程应用，用于分析瞬态电磁散射问题的时域积分方程方法研究日趋活跃。本文以时域积分方程时间步进算法及其快速算法为研究课题，重点研究了时间步进算法的数值实现技术、后时稳定性问题以及两层平面波算法加速计算等，主要研究内容分为四部分。

……

关键词：时域电磁散射，时域积分方程，时间步进算法，后时不稳定性，时域平面波算法

ABSTRACT

With the widespread engineering applications ranging from broadband signals and non-linear systems, time-domain integral equations (TDIE) methods for analyzing transient electromagnetic scattering problems are becoming widely used nowadays. TDIE-based marching-on-in-time (MOT) scheme and its fast algorithm are researched in this dissertation, including the numerical techniques of MOT scheme, late-time stability of MOT scheme, and two-level PWTD-enhanced MOT scheme. The contents are divided into four parts shown as follows.

……

Keywords: time-domain electromagnetic scattering, time-domain integral equation (TDIE), marching-on in-time (MOT) scheme, late-time instability, plane wave time-domain (PWTD) algorithm

目 录

第一章 绪论

第二章 时域积分方程基础

第三章 SDN网络下的并行路由优化算法设计

3.1 引言

路由优化能够有效提高网络性能，提高网络资源利用率，保障用户Qos需求，软件定义网络是一个新兴的，将控制和数据平面分离的网络范型，在SDN网络中，控制平面被分离驻留在一个集中的网络控制器上，它提供了用于网络应用程序和控制数据平面的编程接口（使用标准协议，例如OpenFlow[1]）。SDN的结构使得网络管理者可以根据网络当前情况来进行有效的在线路由优化。微软[1]和谷歌[1]的实验结果证明，在SDN网络结构的数据中心网络中，路由优化能够在网络吞吐量和链路利用率上达到接近最优化性能的表现。但是另一方面，SDN网络下中心控制的路由优化面临大规模计算问题，第一，随着网络应用的快速增加，在SDN网络中短时间内可能会有大量业务到达控制器，所以控制平面必须短时间内为大量业务计算路由。第二，为了适应大业务量的加入，网络规模也快速增大(e.g., a data center network may have hundreds of thousands of switches [9]).因此，对大量业务的快速和高效的路由优化成为了一个重要却困难的问题。为了加快网络优化过程，减小网络延迟，本文利用GPU的大规模的并行能力来加速网络优化算法的计算过程。对大规模业务的路由优化问题可以建模成一个多商品流问题，将网络路由业务作为输入，寻找最优的路由路径来最优化效用函数，效用函数通常设置为对网络拥塞程度的评价水平，比如，最常用的效用函数是最小化最大链路利用率（MLU），简单地被定义为利用率最高的那条链路的链路利用率[],另外一些把所有链路的链路利用率的和作为效用函数（[],[]）。这些效用函数的逻辑是：（1）低链路利用率意味着低的网络延迟。（2）.维持低的链路利用率意味着预留更多的空间给其他将来到达的业务。但是大量基于实际拓扑的实验表明链路利用率效用函数，特别是链路利用率，在网络利用率没有达到拥塞程度的时候，不是对网络优化的较好评价函数[]。在这个实验中，当链路利用率低于0.9的时候会造成不可忽略的网络性能中断。所以作为替代，文章在链路容量约束的情况下来优化路由总代价。我们假设已经知道短时间内到达的一批业务，控制器需要计算出满足链路容量约束的路径，并且最优化链路路由总代价。为了使得加入网络的业务尽量多，我们设定被阻塞的业务代价为一个较大值。本文中考虑的优化问题是一个NP-hard问题，因为他等价于一般的带整数约束的商品流问题[]。路由优化问题是一个组合优化问题，一般来说，在大规模网络中求解路由优化问题是计算困难的，因此，为了在短时间内求解路由优化问题，很多启发式算法被提出来[]。

然而，大部分算法都是单线程的串行算法，串行算法的复杂度随着网络规模大小呈指数上升，SDN网络下的在线路由优化要求很短的路由优化时间，为了加速算法，一个很自然的选择就是设计并行优化算法，采用大量的线程同时计算路由路径。本章主要设计两种路由优化算法，第一种是基于备选路径选择的路由优化算法，采用遗传算法来优化目标函数，并且设计了遗传算法的并行版本，获得几十倍的加速比，第二种是基于拉格朗日松弛的优化算法，算法把链路容量约束松弛到目标函数，并把路由优化问题分解成一堆路由路径计算问题，从而采用GPU进行并行计算。

3.2 网络模型和问题建模

3.2.1 网络模型

本文将SDN网络建模成无向图 $G(V,E)$，V表示所有的点集合， E是所有边的集合， $n=|V|$ 和 $m=|L|$分别表示点数和边数。对每一条边$(i,j)\in E$， w_{ij}表示此边 (i,j)上的权重（传输一单位的流量需要的代价），不失一般性，我们假设每条链路上的w_{ij}是整数，对每一条边 (i,j)， c_{ij}表示此边上的容量，假设D表示需要被路由的业务需求集合，业务$d\in D$是一个元组 (s_d,t_d,bw_d)，其中， s_d表示业务的源节点，t_d表示业务的目的节点，bw_d表示业务d需要的流量带宽。业务量工程问题将网络业务需求和网络拓扑作为输入，计算出每条业务的路由路径以使得效用函数代价最小化，在SDN网络中，业务的路径在中心控制器上计算出来，为了满足用户业务的QoS要求，本文假设链路利用率不能超过一个固定阈值 θ，因此，一些业务会因为链路上容量不足而被阻塞，本文用 $\hat{D}$来表示这些被阻塞的业务集合。

3.2.2 问题建模

本小节，我们把路由优化问题建模成一个混合整数规划模型（MILP），通常，路由优化中应用最广泛的效用函数是[][]，与链路利用率成正比的，但是，最近的研究表明链路利用率不能很好的代表网络表现情况，而且一味的最求低的链路利用率，容易造成出现路由代价过大，路由跳数过长的问题，所以，本文结合了两种情况，采用新的效用函数，如下：

$$f(\mathbf{d})=\begin{cases}\sum\limits_{d\in D} c(p_d) & \text{if available bandwidth is enough}\\ \sum\limits_{d\in \hat{D}} bw_d & \text{otherwise}\end{cases} \tag{3-1}$$

其中p_d是计算出来的对应于业务d的路径，$c(p_d)$ ($c(p_d)=\sum_{(i,j)\in p_d} w_{ij}$)表示的是此路径$p_d$的代价。因为不知道带宽是否足够容纳所有业务，（1）并不是衡量所有情况，为了衡量所有情况，我们对$G(V,E)$构建辅助图$G_a(V_a,E_a)$，初始时，让$G_a(V_a,E_a)=G(V,E)$，然后，对每个点$v\in V$ 和 $u\in V$，在$G_a(V_a,E_a)$中添加一条链路 (v,u)，并且设置链路 (u,v)的容量和代价分别为 ∞和 nM ，其中 M是 $G(V,E)$中最大的链路代价，这样，$G_a(V_a,E_a)$就有足够的容量来容纳业务需求，如果某条业务被路由到$G_a(V_a,E_a)$中，那么就表示这条业务被阻塞了，加入了辅助图$G_a(V_a,E_a)$后，路由优化的效用函数可以表示为：

$$z^* = minimize\ f(\mathbf{d}) = \sum_{d\in D} c(p_d) = \sum_{d\in D}\sum_{e\in p_d} w_e \tag{3-2}$$

在路由优化问题中，每个业务只能够路由到一条路径上，以下整数约束能够保证每个业务只走一条路径

$$\sum_{(i,j)\in E_a} x_{ij}^d - \sum_{(j,i)\in E_a} x_{ji}^d = \begin{cases} 1 & \text{if } i = s_d \\ -1 & \text{if } i = t_d \\ 0 & ohterwise \end{cases} \quad \forall i\in V_a, \forall d\in D \tag{3-3}$$

其中 x_{ij}^d是一个0，1整数变量， $x_{ij}^d=1$表示业务d路由经过链路(i,j)，为了避免链路拥塞，路由路径需要满足以下的链路容量约束:

$$\sum_{d\in D}\sum_{(i,j)\in E_a} x_{ij}^d \cdot bw_d \le \theta \cdot c_{ij} \ \ \forall (i,j)\in E_a \tag{3-4}$$

在这个模型中，变量的数量随着业务量大小和网络规模大小呈指数增长，所以这个MILP模型在大规模情况下很难求解。

3.3 基于遗传算法的路由优化算法

遗传算法是一种模拟自然进化过程搜索问题最优解的启发式算法，遗传算法模仿达尔文进化论和自然选择过程来评价挑选最优解集合，从而找寻较优化的解，遗传算法从一个代表问题的可行解的种群出发，一个种群中不同个体代表了不同的解，每个个体实际上是一个染色体，染色体携带表达当前解的信息编码，初代种群产生后，对每个染色体个体进行评价，按照适者生存，优胜劣汰的原则，使得较优的个体更有可能把自己的遗传信息传递给下一代，从而得到更优化的后代，算法过程中，对一部分基因进行变异，好的变异能够提高解的质量，增加算法的搜索空间，避免算法收敛于局部最优解。遗传算法的步骤流程图如下所示：

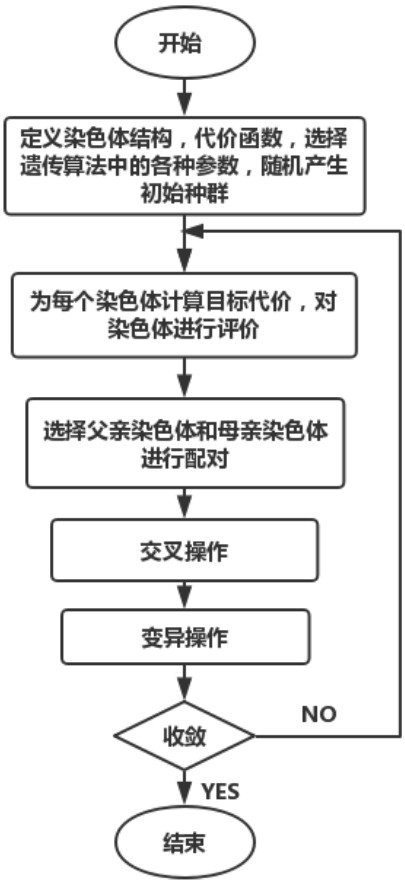
开始
定义染色体结构，代价函数，选择遗传算法中的各种参数，随机产生初始种群
为每个染色体计算目标代价，对染色体进行评价
选择父亲染色体和母亲染色体进行配对
交叉操作
变异操作
收敛
NO
YES
结束

图 3-1 GA算法流程图

3.3.1 遗传算法设计

3.3.1.1 定义染色体结构

[5][], 由前面的讨论可以知道，路由优化问题求解过程是寻找最优的业务路径集合，使得效用函数最小化，在论文[]中作者（效用函数是常见的最小化最大链路利用率）提出了一种路由优化算法，在文章中，为每一个业务$d \in D$,产生K条不同的备选路径作为备选路径集合$P_i = p_d^1, p_d^2, p_d^3 ... p_d^K$,通过遗传算法过程来确定每个业务选择哪一条备选路径，从而找到最优解，本文采用相同的思想来求解路由优化问题，假设业务数量为$|D|$,初始染色体集合大小为POP对于第j（$jin[1:POP]$）号染色体一个染色体C_j是一个$|D|$维数组，$C_j^i = k$fi$k \in [1:K]$表示在第j号染色体中，业务i选择了第k条备选路径，$C_j^i = -1$表示业务i不加入网络中（在辅助图上路由），不占用链路资源，$|p_d^i|$表示业务d的第i条备选路的路径代价，rp_d^i表示路径p_d^i上的最小可用容量,$rp_d^i = min(r_e | e \in p_d^i)$,其中$e \in p_d^i$ 表示路径p_d^i上的边，r_e表示此边e上的剩余容量。

3.3.1.2 初始可行解生成

可行解表示满足容量约束的解，为了使得遗传算法有效，初始解的质量很重要，产生的初始解要尽量好，要有更多的业务要能加入网络中，而且保证业务的路径代价较小，文章中采用一种简单的贪心算法产生出初始可行解，算法过程如下所示： 对某一个染色体，算法随机为每个业务生成所选择的备选路径编号，但是这样选择出来的路径集合有可能会超过网络链路的容量限制，从而使得解变得不可行，要得到可行解，必须从染色体中剔除一部分业务，使得他们阻塞，为了得到比较优秀的初始可行解，本文提出一种启发式过程来确定能加入的业务，以及必须剔除的业务，一方面，要使得目标函数变小，那些流量需求较大的业务应该优先被加入到网络中，但是如果大流量的业务的路由代价很大，经过了一条很长的路径，就会大量的浪费网络中的链路容量资源，所以算法过程对当前染色体j中的业务和其路径按照$\frac{bw_d}{\sqrt{|p_d^{k_j^d}|}}$的值进行排序，其中$bw_d$代表当前业务$d$所需要的流量大小，$|p_d^{k_j^d}|$代表当前染色体$j$所选择的$p_d^j$中的第$k_j^d$条路径的代价大小。这样算法优先加入$\frac{bw_d}{\sqrt{|p_d^{k_j^d}|}}$值较大的业务,观察目标函数，目标函数是优化路由代价最小，而$\frac{bw_d}{\sqrt{|p_d^{k_j^d}|}}$较大意味着较大的流量经过较小代价的链路进行路由，这种路由是很理想的，尽量节省网络的链路使用资源的同时，又减小了总体目标函数，所以这个比例值是对业务路由个体优劣程度的较好评价，于是文章采用这个比例值

Algorithm 1 初始可行解产生

Require: $G(v,E)$:网络拓扑; P:备选路径集合; C:未初始化的染色体集合;

Ensure: C:可行染色体集合;

1: **for** each $c_j \in C$ **do**
2: **for** each $c_j^d \in c_j$ **do**
3: $c_j^d \leftarrow -1$
4: **end for**
5: **end for**
6: **for** each $c_j \in C$ **do**
7: **for** each $c_j^d \in c_j$ **do**
8: $c_j^d \leftarrow k_j^d$，其中k_j^d为1到K之间的随机值，随机选择一条备选路
9: **end for**
10: 对染色体中的的每个业务需求按照值$\frac{bw_d}{\sqrt{|p_d^{k_j^d}|}}$进行降序排序。
11: **for** each $c_j^d \in c_j$ **do**
12: **if** $rp_d^{k_j^d} >= bw_d$ **then**
13: 加入路径$p_d^{k_j^d}$到网络，更新网络链路容量。
14: **else**
15: $c_j^d \leftarrow -1$
16: **end if**
17: **end for**
18: **end for**

来确定路径加入网络的优先级（后面第节也会应用类似的思想来调整路由），每次按照比例值排序的结果将遗传染色体所选择的路径$p_d^{k_j^d}$尝试加入到网络中，如果$rp_d^{k_j^d} >= bw_d$，表示路径经过的链路有足够的容量来容纳这一个业务，所以加入业务到网络中，并且更新网络的链路容量大小，反之，如果$rp_d^{k_j^d} < bw_d$,这个业务选择这一条路径会超过网络链路的容量限制，于是这条业务被阻塞，染色体中的相应基因位置被设置为-1。大量重复以上可行解的产生过程，则可以得到一个较好的初始染色体集合C。

3.3.1.3 评价与交叉

评价过程对本轮产生的染色体，计算其相应的目标效用函数值，并且对染色体按照效用函数目标进行降序排序，由于算法过程随机交叉，可能会产生不可行

的染色体解（链路容量超限），把这样的染色体评价为一个个很大的代价，从而在选优时被排除掉，目标值排在最前面的前α个染色体为最优集合A，这个集合中的染色体为精英染色体，精英染色体将直接保留到下一轮迭代，此后的β个染色体为较优染色体集合B，较优染色体集合中的染色体不会直接保留到下一轮，但是他们有繁殖的权利，可以和精英染色体一样产生后代，遗传自己的选路信息，排在最后面的γ个染色体，组成劣等染色体集合G,由于其目标值一般较大，其选路策略不可取，算法直接扔掉这一部分劣等染色体，而且不让劣等染色体进行繁殖。交叉过程从精英染色体集合中A中随机选取一个染色体$c_i \in A$作为父亲,从精英染色体集合和较优染色体集合的并集$A \cup B$中随机选取一个染色体$c_j \in A \cup B$作为母亲，将c_i和c_j进行均匀交叉得到新的染色体s，均匀交叉过程示意如下所示，均匀交叉的过程是，对s的每一个基因点位以$\%50$的几率选择继承父亲或者母亲的相应点位的路径选择。重复以上过程$\beta+\gamma$次，从而产生$\beta+\gamma$个新的子染色体来替换当前染色体集合评价函数排在后$\beta+\gamma$个的染色体，因此得到新的染色体集合C。

3.3.1.4 变异与迭代终止

变异过程采用随机变异，随机在已经交叉后的集合C中选取$M \in [0:POP]$条染色体，对某一选定的染色体$c_j \in C$,随机选取m个业务基因点位，进行变异，将当前已经选择的路径编号随机改变为备选路集合中的另外一个值，由于变异过程是为了提高算法的的搜索空间，避免算法陷入局部最优解，但是实际实验过程中发现如果M和m值设置较大，可能使得算法收敛较慢，因为大量的变异可能会导致较优秀的可行解变成不可行，因此会丢掉这些优秀的解，因此实验中M和m的值设置得较小，变异的作用有限，算法的收敛效果主要来自于交叉步骤。算法每次迭代都会记录当前可行染色体解的最优目标值，如果当前迭代找到的最优可行解目标值小于全局最优值，则更新全局最优值，并且记录对应的染色体为最优解，如果迭代L次，全局最优值不被更新，则判定算法收敛，算法停止。

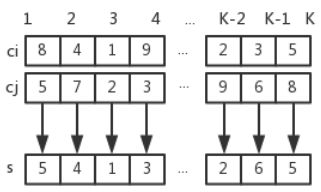

图 3-2 遗传算法均匀交叉过程

3.3.2 基于GPU的并行遗传算法设计

3.3.2.1 并行评价算法设计

遗传算法中最消耗时间的部分是染色体评价部分，由于需要评价大量的染色体，评价每个染色体都需要大量计算开销，但是幸运的是遗传算法具有天然的并行性，每个不同的染色体评价可以并行执行，更进一步，每个染色体中的不同基因的计算也可以并行执行，这样并行粒度是很大的。下面将介绍评价过程的具体并行实现算法，算法主要包括可行性判断和效用函数计算两个步骤，算法计算流程如下图所示： 符号解释：
染色体（chromosome）基因编号：c_i^d表示第i个染色体的第d个基因位置。
备选路径集合（paths），p_i表示第i业务的所有备选路集合。
业务带宽（bandwidth）,bw_i表示第i个业务需要的带宽大小。
链路流量（flow），f_e表示第链路e上占用的流量大小。
链路单位代价（weight）,w_e表示链路e上的代价。
共享内存中间数组（shared），sh_e表示链路e上的总代价。
如图所示，由于每个染色体的计算过程是独立的，算法为每一个染色体开辟一个block，每个block内部每个线程负责染色体上的相应业务，首先通过寻址备选路径集合找到这个业务选择的路径，路径上的每一个链路都需要被占用流量，于是遍历这条路径，将业务的流量加到相应的链路上，所有线程同时计算，最后得到链路上占用的流量大小为数组$flow$,并行比较$flow$和$capacity$数组，如果某一线程发现容量超限，则设置链路代价为无穷大（INF）表示此染色体不可行，最后对得到的链路代价数组进行求和，求和采用GPU上经典的并行规约算法，最后得到染色体对应的效用函数值。其中$flow$和$shared$两个数组被同一个block内部的线程多次访问，利用程序访问的局部性，将两个数组分配到共享内存中，这样避免了对global memery的大量慢速访问，大大提高程序计算速度。 以上代码GPU上的评价过程伪代码，算法开始时先开辟两个大小为边数大小的共享内存数组，然后每个线程负责一个基因点，寻址基因点选择的路径，对路径上的每一条边的流量加上业务的流量大小，注意到这个时候可能同时存在多个线程访问同一个$flow$位置，所以需要同步保护，使得每个线程的加法操作都能正确执行，使用CUDA提供的atomicAdd函数来对$flow$数组进行加法操作，atomicAdd保证其调用的操作是原子操作，从而多个线程对同一$flow$位置的加法操作必修是串行执行的，一个add操作必修一次性执行完成（取址、译码、执行、访存、写回），在当前add操作执行完成之前，其他线程的add操作必修排队等待。$blockDim$表示一个block内的线程数量，代码中的for循环每次迭代标号i偏移$blockDim$的长度，这是因为业务量

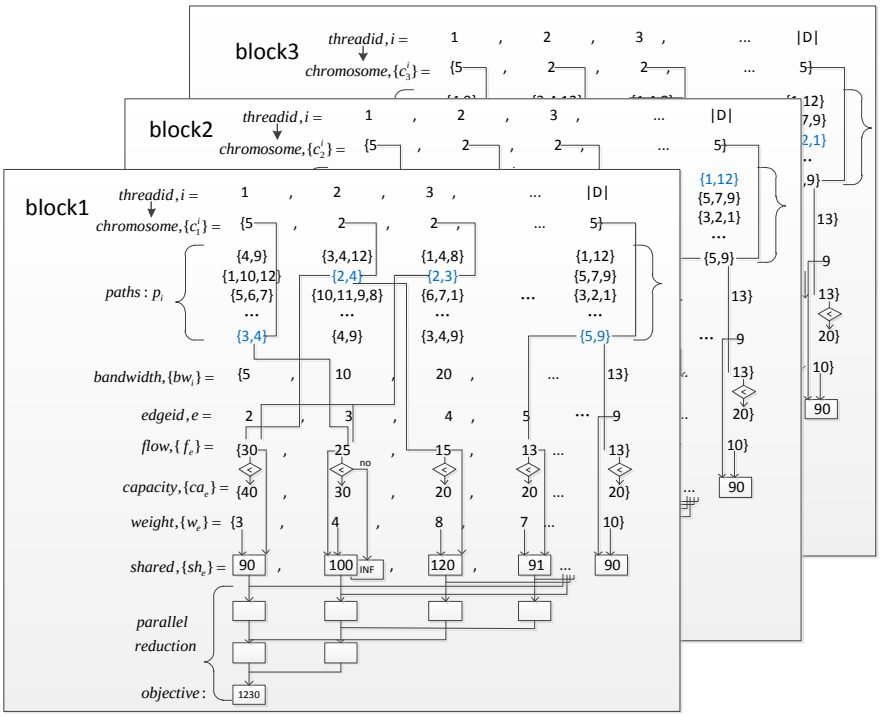

图 3-3 GPU上遗传算法评价函数计算过程图

数$|D|$可能大于$blockDim$,如果这样的话，每个线程可能会负责多个业务的统计计算，每个业务标号相差$blockDim$大小。当线程$flow$大小统计完成之后，必修进行同步（syncthreads()）,同步操作保证block内部的所有线程执行到同一步骤，也就是统计完成的线程必修等待其他统计线程都执行完成后才能继续执行，因为只有所有线程都完成对$flow$数组的加法操作，$flow$数组的统计才完整，才能够进一步进行比较操作。代码第19到25行计算每一条链路的路由代价，并把结果储存在$shared$数组中，如果链路上的流量小于其容量约束，那么链路代价就等于单位代价$weight$和链路流量$flow$的乘积，反之，如果流量超限，就设置$shared$为无穷大。同理，当线程计算完成后必修进行同步（syncthreads()）操作，以使得shared数组正确完整。为了充分利用GPU多线程，代码最后进行并行规约操作进行求和，for循环中每次将后一半的$shared$数组加到前一半，规约过程中必修进行同步（syncthreads()），以保证加法过程计算完整，最终求和值规约到一个下标0，将shared[0]中的值写入到$objective$数组中。另外，由于CUDA每个block支持的shared memory 大小有限，并且分配shared memory太多，会使得SIMT上的资源不足，一个block中的活跃warp数量不足，造成SIMT上不能有足够的活跃warp数量来进行切换，从而掩藏其他warp的访存延迟开销，这样会使得执行速度下降很多，所以在实际设计计算

```
1  void evaluate(float*capacity,
2                    float*bandwith,
3                    float*weight,
4                    int chromsome[][],
5                    int paths[][]
6                    int &objective[]){
7
8  __shared__ float flow[E]={};
9  __shared__ float shared[E]={};
10 for(i=threadid;i<|D|;i+=blockDim){
11         int p[]=paths[i][chromsome[blockid][i]];
12         for(j=0,j<length_of(p);j++)
13                 flow[p[j]]+=bandwidth[blockid];
14 }
15 __syncthreads();
16 for(i=threadid;i<E;i+=blockDim)
17 {
18         if(flow[i]<capacity[i])
19                 shared[i]=flow[i]*weight[i];
20         else
21                 shared[i]=INF;
22 }
23 __syncthread();
24 for(int s=E;s>1;s=(s+1)/2)
25 {       if(e<s/2)
26                 shared[e]+=shared[e+(s+1)/2];
27         __syncthreads();
28 }
29 if(threadid==0)
30         objective[blockid]=shared[0];
31 }
```

图 3-4 GPU上遗传算法评价函数计算伪代码

时$flow$和$shared$使用的是同一段共享内存。

3.3.2.2 并行排序，变异与交叉

由于遗传算法中最消耗时间的部分是评价部分，本设计中对其他部分的并行步骤采用较简单的算法。在评价部分结束后，需要对所有的染色体按照效用函数的大小降序排序，本文采用GPU上的odd-even算法进行排序操作，odd-even算法的计算过程如下图所示： 如图所示，奇偶排序算法每次两两比较数组中的值的大小，然后将较小的那个值交换到前面，奇数次的时候比较下标为$2k2k+1$的值,其中$k \in [0, POP/2]$,偶数次的时候比较下标为$2k-12k$的值，其中$k \in [1, POP/2]$，最终经过$POP/2$轮的奇偶比较，就可以得到排序好的数组，从奇偶排序的执行过程可以看出，每一轮比较中的$POP/2$次比较是是相互独立无关的，其具有天然的可并行性，而且其并行实现较简单。下面介绍染色体排序算法的GPU代码实现： 如图所示，sort函数一共需要执行$POP/2$轮比较，也就要调用kernel_sort共POP/2次，在kernel_sort中，一个线程负责两个数的比较与交换操作，在寻址数组之前，要判断当前的比较轮次的奇偶性，如果为奇数，则当前标号为$threadid$的线程要去比较标号为$2*threadid$和标号为$2*threadid+1$的目标值，如果当前标号为偶数值，则去比较下标为$2*threadid-1$和$2*threadid$的值，如果出现较小的值在后面，需要进行交换操作,交换操作在交换$objective$数组的同时，也要交换相应的染色体($chromsome$)数组,这样一轮比较达到并行度为$POP/2$,一个有$POP/2$个线程参与比较交换计算，因为要循环$POP/2$次才能保证排序完毕，所以算法总的计算复杂度为$O(POP)$。交叉过程分为父母选取和交叉计算两个kernel，父母选取过程随机并行地选取$\beta+\gamma$对父母，每个线程负责选取一对父母，并且将选取的父母下标记录到$father$和$monther$数组中,父母选取GPU计算示意图如下所示： 每个线程进行两次随机，$mother$标号只能在前α个精英染色体中选取，$father$表号的选取在前$\alpha+\beta$中选取，这样既能够给予精英染色体更大的交叉几率，也可以保证解的搜索空间变化足够大，避免陷入局部最优值。父母选取过程结束后，在GPU端记录了所选取的$mother$和$father$数组，$mother$和$father$数组将用于交叉部分的计算，交叉部分的并行粒度更高，其中每个基因点的计算都是并行执行的，如下图示：每个block负责一个新染色体的生成，通过之前的分析，一共需要生产$\beta+\gamma$个新的染色体，所以GPU端一共需要分配$\beta+\gamma$个block，其中每一个block中有—D—个线程来同时负责随机从$monther$和$father$的相应点位选择一个来作为新染色体相应点位的值，如前所说采用均匀交叉策略，选择父亲母亲遗传基因的概率都是50%,这样，这样block内部的并行度达到$|D|$,总的并行粒度达到$|D|*\Phi\beta+\gamma\Psi$。上图为CUDA上

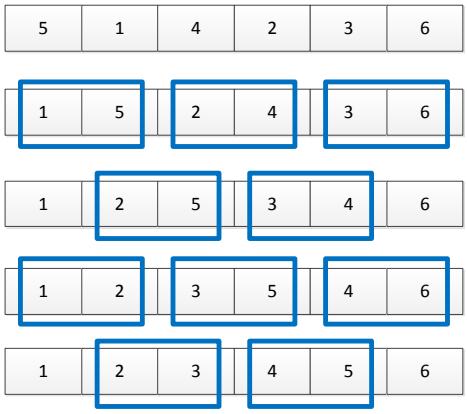

图 3-5 并行奇偶排序例子

```
1  void sort_kernel(float*objective,
2                   int**chromsome,
3                   int round)
4  {
5  int id=2*threadid;
6  if(round/2==0)
7    if(objective[id]<objective[id+1])
8    {
9           swap(objective[id],objective[id+1]);
10          swap(chromsome[id],chromsome[id+1]);
11   }
12 else
13   if(objective[id-1]>objective[id])
14   {
15          swap(objective[id-1],objective[id]);
16          swap(chromsome[id-1],chromsome[id]);
17   }
18 };
19 void sort(float*weight,
20                  int**chromsome,
21                  int round,
22                     int POP
23                     )
24 {
25 for(int i=0;i<POP/2;i++)
26     sort_kernel(objective,chromsome,i);
27 }
```

图 3-6 GPU并行奇偶排序伪代码

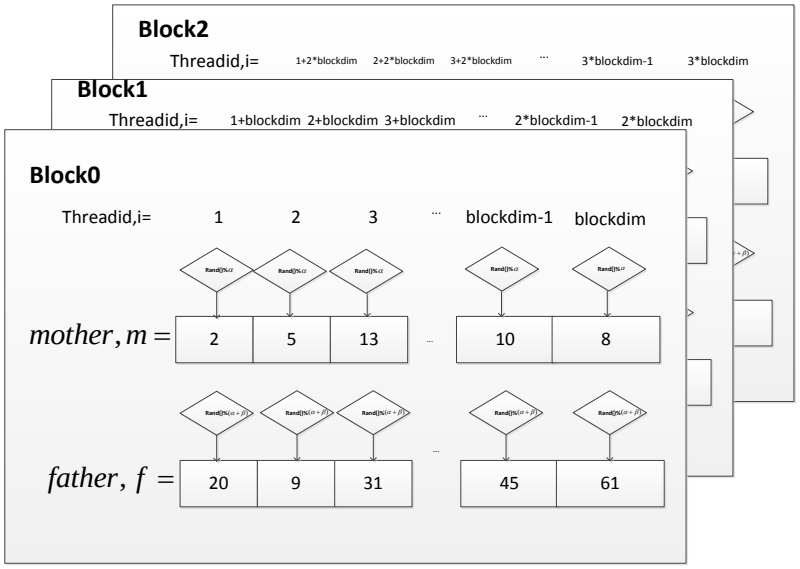

图 3-7 GPU并行父母选取过程

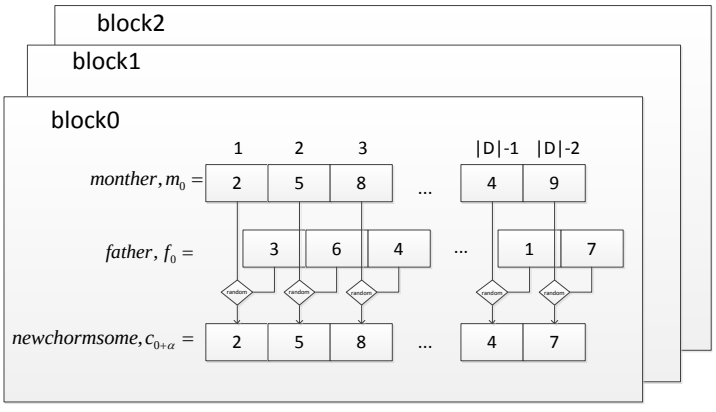

图 3-8 GPU并行均匀交叉过程

```
void cross_kernel(float*monther,
                  float*father,
                  int**chromsome,
            int**newchromsome
                  )
{
int mid=monther[blockid];
int fid=father[blockid];
int a=chromsome[mid][threadid];
int b=chromsome[fid][threadid];
int mark=rand()%100;
if(mark<50)
  newchromsome[blockid+alpha][threadid]=a;
else
  newchromsome[blockid+alpha][threadid]=b;
}
```

图 3-9 GPU并行均匀交叉代码

实现的均匀交叉伪代码，kernel输入为已经随机选好的$monther$和$father$数组，线程进入函数先找到block对应的母亲和父亲的染色体id，通过id寻址到相应的染色体编号，通过threadid寻址到当前的基因位置，然后随机产生一个0到100的数，此数如果小于50，则选择继承母亲的基因，反之，则继承父亲的基因，需要注意的是新生成的染色体数组（$newchormsome$数组），其前α个染色体直接复制排序后染色体集合的前α个染色体，也就是说前面的α个精英染色体直接保留到下一轮，所以代码中交叉产生的新染色体基因在填入$newchormsome$数组中时，需要偏移α个单位。最后对于变异部分，由之前的分析可知，变异步骤的计算量不大，并行粒度也不高，所以其并行算法实现不再赘述。

3.4 基于拉格朗日的优化算法设计

基于遗传算法的路由优化算法，虽然过程简单，但是其具有以下缺点,第一，需要事先计算大量备选路径，不能很好地适应网络的动态变化，一旦网络链路发生变化，又要重新计算备选路径。第二，遗传算法从开始到收敛需要大量的迭代次数，虽然经过GPU加速计算，但是由于收敛缓慢，任然需要大量的计算时间，迭代次数，很大程度上是决定于初始染色体集合的好坏，而随机选择路径的方式很难得到好的初始集合。第三，遗传算法容易陷入局部最优解，交叉范围小，和变异范围小都会使得算法提前陷入局部最优解，而如果交叉范围太大，变异范围太大，又会使得算法收敛缓慢，而且路由优化问题中每一个业务都有大量备选路径，问题的解空间很大，即使增大搜索范围也很难找到更优化的解。于是本节通

过分析采用基于拉格朗日乘子法的GPU并行方法来求解路由优化问题，实验显示基于拉格朗日。。。。。。。。。。

3.4.1 问题建模

路由优化问题就是为每一个业务选取一条路径，以使得效用函数最小化。然而在MILP模型中，网络容量约束，把所有的路由变量联系到一起，因为这些变量的取值必修保证每一条链路(i,j)上占用的流量小于其容量$\beta \cdot c_{ij}$,正是由于存在链路容量约束，每个业务的路由选取才变得不相互独立，但是要利用GPU的并行特性，需要寻找独立计算的可能性，因此，本文采用拉格朗日松弛方法，把一个路由优化问题分解成一些单个业务的寻路问题，而这些单业务的寻路问题是相互独立的，很适合并行计算。将问题。。中的网络容量约束松弛进目标函数得到如下问题：

$$\begin{aligned} L(\lambda) = min \sum_{d \in D} \sum_{(i,j) \in E_a} w_{ij} x_{ij}^d bw_d + \\ \sum_{(i,j) \in E_a} \lambda_{ij} (\sum_{d \in D} (x_{ij}^d bw_d - \beta c_{ij}) \end{aligned} \tag{3-5}$$

其中 λ_{ij} 表示链路 (i,j)的拉格朗日乘子。这个表达式(3-5)还可以表示为:

$$L(\lambda) = min \sum_{d \in D} \sum_{(i,j) \in E_a} (w_{ij} + \lambda_{ij}) x_{ij}^d bw_d - \sum_{(i,j) \in E_a} \lambda_{ij} \beta c_{ij} \tag{3-6}$$

subject to:

$$\begin{aligned} \sum_{(i,j) \in E_a} x_{ij}^d - \sum_{(j,i) \in E_a} x_{ji}^d = \begin{cases} 1 & \text{if } i = s_d \\ -1 & \text{if } i = t_d \\ 0 & ohterwise \end{cases} \\ \forall i \in V_a, \forall d \in D \end{aligned} \tag{3-7}$$

拉格朗日子问题的目标函数中的$\sum_{(i,j) \in E_a} \lambda_{ij} \beta c_{ij}$这一项，不随着拉格朗日乘子的变化而变化，本文将其作为常数项而丢掉不讨论，丢掉$\sum_{(i,j) \in E_a} \lambda_{ij} \beta c_{ij}$这一项后，拉格朗日子问题的目标函数中只含有代价部分 $w_{ij} + \lambda_{ij}$和x_{ij}^d的乘积。注意到，$\sum_{(i,j) \in E_a} (w_{ij} + \lambda_{ij}) x_{ij}^d bw_d$表示业务$d$的总路由代价，因此，拉格朗日子问题的目标函数是最小化所有业务需求的路径路由代价总和，在这个子问题的约束中，没有一个约束同时包含有两个及以上的业务需求相关的变量，所有这个拉格朗日子问题可以被分解成一系列独立的最短路径问题（每个业务需求对于一个最短路问题），只是这些最短路问题的链路单位代价发生改变，链路代价变得和拉格朗日乘子 λ相关，也就是说给定一个拉格朗日乘子λ，我们可以通过并行地计算一系列的最短路径问题来解决这个拉格朗日子问题。因为把容量约束松弛进效用函

数中后，不会增加目标函数的值（¡0），$L(\lambda)$成为原问题最优目标函数值的下界，$z^* \geq L(\lambda)$，为了得到最紧的的下界值，我们要解决以下这个优化问题：

$$L^*(\lambda^*) = maximize_{\lambda} L(\lambda) \tag{3-8}$$

subject to: (3-7)

以上的这个优化问题也被称为原来路由优化问题的对偶问题（），其中λ表示最优拉格朗日乘子，为了得到最优乘子λ*,可以使用次梯度优化算法来解决，次梯度优化计算时，第一次先初始化乘子 λ^0,然后通过以下过程进行迭代求解：

$$\lambda_{ij}^{(k+1)} = \lambda_{\mathbf{ij}}{}^{(k)} + \theta_k g^{(k)} = \lambda_{\mathbf{ij}}{}^{(k)} + \theta_k [(\sum_{d \in D} x_{ij}^d - c_{ij})]^+ \tag{3-9}$$

其中， $\lambda_{ij}^{(k)}$表示第k次迭代的对应于边(i,j)的拉格朗日乘子， $g^{(k)}$是$L(\lambda)$对λ^k的任意一种次梯度，θ_k 表示第k次的迭代的步长，标记$[\alpha]^+$表示α中符号为正的部分，也就说$[\alpha]^+ = max(\alpha, 0)$，从表达式可以看出来如果链路$(i,j)$上的流量总和超过链路$(i,j)$上的容量，链路$(i,j)$上的$\lambda_{ij}^k$拉格朗日乘子会增加，也就是表示一些业务流量需要从链路(i,j)上移除，另外，为了避免产生负权重的链路代价，当链路容量大于其上的流量时，我们并不去减小此链路(i,j)上的λ_{ij}^k。根据以上讨论，我们给出基于拉格朗日乘子法的并行路由优化算法的框架，如图所示，LR-PTEA主要包括以下步骤：步骤一，为$G_a(V_a, E_a)$初始化链路权重。步骤二，计算所有业务的最短路径，其中路径计算任务被分配到GPU进行并行计算。进一步，为了充分利用GPU的大规模计算能力，一种基于GPU的并行最短路算法被用来为每个业务计算路径。步骤三，为了从当前计算出来的路径中得到更好的目标函数值，对步骤二中计算出来的路径进行调整。步骤四，更新链路权重，更新完毕后，如果停止条件不满足，则回到步骤二，进入下一轮迭代。LR-PTEA 如果在连续β次成功的迭代后依然不能找到更优的全局解，则停止算法过程。

3.4.2 基于GPU的并行路由计算

在每次迭代中，LR-PROA为每个业务 $d \in D$在图 $G_a(V_a, E_a)$中计算最短路径，显然，丢掉链路容量约束后，不同业务的最短路径计算可以独立在GPU上并行执行，但是，最短路算法的逻辑对于GPU来说太过复杂，GPU最初是被设计来做大规模的数值计算问题，其只实用于逻辑比较简单，但是数值计算量较大的任务，所以在GPU上直接开辟一个线程来计算一个业务的路径，不仅仅在计算上是低效的，而且这样的并行粒度也不能充分利用GPU的大规模并行能力。为了充分提高最短路径的计算速度，LR-PROA 对最短路径算法进行并行化设计。

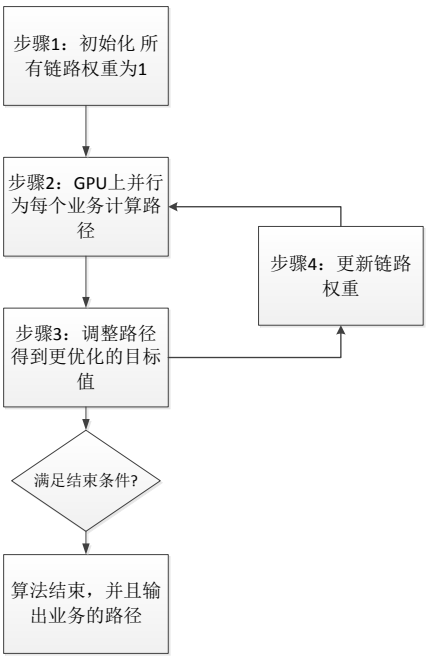

图 3-10 LR-PTEA算法流程图

文章[]提出一种Dijkstra最短路径算法在GPU上的并行实现，但是从算法结构上分析，Dijkstra最短路径算法并不适应于并行算法的设计，所以Dijkstra最短路径算法在GPU上的实现不能得到很好的加速效果，为了期望得到更好的加速效果，LR-PROA选择Bellman-Ford最短路算法来进行并行实现，Bellman-Ford最短路算法逐步地减小距离标记 $Dist[v], v \in V$，直达其收敛与真实的最短距离。Bellman-Ford算法过程如上图，其中 $Dist[v]$表示距离起点s到v的最短路径距离,$Pre[v]$表示点s到点v的最短路径上v的前驱节点，在初始化好了所以节点的距离数组和前驱节点数组之后，Bellman-Ford算法最多迭代 $|V|$次，每一次迭代算法松弛一次图$G(V,E)$中的所有的边（第10-11行）。Bellman-Ford算法的算法复杂度为$(|V| \cdot |E|)$，他的复杂度高于Dijkstra最短路径算法的复杂度，但是，因为Bellman-Ford算法每次松弛边的操作都是独立无关的，通过为每一条边的松弛操作分配一个独立的线程执行，Bellman-For算法很容易在GPU上实现并行化。 上图中显示了LR-PROA的最短路径计算的并行实现框架。首先，将业务量需求根据业务的源节点将业务分配成不同的组，使得每一组内的业务的源节点相同。我们假设第i个组的源节点为s_i。然后，为了高速的计算最短路径，每一组的最短路径计算使用m个GPU线程的并行bellman-ford算法进行计算，如上图所示，线程$T_{i,j}$负责为对应于源点s的

Algorithm 2 Bellman最短路算法

Require: 网络拓扑:$G(V,E)$; 源点:s;

Ensure: 从s开始到其他点的路径集合P;

1: **for** each node $v \in V$ **do**
2: $Dist[v] \leftarrow \infty$
3: $Pre[v] \leftarrow$ NIL
4: **end for**
5: $Dist[s] \leftarrow 0$
6: $Mark \leftarrow 1$
7: **while** $Mark > 0$ **do**
8: $Mark \leftarrow 0$
9: **for** each link $(u,v) \in E$ **do**
10: **if** $Dist[v] > Dist[u] + w_{uv}$ **then**
11: $Dist[v] \leftarrow Dist[u] + w_{uv}$
12: $Pre[v] \leftarrow u$
13: $Mark = 1$
14: **end if**
15: **end for**
16: **end while**
17: 根据前驱数组Pre,重新构建最短路集合，输出路径到集合P **return** P

链路e_j进行松弛操作。因此总的并行执行的线程数是$m \times k$,其中m和k分别表示链路数目和组的数目。在CUDA编程模型中,kernel在执行在一系列的block中，一个block中包含一系列并行执行的线程，在本文的最短路算法并行实现中，每个block内部的线程用于松弛同一条链路(i,j)的不同源节点情况，比如，在图中，集合$\{T_{1j}, T_{2j}, \cdots, T_{ij}, \cdots, T$在block$j$上执行，其中$T_{ij}$为对应源节点为$s_i$的链路$e_i$执行松弛操作，其中，我们设链路$e_i$的头节点和尾节点分别为$h_i$和$t_i$，可以看到，当这次迭代存在链路更新，那么标记$Mark$会被设置成一，这是为了优化算法的迭代次数，当某次迭代结束$Mark = 0$则表示这次迭代没有边进行了更新操作，说明Bellman算法已经提前结束，实验证明这一个优化可以大大地减小Bellman算法的迭代运行次数。最短路径算法CUDA实现算法伪代码如下所示。需要注意的是由于线程在GPU上是独立执行的，在更新节点的距离标记和前驱标记的时候会出现同步问题，假设线程T_1为链路(x,v)执行松弛操作，而线程T_2为链路(y,v)执行松弛操作，假设两个线程更新点v的距离标记和更新前驱标记的顺序如[]所示，如果$Dist[y] + w(y,v) < Dist[x] + w(x,v)$,那

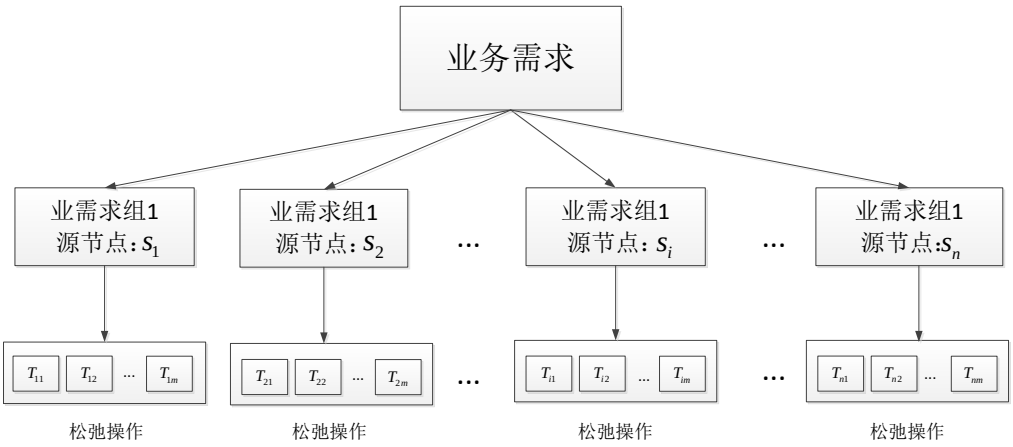

图 3-11 并行业务计算框架

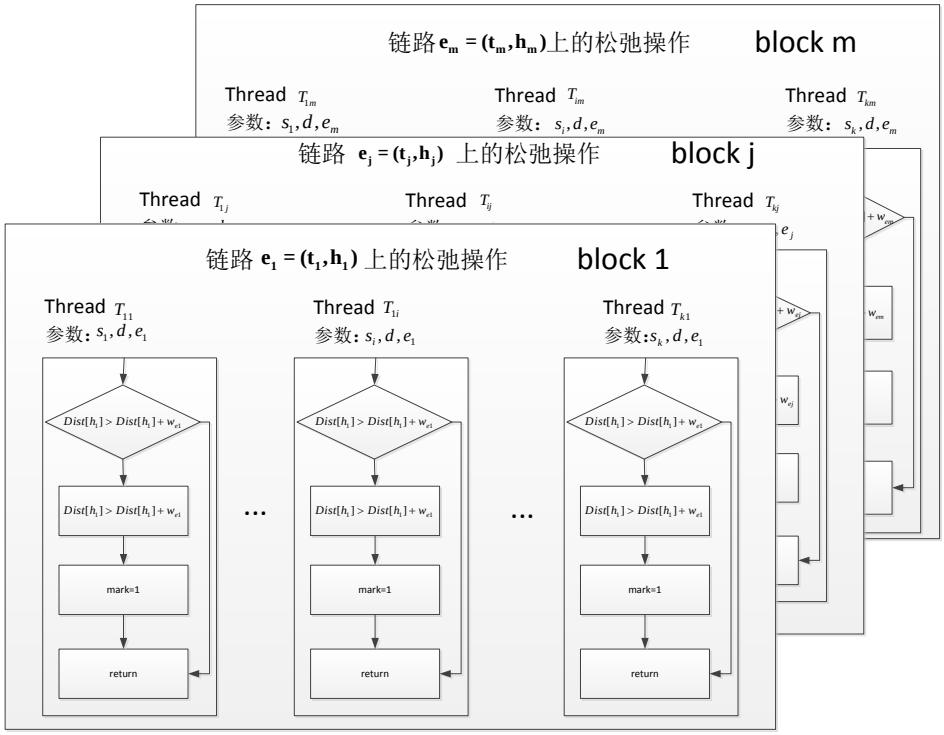

图 3-12 GPU上Bellman算法的实现

么$Dist[v]$被更新为$Dist[y]+w(y,v)$,但是由于更新的顺序发生交叉,节点v的前驱节点被更新成了x,而不是真正正确的前驱节点y。为了避免这个同步问题，算法设计中使用两个kernel，一个用来更新距离标号，一个用来更新前驱节点。

3.4.3 链路权重更新

3.4.3.1 权重更新步长

在LR-PROA算法中，在第$(k+1)$次迭代，链路(i,j)的权重被更新为$w_{ij}^{k}+\lambda_{ij}^{k+1}$,其中$\lambda_{ij}^{k+1}$被更新为：

$$\lambda_{ij}^{k+1}=\lambda_{ij}^{k}+\theta_k[(\sum_{d\in D}x_{ij}^{d}bw_d-\beta c_{ij})]^{+}. \tag{3-10}$$

为了保证收敛性，第k次迭代的更新步长(θ_k)可以被设置为 $\frac{1}{k}$[]。然而，通过仿真发现，当(θ_k)被设置为 $\frac{1}{k}$时，其收敛缓慢。让我们考虑图（）的例子，其中链路(i,j)上标记分别表示链路权重和链路上剩余的容量大小。假设现在有两个业务需求（d_1和d_2）其源和目的节点都是A和D，且每一个业务的流量大小都是4个单位。为了展示这个迭代过程，我们把算法前5次的迭代结果表示在表()中，从表中可以看到，业务计算出的最短路径一直在$A-B-D$和$A-C-D$之间徘徊。算法必修等到$A-B-D$和$A-C-D$的两条路的权重相等时才能停止，只有这样这两个业务才可能分离，其中一个选择$A-B-D$，而另一个选择$A-C-D$。但是，正如表中所示这需要大量的迭代才能达到。 另外一种常用的步长选择是：

$$\theta_k=\frac{\rho[UB-L(\lambda^k)]}{||\mathbf{Ax^k}-\mathbf{b}||^2} \tag{3-11}$$

其中UB是最优化目标函数的上界，ρ是一个取值范围为0到2的常数，$\mathbf{A}$ 和 $\mathbf{b}$分别是链路相关矩阵和链路上的剩余容量向量。但是从实验中发现设置这种步长迭代效果也不令人满意，因此，本设计采用一种简答但是有效的链路权重更新步长，假设θ_k^{ij}为第$k$th次迭代时链路(i,j)上的需更新的步长，那么θ_k^{ij}为：

$$\theta_k^{ij}=\frac{1}{|\beta c_{ij}-\sum\limits_{d\in D}x_{ij}^{d}bw_d|} \tag{3-12}$$

从（）（），我们可以看到，如果一条链路上承载的流量大小超过了这条链路上的容量大小，那么这条链路上的权重在下一次迭代之前就会增加1,对于其他的流量

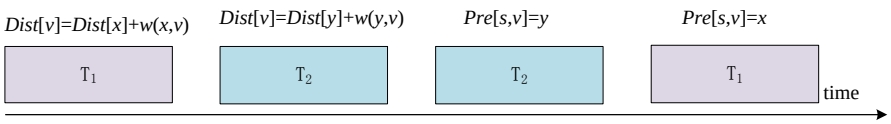

图 3-13 同步问题的例子

Algorithm 3 并行最短路计算

Require: 业务需求集合D; 链路集合E;

Ensure: 业务需求的最短路径结合P

1: 将业务的源节点加入到集合S中
2: $Mark \leftarrow 1$
3: **while** $Mark > 0$ **do**
4: $\quad Mark \leftarrow 0$
5: $\quad$ 发射 kernel_distance_update(S, E, $Dist$)
6: **end while**
7: 发射 kernel_predecessor_update(S, E, $Dist$, Pre)
8: reconstruct the shortest paths for the traffic demands based on predecessor information record in Pre, and put the paths to set P **return** P

Algorithm 4 kernel_distance_update(S, E, $Dist$)

1: $bid \leftarrow$ block ID
2: $tid \leftarrow$ thread ID
3: 将 (bid, tid) 映射到 id sid
4: $s \leftarrow S[sid]$
5: $e \leftarrow E[bid]$
6: **if** $Dist[s][e.tail] + e.weight < Dist[s][e.head]$ **then**
7: $\quad Mark \leftarrow 1$
8: $\quad Dist[s][e.head] \leftarrow Dist[s][e.tail] + e.weight$
9: **end ifreturn**

满足约束的链路吗，其权重不会改变，（）（）可以看到，如果使用（）的步长更新方法，LR-PROA仅仅只需要一次迭代就能够得到例子中（）最优的权重，实验表明，这种粗粒度的更新操作大大的减小了算法的收敛迭代次数，从而大大缩短算法运行时间。

3.4.3.2 随机更新策略

拉格朗日松弛法将原问题分解成了一个个独立的最短路径问题，这样使得算法可以并行化进行设计，但是由于个个子问题独立分离，使得每个问题再求最短路径时都是贪心的，这样可能会使得大量业务抢占同一批链路，造成拥塞，一旦拥塞，链路的权重增加，又会使得大量的业务放弃这一批链路，去抢占其他链路，而这些链路由于权重过分增加，造成没有业务去选择他们，使得链路利用出现浪

Algorithm 5 kernel_predecessor_update(S, E, $Dist$,Pre)

1: $bid \leftarrow$ block ID
2: $tid \leftarrow$ thread ID
3: 将 (bid, tid) 映射到 id sid
4: $s \leftarrow S[sid]$
5: $e \leftarrow E[bid]$
6: **if** $Dist[s][e.tail] + e.weight = Dist[s][e.head]$ **then**
7: $\quad Pre[s][e.head] = e.tail$
8: **end ifreturn**

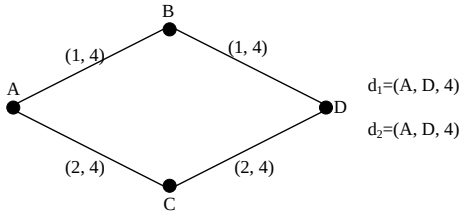

图 3-14 An illustrative example for link weights update

费，甚至会造成其他链路发生拥塞，这样出现恶性循环，最终会使得算法提前收敛到局部最优解，另外，由上一节的分析为了最求收敛快速，我们简单地将补偿设置为$\frac{1}{|\beta c_{ij} - \sum_{d \in D} x_{ij}^{d} bw_{d}|}$，就是对把每一个超过容量约束的边简单地增加一，这样粗粒度的增加，可能会加重上面讨论的拥塞循环。 如上图所示，假设有存在两个业务d_1和d_2,其中d_1的源节点为A,目的节点为G,其流量大小为5;d_2的源节点为B,目的节点为G，其流量大小为5，开始时两个都分别贪心计算最短路径，d_1选择路径$A-C-E-G$，d_2选择路径$B-C-E-G$,这样的话，链路$C-E$和$E-G$出现拥塞，表（）展示了这个迭代过程,图（）中最优的选择是让其中一个业务经过边$C-E-G$进行中继,另一个业务经过边$D-F-G$进行中继。但是的迭代过程始终无法使得两条链路发生分离，图中的链路权重始终无法达到最优条件，这是

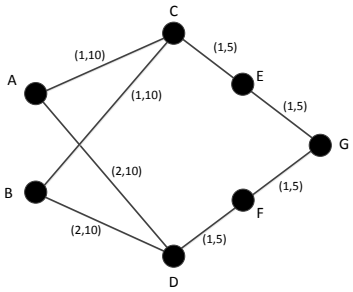

图 3-15 An illustrative example for link weights update

Iteration number	Calculated paths for the two demands	Path weights	θ_k
0	A-B-D A-B-D	4(1+1) 4(1+1)	1
1	A-C-D A-C-D	4(2+2) 4(2+2)	1
2	A-B-D A-B-D	4(1+4+1+4) 4(1+4+1+4)	0.5
3	A-C-D A-C-D	4(2+4+2+4) 4(2+4+2+4)	0.33
4	A-B-D A-B-D	4(1+6+1+6) 4(1+6+1+6)	0.25
5	A-C-D A-C-D	4(2+5.33+2+5.33) 4(2+5.33+2+5.33)	0.2

表 3-1 The illustration for the Laglanrian iteration process

因为算法的权重迭代增加粒度太大，由于链路$C-E-G$和链路$D-F-G$每次超限都会为路径的总权重增加2个单位，假设每次迭代时链路$C-E-G$和链路$D-F-G$每次迭代时一共只贡献1个单位的权重增加，那么算法只需要一次迭代就能达到最优条件，此时链路$C-E-G$由于超限，权重一共增加1个单位，那么路径$A-C-E-G$和路径$A-D-F-G$权重相等都为4，同样，路径$B-C-E-G$和路径$B-D-F-G$的权重也相等了，这样两个业务才会分离开（比如业务d_1选择链路$A-C-E-G$,业务d_2选择链路$B-D-F-G$）。但是在算法设计时，我们难以分辨哪些链路的组合会引起业务出现这种徘徊情况，为了解决链路增加粒度过大的情况，在本设计中我们采用随机选择执行更新的策略，也就是对一条流量超过容量约束的边(i,j)，我们以概率ϕ来对他进行权重更新，每条边的权重增加粒度依然为1个单位，假设$\phi=0.5$，这种方法在一定概率上保证图（）中的例子可以在一次迭代中收敛。实际实验中发现这种更新策略能够保证算法得到较优解的同时，保证收敛速度较快。

3.4.4 路径调整

注意到，在最优的权重代价下，LR-PROA 求解到的路径集合解对于拉格朗日对偶问题的优化解（eq）,但是他不是原来路由优化问题的优化解（Eqs）。作为一个例子，考察图()中的情况

3.4.5 仿真实验分析

Iteration number	Calculated paths for the two demands	Path weights
0	A-C-E-G B-C-E-G	3(1+1+1) 3(1+1+1)
1	A-D-F-G B-D-F-G	3(2+1+1) 3(2+1+1)
2	A-C-E-G B-C-E-G	3(1+2+2) 3(1+2+2)
3	A-D-F-G B-D-F-G	3(2+2+2) 3(2+2+2)
4	A-C-E-G B-C-E-G	3(1+3+3) 3(1+3+3)
5	A-D-F-G B-D-F-G	3(2+3+3) 3(2+3+3)
6	... ...	... ...

表 3-2 The illustration for the Laglanrian iteration process

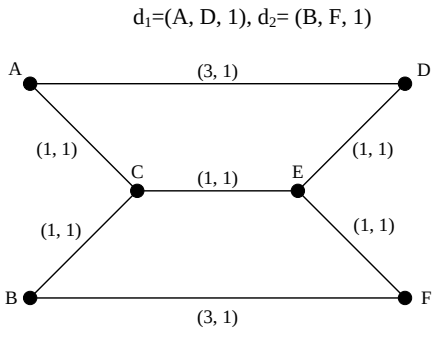

图 3-16 An illustrative example for link weights update

第四章 全文总结与展望

4.1 全文总结

本文以时域积分方程方法为研究背景，主要对求解时域积分方程的时间步进算法以及两层平面波快速算法进行了研究。

……

4.2 后续工作展望

时域积分方程方法的研究近几年发展迅速，在本文研究工作的基础上，仍有以下方向值得进一步研究：

……

致 谢

在攻读博士学位期间，首先衷心感谢我的导师 XXX 教授，……
……

参考文献

[1] 王浩刚, 聂在平. 三维矢量散射积分方程中奇异性分析[J]. 电子学报, 1999, 27(12):68–71

[2] X. F. Liu, B. Z. Wang, W. Shao. A marching-on-in-order scheme for exact attenuation constant extraction of lossy transmission lines[C]. China-Japan Joint Microwave Conference Proceedings, Chengdu, 2006, 527–529

[3] 竺可桢. 物理学[M]. 北京: 科学出版社, 1973, 56–60

[4] 陈念永. 毫米波细胞生物效应及抗肿瘤研究[D]. 成都: 电子科技大学, 2001, 50–60

[5] 顾春. 牢牢把握稳中求进的总基调[N]. 人民日报, 2012年3月31日

[6] 冯西桥. 核反应堆压力容器的LBB分析[R]. 北京: 清华大学核能技术设计研究院, 1997年6月25日

[7] 肖珍新. 一种新型排渣阀调节降温装置[P]. 中国, 实用新型专利, ZL201120085830.0, 2012年4月25日

[8] 中华人民共和国国家技术监督局. GB3100-3102. 中华人民共和国国家标准–量与单位[S]. 北京: 中国标准出版社, 1994年11月1日

[9] M. Clerc. Discrete particle swarm optimization: a fuzzy combinatorial box[EB/OL]. `http://clere.maurice.free.fr/pso/Fuzzy_Discrere_PSO/Fuzzy_DPSO.htm`, July 16, 2010

攻硕期间取得的研究成果

[1] J.-Y. Li, Y.-W. Zhao, Z.-P. Nie. New Memory Method of Impedance Elements for Marching-on-in-Time Solution of Time-Domain Integral Equation[J]. Electromagnetics, 2010, 30(5):448–462

[2] 张三, 李四. 时间步进算法中阻抗矩阵的高效存储新方法[J]. 电波科学学报, 2010, 25(4):624–631

[3] 张三, 李四. 时域磁场积分方程时间步进算法稳定性研究[J]. 物理学报, 2013, 62(9):090206–1–090206–6

[4] 张三，李四.时域磁场积分方程时间步进算法后时稳定性研究.电子科技大学学报[J]（已录用，待刊）

[5] S. Zhang. Parameters Discussion in Two-Level Plane Wave Time-Domain Algorithm[C]. 2012 IEEE International Workshop on Electromagnetics, Chengdu, 2012, 38–39

[6] 张三, 李四. 时域积分方程时间步进算法研究[C]. 电子科技大学电子科学技术研究院第四届学术交流会, 成都, 2008, 164–168

[7] 张三（4）.人工介质雷达罩技术研究.国防科技进步二等奖，2008 年

[8] XXX，XXX，XXX，XXX，王升. XXX的陶瓷研究. 四川省科技进步三等奖，2003年12月