福州大学

本科生毕业设计（论文）

题　　目：超大规模集成电路下高效层分配算法研究

姓　　名：吴逸凡

学　　号：041903101

学　　院：计算机与大数据学院

专　　业：计算机科学与技术

年　　级：2019 级

校内指导教师：_____（签名）

校外指导教师：_____（签名）

2023 年 05 月 20 日

福州大学本科生毕业设计（论文）诚信承诺书

学生姓名	吴逸凡	年　级	2019 级	学　号	041803101
所在学院	计算机与大数据学院			所学专业	计算机科学与技术
毕业设计（论文）题目	中文：超大规模集成电路下高效层分配算法研究				
	外文：Research on Efficient Layer Assignment Algorithm for VLSI				

学生承诺

我承诺在毕业设计（论文）过程中遵守学校有关规定，恪守学术规范，未存在买卖、代写、作假等行为。在本人的毕业设计（论文）中未剽窃、抄袭他人的学术观点、思想和成果，未篡改实验数据。如有违规行为发生我愿承担一切责任，接受学校的处理。

学生（签名）：

年　　月　　日

指导教师承诺

我承诺在指导学生毕业设计（论文）过程中遵守学校有关规定，恪守学术规范，经过本人认真的核查，该同学未存在买卖、代写、作假等行为，毕业设计（论文）中未发现有剽窃、抄袭他人的学术观点、思想和成果的现象，未发现篡改实验数据。

指导教师（签名）：

年　　月　　日

超大规模集成电路下高效层分配算法研究

摘　　要

超大规模集成电路的发展中，多层布线结构由于比单金属层结构在单位面积上具有更大布线空间而更适应高密度布线的需求。在该背景下，需要将电路中的电子元件按照指定方式连接并映射到不同的电路层上。而层分配算法的目标则是在满足特定设计约束条件的情况下，将逻辑元素分配到合理的电路层上。

层分配方案对集成电路的整体性能可以起到很大的影响，也因此吸引了不少研究者的目光。于是，3 维布线器和 2.5 维布线器被相继提出。其中 3 维布线器直接在多层布线空间上计算布线方案，导致算法复杂度过高。为缩短运行时间，2.5 维布线器先将不同层的布线集中到一层上，在 2 维空间上快速得到一个布线方案，之后再通过层分配算法将该 2 维布线方案拓展成 3 维详细布线方案，显著提高了算法效率，是当前主流布线器。

现有关于层分配算法的研究，大多集中在优化布线方案的时延、拥塞和通孔数等指标。其中，在 2.5 维布线器下基于动态规划方法的层分配算法已经可以得到效果很好的详细布线方案。然而，在处理复杂数据时，许多算法的执行效率较低，难以适应日趋复杂化的集成电路物理设计要求。优化层分配算法效率可以更好地解决多层布线困难的问题。

本文旨在改进层分配算法的算法效率，分三个步骤展开研究：

（1）检测当前效果较好的层分配算法中最耗时的瓶颈代码。本文通过对代码各个部分运行时间的统计找出影响算法效率的瓶颈代码，通过优化瓶颈代码执行效率减少了层分配算法执行时间。

（2）在代码层面优化层分配算法的函数调用逻辑。由于线网的单发信器与多接收器结构能很容易地与树形结构对应，故在基于动态规划的层分配算法中往往会使用大型递归函数进行树形动规。本文通过将递归调用的函数逻辑转写为非递归方式而改进了算法执行效率。

（3）引入并行算法。树形动规的特点决定了其可以进行多线程并行。本文使用并行算法改进层分配算法，并使用线程池减少开销，从而进一步提高算法效率。

实验结果表明，本文能在不降低层分配算法效果的前提下，有效优化层分配算法效率。

关键词：层分配算法，大规模集成电路，并行算法，内存调度

Research on Efficient Layer Assignment Algorithm for VLSI

Abstract

In the development of very large scale integration circuits, multilayer routing structures gradually become mainstream due to their larger wiring density compared to single-metal layer structures. In this context, it is necessary to map the electronic components to different circuit layers. The goal of layer assignment algorithms is to assign logic elements to reasonable circuit layers while satisfying specific design constraints.

Layer assignment schemes can have a significant impact on the performance of integrated circuits, which has attracted the attention of researchers. Therefore, 3D and 2.5D routers have been proposed. Among them, the 3D wiring tool calculates the wiring scheme on the multi-layer wiring space, leading to high algorithm complexity. To accelerate that, the 2.5D router first designs the wiring on one layer quickly, then expands the 2D wiring scheme into a detailed 3D wiring scheme through layer assignment algorithms. This improves the efficiency, and is the mainstream router.

Existing research mostly focuses on optimizing indicators such as delay, congestion, and via count in wiring schemes. Among them, the layer assignment algorithm based on dynamic programming under the 2.5D router can already obtain a detailed wiring scheme with good effect. However, in dealing with complex data, the execution efficiency of many algorithms is low and it is difficult to adapt to the increasingly complex of integrated circuits physical design. Optimizing the efficiency of layer assignment algorithms can better solve the multilayer routing problems.

This paper aims to improve the algorithm efficiency of layer assignment algorithms and carries out research in three steps:

(1) Detect the most time-consuming bottleneck code in the layer assignment algorithm with better effect. This paper finds the bottleneck code that affects the algorithm efficiency by statistical the running time of each part of the code, and reduces the execution time of the layer assignment algorithm by optimizing the efficiency of the bottleneck code.

(2) Optimize the function call logic of the layer assignment algorithm. Since the single-transmitter and multiple-receiver structure of the wire network can easily correspond to the tree structure, large recursive functions are often used for tree-based dynamic programming in the layer assignment algorithm based on dynamic programming. This paper improves the algorithm execution efficiency by rewriting the logic of the recursive function call to a non-recursive manner.

(3) Introduce parallel algorithms. The characteristics of tree-based dynamic programming determine that it can be multi-threaded. This paper uses parallel algorithms to improve the layer assignment algorithm and uses thread pools to reduce overhead, thereby further improving the algorithm efficiency.

Experimental results show that this paper can effectively optimize the algorithm efficiency of layer assignment algorithms without reducing their effectiveness.

Keywords: layer assignment, VLSI, parallel algorithm, Memory scheduling

目　　录

第一章 绪论

1.1 研究背景及意义

本文所研究的层分配算法是集成电路物理设计过程中决定逻辑元素，主要是发信器、接收器和布线，以何种方式分配到集成电路的不同金属层上的算法。在传统单金属层下的布线已经走到瓶颈，为保证集成电路性能，以三维集成技术制造的三维集成电路应运而生[1]。多层布线器于是承担起将发信器、接收器和连接它们的线网分配到不同金属层的责任。

随着多层布线器所得到的布线方案的不断改善，不少研究者也关注起如何继续优化布线器本身的算法效率[2]。大规模集成电路的规模渐长，使得布线器需要处理的线网数量显著增加，于是布线器执行层分配算法的时间逐渐成为影响高效布线方案设计的重要因素。多核处理器的发展为提高多层布线器的布线效率提供了一个方向，因此如何利用并行算法设计一套适应多核处理器环境的层分配算法成为重要问题。线网的并行处理又可以继续分为线网内部的并行和线网之间的并行。本文的实现高效层分配算法的策略之一，就是进行线网内部的并行处理。

通过加快层分配算法的执行速度，能实现更高的布线器布线效率，以进一步缩短芯片物理设计的周期。同时，更快的执行速度，也使得层分配算法能在相同运行时间下使用更加复杂的布线方案优化策略。

1.2 研究现状

在超大规模集成电路的设计中，通常需要将电路中的电子元件按照指定方式连接并映射到不同的电路层上。为了高效完成集成电路的全局路由设计，通常会对集成电路建立网格图模型，在该模型下，集成电路被抽象成多个层和块。层分配算法的目标就是在满足特定的设计约束条件的情况下，将逻辑元素分配到合理的电路层的合理位置上。评价分配方案的常用指标有电路的通孔数、时延、布局、连接方式、功耗、可靠性等。层分配方案对集成电路的整体性能可以起到很大的影响，也是集成电路设计过程中重要的一环。

为适应复杂化的电路设计和高密度化的布线需求，传统的单金属层布线模式逐渐被多金属层布线模式替代，多层布线结构应运而生[3]。多层布线器又可继续分为 3 维（3-Dimensional， 3D）布线器和 2.5 维（2.5-Dimensional， 2D）布线器。3 维布线器布线器直接在整个多金属层的三维布线空间中进行布线，并得到 3 维的详细布线方案；2.5 维布线器会先在单金属层的二维布线空间上布置线网，之后利用层分配算法将二维空间上线网的各个逻辑器件及连接线路分配到不同的层上，从而得到 3 维的详细布线方案。层分配算法是布线器的核心算法，它能有效提高布线方案的质量[4]。以文献[5-6]为代表的 3 维布线器直接在复杂的多层布线区域中进行布线，虽然可以得到不错的布线方案，但其算法复杂

度太高，计算效率太低，而且也不利于进行进一步的优化。以文献[7-10]为代表的 2.5 维布线器先在 2 维平面上快速设计一个可行的线网布线方案，之后利用层分配算法将其中的发信器、接收器、连接线路等逻辑元素分配到不同层上，从而极大地减少了计算布线方案的时间开销。同时，这种方法也将布线问题进一步拆分成两个步骤，即得到 2 维布线方案的步骤，和从 2 维布线方案得出对应 3 维布线方案的步骤，这使得这两个步骤可以分别较为独立地进行优化。因此，2.5 维布线器成为目前主流的布线器。

在布线器的优化中，一般会关注布线器得到的布线方案是否满足各类约束，同时拥有尽量小的时延和通孔数，尽量短的布线长度，尽量同步的总线内部信号到达时序。在这当中，动态规划方法、线网排序方法和基于协商的方法等被广泛应用。近年也有研究者在布线器中引入机器学习的方法[11-13]，但由于训练数据不足，训练时间长等问题，目前尚未得到广泛应用。

在 2.5 维布线器中，层分配算法主要有基于整数线性规划的方法[14]和基于动态规划的方法[15-16]两大类。其中，基于整数线性规划得到的布线方案总体效果不错，但需要占用大量时间进行计算且难以改进效率；而基于动态规划得到的布线方案还不够优秀，但其拥有极佳的运行效率。

文献[17]首次将基于协商的层分配算法引入布线器中。在这个算法中，每个逻辑元素都有一个代表它的代理，这些代理之间可以进行协商，以决定最佳的层分配方案。在协商的过程中，代理之间可以交换信息，提出建议，并进行讨论，最终达成共识。文献[18]在基于协商的层分配算法中，提出了时延驱动层分配（Delay-driven Layer Assignment， DLA）算法，使其布线方案能在延迟、拥塞和通孔数之间取得良好平衡。文献[19]在其基础上优化了算法在不同阶段的侧重点，同时引入了新的拥塞评估函数，从而得到了更好的布线方案。

文献[20]提出了一种考虑总线时序匹配的多层分配算法，使层分配算法首次具备考虑总线时序匹配的能力。文献[21]设计了一种基于区域划分的并行策略和一种基于线网等效布线方案感知的通孔优化策略，避免算法在布线阶段出现布线资源冲突问题的同时，使算法能在多种线网等效布线方案中选出较优方案。文献[22]提出了一种基于多策略的时延驱动层分配算法，在保证不溢出的同时明显优化了布线方案的时延和通孔数这两个最重要的优化目标。

文献[2]提出了一个多线程、碰撞感知的全局布线工具。它在全局布线时会考虑到线路之间的碰撞，以避免出现布线冲突的情况。它采用了多线程技术，可以在短时间内对大规模的电路板进行全局布线。此外，它还使用了有限长度迷宫布线技术，可以在保证布线质量的前提下，大大缩短布线时间。文献[23]提出了一种时序驱动的递增式布线算法，采用逐步增量的方式进行布线，每次只布线一个逻辑单元，并在布线过程中动态调整时序约束，以保证布线结果满足时序约束。具体来说，该算法首先对电路进行初步的布线，然后根据初步布线结果计算出每个逻辑单元的斜率约束，接着对每个逻辑单元进行逐一布线，并在布线过程中动态调整相邻逻辑单元的斜率约束，以避免电压转换速率违规。

1.3 总结

综上所述，当前 2.5 维布线器中基于动态规划方法得到的布线方案已经具有很好的效果。但随着线网数量和线网复杂度的不断提高，算法生成布线方案所需的时间也越来越长。另一方面，在基于协商的算法框架下，一个线网往往需要进行多次层分配，这也进一步延长了算法运行时间。为适应日趋复杂化的集成电路物理设计要求，同时方便层分配算法的进一步优化，本文在文献[19]的基础上对层分配算法的效率进行优化。通过将该文献使用的树形动态规划以非递归方式实现，同时引入Taskflow库对动态规划过程进行多线程并行，从而提高了算法效率。

1.4 主要研究内容

本文以文献[19]所设计的一种经典的 2.5 维布线器上的层分配算法为基础，进行了以下研究：

（1）研究原算法处理不同数据时最耗费时间的函数或程序段。

（2）重组织算法函数调用逻辑以提高算法执行效率。

（3）引入 Taskflow 并行库，利用并行算法将层分配算法对每个线网的计算改为线网内部的多线程并行计算。

（4）通过调整线程数，分析不同输入数据和不同线程数下并行算法的表现。比较不同数据在不同线程数下的运行时间，得到适合不同数据的最佳线程数。

（5）分析不同数据下的线网结构，找到线网结构与最佳线程数之间的关系。引入线程数自适应算法，使层分配算法能自动根据线网复杂程度选择合适的线程数进行并行。

（6）使用单一线程池优化线程创建和释放的开销。

1.5 论文组织架构

本文一共有五章，具体内容安排如下：

第一章对论文整体进行了介绍。首先介绍了研究背景与意义，然后介绍了层分配算法的研究现状，接着对研究现状进行总结，指出现有研究的不足和本文的改进，最后介绍本文的主要研究内容和组织架构。

第二章对层分配问题进行具体阐述，介绍两个经典的层分配算法：DLA 层分配算法和 MiniDelay 层分配算法，本文将在 MiniDelay 算法基础上进行改进。同时建立层分配问题模型，并描述层分配算法中动态规划算法的使用。

第三章介绍了 Taskflow 并行框架的功能以及它的主要类和使用方法。Taskflow 框架使用任务流程图对任务执行顺序进行约束，可以使用户创建的任务在用户设定的流程下并行

执行，能满足不同场景的并行需求。

第四章将介绍本文为提高层分配算法效率而使用的三个方法：第 1 部分为引言，概括本文的主要工作内容；第 2 部分对 MiniDelay 层分配算法的运行时间进行分析；第 3 部分介绍了本文对算法函数调用逻辑进行的更改；第 4 部分介绍了本文如何引入并行算法；第 5 部分对本文使用的数据集进行分析；第 6 部分介绍了本文提出的自适应线程数的并行算法；第 7 部分介绍了本文提出的使用单一线程池的并行算法。

第五章将分 3 部分展示本文的实验内容：第 1 部分介绍了本文的实验环境；第 2 部分介绍了本文使用的数据集；第 3 部分展示了本文重组织算法函数调用逻辑、引入并行算法后，层分配算法的表现，并对实验结果进行分析。

最后是对全文的总结和展望，分析了本文所研究内容中存在的局限性，对未来进一步优化工作做出展望。

第二章 层分配算法研究

2.1 层分配问题描述

层分配算法是一个在 3 维布线空间下，将线网中的发信器、接收器和导线分配到不同布线层，并实现线网中元器件互连的算法。在全局布线和层分配问题中，一般会对集成电路建立网格图模型。一个k层布线区域可以被进一步分割成多个块，如图 2-1 所示，其中的每一个块可以进一步映射到图 2-2 的一个点上。

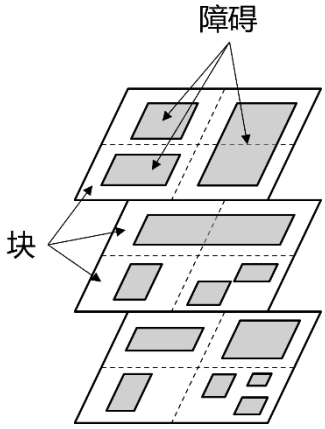

图 2-1 布线区模型[18]

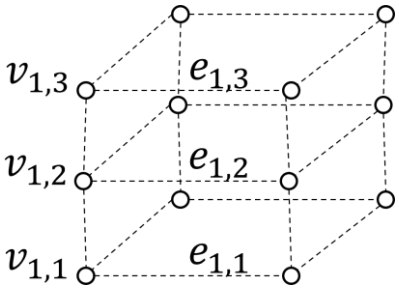

图 2-2 三维网格图模型[18]

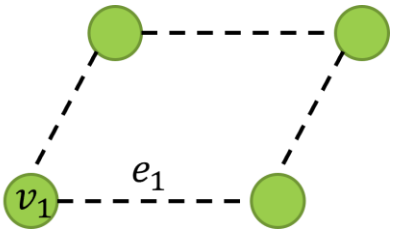

图 2-3 压缩二维网格图模型[18]

由于对图 2-1 中各个布线层的分割仅有水平分割线和垂直分割线，故其分割成的块也至多只有上下左右四个相邻块。使用通孔可以在不同层的块之间建立连接。通过把布线区模型中的块映射到三维网格图模型中的点上，把布线区模型中相邻块的边界映射到三维网

格图模型的边上，可以得到图 2-2 所示的三维网格图模型。该模型中的每个点至多与同层的 4 个点连接，与不同层的 2 个点连接。将 k 层的网格图模型继续压缩，可以得到如图 2-3 所示的压缩二维网格图模型。其中，图 2-2 中的$v_{1,1}$、$v_{1,2}$、$v_{1,3}$这 3 个点都对应图 2-3 中的v_1这 1 个点，类似地，图 2-2 中的$e_{1,3}$、$e_{1,2}$、$e_{1,3}$这 3 条边都对应图 2-3 中的e_1这 1 条边。

层分配问题的问题描述如下：给定一个三元组(G^k, G, S)，其中$G^k(V^k, E^k)$表示一个k层的三维网格图模型，$G(V, E)$表示与$G^k(V^k, E^k)$对应的压缩二维网格图模型，S表示在G上的一个整体布线方案。根据该三元组，求得S^k，即一个在G^k上的三维详细布线方案。具体来说，就是要将S上的每一个点v_i分配到G^k上的$v_{i,z}(1 \le z \le k)$上，同时将每一条边e_i分配到G^k上的$e_{i,z}(1 \le z \le k)$上。

由于布线轨道数量有限，图 2-2 中的每一条边所能容纳的导线数也是有限的。所以尽管不同线网都能将其二维布线方案的某一条边分配到三维网格图模型上的同一条边上，但并无法无限制的重复使用同一条边。图 2-4 展示了一个线网在二维网格图上的整体布线方案，图 2-5 展示了该线网在三维网格图上的一种详细布线方案。在图 2-4 展示的线网中，点v_1代表线网的发信器，点v_2和点v_3代表线网的接收器，边e_1，e_2，e_3和e_4分别为线网中的导线段。将该二维布线拓展到三维，可以得到如图 2-5 所示的三维布线，与图 2-4 对应的三维布线方案有很多，图 2-5 仅展示一种可能的分配方案。在层分配过程中，为连接不同层的线路，会使用通孔。在图 2-5 中，连接不同层的边即为通孔，该图所示的分配方案将产生 3 个通孔。本文讨论的集成电路布线中，同一金属层上的线路方向应当一致。

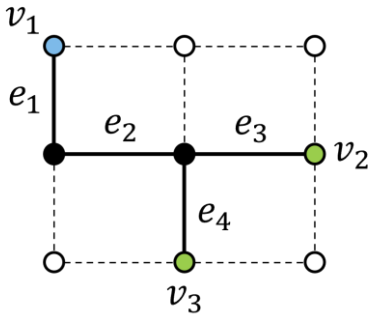

图 2-4 单线网的二维布线

3 维布线空间下，不同布线层上有不同障碍，也有不同的默认线宽和线间距。上层的导线通常具有更大的线宽和线间距，因此也具有更小的电阻。然而，每一层的布线空间是有限的，而且当走线密度增高时，耦合电容也会随之增高，从而对线网的连通时延产生负面影响。在低纳米节点下，通孔数和耦合效应对时序的影响很大，因此层分配算法需要充分考虑通孔数和耦合效应带来的影响。

由于制造工艺的限制，布线过程中使用的导线不能具有任意宽度，而只能挑选某些预定义的导线宽度。使用特殊预定义宽度而非默认宽度的导线称为非默认规则线（Non-Default-Rule， NDR）线。NDR 线包括宽线和平行线两种。宽线占用更多布线轨道的同时，也具有更小的电阻。平行线布线方法也具有类似效果。合理使用 NDR 线可以加快线网的

连通速度。故在层分配算法中，需要在满足拥塞约束的条件下，尽量多地使用 NDR 线，并优先将其用于时序关键线网上，来降低线网连通时延。

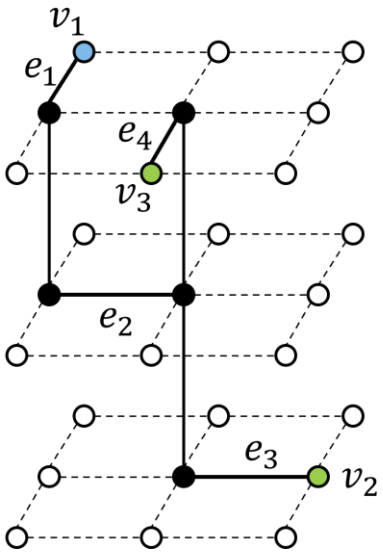

图 2-5 单线网的三维布线

2.2 DLA 层分配算法

本部分内容将描述文献[18]提出的 DLA 层分配算法，该算法在作为本文改进基础的文献[19]中也有使用。

如图 2-6 所示，DLA 算法可分为 3 个大阶段，分别是时延主导层分配阶段、基于协商的时延驱动层分配（Negotiation-based Delay-driven Layer Assignment， NDLA）阶段和后续优化阶段。

在时延主导层分配阶段，将为每个线网进行初步层分配，而不考虑拥塞约束。该阶段不允许使用平行线和 NDR 线，从而避免在这个早期阶段平行线和 NDR 线消耗额外的布线资源。由于忽略了拥塞约束，该阶段产生的层分配方案可能使 3D 网格图中的边出现不必要的溢出。如果一个线网通过了出现非必要溢出的网格边缘，就称这个线网为非法线网。

在 NDLA 阶段，将分配方案中的非法线网迭代进行层分配，以使分配方案满足拥塞约束。同时，该阶段将有效使用布线资源来减小线网时延。为了有效减小关键线网的时延，该阶段也将识别关键线网。这个阶段可以继续分为三个步骤：（1）计算每个线网的时延，并按时延非递增的顺序对线网排序。（2）对线网的导线段加权，按每个导线段的时序关键性为其分配不同的权重。这将有利于时序关键导线段分配到较低电阻层。（3）第三步是时延感知的拥塞减少。根据先前的层分配结果，将非法线网重新按排序顺序逐个分配。当重新分配完成后，再次检查拥塞约束。如果分配结果违法拥塞约束，则将每条溢出的 3D 网格图边缘的拥塞代价增加，然后开始新一轮 NDLA 迭代，直到分配结果满足拥塞约束。为

了加快 NDLA 收敛过程，当迭代次数超过用户设定的阈值时，第二步将被跳过。

在后续优化阶段，将在满足拥塞约束的条件下，对所有线网重新进行一次层分配，以进一步减小时延和通孔数。

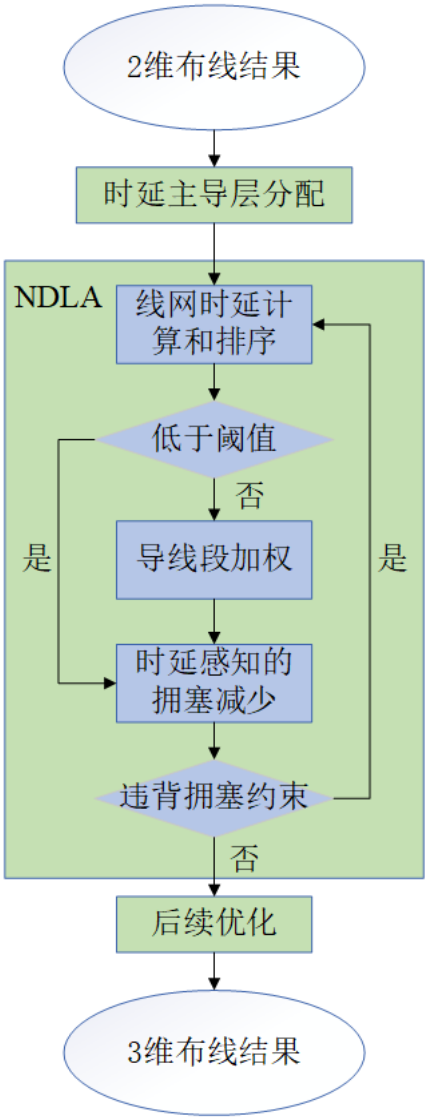

图 2-6 DLA 算法流程图

该算法通过考虑平行线和 NDR 线的使用，和耦合效应对时延的影响，有效优化了布线方案的时延，但它也存在一些不足：一是它对拥塞的评估是有限的，无法区分层分配方案中有无非必要的溢出；二是没有针对最大时延的优化，最大时延是限制芯片性能的关键因素；三是对时延关键线网的时延优化有限。这些不足将在文献[19]中被改进。

该算法的每个阶段都将使用动态规划方法对线网进行层分配，而其并没有针对动态规划运行时间的优化策略。

2.3 MiniDelay 层分配算法

本部分内容将描述作为本文基础的文献[19]所使用的基于协商的层分配算法，MiniDelay。

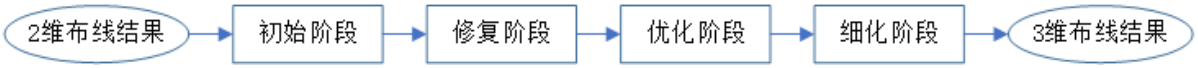

图 2-7 MiniDelay 算法流程图

图 2-7 展示了 MiniDelay 的算法流程，该算法共有 4 个阶段，分别为初始阶段、修复阶段、优化阶段和细化阶段。

在算法的初始阶段，会按顺序将各个线网分配到 3 维布线空间上。此阶段将为后续阶段构造一个初始层分配结果。该阶段不仅考虑时延，也同时考虑拥塞，这样能避免在后续阶段浪费时间。该阶段同时还会考虑通孔数。通孔时延是影响总时延的重要因素，因此减少通孔数能有效优化分配代价。要得到一个理想的初始层分配结果，需要综合考虑时延、拥塞和通孔数。但在初始阶段，该算法允许布线的拥塞代价超过拥塞限制。不过尽管如此，它依然能减轻后续阶段调整拥塞代价的负担。

在算法的修复阶段，会将上一阶段得到的布线方案中不满足拥塞约束的线网进行再分配。该阶段会使用协商方法，多次分配线网，直到所有线网都能满足拥塞约束。当一个线网的导线段被分配到一个布线轨道数已满的边上时，使用这条边的拥塞代价将会提高。NDR 线在此阶段被禁止使用，这是因为该阶段重分配的非法线网多集中于相对拥挤的布线区域，而并不代表这些线网是时序关键线网。将 NDR 线分配到这些线网上是不合适的。

在算法的优化阶段，将使用手术刀算法来优化时延。该阶段将重分配时序关键线网，并在时序关键线网中的时序关键线段使用 NDR 线来进一步优化时延。时序关键线网是经过前两个阶段的分配后，互连时延最长的那些线网。时序关键线段就是指时序关键线网中靠近发信器的导线段。为了挑选出时序关键线段，线段区别算法将根据线段与线网发信器之间的距离为每个线段评分，距离近的将得到高分，距离远的将得到低分。通过这种方法可以有效降低最大时延。同时，严格限制 NDR 线的使用，可以避免可布线性变差。

在算法的细化阶段，线网将按照时延从高到低的顺序排序。随后，每个线网都将被拆开重分配一次。该阶段的重分配将在保证满足拥塞约束的前提下进行，同时只有时延最长的 5%线网能够使用 NDR 线，其它线网只能使用默认宽度的导线。该阶段依然同时考虑时延、通孔数和拥塞约束，但会更侧重于优化通孔数。

该算法已经能得到不错的层分配方案，但该算法依然没有针对算法运行时间提出具体优化策略，使得该算法运行时间依然较长。

2.4 层分配算法中动态规划算法的使用

动态规划是一种算法思想，用于解决具有重叠子问题和最优子结构性质的问题。它将问题分解为子问题，通过保存子问题的解来避免重复计算，从而实现高效的求解。动态规划算法通常用于优化问题，例如最长公共子序列、最短路径和背包问题等。

动态规划算法的基本思想是将问题分解为更小的子问题，并使用子问题的解来求解原始问题。通过保存子问题的解，动态规划算法可以避免重复计算，从而提高算法的效率。动态规划算法通常有两种实现方式：自顶向下和自底向上。

自顶向下的实现方式称为记忆化搜索。在记忆化搜索中，算法首先检查是否已经计算了当前子问题的解，如果已经计算，则直接返回该解。否则，算法计算当前子问题的解，并将其保存在一个表格中，以便以后使用。

自底向上的实现方式称为迭代法。在迭代法中，算法首先计算最小的子问题的解，然后逐步计算规模更大的子问题的解，直到计算出原始问题的解。

在层分配问题中，一个线网由一个发信器和多个接收器组成，在 2 维网格图中，该线网可以看成是一个树形结构。如图 2-7 所示，点v_1为线网的发信器，点v_2、点v_3和点v_4为线网的接收器，边e_1、e_2、e_3、e_4和e_5为连接接收器与发信器的导线段。二维线网中的发信器可以映射为树形图的根节点，接收器可以映射为树形图的叶子节点，导线段可以映射为树形图中连接父节点和子节点的边。为了得到不同分配方案下线网的最小代价函数值，层分配算法往往需要在线网上使用动态规划算法，依照其树形结构，从叶子节点不断向上迭代计算。当计算到根节点时，就可以得到整个线网的最小代价函数值。

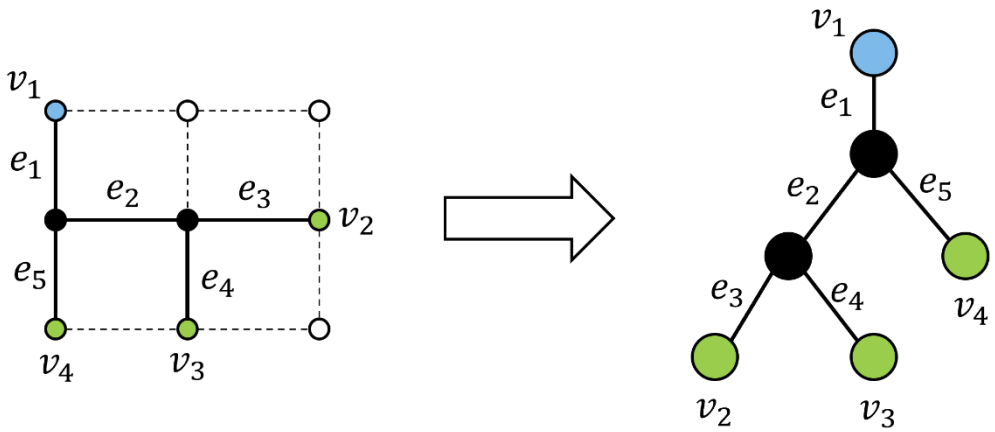

图 2-8 线网结构与其对应的树形结构

具体来说，层分配算法为了得到分配一棵线网树的最小代价值，需要先计算其所有子树的最小代价值。于是，该算法先从根节点开始向子节点方向搜索，搜索到叶子节点后，该算法会从叶子节点开始，计算该节点连向父节点的边，也即一条导线段分配到 3 维网格图中的不同层，使用不同导线类型的不同代价。接着，再根据父节点的所有子节点的分配

代价，计算出以父节点为根的子树在不同分配方案下的最小代价值。如此向根节点递归计算，从而得到整个线网在不同分配方案下的最小代价值。

表 2-1 层分配算法伪代码[18]

Algorithm: Layer Assignment

Input: net $N(V_N, E_N)$, pin set P_N^k, 3D grid graph G^k

```
1  Rec(v_i, N, P_N^k)
2  begin
3      foreach node v_j ∈ ch(v_j) do
4          if v_j is not visited then
5              Rec(v_j, N, P_N^k)
6          end
7      end
8      if v_i is a leaf then
9          foreach layer L do
10             wt ←Rem(e_i, L)
11             for j ← 0 to |wt| − 1 do
12                 R(v_{i,L}, t_j) ←LeafSol(v_{i,L}, t_j, P_N^k)
13                 S(v_i) = S(v_i) ∪ R(v_{i,L}, t_j)
14             end
15         end
16     end
17     else if v_i is a root then
18         S(v_i) ← EnumSol(v_i, x, P_N^k)
19         TopDownAssignment(S(v_i), G^k)
20     end
21     else
22         foreach layer L do
23             wt ←Rem(e_i, L)
24             for j ← 0 to |wt| − 1 do
25                 R(v_{i,L}, t_j) ←EnumSol(v_{i,L}, t_j, P_N^k)
26                 S(v_i) = S(v_i) ∪ R(v_{i,L}, t_j)
27             end
28         end
29     end
30 end
```

表 2-1 展示了层分配算法中的动态规划过程，该过程被应用在文献[19]中基于协商的层分配算法 MiniDelay 里的全部四个流程里。该过程使用了递归函数*Rec*，在递归过程中根据当前搜索到的节点的类别：叶子节点、根节点和其它分支节点，执行各自对应的动态规划过程，并将所有分配方案的信息记录进S中。

虽然相比于基于整数的线性规划算法，基于动态规划的算法已经有更好的算法效率。但由于线网数量的增加，3 维布线区域的扩大，单次执行动态规划算法计算线网代价值所需要的时间也不断增加。加上现有层分配算法为得到更理想的布线方案，会多次修改代价函数计算方法，并对已分配线网进行多次重分配，从而使同一线网会多次使用动态规划算法计算代价值。因此，动态规划部分不可避免地成为限制整个层分配算法效率的重要因素。

第三章 Taskflow 的研究和使用

3.1 引言

在研究原算法的实现发现，原算法采用了记忆化搜索的方式实现树型动态规划，根节点作为发射器，叶子节点作为接收器，对节点进行状态转移的时候，需要该节点的所有孩子节点的状态计算完毕时才计算自身状态。这种方式无法很好的利用树型动态规划的结构特性，为此，本文引入了 TaskFlow 任务编排框架，提高执行效率。本章将介绍 Taskflow 轻量并行库。

3.2 Taskflow 功能介绍

Taskflow 库是一个开源、跨平台的多线程任务管理编排库。它具有简单、高效、易拓展、线程安全等优点。它允许用户创建任务结点，设定任务之间的依赖关系，从而构建出一整个任务流。一个任务流中的依赖关系可以被描述为一张有向无环图。执行任务流时，Taskflow 将会根据依赖图和其内置的调度算法，决定哪些任务优先分配线程执行，从而保证并行算法的正确性。当多个线程需要访问同一临界资源时，也可以在 Taskflow 框架下继续通过加锁访问以避免出现错误。

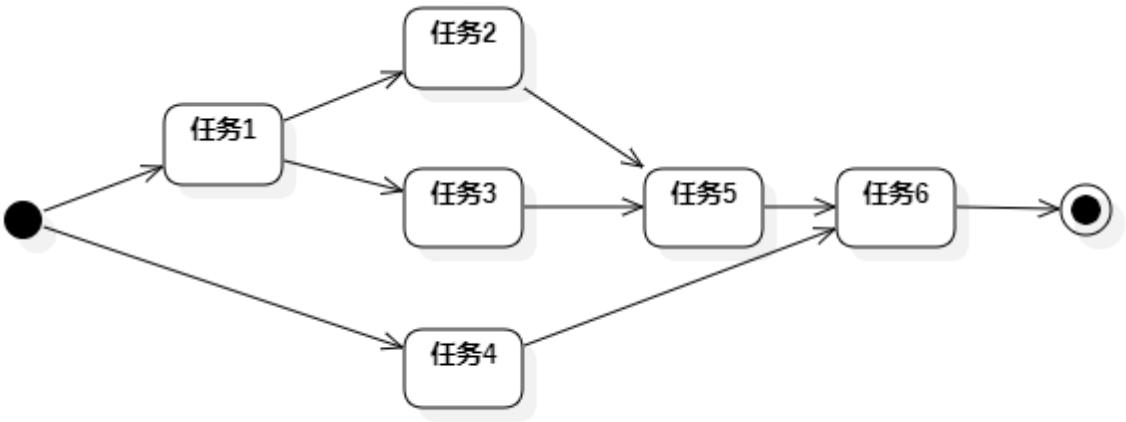

图 3-1 一种 Taskflow 可接受的任务图

图 3-1 展示了 Taskflow 能接受的一种典型任务流程图。该任务图为有向无环图，包含 6 个任务节点、一个起始点和一个终止点。该任务流程图显示了任务之间的依赖关系，Taskflow 将根据该依赖关系进行任务的并行执行。例如，任务 5 必须在任务 2 和任务 3 执行结束后才可执行，而任务 6 则必须等任务 5 和任务 4 都执行结束后才可执行。最终任务的执行顺序，必然是该有向无环图的一种拓扑排序。若执行该任务流程，则在初始时刻，

只有任务 1 和任务 4 是可并行执行的。若任务 1 执行结束，则接着就是任务 2、任务 3 和任务 4 一起并行执行。

3.3 Taskflow 主要类介绍

表 3-1 Taskflow 框架中的主要类介绍

类名	作用
Node	保存前驱节点指针和后继节点指针等节点信息，提供节点执行方法
Task	对 Node 类的包装，防止直接操作内部存储，包含添加任务依赖方法
Graph	负责管理 Node 对象的内存，含有一个管理 Node 的内存池
Topology	负责管理 Graph 的拓扑结构
TaskFlow	继承于 FlowBuilder，保存了 Graph 对象，能为 Graph 创建节点
Executor	负责执行 TaskFlow，创建工作线程

TaskFlow 是一个通用的并行异构任务流程框架，基于有向无环图(DAG)对任务进行编排管理。利用 TaskFlow 可以快速实施任务分解，利用多线程提高性能，而文献[24]中为 TaskFlow 的流程管理提供了理论依据。TaskFlow 使用了 C++17 的语法特性，需要 C++17 的编译环境，为此，在编译中需要添加-std=c++17 的编译指令。

Node 类是 TaskFlow 中最重要的类，该类保存了当前节点的执行函数、前驱节点指针、后继节点指针、前继节点完成计数器。执行函数使用 std::variant 保存，std::variant 是 c++17 新加入的容器，提供了更安全的 union, 在 TaskFlow 中，目前可以接受的函数类型为 Static、Dynamic、Condition、MultiCondition、Module、Async、SilentAsync、cudaFlow、syclFlow、Runtime10 类。_precede 用于向该节点添加后继节点，是构建任务拓扑关系的重要函数。

Task 类是对 Node 类的包装，该类弱引用一个 Node 对象指针，可以防止使用方直接操作内部存储。Task 提供了 succeed 和 precede 来为 Task 间添加依赖关系，A.succeed(B)代表 A 依赖 B，与 B.precede(A)等价，实际上 A 的 succeed 方法就是调用了 B 的 Node 的_precede 方法。

Graph 类负责管理 Node 对象的内存，为了避免赋值和拷贝引起的问题，Graph 显示删除了拷贝构造函数和赋值操作函数。Graph 中的_nodes 使用 vector 容器维护 Graph 中 Node 对象的指针。node_pool 是一个管理 Node 的内存池，采用内存池可以避免频繁调用 malloc/free, 减少内存碎片化，提高内存使用率，提高内存分配速度，降低分配开销。该类使用 emplace_back 将执行函数封装成 Node 节点，emplace_back 方法从 node_pool 中管理的内存池里获取一个 node，并初始化 node 对象，然后指向该 node 对象的指针会被加入到 Graph 的_nodes 中。

Topology 用于管理 Graph 的拓扑结构，_bind 将绑定一个 Graph 对象，保存了所有入口节点指针，出口个数，出口节点完成计数器。

TaskFlow 类继承于 FlowBuilder，保存了 Graph 对象。提供了 emplace 方法在 Graph 中创建新节点，并使用 callable 参数作为现新节点的执行函数。

Executor 类是负责执行 TaskFlow 的类。在该类中会创建工作线程，如果没有指定工作线程数量，调用 hardware_concurrency 获取能并发在一个程序中的线程数量，hardware_concurrency 在 linux 系统中一般返回 CPU 核心数量。

同时，TaskFlow 提供了内置分析器 TFProf，可以在 web 页面中分析和可视化任务流，还可以将任务存储为 DOT 格式，使用 GraphViz 工具进行可视化。

3.4 本章小结

Taskflow 框架的特点可以归结为以下四点：

轻量级：Taskflow 的核心库非常小巧，并且可以与其他库和框架集成。

灵活性：Taskflow 提供了多种不同的任务调度策略，可以根据应用程序的需求进行配置。

易用性：Taskflow 提供了一个简单易用的 API，使得开发人员可以很容易地编写并行程序。

异构性：Taskflow 支持多种异构设备，并提供了一个统一的编程模型，使得开发人员可以轻松地将任务图映射到这些设备上。

总之，Taskflow 是一个非常有用的并行和异构任务图计算系统，可以帮助开发人员轻松地编写高效的并行程序。本文使用该框架，从而很方便地实现了线网内的并行算法。

第四章 高效层分配算法的设计

4.1 引言

第二章中，本文指出了优化层分配算法中的动态规划效率的重要性。第三章中，本文介绍了 Taskflow 并行库。本章将系统阐述本文针对文献[19]中的原算法 MiniDelay 进行的具体分析和优化，分为以下几点：

（1）层分配算法运行时间分析：分析原算法中动态规划算法的执行时间占比；

（2）函数调用逻辑改写：分析和改进原算法的函数调用逻辑；

（3）并行算法的引入：介绍本文为层分配算法设计的并行算法，并介绍该算法基于 Taskflow 框架的实现方法；

（4）线网结构分析：分析数据集中的线网结构，以及该结构对并行效率的影响；

（5）自适应线程数算法设计：介绍如何改进并行算法，实现自适应线程数并行算法；

（6）使用单一线程池：介绍如何通过线程池优化线程重复创建和释放的开销。

4.2 层分配算法运行时间分析

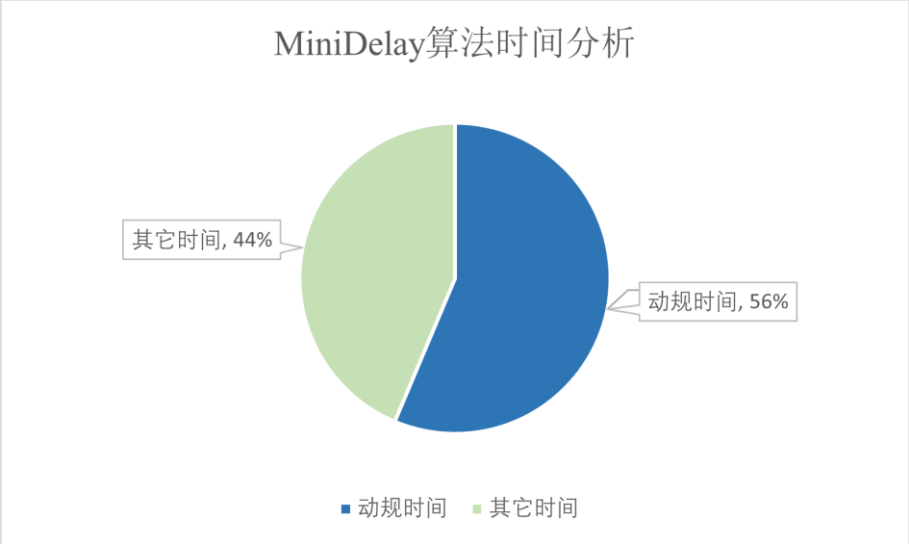

图 4-1 MiniDelay 算法时间分析

MiniDelay 使用了通过动态规划算法实现的层分配算法，每个线网在进行层分配时，都需要通过动态规划算法计算各个导线段分配到各个层上的代价。同时由于使用了基于协商的方法，每个线网都需要进行多次层分配，这也使得动态规划算法被更多次地使用。本文经过多次实验统计，得到了如图 4-1 所示的动规算法运行时间占层分配算法总运行时间的百分比，可以看到，动规时间占比达到了 56%，是算法所需时间的主要部分。因此，要

优化该层分配算法的效率，最有效的方法就是优化算法中动态规划部分的效率。

4.3 函数调用逻辑改写

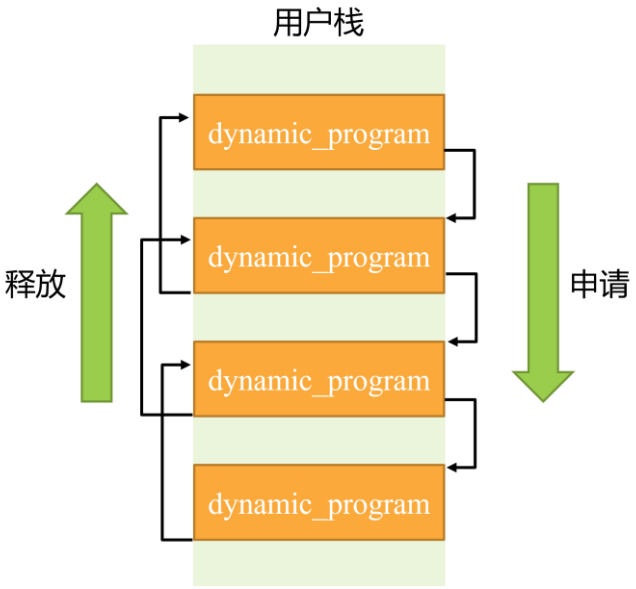

图 4-2 递归函数的内存使用

由于层分配算法涉及的变量与条件数量很多，计算代价函数值的过程也很复杂，因此原算法在动态规划函数中编写得非常复杂且冗长，并且需要开辟较大的栈空间用于储存临时数据。而原算法编写的树形动态规划方法又需要对每一个线网对应的树进行深度优先搜索，并用递归的方式逐步得到整个树的代价函数值。这就使得算法在函数调用和堆栈空间的使用上消耗了大量时间。

图 4-2 展示了层分配算法中递归写法的动规函数使用内存的情况，dynamic_program 代表该动规函数的函数栈帧，向下为函数递归申请主存的过程，向上为递归返回时函数释放主存的过程。每个函数栈帧对应一个仍在运行中的函数，保存了该函数的返回地址以及函数内使用的局部变量和函数返回值等信息。当递归函数调用自身时，需要向主存申请一块新的空间，成为一个新函数栈帧，同时递归前的函数栈帧依然保持不能释放。等待其调用的函数返回后，自身才能继续执行，并在执行完毕后释放函数栈帧。

非递归函数一般拥有比递归函数更快的执行速度，这有以下几个原因：

首先，递归函数需要在每次调用时保存当前函数的状态，并将控制权转移到新的函数调用。这样的操作会增加函数调用的开销，使得递归函数的运行速度比非递归函数慢。

其次，递归函数在每次调用时都会在堆栈中创建一个新的函数帧，并在返回时销毁该帧。如果递归深度很大，将会占用大量的堆栈空间，甚至可能导致栈溢出或者程序崩溃。而非递归函数每次只需要使用一个函数帧，并在调用结束后立即释放函数帧，因此可以避免这个问题。

此外，当主存不足以继续保存新的函数栈帧时，操作系统会使用页面置换算法，将主存中的页置换到磁盘空间上，这也进一步增加了递归函数的时间开销。

基于此，本文将原算法递归执行的树形动态规划函数改为使用非递归方式调用和执行，该改写可以通过使用 Taskflow 框架很方便地完成。

4.4 并行算法的引入

4.4.1 并行算法设计

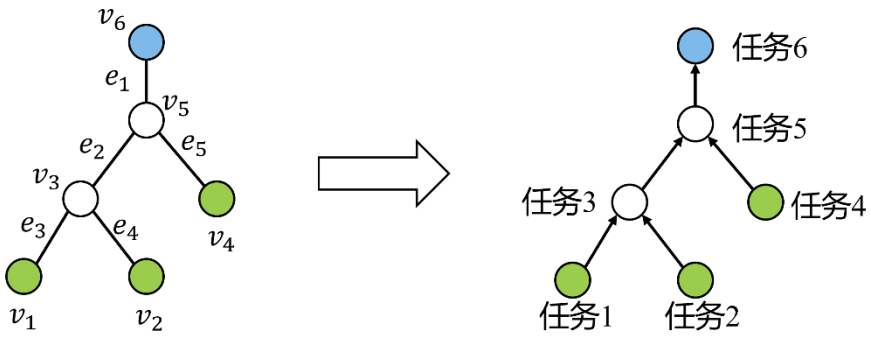

图 4-3 一种线网树和为其建立的任务流程图

层分配算法中，使用树形动态规划求解线网代价函数值的过程可以用任务流程图表示。如图 4-3 所示：左侧为一种线网结构对应的树结构，其中根节点v_6为线网的发信器，叶子节点v_1、v_2、v_4为线网的接收器，中间节点v_3和v_5是连接发信器和接收器的导线段，边e_1、e_2、e_3、e_4和e_5为线网中导线段通过的 2D 网格图边缘。右侧为计算线网分配代价时使用的动态规划方法，该方法中，每个节点都需要计算该节点对应的导线段、发信器或接收器分配到不同金属层时所需的代价，这个计算过程可被封装成一个任务。其中，任务的编号与节点的编号一一对应，编号顺序为树的后序遍历顺序。为了使任务的执行顺序正确，即所有子节点的任务必须全部执行完毕，才允许父节点的任务执行，本文使用 Taskflow 框架为任务建立流程图，这是一个有向无环图。Taskflow 可以保证任务的执行顺序，必然是该任务流程图的一种拓扑排序。

图 4-4 展示了图 4-3 中建立的任务流程图在多线程下的执行情况，其中，假设每个任务的执行时间相同，都为 1 单位时间，使用 3 个线程执行，且每个线程各自占用一个中央处理器（Central Processing Unit， CPU）核心，不考虑操作系统对线程的调度。由图可见，在 3 线程下，这 6 个任务的总执行时间为 4 个单位时间，也是多线程下最快的执行时间。

图 4-5 展示了图 4-3 中的任务流程图在单线程下的执行情况，同样假设每个任务的执行时间都为 1 单位时间。该线程仅使用一个 CPU 核心。由图可见，单线程下执行同样的任务需要 6 个单位时间。

对比图 4-4 和图 4-5 可知，多线程并行由于使用了更多的 CPU 核心进行计算而明显比单线程串行的执行时间短。

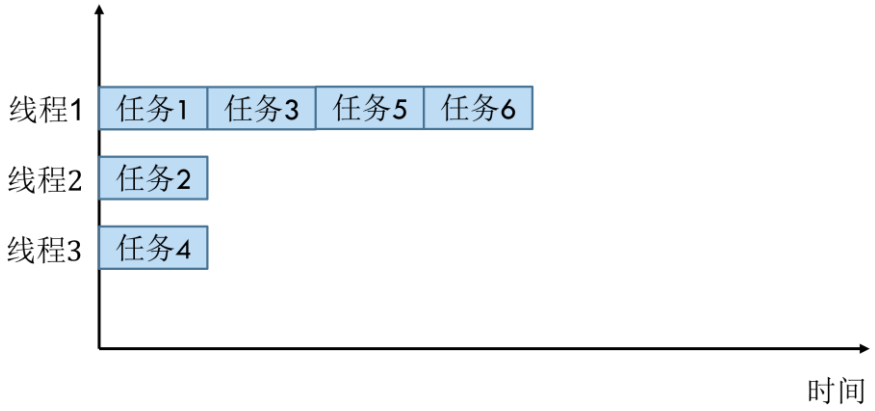

图 4-4 多线程下任务执行情况

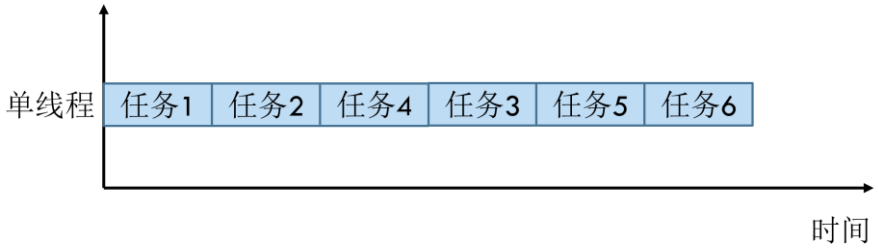

图 4-5 单线程下任务执行情况

综上所述，通过建立从子节点指向父节点的流程依赖，就能确保先计算子节点，后计算父节点的算法执行顺序，满足该树形动态规划算法的要求。同时，多个彼此没有依赖关系的节点计算任务将通过 Taskflow 的调度算法并行执行，可以充分利用多核处理器带来的高算力支持。

4.4.2 并行算法具体实现

本文在执行动态规划前，预先利用 Taskflow 编排任务，将树上每个节点视为一个 Task，依据所有孩子节点执行完才能执行自身节点这条规则，建立 Task 间的拓扑关系，从而构建 TaskFlow 类对象，也即一个任务流。每个节点对应的任务，都依赖于该节点的所有孩子对应的任务，只有当所有孩子对应的任务均执行完毕后，父节点对应的任务才被允许执行。

接着利用 Executor 类生成执行器对象，然后将任务流交由执行器执行。执行器将根据设定的线程数，向系统内核申请线程，然后用这些线程执行任务流，实现任务流内各个可并行任务的并行执行，提高了执行效率。

表 4-1 展示了建立任务流的简易过程。通过使用节点队列和任务队列，buildDependency 对线网树进行了广度优先遍历，在遍历过程中，创建每个节点对应的任务节点，并为其添加依赖关系，从而建立完整的任务流图。

表 4-1 buildDependency 建立任务流过程

Algorithm buildDependency

```
Input: _rt, taskflow
let Q be a queue for tree nodes
let TQ be a queue for tasks
task_root ← create a new task for root use taskflow
Q.enqueue(root)
TQ.enqueue(task_root)
while Q is not empty do
    pa ← Q.dequeue()
    task_pa ← TQ.dequeue()
    forall ch ← child of pa in _rt do
        task_ch ← create a new task for ch use taskflow
        Q.enqueue(ch)
        TQ.enqueue(task_ch)
        task_pa.succeed(task_ch)
    end
end
```

具体来说，任务节点为一个 Task 对象，Task 弱引用了 Node 对象指针，而 Node 对象保存当前节点的执行函数，这里树上节点的执行函数即为动态规划的状态转移方程，而为了让 Task 更好的执行，需要将状态转移函数和以及函数运行所需的上下文一同封装。C++11 提供了 Lambda 表达式定义匿名函数，这里通过 Lambda 的捕获能力获取表达式的外部变量。

通过对于原论文实现的分析可知，对于 TREE_NODE 列表、RECORD_DP 列表、SEGMENT 列表使用引用捕获，其余采用值捕获符合使用情况。TREE_NODE 列表和 SEGMENT 列表在状态转移方程内不会被修改，如果使用值捕获，执行时会重新构造一个一摸一样的对象，所需内存巨大，浪费资源，故本处采用引用捕获合适；RECORD_DP 列表每个 Task 只会对其中一个节点的值修改，而并行过程中，不会用两个 Task 修改同一个位置，所以直接采用引用捕获，也不需要对其进行加锁。

TaskFlow 的 emplace 接受节点对应的 Lambda 表达式，转交给 Graph 对象的 _emplace_back 方法，_emplace_back 将从内存池获取一个 Node 对象，利用传入参数初始化 Node 对象。Node 类接受的函数类型中的 Static 类型是对 std::function 的封装，std::function 是 C++11 提供的一个模板类，可以存储任意可调用对象，包括函数指针、Lambda 表达式等，故该 Lambda 表达式可以被接受。

在 DFS 访问节点 V 的完全部孩子节点后，可以构造该节点对应的 Task 和孩子节点对应的 Task 的执行关系，利用 Task 中的 succeed 方法保证在孩子节点完成后再执行自身。在这种模式下，减少了由于递归引起的压栈开销，相比于源论文，提升了代码执行效率。

在编排完 TaskFlow 以后，使用 Executor 去执行这个 TaskFlow。Executor 初始化接受

一个 size_t 类型的值作为线程池的数量，本文对 12 线程、8 线程、4 线程下的算法表现进行了测试，具体结果将在第五章展示。

4.5 线网结构分析

本文在线网内部进行并行，并行效果受到线网结构影响。经过实验测试，低线程数下的算法表现在大多数数据下表现优于高线程数。针对此现象，本文对线网结构进行了进一步研究。

4.5.1 线网结构差异对并行算法的影响

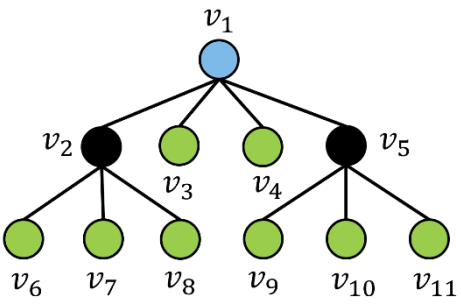

图 4-6 一种并行效果好的线网结构

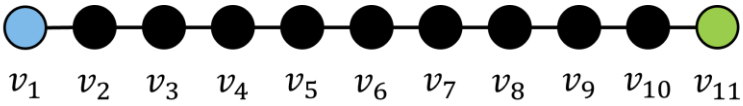

图 4-7 一种并行效果差的线网结构

图 4-6 展示了一种并行效果好的线网结构，该结构中共有 1 个发信器和 8 个接收器，总共占用 11 个网格。在图中体现为 1 个根节点和 8 个叶子节点，总共有 11 个节点。由于每个节点对应一个任务，故在任务流程图中，该线网将产生 11 个任务节点。

图 4-7 展示了一种并行效果差的线网结构，该结构中共有 1 个发信器和 1 个接收器，总共占用 11 个网格。在图中体现为 1 个根节点和 1 个叶子节点，总共有 11 个节点。该线网同样将产生 11 个任务节点。

虽然同样使用 11 个任务节点，但图 4-6 中的线网结构与图 4-7 中的线网结构会在并行算法中产生很大的执行效率差异。

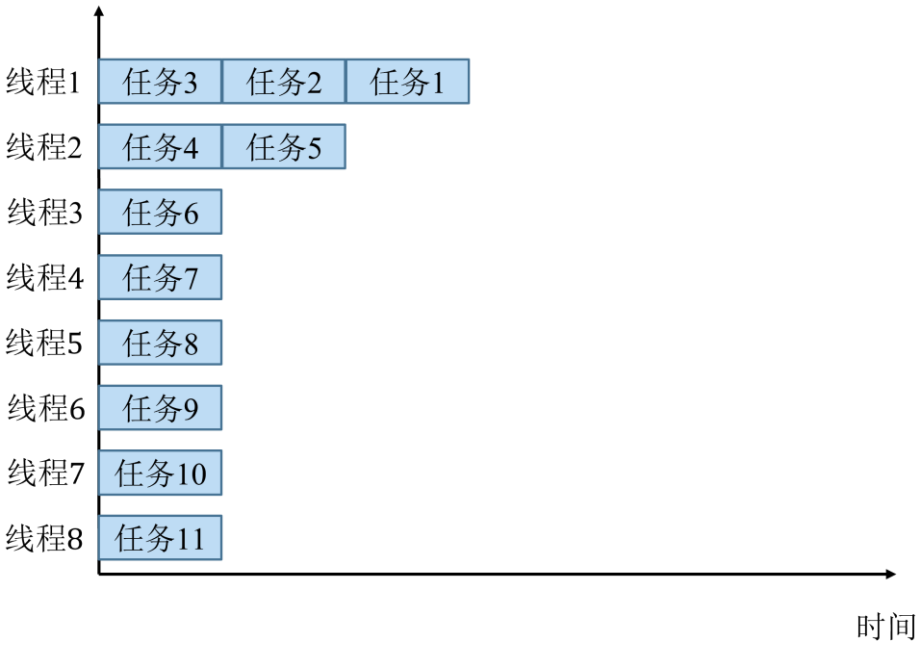

图 4-8 多线程下图 4-6 所示线网结构的执行情况

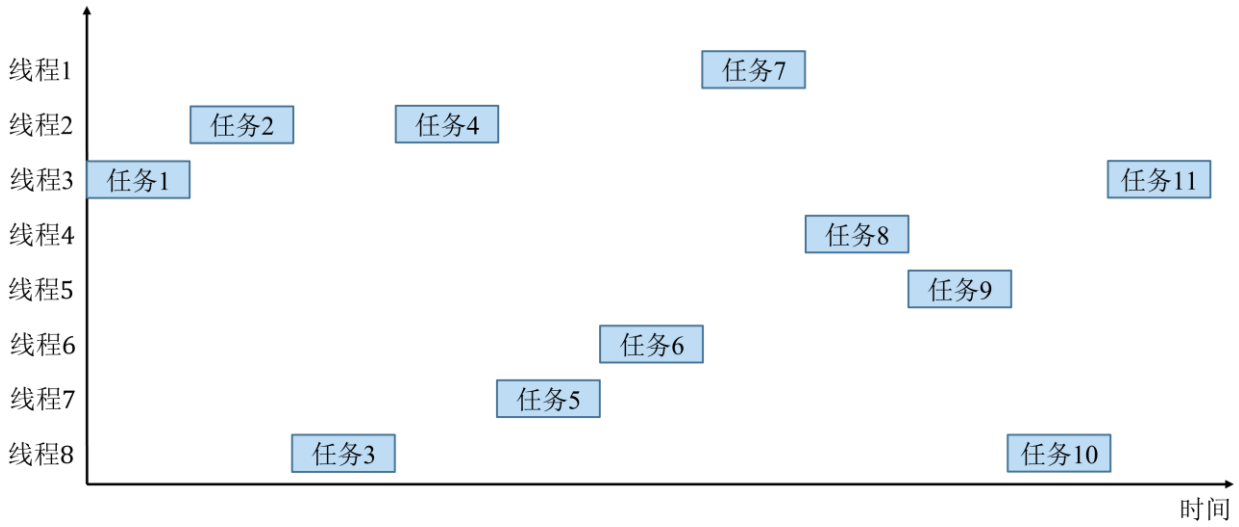

图 4-9 多线程下图 4-7 所示线网结构的执行情况

图 4-8 展示了在多线程下，图 4-6 所示的线网结构对应的 11 个任务节点的执行情况，其中，假设每个任务节点的运行时间均为 1 个单位时间，每个线程独立占用一个 CPU 核心。由图可见，该结构下，使用 8 个线程，只需 3 个单位时间即可将全部 11 个任务执行完毕。

图 4-9 展示了在多线程下，图 4-7 所示的线网结构对应的 11 个任务节点的执行情况，假设同上。由图可见，在该结构下，同样使用 8 个线程，需要 11 个单位时间才能将全部任务执行完毕。而在单线程下，这 11 个任务同样需要 11 个单位时间来执行，可见此时多线程并无法为该线网的层分配过程加速。相反，由于线程之间的调度需要额外时间代价，这

种线网在多线程下的执行速度反而不如单线程。这个时间代价还会随着线程数的增多而进一步增加。

4.5.2 superblue 数据中线网结构分析

本文使用的数据来自 2012 年电子设计自动化国际会议所举办的可布线性驱动放置比赛[25]，数据集名称为 superblue。由于不同线网结构的并行效果不同，本部分内容将对 superblue 中的线网结构进行分析。

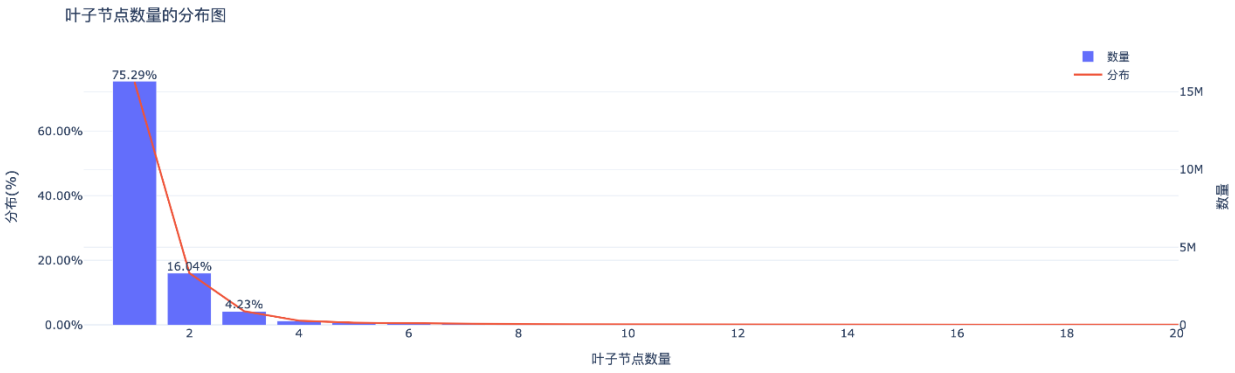

图 4-10 superblue 数据的叶子节点数分析

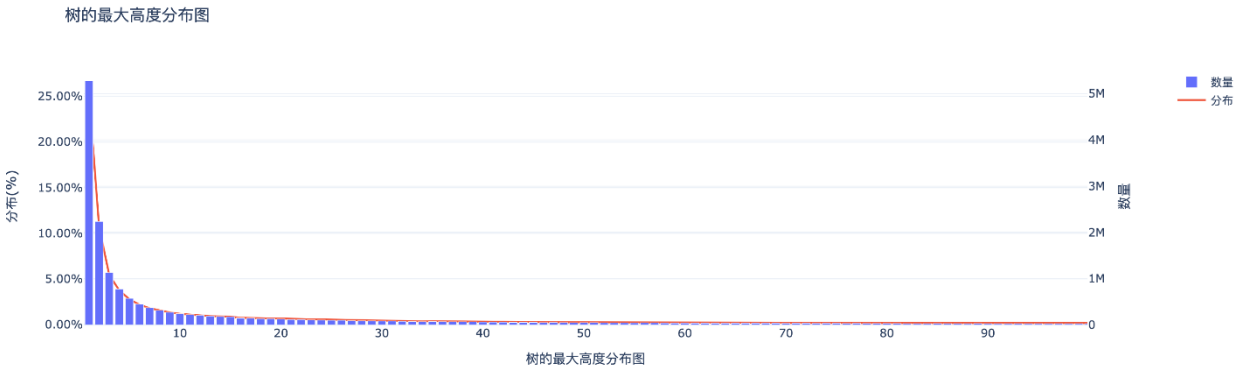

图 4-11 superblue 数据的树高分析

图 4-10 展示了 superblue 中 2D 线网结构经过变换得到的树的叶子节点数。可以看到，75.29%的线网树只有 1 个叶子节点，也即单链结构。对于这部分线网，多线程不仅不能加快它们的层分配效率，反而会因线程调度而浪费时间。16.04%的线网树有 2 个叶子节点，4.23%的线网树有 3 个叶子节点，其余线网有更多的叶子节点，这些线网可能单线程更快也可能多线程更快，使不同线网达到最佳层分配效率所需的线程数也可能不同。根据 4.5.1 的分析，叶子节点多的线网，使其层分配效率达到最高所需的线程数一般也会越多。

图 4-11 展示了 superblue 中 2D 线网结构经过变换得到的树高。可以看到，大部分线网树的树高集中在 10 以下，但也存在少部分线网树高能达到 90 以上。树高也会影响并行效

果，树高越大，叶子节点数越少，树的结构就越接近单链结构，这种结构最不利于并行。并行效果好的线网树应当有较多的叶子节点和较小的树高。

4.6 自适应线程数算法设计

通过实验和分析，可以发现对所有线网都使用相同线程数进行并行是不合适的。因此，本文提出使用自适应线程数算法为不同线网使用不同线程数进行并行。

本文使用以下方式计算线网所需线程数：

$$threads = min(cores, leaves) \qquad \text{公式}(4-1)$$

其中，$threads$表示线程数，$cores$为 CPU 核心数，$leaves$表示线网树的叶子节点数min表示取最小值。此方法将使尽可能多的叶子节点同时计算，同时又避免在叶子节点数少时过多的线程数引入额外线程调度开销。虽然线程数可以大于 CPU 核心数，但那将会使 CPU 在使用时间片调度算法时决定当前由哪些线程使用 CPU 核心，并无法使每个线程都并行执行。所以，当线程数超过 CPU 核心数时将无法继续提高并行效率。因此，本文限制了线程数不能超过 CPU 核心数。

使用自适应线程数算法就能避免在 75.29%的线网中引入线程调度的额外开销，而又能在接收器多的线网中发挥并行算法的作用。使用该方法也会引入少量开销，用于计算每个线网树的叶子节点。本文也有考虑使用树高来确定线程数，但树高和线程数之间的关系比较不直观，经过实验，直接使用叶子节点数来确定线程数的方法更好。

4.7 使用单一线程池

通过实验和分析，使用自适应线程数的方法后虽然效率提升明显，但依然存在较大线程开销。研究发现，该算法引入的主要开销并非来自线程调度，而是来自线程的创建和释放。由于每个线网在进行并行时，都需要先创建线程，然后当执行结束后又需要将线程释放，因此需要较大的时间开销。为减少此开销，本文使用了单一线程池对并行算法进行优化。具体来说，在算法开始时，直接创建足够的线程数并放入线程池中，然后在每个线网进行层分配过程时，都从该线程池中获取线程，层分配结束后由线程池回收线程，而并不将线程直接释放。该方法使得算法无需多次重复创建和释放线程，而是在算法开始前一次性申请线程，并在算法结束后一次性释放线程。

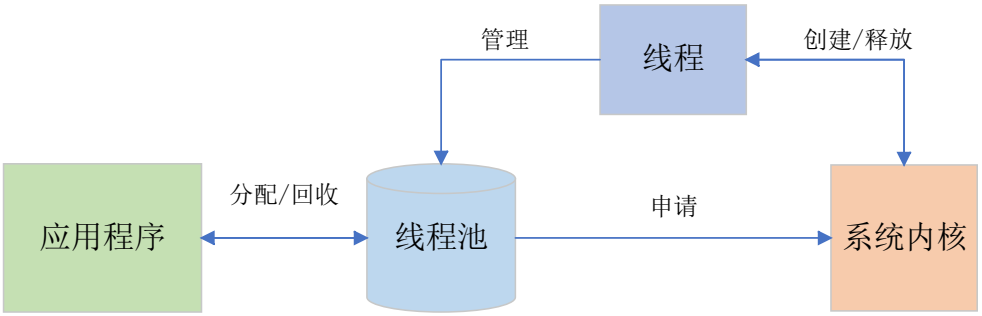

图 4-12 线程池工作方式

图 4-12 展示了线程池的工作方式，线程池向系统内核申请线程，然后接管系统内核创建的线程。当应用程序需要使用线程时，不必直接向系统内核申请，而是直接由线程池负责线程的分配和回收。

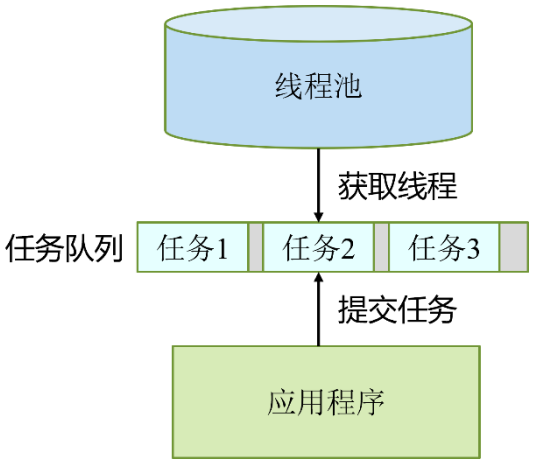

图 4-13 任务队列工作方式

在线程的分配和回收过程中，使用了生产者消费者模式。如图 4-13 所示，应用程序负责提交任务到任务队列中，生产任务。线程池分配线程给任务队列执行任务，也即消费任务。这种方式使得每个任务在执行时不必额外创建线程，而是使用线程池中已有的线程来执行任务，节约了线程创建和释放的时间。

在之前的并行方法中，虽然也使用了线程池，但是线网和线网之间的线程池并不共享使用，浪费了很多创建线程和释放线程的时间。使用单一线程池即可解决此问题，从而改善算法效率。

第五章 实验结果

5.1 实验环境

本文的所有代码均运行在腾讯云服务器上，使用的实例类型为计算型 C6，实例配置为 16 核 32GB 内存，所用系统盘为高性能云硬盘。本文使用的操作系统为 64 位操作系统 Ubuntu Server 22.04 LTS。本文对服务器的使用为独占方式使用，因此不存在对服务器资源的竞争使用。

5.2 实验数据

本文使用的数据来自 2012 年电子设计自动化国际会议所举办的可布线性驱动放置比赛[25]，数据集名称为 superblue。该数据来自实际的工业级电路设计数据，具有很高的真实性。同时，数据涵盖了多种不同类型的电路设计，包括数字电路、模拟电路和混合信号电路等，因此可以更综合地评测层分配算法在不同类型的电路设计下的表现。这些数据包含十万数量级的线网数，可以保证实验结果的客观性。

5.3 结果展示

表 5-1 12 线程下五次实验结果

电路	第一次（s）		第二次（s）		第三次（s）		第四次（s）		第五次（s）	
	MD	12 线程	MD	12 线程	MD	12 线程	MD	12 线程	MD	12 线程
sp2	653.919	1137.78	655.249	1139.30	654.280	1132.54	655.720	1134.40	650.304	1202.68
sp3	471.347	874.890	438.621	868.707	431.947	870.607	432.004	872.672	453.034	870.953
sp6	335.900	839.846	326.410	837.835	330.662	839.003	326.177	841.103	329.896	836.552
sp7	442.933	1129.21	428.009	1125.19	427.186	1130.76	423.204	1125.27	419.968	1123.91
sp9	266.150	656.419	255.852	647.816	255.755	647.514	255.084	648.202	257.622	648.204
sp11	407.348	932.656	398.266	928.663	394.616	927.884	394.253	924.140	396.860	934.475
sp12	397.695	1099.39	389.271	1095.68	375.947	1099.45	379.579	1096.45	376.372	1097.68
sp14	196.100	553.453	190.552	548.969	193.203	548.977	193.014	548.711	192.975	549.335
sp16	427.540	997.283	421.103	998.720	424.620	997.331	421.599	998.960	421.053	998.911
sp19	115.181	375.615	109.962	372.363	109.915	363.574	111.030	346.706	109.883	345.039
平均	371.411	859.654	361.330	856.324	359.813	855.764	359.166	853.661	360.797	860.774

表 5-2 12 线程下五次实验结果的平均值和标准差

电路	平均（s）			标准差	
	MD	12 线程	加速比	MD	12 线程
sp2	653.894	1149.34	0.57	2.13	29.94
sp3	445.391	871.566	0.51	16.87	2.33
sp6	329.809	838.868	0.39	3.96	1.76
sp7	428.260	1126.87	0.38	8.82	2.95
sp9	258.093	649.631	0.40	4.60	3.81
sp11	398.269	929.564	0.43	5.34	4.08
sp12	383.773	1097.73	0.35	9.46	1.70
sp14	193.169	549.889	0.35	1.97	2.00
sp16	423.183	998.241	0.42	2.85	0.86
sp19	111.194	360.659	0.31	2.28	14.21
平均	362.503	857.236	0.42	-	-

表 5-3 8 线程下五次实验结果

电路	第一次（s）		第二次（s）		第三次（s）		第四次（s）		第五次（s）	
	MD	8 线程	MD	8 线程	MD	8 线程	MD	8 线程	MD	8 线程
sp2	653.919	946.659	655.249	945.999	654.280	949.275	655.720	947.365	650.304	948.219
sp3	471.347	666.285	438.621	665.616	431.947	666.523	432.004	666.927	453.034	665.438
sp6	335.900	623.654	326.410	622.913	330.662	621.377	326.177	622.914	329.896	623.161
sp7	442.933	837.036	428.009	836.282	427.186	838.339	423.204	838.382	419.968	835.814
sp9	266.150	485.159	255.852	484.607	255.755	483.224	255.084	484.452	257.622	485.202
sp11	407.348	698.843	398.266	697.259	394.616	701.589	394.253	701.146	396.860	699.405
sp12	397.695	796.439	389.271	800.485	375.947	796.753	379.579	797.625	376.372	796.546
sp14	196.100	405.071	190.552	404.850	193.203	406.750	193.014	404.908	192.975	405.257
sp16	427.540	748.911	421.103	752.031	424.620	749.170	421.599	750.067	421.053	749.362
sp19	115.181	271.147	109.962	271.142	109.915	272.098	111.030	271.269	109.883	270.607
平均	371.411	647.920	361.330	648.118	359.813	648.510	359.166	648.506	360.797	647.901

表 5-4 8 线程下五次实验结果的平均值和标准差

电路	平均（s）			标准差	
	MD	8 线程	加速比	MD	8 线程
sp2	653.894	947.503	0.69	2.13	1.29
sp3	445.391	666.158	0.67	16.87	0.62
sp6	329.809	622.804	0.53	3.96	0.85
sp7	428.260	837.171	0.51	8.82	1.17
sp9	258.093	484.529	0.53	4.60	0.80
sp11	398.269	699.648	0.57	5.34	1.76
sp12	383.773	797.570	0.48	9.46	1.70
sp14	193.169	405.367	0.48	1.97	0.79
sp16	423.183	749.908	0.56	2.85	1.26
sp19	111.194	271.253	0.41	2.28	0.54
平均	362.503	648.191	0.56	-	-

表 5-5 4 线程下五次实验结果

电路	第一次（s）		第二次（s）		第三次（s）		第四次（s）		第五次（s）	
	MD	4 线程	MD	4 线程	MD	4 线程	MD	4 线程	MD	4 线程
sp2	653.919	724.771	655.249	724.487	654.280	729.354	655.720	729.059	650.304	727.537
sp3	471.347	508.204	438.621	507.260	431.947	507.771	432.004	504.278	453.034	508.305
sp6	335.900	440.865	326.410	440.077	330.662	442.436	326.177	437.658	329.896	439.451
sp7	442.933	586.607	428.009	586.009	427.186	587.257	423.204	589.354	419.968	586.750
sp9	266.150	347.955	255.852	347.514	255.755	347.878	255.084	349.021	257.622	349.077
sp11	407.348	512.466	398.266	513.814	394.616	512.967	394.253	513.218	396.860	514.635
sp12	397.695	546.258	389.271	548.029	375.947	546.604	379.579	549.066	376.372	548.351
sp14	196.100	286.274	190.552	286.354	193.203	286.740	193.014	285.126	192.975	285.924
sp16	427.540	556.220	421.103	554.859	424.620	552.680	421.599	554.015	421.053	554.247
sp19	115.181	185.013	109.962	184.716	109.915	185.670	111.030	185.321	109.883	185.243
平均	371.411	469.463	361.330	469.312	359.813	469.936	359.166	469.612	360.797	469.952

表 5-6 4 线程下五次实验结果的平均值和标准差

电路	平均（s）			标准差	
	MD	4 线程	加速比	MD	4 线程
sp2	653.894	727.042	0.90	2.13	2.31
sp3	445.391	507.164	0.88	16.87	1.67
sp6	329.809	440.097	0.75	3.96	1.76
sp7	428.260	587.195	0.73	8.82	1.29
sp9	258.093	348.289	0.74	4.60	0.71
sp11	398.269	513.420	0.78	5.34	0.83
sp12	383.773	547.662	0.70	9.46	1.19
sp14	193.169	286.084	0.68	1.97	0.61
sp16	423.183	554.404	0.76	2.85	1.29
sp19	111.194	185.193	0.60	2.28	0.36
平均	362.503	469.655	0.77	-	-

表 5-1 和表 5-2 展示了在 12 线程下并行算法的表现，表 5-3 和表 5-4 展示了在 8 线程下并行算法的表现，表 5-5 和表 5-6 展示了在 8 线程下并行算法的表现。可以看到，由于数据中单链型线网较多，简单地并行并不能提高算法的效率。相反，由于线程创建、调度和销毁引入了额外开销，故该并行算法的效率比 MiniDelay 更低。可以看到，4 线程的速度反而比 8 线程和 12 线程更快。

为了解决以上问题，本文继续引入了自适应线程数算法，降低不必要的线程开销，为每个线网选定适合的线程数进行并行。表 5-7 和表 5-8 展示了使用自适应线程数算法后的实验结果，可以看到算法效率有了进一步提升，在复杂数据下表现较好，在简单数据下表现较差。在 superblue2 数据集和 superblue3 数据集下，算法速度超过了 MiniDelay 算法，但在其它数据集的表现依然不如 MiniDelay。总体算法表现略逊于 MiniDelay 算法。通过第四章的分析，该算法效率不高的问题在于线程的反复创建和释放。线程的创建和释放需要向系统内核申请，从而产生较大时间开销。单一线程池的方法可以使充分利用每一个申请到的线程，避免提前释放线程而产生额外开销。

表 5-9 和表 5-10 展示了使用单一线程池后的实验结果，可以看到算法效率有了明显提升。在大多数数据中算法效率优于 MiniDelay，在少部分数据中由于线网结构简单而无法优于单线程的 MiniDelay 算法。整体算法效率优于 MiniDelay，使算法综合执行时间减少了大约 4.8%。对比 MiniDelay 算法对 DLA 的算法在时间上的优化率为 1.6%[19]可以看到本文有不错的优化效果。

表 5-7 自适应线程数五次实验结果

电路	第一次（s）		第二次（s）		第三次（s）		第四次（s）		第五次（s）	
	MD	自适应	MD	自适应	MD	自适应	MD	自适应	MD	自适应
sp2	653.919	615.426	655.249	605.819	654.280	605.501	655.720	610.165	650.304	606.947
sp3	471.347	443.109	438.621	435.912	431.947	434.885	432.004	434.319	453.034	434.422
sp6	335.900	373.982	326.410	362.931	330.662	362.467	326.177	362.419	329.896	363.547
sp7	442.933	497.951	428.009	486.004	427.186	485.870	423.204	485.225	419.968	485.700
sp9	266.150	296.830	255.852	288.674	255.755	288.927	255.084	288.986	257.622	288.945
sp11	407.348	428.093	398.266	419.401	394.616	420.784	394.253	427.051	396.860	419.286
sp12	397.695	450.433	389.271	436.112	375.947	436.182	379.579	436.418	376.372	436.811
sp14	196.100	234.382	190.552	228.411	193.203	227.454	193.014	227.974	192.975	227.975
sp16	427.540	456.102	421.103	449.955	424.620	449.023	421.599	449.327	421.053	450.375
sp19	115.181	149.671	109.962	144.578	109.915	144.136	111.030	144.799	109.883	144.355
平均	371.411	394.598	361.330	385.780	359.813	385.523	359.166	386.668	360.797	385.836

表 5-8 自适应线程数五次实验结果的平均值和标准差

电路	平均（s）			标准差	
	MD	自适应	加速比	MD	自适应
sp2	653.894	608.772	1.07	2.13	4.15
sp3	445.391	436.529	1.02	16.87	3.73
sp6	329.809	365.069	0.90	3.96	5.00
sp7	428.260	488.150	0.88	8.82	5.49
sp9	258.093	290.472	0.89	4.60	3.56
sp11	398.269	422.923	0.94	5.34	4.30
sp12	383.773	439.191	0.87	9.46	6.29
sp14	193.169	229.239	0.84	1.97	2.89
sp16	423.183	450.956	0.94	2.85	2.92
sp19	111.194	145.508	0.76	2.28	2.34
平均	362.503	387.681	0.94	-	-

表 5-9 单一线程池五次实验结果

电路	第一次（s）		第二次（s）		第三次（s）		第四次（s）		第五次（s）	
	MD	线程池	MD	线程池	MD	线程池	MD	线程池	MD	线程池
sp2	653.919	598.578	655.249	596.818	654.280	600.359	655.720	602.412	650.304	598.343
sp3	471.347	391.446	438.621	391.167	431.947	391.522	432.004	394.399	453.034	391.693
sp6	335.900	314.139	326.410	314.288	330.662	313.862	326.177	314.390	329.896	314.376
sp7	442.933	409.652	428.009	409.811	427.186	408.654	423.204	410.851	419.968	410.200
sp9	266.150	253.029	255.852	253.587	255.755	253.281	255.084	254.033	257.622	251.867
sp11	407.348	386.683	398.266	386.197	394.616	387.288	394.253	387.909	396.860	386.678
sp12	397.695	350.147	389.271	352.208	375.947	351.743	379.579	351.148	376.372	350.723
sp14	196.100	199.510	190.552	199.851	193.203	199.145	193.014	201.973	192.975	199.569
sp16	427.540	425.381	421.103	425.072	424.620	424.577	421.599	425.639	421.053	424.465
sp19	115.181	120.325	109.962	120.359	109.915	120.477	111.030	120.319	109.883	120.368
平均	371.411	344.889	361.330	344.936	359.813	345.091	359.166	346.307	360.797	344.828

表 5-10 单一线程池五次实验结果的平均值和标准差

电路	平均（s）			标准差	
	MD	单线程池	加速比	MD	单线程池
sp2	653.894	599.302	1.09	2.13	2.14
sp3	445.391	392.045	1.14	16.87	1.33
sp6	329.809	314.211	1.05	3.96	0.22
sp7	428.260	409.834	1.04	8.82	0.81
sp9	258.093	253.159	1.02	4.60	0.81
sp11	398.269	386.951	1.03	5.34	0.66
sp12	383.773	351.194	1.09	9.46	0.81
sp14	193.169	200.010	0.97	1.97	1.13
sp16	423.183	425.027	1.00	2.85	0.51
sp19	111.194	120.370	0.92	2.28	0.06
平均	362.503	345.210	1.05	-	-

总结与展望

总结

随着超大规模集成电路的发展，集成电路的设计空间快速增大，这也使得设计空间内的布线方案急剧增加，如何发挥多金属层集成电路的布线优势，设计出高性能的集成电路布线方案也困难重重。本文虽未在集成电路布线方案本身进行改进，但是通过引入并行算法提高了层分配算法的算法效率，加快了得到集成电路布线方案的过程。

当前集成电路中的晶体管尺寸早已进入纳米级范畴，这意味着集成电路将具有更大的布线密度，并能在单位空间中容纳更多的线网，这从另一方面提高了层分配算法的负担，使优化层分配算法的算法效率变得更加重要。本文通过研究发现，当前线网呈现数量多、长度长而接收器数量少的特点。这意味着在层分配算法计算线网代价时，简单地在线网内部对各个接收器的计算过程进行并行是难以取得较好并行效果的。为解决此问题，本文的并行算法不会使用固定线程数进行并行，而是根据当前线网的接收器数量，也即叶子节点数来决定并行的线程数，从而得到理想的并行效果。

本文充分利用了多核处理器带来的高线程并行能力，通过充分发挥 CPU 性能，提升了算法效率。

展望

本文虽通过线网内部的并行实现了层分配算法效率的提升，但在线网和线网之间，依然是串行进行计算的。也就是，下一个线网想计算代价函数值，就必须等待上一个线网计算完成才可开始。想进一步提高层分配算法效率，可以在线网和线网之间做粗粒度并行。考虑到在单位三维布线区域上需要分配的线网数量很大，且仍在持续上升，线网之间的并行应当可以取得不错的加速效果。

在本文的基础上，可以继续考虑直接对多个线网使用 Taskflow 库建立依赖，使不同线网的节点之间也可以并行计算代价函数值，从而实现更好的并行效果。

参考文献

[1] 李晨, 马胜, 王璐, 等. 三维片上网络体系结构研究综述[J]. 计算机学报, 2016, 39(9): 1812-1828.

[2] Liu W H, Kao W C, Li Y L, et al. Multi-threaded collision-aware global routing with bounded-length maze routing[C]//Proceedings of the 47th Design Automation Conference. 2010: 200-205.

[3] Alpert C J, Li Z, Moffitt M D, et al. What makes a design difficult to route[C]//Proceedings of the 19th international symposium on Physical design. 2010: 7-12.

[4] Li Z, Alpert C J, Hu S, et al. Fast interconnect synthesis with layer assignment[C]//Proceedings of the 2008 international symposium on Physical design. 2008: 71-77.

[5] Wu T H, Davoodi A, Linderoth J T. GRIP: Global routing via integer programming[J]. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2010, 30(1): 72-84.

[6] Roy J A, Markov I L. High-performance routing at the nanometer scale[J]. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2008, 27(6): 1066-1077.

[7] Ozdal M M, Wong M D F. Archer: A history-based global routing algorithm[J]. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2009, 28(4): 528-540.

[8] Xu Y, Zhang Y, Chu C. FastRoute 4.0: Global router with efficient via minimization[C]//2009 Asia and South Pacific Design Automation Conference. IEEE, 2009: 576-581.

[9] Chen H Y, Hsu C H, Chang Y W. High-performance global routing with fast overflow reduction[C]//2009 Asia and South Pacific Design Automation Conference. IEEE, 2009: 582-587.

[10] Chang Y J, Lee Y T, Gao J R, et al. NTHU-Route 2.0: a robust global router for modern designs[J]. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2010, 29(12): 1931-1944.

[11] Liao H, Zhang W, Dong X, et al. A deep reinforcement learning approach for global routing[J]. Journal of Mechanical Design, 2019, 142(6): 1-17.

[12] Zhou Z, Chahal S, Ho T Y, et al. Supervised-learning congestion predictor for routability-driven global routing[C]//2019 International Symposium on VLSI Design, Automation and Test (VLSI-DAT). IEEE, 2019: 1-4.

[13] Zhou Z, Zhu Z, Chen J, et al. Congestion-aware global routing using deep convolutional generative adversarial networks[C]//2019 ACM/IEEE 1st Workshop on Machine Learning for CAD (MLCAD). IEEE, 2019: 1-6.

[14] Cho M, Pan D Z. BoxRouter: A new global router based on box expansion and progressive ILP[C]//Proceedings of the 43rd annual Design Automation Conference. 2006: 373-378.

[15] Liu W H, Li Y L. Negotiation-based layer assignment for via count and via overflow minimization[C]//16th Asia and South Pacific Design Automation Conference (ASP-DAC 2011). IEEE, 2011: 539-544.

[16] Lee T H, Wang T C. Congestion-constrained layer assignment for via minimization in global routing[J]. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2008, 27(9): 1643-1656.

[17] McMurchie L, Ebeling C. Pathfinder: A negotiation-based performance-driven router for FPGAs[C]//Proceedings of the 1995 ACM third international symposium on Field-programmable gate arrays. 1995: 111-117.

[18] Han S Y, Liu W H, Ewetz R, et al. Delay-driven layer assignment for advanced technology nodes[C]//2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC). IEEE, 2017: 456-462.

[19] Zhang X, Zhuang Z, Liu G, et al. Minidelay: Multi-strategy timing-aware layer assignment for advanced technology nodes[C]//2020 Design, Automation & Test in Europe Conference & Exhibition (DATE). IEEE, 2020: 586-591.

[20] 刘耿耿, 魏凌, 徐宁. 考虑总线时序匹配的多策略层分配算法[J]. 计算机辅助设计与图形学学报, 2022, 34(4): 545-551.

[21] 刘耿耿, 李泽鹏, 郭文忠, 等. 面向超大规模集成电路物理设计的通孔感知的并行层分配算法[J]. 电子学报, 2022, 50(11): 2575-2583.

[22] 刘耿耿, 鲍晨鹏, 王鑫,等. 非默认规则线技术下基于多策略的时延驱动层分配算法[J]. 计算机学报, 2023, 46(4): 743-760.

[23] Liu D, Yu B, Chowdhury S, et al. TILA-S: Timing-driven incremental layer assignment avoiding slew violations[J]. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2017, 37(1): 231-244.

[24] Huang T W, Lin D L, Lin C X, et al. Taskflow: A lightweight parallel and heterogeneous task graph computing system[J]. IEEE Transactions on Parallel and Distributed Systems, 2021, 33(6): 1303-1320.

[25] Viswanathan N, Alpert C, Sze C, et al. The DAC 2012 routability-driven placement contest and benchmark suite[C]// Proceedings of the Design Automation Conference, 2012: 774-782.

致谢

转眼间，四年的本科学习就要抵达终点了。回顾这四年的本科生活，我从对计算机科学的一无所知到现在能够完成一篇较为完整的毕业设计，这期间离不开老师和同学们的帮助。

感谢傅仰耿老师。傅老师在我一进入大学后，就帮助我进入 ACM 协同创新实验室进行学习。此后，在傅老师的指导下，我对算法与数据结构的理解逐渐加深，对它们的应用也逐渐熟练。在实验室里跟老师和同学们一起学习的日子，是我本科生活中最珍贵、最难忘也最快乐的回忆。在论文完成之际，再次由衷地向傅老师表示深深地感谢。

感谢刘耿耿老师。刘老师针对我的毕业论文提出了不少深刻的问题和很多鞭辟入里的建议。在遇到问题时，刘老师既会引导我思考，也会耐心答疑解惑。同时十分感谢李泽鹏师兄，在选题方面给了我很大的帮助，并且为我提供了良好的实验环境，让我的实验能够顺利进行。当实验遇到困难时，他愿意为我指出问题所在，给我许多建设性的建议。在此向他们表示由衷的感谢。

感谢计算机与大数据学院的所有老师、辅导员与工作人员。你们为我提供了良好的学习环境，帮助我在本科四年期间掌握了足够的专业知识，给予我继续在计算机科学与技术领域深入学习的能力和动力。

感谢我的父母与家人们，感谢你们这么多年来含辛茹苦的付出。你们全力支持我学习，支持我在学习路上的各种决定。没有你们坚定的支持，我不可能走到今天这一步。你们的一声声鼓励和一句句关心是我不愿辜负的温暖。

感谢我在福大遇到的好朋友们，能够和他们相遇相识成为好朋友，是我最大的幸运。是他们让我的四年大学生活充满了缤纷的色彩，让我的每一份快乐或烦恼都有了可以分享的地方。祝愿他们都有一个美好的未来。

感谢福州大学，是她为我们每一位学生提供了优秀的学习平台和丰富的校园生活，祝愿福州大学越办越好。

最后，感谢在百忙中参加论文评审和答辩的诸位老师和专家！