学士学位论文

基于模拟退火的FPGA并行布局算法研究

院　　系：信息科学与工程学院

专　　业：微电子学系

姓　　名：严健康

学　　号：10300720125

指导教师：陈更生 高级工程师

摘 要

FPGA 布局流程位于工艺映射和布线之间，布局时间约为整个 FPGA 设计流程的四分之一，而且布局的质量对最终电路布线的布通率、时延和面积等性能都有非常大的影响。

模拟退火算法是 FPGA 中采用最广泛的布局算法，它通过多次迭代的模拟退火降温过程来求出全局最优解。本文在模拟退火算法的基础上，使用多核 CPU 的并行交换来加快 FPGA 的布局速度。该算法使用 MapReduce 模型，将一次交换过程分为处理和提交两个阶段，多核作为 Worker 同时进行处理阶段的计算，Reducer 负责提交所有完成的交换，并检查交换之间的冲突，这样就将上百万次的交换分配到多核 CPU 上并行处理。

同时，本文基于多核并行交换的算法，使用 C++语言实现了相应的软件工具。由于 VPR 只能处理通用的 FPGA 结构，考虑到软件的实用性，本文的软件工具基于 Virtex4 平台开发，以 XDL 文件作为输入输出文件，并基于多个硬件平台，使用 MCNC 的基准电路对软件的布局质量和运行时间进行了相应测试。

本文为现代 FPGA 的布局算法设计，在保证算法确定性、串行等价性和良好的布局质量的基础上，提供了基于多核交换的并行化解决方法。

关键词：模拟退火；FPGA 布局算法；并行布局；多核计算；MapReduce 框架

Abstract

Placement is an intermediate step between packing and routing, and the time of placement is the quarter of the whole FPGA design flow. The quality of placement has great impact on the final routability and time closure.

Simulated annealing algorithm is the most widely used in FPGA placement. This algorithm figure out the global optimal solution through the multiple iterations. In this paper, we propose the parallel move and process it on multi-core based on the simulated annealing algorithm. The algorithm split the phase of one move to the processing step and finalization step according to the MapReduce framework. The first step is processed on the multi-core and the reducer commit all moves in order and check out the collisions. So we distribute the millions of move to all cores to reduce the runtime.

In addition, we achieved the programing of the software to approve the correction the of parallel placement algorithm. In view of the practicability of the tool, we developed the tool based on the Virtex-4 FPGA instead of VPR tool. And the tool read the XDL file to finish the placement and output the XDL file. We also test the performance of the tool with the standard MCNC circuits.

This paper come up with a practical parallel placement solution based on the structure of the real FPGA chip while keep the good quality and deterministic and serial-equivalent result.

Key words：Simulated annealing；FPGA placement algorithm；Parallel placement；Multicore computing；MapReduce framework

目 录

第一章 绪论

1.1 研究背景

当今是信息化时代，以集成电路为基础的信息化产业的发展促进了人类社会的进步。从 1958 年美国仙童公司和德克萨斯仪器公司各自独立发明集成电路以后，集成电路产业经历了大规模，超大规模（VLSI），甚大规模（ULSI）的过程，沿每十八个月翻一番的摩尔定律轨道发展。因此深入研究集成电路设计、生产和加工技术，有利于推动我国微电子行业的发展。

1.1.1 集成电路的发展现状

大规模集成电路是信息产业的核心部分。目前，集成电路主要可以分为以下四类：微处理器（CPU、DSP），存储器（Memory），专用芯片（ASIC，Application Specific Integrated Circuit），通用芯片或可编程逻辑器件（PLD, Programmable Logic Device）。其中，ASIC 芯片是根据用户的需求或特定嵌入式系统的需求而设计开发的，这种芯片开发周期虽长但性能稳定，然而它的灵活性太差，不可修改。因此随着技术的发展，市场上出现了可快速编程开发的通用芯片。

随着数字集成电路技术的发展,可编程逻辑器件（PLD, Programmable Logic Device）成为一种有生命力的新型集成电路。这类芯片可以由用户通过软件进行配置和编程，来完成特定的逻辑功能设计，因此电路设计变得相当灵活。可编程逻辑器件分为简单 PLD（SPLD, Simple Programmable Logic Device)、复杂 PLD（CPLD, Complex Programmable Logic Device）以及片上可编程门阵列 FPGA（Field Programmable Gate Array）三类，它们的区别在于内部的可编程结构实现方式不同，因此功能上也不同。PLD 器件主要是基于 PROM 存储器结构和与或、乘积项等逻辑操作来实现特定的函数功能，结构简单，成本低，在早期得到了广泛应用。但是 CPLD 的数据处理能力明显落后于当时的数字微处理系统[1]，为了进一步提高可编程芯片的数据计算能力，1985 年，美国 Xilinx 公司推出了世界上第一款 FPGA 系列产品，包括两个可编程芯片和支持布局布线、编程下载的软件系统。由于 FPGA 器件采用了基于 LUT 的结构，而且多个 LUT 可以通过互连资源级联起来以实现更加复杂的组合逻辑函数，因此 FPGA 比 CPLD 具有更强的逻辑函数实现能力，而且 FPGA 内部包含了大量的触发器，可以实现各种复杂的数字信号处理系统，因此 FPGA 快速发展并取代了 PLD 器件。

目前，功能不断强大和完善的 FPGA 产品逐渐侵袭专用芯片，DSP 芯片和微

处理器的市场。

1.1.2 EDA 技术的发展

随着可编程芯片的发展，传统的集成电路设计方法已经不能满足日益复杂的设计要求，从而 EDA（Electronic Design Automation）技术发展起来。EDA 技术是融合了计算机应用科学和微电子结构与工艺技术的计算机辅助设计技术。EDA 技术通过采用自顶向下（Top-Down）的设计流程来缩短集成电路的开发周期，节省开发费用，促进了集成电路的发展。

EDA 技术依托计算机工具，让设计者在 EDA 软件平台上，用硬件描述语言完成设计文件，然后由计算机自动地完成逻辑编译、化简、分割、综合、优化、布局、布线和仿真，直至对于特定目标芯片的适配编译、逻辑映射和编程下载等工作。EDA 技术的出现，极大地提高了电路设计的效率和可操作性，减轻了设计者的劳动强度。

由于集成电路的工艺水平已经进入深亚微米乃至纳米级别，互连线延时成为总延时的关键部分，因此 EDA 软件的设计能力必须能够超过工艺增长速度，才能适应工艺的发展。

1.2 FPGA 的结构与设计流程

1.2.1 FPGA 的通用结构

现场可编程门阵列(FPGA)是一种可由用户根据需要自行配置的高密度专用集成电路,它作为专用集成电路(ASIC)领域中的一种半定制电路出现,既解决了定制电路的不足,又克服了可编程器件门电路数有限的缺点,成为了一类标准产品。目前,FPGA 的品种很多,有 Xilinx 公司的 Spartan、Vertex 系列，Altera 公司的 FLEX 系列，Actel 公司的 ProASIC 系列以及 Tl 公司的 TPC 系列等。

FPGA 一般由可配置逻辑块（CLB ，Configurable Logic Block），连线资源（IR ，Interconnection Resource）,和输入输出模块（IOB ， Input Output Block）,以及一个用于存放编程数据的 SRAM（Static Random Access Memory）组成。FPGA 的核心部分通常由逻辑单元阵列及布线资源两部分组成，现在大部分的 FPGA 都采用了岛型结构,如图 1-1 是一个岛型结构 FPGA 示意图。

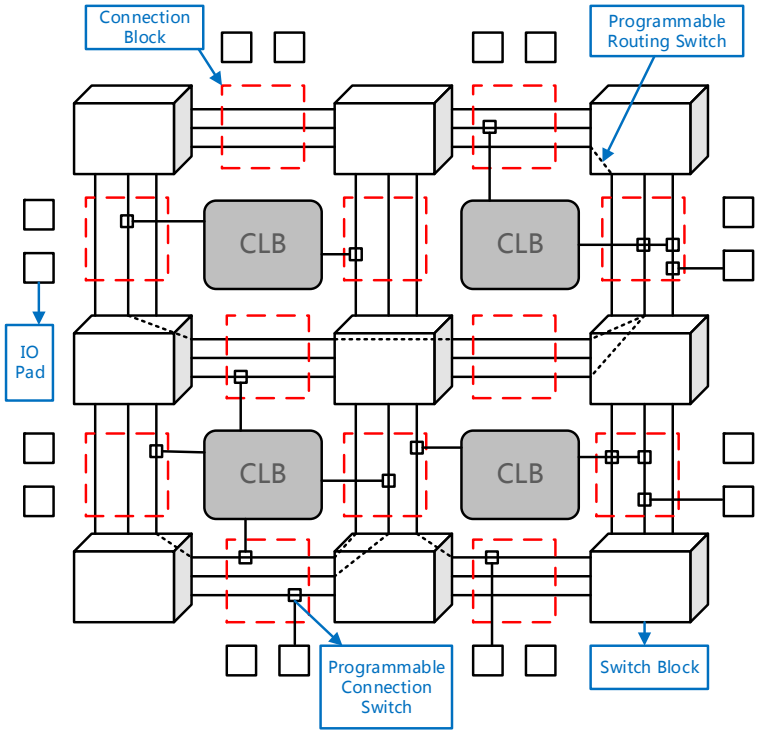


图 1-1 FPGA 的岛状结构示意图

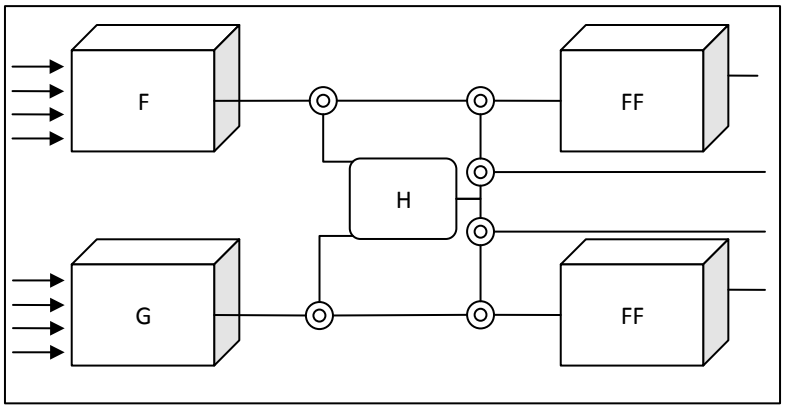


图 1-2 CLB 结构示意图

图中标示为 CLB 的正方形表示逻辑单元,大多数商业 FPGA 都使用基于查找表（LUT ，Lookup Table）的逻辑块结构。在基于查找表逻辑块的 FPGA 中,逻辑块主要是由若干个基本逻辑单元（BLE，Basic Logic Element）组成的,如图 1-2 所示是 Xilinx XC400OE 系列 FPGA 的逻辑块结构。BLE 由两个基于 LUT 结构的 4 输入逻辑函数发生器和一对触发器（FF，Flip-Flop）组成，其中每个 CLB 含有 BLE 的个数以及基本逻辑单元含有 LUT 的输入端个数对 FPGA 的速度、面积和功耗等性能有很大的影响,大多数商业 FPGA 是基于 4 输入查找表的。布线资源连通 FPGA 内部的所有单元,主要由连线通道(Track),连线连接盒 CB(Connection

Box)和连线开关盒 SB(Switch Box)组成,其中连接盒 CB 实现 CLB 与其周围行列分布的布线通道的连接,而开关盒 SB 决定水平与垂直方向的走线。

FPGA 芯片内部有四类连线资源:第一类是全局布线资源,用于芯片内部全局时钟和全局复位/置位的布线;第二类是长线资源,用以完成芯片 Bank 间的高速信号和第二全局时钟信号的布线;第三类是短线资源,用以完成基本逻辑单元之间的逻辑互连和布线；输入输出模块 IOB 是芯片与外围电路的接口,对内部逻辑单元阵列与外围器件之间进行连接,主要是由输入触发器，输入缓冲器，输出触发/锁存器和输出缓冲器组成。

1.2.2 FPGA 的 CAD 设计流程

一旦我们有了对于一个系统的设计框架，FPGA 电路系统的设计可以通过 EDA 技术很快完成，最终芯片可以通过可编程逻辑单元及互连单元的配置很快生产出来（实际上是软件生成的配置信息下载到 FPGA 芯片中），其中的过程无需任何第三方厂商的帮助。这种“非定制”的设计方法使得 FPGA 产品能以比用 ASIC 方法制作短得多的周期就进入市场。但另一方面，这种短时间上市的优点让 FPGA 设计软件承担了很大的压力。为了达到产品的快速上市周期，FPGA 设计软件必须能快速地将 HDL(硬件描述语言)描述转化成配置比特流。

对于 FPGA 而言，实现一个电路就是将芯片内部的可编程开关和配置存储器设置成合适的状态。如图 1-3 所示，首先电路设计者采用高级抽象语言(VHDL 或 Verilog)或者原理图来描述电路，然后进行综合、工艺映射、布局、布线和编程下载这五个子过程。

综合的第一步是把通常以硬件描述语言或电路原理图形式描述的电路转化为基本门级网表，然后再把这个基本门级网表转化为由 FPGA 逻辑单元块组成的网表，使得电路需要的逻辑单元块数目较少，并且/或者电路速度够快。综合过程第二步会执行与工艺无关的逻辑优化，该过程一般会删除冗余逻辑并尽可能的简化逻辑。在综合完成之后，一般会进行逻辑功能的验证仿真，以确保电路设计的功能正确。

工艺映射又分为映射（Mapping）和打包（Packing）两个子步骤。映射子步骤的目标是把基本门级网表转换成由查找表(look-up table, LUT)、多路选择器(multiplexer, MUX)和触发器(flip-flop, FF)等电路功能元件组成的网表；打包子步骤的目标就是在考虑约束(如一个逻辑块中所容纳的查找表、不同的输入信号和时钟的数目等)情况下，把 LUT、MUX 和触发器等电路功能元件进行组合，尽可能地放到一个逻辑块中。打包完成后就可以进行逻辑块的布局。

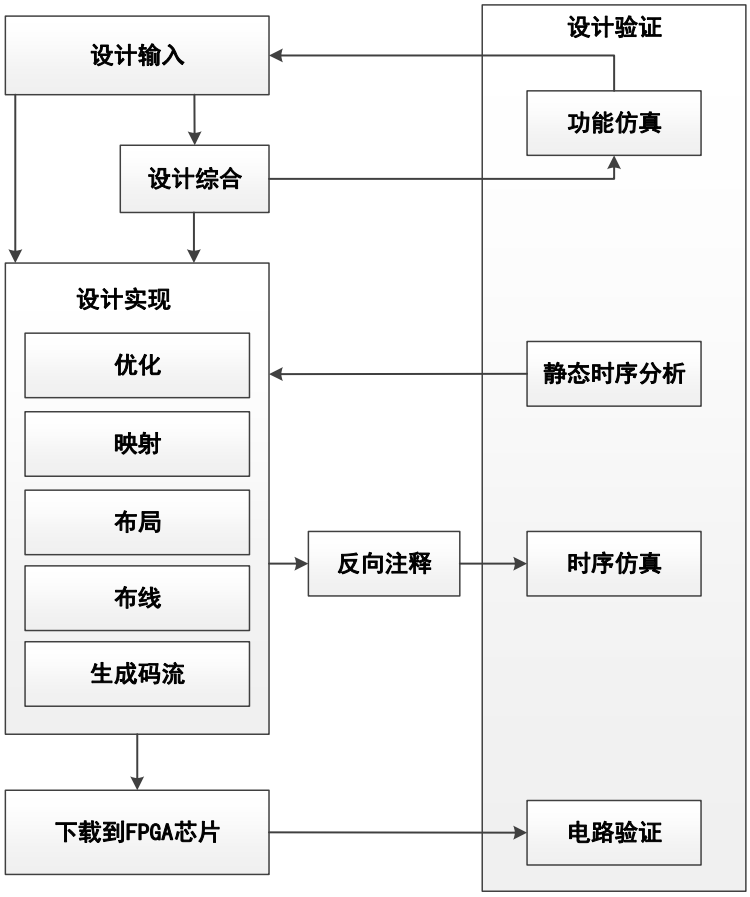


图 1-3 FPGA 软件 CAD 流程图

布局流程确定了实现电路功能所需要的各逻辑单元块（或逻辑簇）在 FPGA 中的位置，它的优化目标是把相连的逻辑单元块靠近放置以最大限度的减少所需的布线资源（线长驱动布局），但有时也要平衡 FPGA 中所需要的布线密度（布通率驱动布局）或者最大限度的提高电路速度（时序驱动布局）。大多数 FPGA 布局工具仍采用与传统 ASIC 布局算法十分相似的算法。例如，一种流行的 FPGA 布局方法是基于一种具有标准度量的迭代移动筛选算法，如模拟退火算法。

一旦确定了电路中所有逻辑单元块的位置，布线器就可以打通所有合适的可编程开关以连接电路需要的所有逻辑单元块的输入和输出引脚。FPGA 布线问题可定义为：找到将布线资源分配给每个线网连线的一种配置方法，同时满足所有其上的时延约束条件。FPGA 与 ASIC 布线算法的虽大差别是：ASIC 布线法尽力将布线资源(电路的布线路径，线路的数目等)的使用最小化。而 FPGA 布线主要致力于优化可用资源的使用以将获得可布线布局的可能性最大化。这一差别来自这样一个事实：FPGA 布线资源使用的最小化不一定意味着在设计中就可以提高可布线能力。换句话说，由于 FPGA 上的布线资源数是固定的，拥挤控制及解决资源竞争是 FPGA 布线法中必须综合考虑的重要因素。一般用有向图来描述

FPGA 的布线结构，即布线资源图，并使用基于有向图的迷宫算法来确定最优路径。

在设计实现的过程中，可以根据产生的时序信息进行时序仿真，用以验证设计电路在时序上是否满足要求。

布局布线完成以后，FPGA 就可以根据所有的配置点信息来生成对应的码流文件，并可以下载到 FPGA 芯片中执行以得到所需的电路功能。

1.3 布局算法的发展现状与国内外研究

布局算法是 FPGA 电路设计中的重要环节，它直接影响到布线阶段的布通率、时延、功耗和面积等关键电路参数。布局简单的讲就是建立技术映射后的线网元素与 FPGA 芯片资源物理位置之间的对应关系。具体的 FPGA 布局就是根据一定的优化目标，将网表文件中描述的电路元素如 CLB、IOB 等分配到芯片上对应的可编程逻辑单元上，然后再通过布线阶段实现整个系统的连接。

布局问题是一个相当难的问题，至今都没有找到能在多项式时间内得到最优解的过程。布局的算法也相当多，这个算法主要分为三类：构造式布局算法，启发迭代式布局算法和解析式布局算法。

构造式布局算法的主要代表是基于划分的最小分割法，它的主要是在一定的约束条件下，通过不断迭代，将完整的布局实例逐步划分为比较小的布局实例，然后再通过自底向上的布局调整，最终完成整体的布局过程。

迭代式算法的主要代表是模拟退火算法，这是现在使用最多的布局算法，因为该算法可直接处理 FPGA 特有的离散坐标约束等问题[2]，并且对多优化目标有天然的适应性，学术界以 VPR（Versatile Placement and Routing）[3]工具为代表，工业界以 Altera 公司的 Quartus II[4]为代表。它是根据物理学中金属在退火过程中的分子运动原理而设计的，算法在某一温度下不断进行逻辑块的交换，并通过代价函数判断是否接受，再继续降温进行。通过这样的迭代过程，可以使布局的结果趋于全局最优解。

解析式布局算法是通过数学抽象，把布局问题转化为数学问题，利用最小二乘法来解决。算法可以得到良好的布局结果，但是对数学问题的抽象却非常困难。

目前使用最广泛的布局算法是模拟退火算法，Altera 和 Xilinx 的 FPGA 可视化设计软件中均采用了模拟退火算法。但是模拟退火算法的退火速度慢，特别是在芯片规模越来越大的今天，因此提高退火速度算法迫在眉睫。国内外都在研究和改进布局算法，还有很多人提出了新的算法，如遗传算法，蚁群算法，粒子群算法等，这些算法的效率都有待实验验证，而本文研究的重点是模拟退火算法。

1.4 研究意义

由于模拟退火算法收敛的速度有限，导致找到全局最优解的的时间比较漫长。

而 FPGA 的布局流程在整个 FPGA CAD 流程中占用了大约四分之一的时间，其随着电路规模的增长，布局时间也在不断上升。同时，FPGA 布局的质量对最终电路布线的布通率、时延和面积等性能都有非常大的影响。因此优化布局算法是 FPGA 的软件设计重点之一。

随着工艺技术的发展，计算机的性能也在不断提升，多核 CPU 的发展也异常迅速，多核计算机已经相当普遍。而串行的程序并没有充分利用计算机的硬件资源，所以并行程序将会蓬勃发展。

模拟退火算法的本质就是基于成百万次的随机交换来找到全局最优解，每次交换的过程都是相同的，因此特别适合使用多核来并行处理，主要处理好每次交换之间可能存在的冲突即可。

目前已有一些研究开始尝试挖掘布局算法的并行性，但现有的并行算法都不能在维持布局质量、满足布局的限制和达到高的加速比之间有很好表现，因此模拟退火算法的并行化仍是一个亟待解决的问题。

本文希望通过并行在不显著影响布局质量且满足布局算法的相应约束条件的前提下的情况下得到较好的加速比。

1.5 论文主要工作与组织

1.5.1 本文内容

本文的主要内容可以概括为以下三个部分点：

1）简单介绍岛式通用 FPGA 的结构，并针对该结构研究现有的模拟退火算法的详细实现方法；

2）设计一种可以多核并行的模拟退火布局算法，并且使得该算法在取得很好的加速比的同时维持原有的高质量布局水平，并对该算法进行优化。

3）在现有的 VPR 布局工具的基础上进行改进，实现多核并行的模拟退火算法 CAD 工具，并通过工具来测试该算法的正确性。

本文的创新之处在于：

1）明确了布局算法的重要目标，即确定性和串行等价性，而过去很多研究在追求并行加速比和布局质量的时候忽视了程序的布局结果确定性和不同硬件平台的等价性；

2）本文基于 MapReduce 框架提出了一种实用的基于多核交换的 FPGA 并行布局

算法框架，并对冲突的解决和算法的运行时间优化提出了行之有效的方法；
3）本文在 VPR 基础上实现了基于第四章描述的并行算法的软件工具，并应用于 Virtex-4 平台，可以输入 Xilinx 软件生成的 XDL 进行多核布局测试，证实了算法的实用性。

1.5.2 本文组织

本文的主要章节安排如下：
第一章介绍论文课题的研究背景和研究意义、FPGA 的内部结构、设计流程和布局算法及其发展现状，最后是本文的内容和章节安排；
第二章介绍了 FPGA 的三种主要布局算法：构造式、解析式和迭代式，并详细研究基于 VPR 的 FPGA 的模拟退火布局算法；
第三章主要介绍了多核计算的相关概念，并介绍了两个常见的多核硬件架构和相应芯片，然后介绍了多核相关的编程语言，为多核算法的工具实现进行铺垫；
第四章介绍了已有的一些基于模拟退火的并行布局算法，并指出其优点和缺点，然后提出了一种新的基于多核交换的并行 FPGA 布局算法，并详细阐释了算法的实现和优化；
第五章实现了基于第四章阐述的并行布局算法的软件工具，并在多核硬件平台上进行了相关测试；
第六章对全文进行了总结，并对进一步的研究工作进行了展望。

1.6 本章小结

本章介绍了本文的研究背景和研究意义，再简要介绍了通用的岛型 FPGA 结构和 FPGA 的 CAD 设计流程，然后介绍了目前三种主流的 FPGA 布局算法，最后介绍了本论文的主要研究工作和章节的组织安排。

第二章 布局算法综述

2.1 FPGA 的布局问题和优化目标

布局是芯片物理设计中一个非常重要的过程，是由技术映射向布线过程转换的中间过程，因此布局质量的好坏直接决定芯片的最终性能。FPGA 的布局问题是在满足一定约束的前提下，将技术映射后的线网与 FPGA 芯片上的物理资源建立位置对应关系的过程。布局阶段主要关心的问题如下：

1） 确保布线工具能够完成布线（优化互连线的拥塞）；

2） 使电路的互连线长度最小（优化线长）；

3） 使得关键路径上的延时最小（优化时延）；

4） 芯片的分布尽量密集（优化面积）；

5） 芯片功耗小（优化功耗）；

由于布线也是一个相当复杂的问题，因此不可能在布局阶段就能为后面的布线阶段考虑得很完整，因此布局阶段只能粗略的考虑布线因素，如互连线长度等。同时，应该注意到上述目标并不是都能达到的，因此布局问题是多目标优化问题，需要在其中进行折中。

2.2 FPGA 的布局算法

布局问题从提出到现在已经经过了二十年时间，其中出现了很多的算法，并且直到今天人们还在寻找最优的布局算法。常见的布局算法主要有三类：构造式布局算法、解析式布局算法和划分式布局算法。

2.2.1 构造式布局算法

构造性的布局算法主要分为两类：自顶向下的基于划分的算法和自底向上的基于结群的算法。而基于划分的布局算法由于运算速度快，求解规模大，在布局领域得到较为广泛的应用。

基于划分的算法的思想是：在一定的约束条件下，将电路和布局区域进行同步的划分，每个划分的子电路对应一个电路子区域。然后通过不断的迭代过程，将完整的布局实例逐步划分为比较小的原子布局实例。例如，设在一个矩形域 R(A)内，存在一个与该区域对应的子电路元素集 E(A)，每个属于 E(A)的元素都在 R(A)中有对应的位置。再设割线 C 将 R(A)分割为两个子区域 R1(A)和 R2(A)，将元素 E(A)分为 E1(A)和 E2(A)两个子集，如此重复下去。划分的目标函数为子电路之间的连线数目。划分完成之后，再通过自底向上的布局调整，最终完成整

体的布局过程。划分算法如图 2-1 所示。

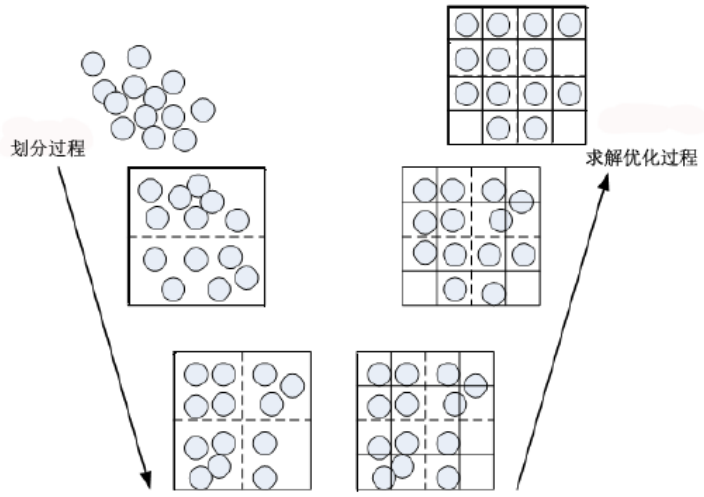


图 2-1 划分布局示意图

划分算法可以分为二划分算法和多划分算法两种，其中二划分算法是多划分算法的基础。划分算法的伪代码如下：

```
//Algorithm Min-Cut(S,n,C)
//S is the layout surface
//n is the number of cells to be placed
//C is the connectivity matrix
Begin
    if(n<=0) Then place-cells(S,n,C);
    else begin
        (S1,S2)=cut-surface(S);
        (n1,c1),(n2,c2)=partition(n,C);
        Call Min-cut(S1,n1,c1);
        Call Min-cut(S2,n2,c2);
    endif
End
```

2.2.2 解析式布局算法

基于数学解析的布局算法是将二次线长模型抽象成数学模型，然后通过数学规划布局的方法来完成布局，这种算法可以在数学上严格的证明出算法的布局质

量。解析式布局算法的核心在于给布局问题定义恰当的解析目标函数，并通过数学优化方法最小化目标函数。基于数学解析的布局算法主要代表有二次规划布局算法。二次规划布局算法首先根据线网结构抽象出一个凸规划的问题，然后解这个凸规划问题从而求出全局最优解。该算法的主要目标函数是二次线长模型，采用拉格朗日乘子方法来求解非线性规划，不依赖于初始值，能够并行求出所有的单元位置，速度快，已被广泛应用于 VLSI 的布局。

由于 FPGA 的结构和标准单元 VLSI 的结构有很大的不同，因此有关标准单元的物理设计方法不能直接应用到 FPGA 的设计上，因此有关 FPGA 的数学规划布局算法的研究成果还比较少，目前比较成功的基于数学解析的布局算法是二次规划 FPGA 布局法（QPF，Quadratic Placement Tools for FPGAs）。QPF 算法主要分为三个阶段。第一阶段中通过重复建立、修改和求解线性方程式来得到好的初始布局；然后在第二个阶段应用线长优化策略来改善第一阶段得到的布局；第三个阶段是低温模拟退火阶段，进一步优化总线长。

2.2.3 迭代式布局算法

迭代式布局算法的基本思想是快速形成问题的初始解，然后在初始解的基础上，不断地进行迭代过程使结果朝布局优化目标接近，从而最终达到全局最优解。

为了提高迭代法的效率，需要解决三个方面的问题：确定迭代变量，建立迭

代关系式和对迭代过程的控制。布局过程中每一个迭代变量均对应一个布局结果。迭代关系式是指从当前解推导出下一个解的公式或关系，迭代关系式直接决定迭代算法的效率。为了平衡求解过程中的时间与空间之间的平衡，需要对迭代过程进行控制。一般来说，通过迭代次数或者解的精度来控制迭代过程。

二维 FPGA 中最常见的迭代式算法是模拟退火算法。模拟退火算法是借鉴工业界冶金的退火原理进行的。当金属加热以后，再以某个特定的速率冷却下来，原子自然地结合成一种低能量结构，但是错误的冷却调度会导致一种错误的结构。在模拟退火算法中，温度是根据电流流动过程中元素的移动概率来测量的。冷却进度是温度值序列在时间上的变化概率。当温度升高，电流的反复过程中就有更多的元素发生移动；而温度降低时，发生移动的元素相对减少。

因此模拟退火算法的基本思想为：先把逻辑单元块随机分配到 FPGA 从而得到一个初始分布，随后进行大量的移动或者局部优化来改进布局。移动的过程就是随机选取一个逻辑模块，随机分配一个新的位置，计算由移动造成的代价函数差值，如果代价函数减少，就接受移动；若增大，仍然存在接受的可能。接受一个使布局变差的移动带来的爬坡能力，使得模拟退火避免收敛到成本函数上的局部最优解。

2.2.4 优缺点比较

构造式布局算法的优点是求解速度快，规模大，但是由于将电路不断划分，电路的不完整性难以保证布局质量的可靠性。

解析式布局算法目前主要应用于 ASIC 布局，由于 FPGA 的结构和标准单元 VLSI 的结构有很大的不同，因此有关标准单元的物理设计方法不能直接应用到 FPGA 的设计上。

模拟退火算法的优点是能越过迭代过程中的局部最优解而得到接近于全局最优解，同时，该算法可直接处理 FPGA 特有的离散坐标约束等问题，缺点是需要花费较长的时间。但是随着硬件性能的改善和多核 CPU 的普及，算法的运行时间可以大大改善。所以模拟退火算法应用最为广泛。

目前学术界研究最为广泛的 VPR 算法采用的即为模拟退火算法。VPR 是一款 FPGA 布局布线的通用软件，由加拿大多伦多大学的 Vaughn Betz 教授等人提出，提出了较为完整的解决方案[3]。其作者改进了传统的退火表，而使用自适应的退火表，使得退火时间大大缩短，且布局质量较好。

2.3 VPR 布局算法详解

2.3.1 算法流程

模拟退火算法的基本步骤为：

1) 初始化过程：初始温度T（充分大），初始的解状态（算法迭代的起点）和每个温度T下的迭代次数L；
2) 内循环过程：对k=0,1,2,...,L，做下面（3）到（6）步；
3) 产生新解S′;
4) 计算增量$\Delta C = C(S') - C(S)$，其中C(S) 为代价函数；
5) 若$\Delta C < 0$，则接受S′作为新的当前解，否则以概率exp（-ΔC/T） 接受S′作为新的当前解。
6) 如果满足终止条件则输出当前解作为最优解，结束程序，终止条件通常取为连续若干个新解都没有被接受；
7) 按退火表减小温度T，且T大于终止温度，然后转第二步；

退火算法的伪代码如下：

```
S=RandomPlacement();                    //Initial random place
T=InitialTemperature();
D_limit=InitialD_limit();
```

```
while(ExitCriterion() == false) {              // Outer Loop
    while(InnerExitCriterion == false) {       // Inner Loop
        S_new = GenerateViaMove(S,D_limit);
        C_delta = Cost(S_new) - Cost(S);
        r = random(0,1);
        if(r < exp(-C_delta / T)) {
            S = S_new;
        }
    }                                          // End inner loop
    T = UpdateTemperature();
    D_limit = UpdateD_limit();
}                                              // End outer loop
```

算法分为内外两层循环：内循环在特定的退火温度上进行有限次的 CLB 交换，如果交换后得到的布局比原先的布局质量好，即接受新的布局；但如果交换后得到的布局比原先的布局质量差，也以一定的概率接受布局，从而避免算法过早的陷入局部最优解而无法找到全局最优解。交换次数达到一定后，交换已无法改变布局质量，因此内循环将会退出。更新退火温度后，重新在新的退火温度上进行交换，如此进行直到达到退火终止条件，此时完成整个布局算法。

2.3.2 算法控制条件

2.3.2.1 代价函数的计算（线性阻塞函数）

VPR 模拟退火算法中计算代价函数采用线性阻塞函数，表达式如下：

$$\text{Cost} = \sum_{n=1}^{N_{nets}} q(n)\left[\frac{bb_x(n)}{C_{av,x}(n)} + \frac{bb_y(n)}{C_{av,y}(n)}\right] \qquad \text{式(2-1)}$$

其中，Cost 为当前代价函数值，N_{nets} 为线网数量,总成本为每条线网的成本之和。一个线网连接若干引脚，可以是一条线，也可以是一个网络，*bbx* 和 *bby* 是指线网在 *x* 和 *y* 方向上的曼哈顿跨度，如图 2-2 所示即为一条线网，其 *bbx* 为 4，*bby* 为 3。$C_{av,x}$ 和 $C_{av,y}$ 为线网所跨范围内 x 和 y 方向上的平均轨道数。q(n) 为线网扇出超过3的非线性补偿因子，与扇出数密切相关，当删除小于50时为常数，可以通过查表得到，当扇出大于50时需要进行修正。

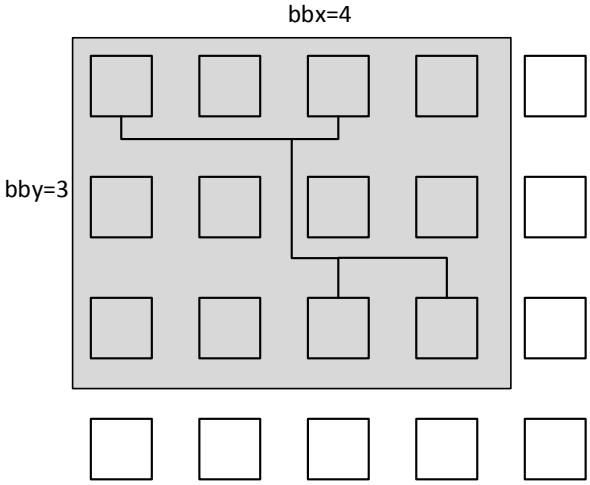


图 2-2 线网的曼哈顿跨度

2.3.2.2 循环终止条件

外循环的终止条件为：

$$T < 0.005 * \frac{\cos t}{num\_nets} \qquad \text{式(2-2)}$$

其中，cost 为当前代价函数值，num_nets 为线网数量。由于等式右边一般是很小的数，因此模拟退火在温度非常低的时候会终止。因为温度很低时,不能够改善布局的交换几乎不能够被接受,即继续退火也不能够跳出当前最优解,这样就失去了继续退火的必要。

内循环的终止条件为：

$$move\_\lim = inner\_num * N_{block}^{4/3} \qquad \text{式(2-3)}$$

其中，$move_\lim$为内循环迭代的次数，$inner\_num$ 是一个用户可以设置的参数,缺省值为 10，N_{blocks} 表示电路中逻辑单元的个数，因此等式右边为常数，每个温度上的退火时间相同。

本来按照模拟退火理论,内循环迭代应当是在布局在该温度达到/热平衡 0 时能够中止,然而,热平衡的判断需要较大的计算量,也难以控制模拟退火的算法复杂度。因此 VPR 采用了固定的内循环迭代次数，仍然可以得到较好的布局质量，同时缩短了退火时间。

2.3.2.3 退火初始温度

退火表中初始温度的设置采用了 Huang 等人的方法，即首先创建一个随机的

电路布局再对逻辑单元块和 IO 端口进行 N_{blocks} 次移动（成对交换），计算这 N_{blocks} 中不同配置成本的均方差 σ，则初始温度被设置为 20σ，这样可以保证在退火初始阶段几乎每次移动都能够被接受。

2.3.2.4 温度更新的策略

太高的温度使得不能改善布局的交换更容易被接受,降低布局质量;相反,太低的温度使得退火容易陷入局部最优解。而 VPR 力求使接受率在 44%左右，以此调节温度下降速度。因此在 VPR 中温度是动态调整的：$T_{new} = \alpha T_{old}$

其中 α 因子由接受移动的百分比 R_{accept} 来调控，两者关系如下表：

表 2-1 接受移动的百分比与温度调节因子的关系

Fraction of moves accepted（R_{accept}）	α
R_{accept}>0.96	0.5
0.8<R_{accept} ≤ 0.96	0.9
0.15 <R_{accept} ≤ 0.8	0.95
R_{accept}≤ 0.15	0.8

2.3.2.5 随机交换范围

同样为了使接受率稳定在 44%左右，我们用接受移动的百分比 R_{accept} 来控制交换的范围。交换范围的表达式为：

$$D_{limit}^{new} = D_{limit}^{old} \times (1 - 0.44 + R_{accept}^{old}) \qquad \text{式(2-4)}$$

在一开始，交换范围很大，一般是整个芯片，随着退火的进行，如果交换接受率小于 0.44 就减小交换范围；如果交换接受率大于 0.44 就增大交换范围，从而使交换接受率稳定在 44%左右。

2.4 本章小结

本章首先介绍了布局算法的问题和优化目标，然后介绍了 FPGA 布局的三种主要算法：构造式、解析式和迭代式布局算法，最后以 VPR 中实现的模拟退火算法为例，详细讲述了该算法的实现细节，便于下一章的并行化实现的展开。

第三章 多核计算概述

3.1 多核计算

并行是一个广义的概念，从定义上来讲，并行处理是一种有效的强调开发计算过程中并行事件的信息处理方式。并行性包含三种含义：

同时性指两个或多个事件在同一时刻发生在多个资源中；

并发性指两个或多个事件在同一时间间隔内发生在多个资源中；

流水线指两个或多个事件发生在可能重叠的时间段内。

根据实现层次的不同，并行计算可以分为几种方式。如图 3-1 所示，单核指令级并行（Instruction-Level Parallelism）是最微观、最底层的并行层次，它让单个处理器的执行单元可以同时执行多条指令；向上一层是多核并行，即将多个处理器核心集成在一个芯片上，实现线程级并行（Tread-Level Parallelism）；再往上是多处理器并行，即将多个处理器安装在一块电路板上，从而实现线程和进程级并行；最后，集群或分布式并行是借助网络使每台独立的计算机成为一个节点，以此来实现更大规模的并行计算。

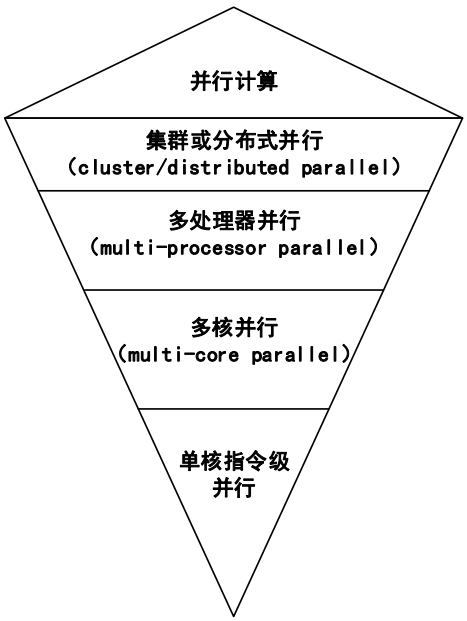


图 3-1 并行层次图

伴随着并行架构的发展，并行算法也在不断成熟与完善。在二十世纪七八十年代出现了许多采用不同互连架构和存储器模型的 SIMD（Single Instruction Multiple Data）机，同期也涌现出了许多优秀的并行算法。二十世纪九十年代中期以后，并行算法的研究开始以应用为向导，更加关注到应用层面，在对并行算法进行研究、设计和分析的同时，也越来越多地兼顾到并行机体系结构以及与之相关的并行应用程序的设计。

当前，主流的计算机处理器通常是拥有二至八个核心的多核 CPU 以及拥有数十乃至上百个核心的众核 GPU，并且 CPU 与 GPU 的核心数量呈持续上升的趋势。近年来随着多核和众核处理器越来越普及，并行处理也逐渐走入了寻常百姓家，已经不仅仅是大型机和集群的专利。

3.2 CPU 多核并行

随着计算机技术的迅猛发展，现代社会对于计算能力和信息处理速度的要求也在不断地提高，这使得提高 CPU 处理能力来提升系统性能成为了解决问题的关键。提高处理器的工作频率以及增加指令级并行是提高 CPU 单个核心性能的主要方法，但是这两种传统的手段在实际运用中都会遇到问题。晶体管的尺寸随着制造工艺的不断提高越来越接近原子的数量级，伴随而来的便是更加显著的漏电流问题，芯片集成的规模已经逐步接近其承受极限，尤其是散热和功耗等问题，使得处理器的频率提高越来越艰难。另一方面，通用计算中的指令级并行并不多，当和投入的大量晶体管成本比较起来，费尽苦心设计所获得的性能提高往往显得很不划算。如果要实现更高层次的指令级并行，就必须采用复杂的猜测执行机制和大块的缓存，以此来保证指令与数据的命中率。然而，现代计算机技术的发展已经使得 CPU 的分支预测正确率能够达到 95%以上，在这一方面可供提高的空间不大。缓存的大小对 CPU 的性能影响很大，但是倘若继续增加缓存大小，最多也只能让真正用于计算的少量执行单元满负荷运行，这对于进一步提高 CPU 性能的影响微乎其微。

由于存在上述因素，单核 CPU 的性能提升一直受到限制，没有很大的突破空间。因此，CPU 厂商开始寻找新的方式来突破计算能力的限制。多核心 CPU 解决方案的出现，给人们带来了新的希望，CPU 厂商开始在单块芯片内集成更多的处理器核心，使 CPU 向多核方向发展。Intel 和 AMD 于 2005 年正式向主流消费市场发布了各自的双核 CPU 产品。当年 4 月，Intel 推出简单封装双核的奔腾 D 和奔腾四至尊版 840，AMD 在之后也发布了双核皓龙（Opteron）和速龙（Athlon）处理器。2006 年 7 月 23 日，Intel 基于酷睿（Core）架构的处理器正式发布，宣告了多核处理器的应用范围从大型机扩展到了家用机。同年 11 月，Intel 又推出面向服务器、工作站和高端个人电脑的至强（Xeon）5400 以及酷睿双核和四核至尊版系列处理器，与上一代台式机处理器相比，酷睿 2 双核处理器在功耗降低 40%的基础上提升了 40%的处理性能。

3.3 多核 CPU 架构

CPU 芯片的诸多厂商都有各自的多核芯片产品，然而他们在设计时采用的方

法各不相同，但都得到了有效的实现，与其他设计相比都有各自的优缺点。目前的片上多核处理器多个 CPU 核心之间数据共享与同步的通信机制主要分为两种：

1) 总线共享 Cache 结构：每个 CPU 内核拥有共享的二级或者三级 Cache，用于保存比较常用的数据，并通过连接核心的总线进行通信；
2) 基于片上互连的结构：每个 CPU 核心具有独立的处理单元和 Cache，各个 CPU 核心通过交叉开关或片上网络等方式连接在一起。

上述两种结构分别对应均匀存储访问(Uniform Memory Access, UMA)和非均匀存储访问(Non-Uniform Memory Access, NUMA)这两种访存模型。

3.3.1 UMA 访存

UMA 结构的简化框图如图 3-2(a)所示。这种结构中，多个处理器共享单一的内存，每个处理器的内存访问时间与其他处理器一致，因此也常被称为对称（共享内存）多处理器（SMP）。主要特点如下：

物理存储器被所有节点共享；

所有节点访问任意存储单元的时间相同；

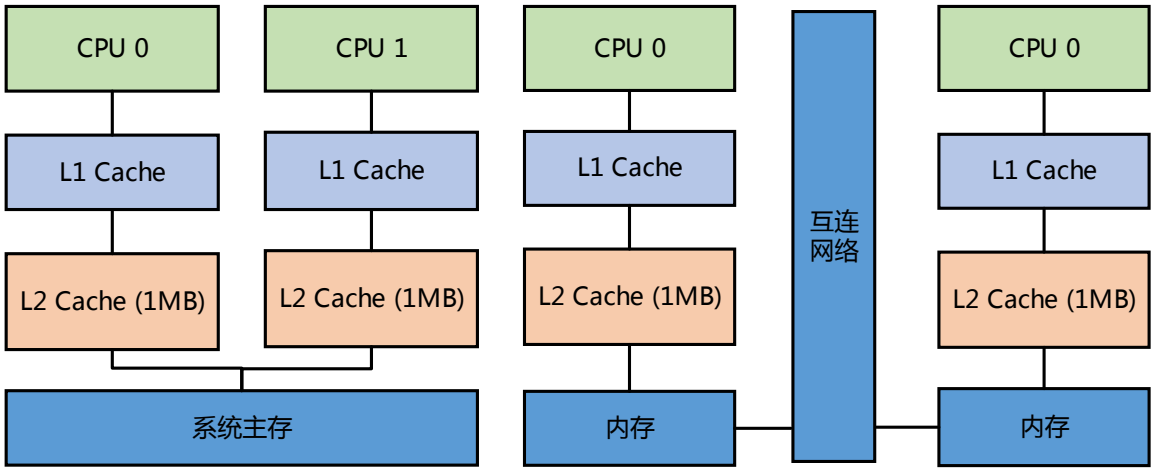
发生访存竞争时，仲裁策略平等对待每个节点，即每个节点机会均等；

各节点的 CPU 可带有局部私有高速缓存；

外围 I/O 设备也可以共享，且每个节点有平等的访问权利；

共享地址空间是从物理角度共享的；

唯一的内存控制器可能成为性能限制的瓶颈。



(a) UMA 访存模型简化框图　　(b) UMA 访存模型简化框图

图 3-2 两种多核内存模型的结构框图

3.3.2 NUMA 访存

NUMA 结构的简化框图如图 3-2(b)所示。这种结构中，每个处理器有自己的内存块，但每块内存共享同一个地址空间，也就是说，两个处理器上相同的物理地址指向内存中相同的位置。在 UMA 和 NUMA 两种结构中，处理器均共享内存。主要特点如下：

物理存储器被所有节点共享，任意节点可以直接访问任意内存模块；

节点访问内存模块的速度不同，访问本地存储模块的速度一般是访问其他节点内存模块的 3 倍以上；

发生访存竞争时，仲裁策略对节点可能是不等价的；

各节点的 CPU 可带有局部私有高速缓存 （cache）；

外围 I/O 设备也可以共享，但对各节点是不等价的；

共享地址空间是从逻辑角度共享的；

3.3.3 AMD Multicore Opteron

AMD 公司的双核 Opteron 芯片是 NUMA 类型的典型芯片。双核 Opteron 是最基本的配置，它采用了 AMD 实现多核架构的基本方法。图 3-3 是 Opteron 的简化框图。

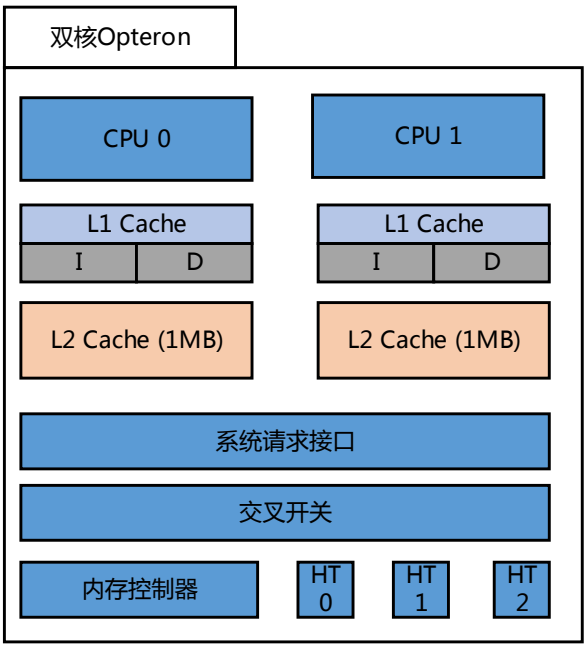


图 3-3 Opteron CPU 架构简化框图

Opteron 处理器摆脱了基于总线的架构，它使用直连架构结合 HyperTransport(HT) 技术来避免基本的前端总线(FSB)，后端总线(BSB)和外围设

备互连(PCI)结构带来的一些性能瓶颈。Opteron 直连架构将处理器、内存控制器和 I/O 直接连接到 CPU。这种专用连接方法避免了 CPU 和内存控制器间基于总线通信的潜在性能问题。同时由于连接是专用的，即每个内核都直接连接到自己的内存控制器和 I/O 控制器，因此避免了竞争问题。

HyperTransport 技术是一种高速、低延时、点对点的连接，旨在提高电脑、服务器、嵌入式系统，以及网络和电信设备的集成电路之间的通信速度。它的速度比某些现有技术高出 48 倍。HyperTransport 有助于减少系统之中的布线数量，从而能够减少系统瓶颈，让当前速度更快的微处理器能够更加有效地在高端多处理器系统中使用系统内存。

Opteron 使用 HT 作为芯片-芯片的 CPU 和 I/O 间的互联，使用 HT 连接的部件是以端到端的方式连接的，因此能够直接相互通信，不需要数据总线。

3.3.4 Intel Core 2 Duo

Intel 的 Core 2 Duo 只是 Intel 的多核处理器中的一个系列。有些为双核，其他的为四核。有些多核处理器通过超线程进行了增强，使得每个内核成为两个逻辑处理器。Intel 的第一款多核处理器是 2005 年推出的的 Intel Pentium Extreme Edition。它是双核的，且支持超线程，使系统具有 8 个逻辑内核。2006 年推出的 Core Duo 多核处理器提供了两倍的内核，而且功耗更低。[5]

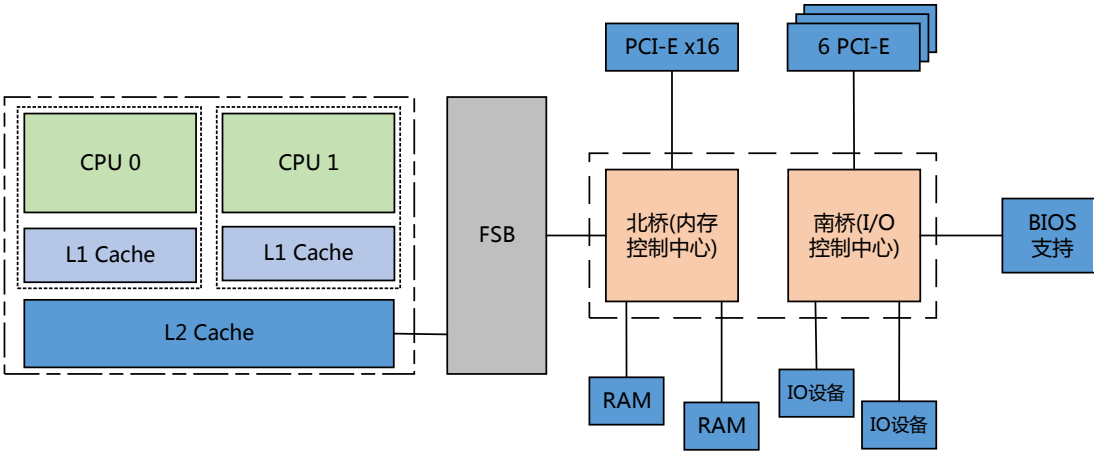


图 3-4 Intel Core 2 Duo 处理器主板结构框图

图 3-4 显示了 Intel Core 2 Duo 的主板框图。Core 2 Duo 处理器有两个 64 位内核，每个内核有 64KB L1 Cache。L2 Cache 在内核间共享。L2 Cache 最多可以达到 4MB。两个内核都最多可以利用 100%的 L2 Cache，这意味着当另一个内核未被充分利用。

其中，北桥作为内存控制中心，直接通过前端总线(FSB)与 CPU 通信。它将

CPU 同高速设备连接起来，如主存。它还通过一条内部总线将 CPU 同 PCI-E 插槽及南桥连接起来。南桥是 I/O 控制器，负责主板上速度较慢的部分。

3.4 多核编程语言

随着 CPU 从单核发展为多核，多线程编程吸引了日益增多的关注目光，也越来越多地受到程序员们的重视。通过多线程编程，程序员不仅可以在多个 CPU 核心间实现线程的并行运行，还能够通过超线程等技术使每一个 CPU 核心内的资源得到更好的利用，充分发挥 CPU 强大的计算能力。除了操作系统提供的线程管理 API，程序员还可以通过一些库或者语言扩展等方法来实现多线程并行计算。在过去的几十年里，人们相继提出了很多并行编程语言和模型。

3.4.1 MPI(Message Passing Interface，消息传递接口)

MPI 是目前最重要的并行编程工具，它具有移植性好、功能强大、效率高等多种优点，而且有多种不同的免费、高效、实用的实现版本，几乎所有的并行计算机厂商都提供对它的支持，这是其它所有的并行编程环境都无法比拟的。MPI 于 1994 年产生，虽然产生时间相对较晚，由于它吸收了其它多种并行环境的优点，同时兼顾性能、功能、移植性等特点，在短短的几年内便迅速普及，成为消息传递并行编程模式的标准。MPI 其实就是一个库，共有上百个函数调用接口，在 FORTRAN77 和 C 语言中可以直接对这些函数进行调用 MPI 提供的调用虽然很多，但最常使用的只有 6 个，只要会使用 FORTRAN 77 或者是 C 就可以比较容易地掌握 MPI 的基本功能，只需通过使用这 6 个函数就可以完成几乎所有的通信功能。由于 Fortran90 和 C++的使用也十分广泛，MPI 后来又进一步提供对 Fortran 90 和 C++的调用接口，这更提高的 MPI 的适用性。

3.4.2 OpenMP

1997 年，一些厂商携手合作，在硬件制造商 Silicon Graphics 的支持下组成了新的线程接口。他们共同的问题是主要的时间操作系统全部利用了完全不同的线程编程方法。UNIX 使用 Pthreads、Sun 使用 Solaris 线程、Windows 使用自己的 API、而 Linux 则使用 Linux 线程（直到其后来采用 Pthreads）。委员会希望设计出能够支持不需要改变代码库即可在 Windows 和 UNIX/Linux 上平等运行的 API。1998 年，它提供了第一个称为 OpenMP 的 API 规范（在这些日子里，“开放”这个词汇与多厂商支持的概念相关，是指开放系统，而不是现在的开放源代码的含义。）

OpenMP 规范包括 API、一组编译指示、以及对 OpenMP 指定环境变量的几种设置。随着对标准的进一步修订，OpenMP 最实用的特性之一就是那一组编译指示，这一点逐渐明朗。通过明智地使用这些编译指示，单线程程序不需要向 API 或环境变量求助即可实现多线程。通过 OpenMP 2.0 的最新版本，OpenMP 架构审核委员会（ARB）是提出 OpenMP 规范的委员会的正式名称，显然开发人员使用编译指示而不是 API 作为优先选择。以下关于编译指示的定义（摘自微软的文档）是最清晰的解释之一：“#pragma 指示为每个编译器提供了一种方法，能够提供机器指定和操作系统指定特性，同时保留与 C 和 C++ 语言的整体兼容性。根据定义，编译指示是机器指定或操作系统指定的，通常所有的编译器都不同。编译指示最常用于 C 和 C++ 中，格式如下：#pragma token-string。”

编译指示的主要方面是如果编译器不能识别指定的编译指示，则它必须忽略（根据 ANSI C 和 C++ 标准）。因此，将库指定的编译指示放入代码中是安全的，不需要担心如果采用不同的工具包进行编译会将代码破坏。OpenMp 支持的编程语言包括 C 语言、C++和 Fortran；而支持 OpenMp 的编译器包括 Sun Compiler，GNU Compiler 和 Intel Compiler 等。OpenMp 提供了对并行算法的高层的抽象描述，程序员通过在源代码中加入专用的 pragma 来指明自己的意图，由此编译器可以自动将程序进行并行化，并在必要之处加入同步互斥以及通信。当选择忽略这些 pragma，或者编译器不支持 OpenMp 时，程序又可退化为通常的程序(一般为串行)，代码仍然可以正常运作，只是不能利用多线程来加速程序执行。

3.4.3 POSIX 线程(POSIX threads)

Pthreads 是 IEEE（电子和电气工程师协会）委员会开发的一组线程接口，负责指定便携式操作系统接口（POSIX）。Pthreads 中的 P 表示 POSIX，实际上，Pthreads 有时候也代表 POSIX 线程。基本上，POSIX 委员会定义了一系列基本功能和数据结构，它希望能够被大量厂商采用，因此线程代码能够轻松地在操作系统上移植。委员会的梦想由 UNIX 厂商实现了，他们都大规模实施 Pthreads。（最著名的例外就是 Sun，它继续采用 Solaris* 线程作为其主要线程 API。）由于 Linux 的采用和移植到 Windows 平台，Pthreads 的可能性一直被进一步扩展。

Pthreads 定义了创建和操纵线程的一整套 API。这些行为包括创建和终止线程、等待线程完成、以及管理线程之间的交互。后面的目录中存在各种锁定机制，能够阻止两个线程同时尝试修改相同的数据值：互斥体、条件变量和信号量。（从技术上讲，信号量不是 Pthreads 的一部分，但是它们从概念上更接近于与线程合作，而且可用于 Pthreads 能够运行的所有系统上。）

在类 Unix 操作系统（Unix、Linux、Mac OS X 等）中，都使用 Pthreads 作为

操作系统的线程。Windows 操作系统也有其移植版 pthreads-win32。POSIX 线程具有很好的可移植的性，使用 Pthreads 编写的代码可运行于 Solaris、FreeBSD、Linux 等平台，Windows 平台亦有 pthreads-win32 可供使用。Pthreads 定义了一套 C 语言的类型、函数与常量，它以 pthread.h 头文件和一个线程库实现。

3.5 本章小结

本章首先介绍了多核计算的概念，并详细介绍了 UMA 和 NUMA 两种多核 CPU 内存模型，最后介绍了当前的一些主流多核编程语言。同一个并行算法在不同的硬件架构上有不同的表现，这也是本文必须要考虑的因素。

第四章 模拟退火布局算法的并行化研究

在整个 FPGA 软件执行流程中，布局大约占据了大约四分之一的时间，因此在电路规模不断增大的今天，模拟退火算法速度的提高刻不容缓；同时，FPGA 的布局结果对于后续的布线结果影响很大，器件的摆放位置大体上决定了 FPGA 的全局布线结构，因此在布局时还必须保证 FPGA 软件的布局质量。综合权衡速度和布局质量，成为并行化研究算法的主要难题。

4.1 并行化研究目标

并行化布局算法的设计初衷是为了减少布局的运行时间，但是不代表我们可以牺牲布局的质量。因此，一个真正使用的布局算法应该遵循以下原则：

1) 算法必须能够在通用的硬件处理器平台上运行，那些需要借助特殊硬件如图形处理器（GPU，Graphic Processing unit）或工作站等并行算法很难在实际中推广；
2) 布局质量的下降必须在能接受的范围内（一般 5%为容忍点），即布局质量不能远差于现有的布局工具；
3) 布局算法必须是确定的算法。什么是确定的算法？就是当我们运行同一个布局算法重新很多遍，每次都能得到相同的布局结果。这一点在过去的研究中不太重视，但是却相当重要。例如，在某次运行过程中算法出现了 Bug，我们希望能重现 Bug 发生的情形。同时，确定性有利于我们进行算法的测试；
4) 布局算法必须是串行等价的，这是最严格的约束。我们的算法需要在单核、双核、四核以及更多核上得到相同的运行结果。串行等价性的算法有很多的优势。首先，保证串行等价性就保证了第二点要求的满足；其次，串行等价性可以简化测试；最后，串行等价性可以使我们的算法适应更多的平台，用户在不同的电脑上可以得到相同的布局结果；

4.2 并行化研究现状

现有的并行化模拟布局算法主要归纳如下：

文献[6]等尝试将模拟退火算法中的移动交换过程进行流水线切割，例如双核系统 C0 和 C1，使用 C0 核负责交换，使用 C1 核负责交换后的代价评估，这种算法是满足串行等价性的，但是由于流水线深度有限，难以实现很好的加速效果。

文献[7][8]等尝试将布局区域进行划分，在各个子区域中独立的进行布局，并且忽略在代价评估时由于区域划分而导致的代价函数失真，同时定期在各子区域间进行布局结果的同步。这种算法由于各子区域的独立性从而有很大的并行加速，但是很显然布局质量明显下降，因为交换被限制在子区域中。

文献[9]等尝试用分支预测的方法来进行流水线的优化。因为一次交换的接受与否决定了下一次提出的交换逻辑块能否继续交换。如果这一次的交换的逻辑块是 A 和 C，而下一次提出的交换逻辑块是 C 和 D，那么如果这一次交换被接受，下一次交换必然要被拒绝，这就产生了“冲突”，因此需要重新提出交换。而分支预测一共需要三个核 C0、C1 和 C2，当 C0 核正在处理交换 M0 的时候，C1 核在假设交换 M0 被接受的情况下提出新的交换迭代，而 C2 核则在假设交换 M0 被拒绝的情况下提出新的交换迭代。这种方法理论上有很好的加速比，但是由于实际处理器核之间的同步开销太大使得加速效果很差。这种算法也是串行等价性的。

文献[10]等尝试使用异步马尔科夫链的方法来并行化模拟退火算法，即多个线程采用不同的马尔科夫序列进行独立查找，在一段时间后线程查找结果进行汇总，找出最优的搜索结果，然后所有线程又同时从当前的最优结果出发，继续沿不同的马尔科夫序列进行查找。如果将每个线程的查找次数减为串行查找次数的 1/N，则理论上是可以达到 N 倍的加速比，而且布局质量基本可以和串行版本持平。而且由于算法采用同步的线程通信策略，因此可以满足布局结果的确定性。但是由于核数越多，参与搜索的线程越多，最优化结果不相同，因此该算法不满足串行等价性。

文献[11]提出了两种基本的模拟退火算法并行化实现方法。第一种是是基于流水线操作的管道式并行布局算法。算法将模拟退火的一次交换分为两个阶段，第一阶段提出交换的逻辑块，占据 40%的时间；第二阶段计算交换的代价差并判定是否接受，占据 60%的时间。两个核分别处理第一阶段和第二阶段。第一个核完成第一阶段之后将 Move 放到队列中并继续提出下一次交换；第二个核从队列中取出 Move 并计算、检测冲突、判定和提交。如果检测冲突，则该次 Move 需要重新提出，同时需要检测队列中已经提出的 Move 是否存在冲突。这种算法是确定性和串行等价性的，但是流水线的方式深度有限，难以扩展，因此加速比是有限的，本算法最理想的加速比是 1/0.6=1.7 倍，实际上只能达到 1.3 倍左右的加速比。第二种方法是基于多核并行交换的并行布局算法。该算法的基本思想为：把模拟退火算法中一次迭代的过程分为两个阶段，第一阶段提出 Move 的对象并计算 Move 后的代价函数差，记为 Processing 阶段；第二阶段判定 Move 是否被接受，记为 Finalization 阶段。多个核同时进行第一阶段，第一阶段计算的结果

进入一个有序的缓冲队列。该队列允许 Move 乱序进入，但是 Move 的 Finalization 阶段必须有序进行，即 Move 出队列是按序的，目的是为了保证算法的串行等价性。Move 的 Finalization 阶段由管理员核执行，任何一个核都可以充当管理员的身份，一般由下一个将要 Finalize 的 Move 提出核来承担。一旦某个核进入管理员角色，那么这个核需要将队列里能够按序 Finalization 的 Move 处理完毕。每个核都保存一份当前所有 Block 坐标信息的数据。每次有交换被接受时，每个核在当前正在进行的 Move 完成之后更新自身的 Block 信息。一旦 Finalization 阶段中某一 Move 被接受，那么管理员核需要对其他核已完成或当前正在进行的 Move 进行检查，查看是否有冲突，一旦冲突，重新去处理该 Move。该算法是确定性的串行等价性的，而且理想情况下 N 核可以达到的加速比为 N，实际上四核可以达到 2.5 倍的加速比，8 核在熄火阶段可以达到 4.0 倍的加速比。但该算法的模型过于复杂，难以实现，本文算法借鉴了该算法的基本思路，并将 MapReduce 的框架模型引入其中，大大简化了算法的模型。

4.3 基于多核交换的并行模拟退火布局算法

综上可知，要同时满足确定性和串行等价性，而又不能使布局质量下降太多，是比较困难的，对算法的要求比较高。本文采用了一种基于多核并行交换的策略，同时加上限制条件来控制算法的串行等价性。算法的主要思想为：(在本文中每一次交换称为一次 Move)

把模拟退火算法中一次交换的过程分为两个阶段，第一阶段提出 Move 的对象并计算 Move 后的代价函数差，记为 Processing 阶段；第二阶段判定 Move 是否被接受，记为 Finalization 阶段。第一阶段理论上占据一次 Move 99%的时间，而第二阶段只占据 1%的时间(实际上在并行时第二阶段还要完成更新数据和解决冲突，需要的时间会多一些)。如图 4-1 所示。

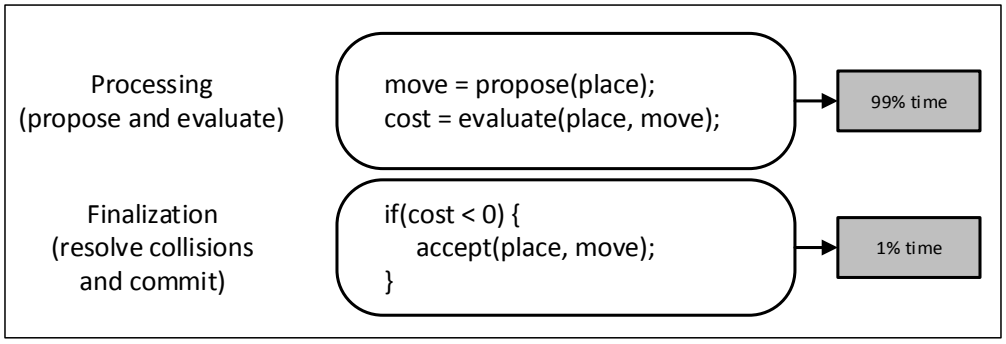


图 4-1 多核并行布局阶段划分

多个核同时进行第一阶段的 Move 操作，然后每个核把第一阶段计算的结果

放入一个有序的缓冲队列。该队列允许 Move 乱序进入，但是 Move 的 Finalization 阶段必须有序进行，即 Move 出队列是按序的，目的是为了保证算法的串行等价性。另外有一个核专门用于管理该缓冲队列，它从缓冲队列中按序取出 Move，并判断是否接受。一旦交换被接受，该核需要去检查该交换是否与已经完成的 Move 存在冲突，若存在冲突，交换必须重新处理。

4.3.1 MapReduce 并行算法模型

为了简化多核程序的编写，本论文采用了 Google 公司提出的 MapReduce 并行编程模型[12]。MapReduce 是一个编程模型，也是一个处理和生成超大数据集的算法模型的相关实现。用户首先创建一个 Map 函数处理一个基于 key/value pair 的数据集合，输出中间的基于 key/value pair 的数据集合；然后再创建一个 Reduce 函数用来合并所有的具有相同中间 key 值的中间 value 值。

由于 Google 公司的工程师每天需要处理使用网络爬虫抓取的大约 20PB 的数据，面对大数据的运算，他们研究，发现大多数的运算都包括这样的操作：在输入数据的“逻辑”记录上应用 Map 操作得出一个中间 key/value pair 集合，然后在所有具有相同 key 值的 value 值上应用 Reduce 操作，从而达到合并中间的数据，得到一个想要的结果的目的。使用 MapReduce 模型，再结合用户实现的 Map 和 Reduce 函数，就可以非常容易的实现大规模并行化计算。借鉴于 Lisp 和函数式编程的概念，他们实现了这个框架并取得了很好的并行速度和性能。

MapReduce 架构的程序能够在大量的普通配置的计算机上实现并行化处理。这个系统在运行时只关心：如何分割输入数据，在大量计算机组成的集群上的调度，集群中计算机的错误处理，管理集群中计算机之间必要的通信。采用 MapReduce 架构可以使那些没有并行计算和分布式处理系统开发经验的程序员有效利用分布式系统的丰富资源。

MapReduce 模型的执行过程为：通过将 Map 调用的输入数据自动分割为 M 个数据片段的集合，Map 调用被分布到多台机器上执行。输入的数据片段能够在不同的机器上并行处理。使用分区函数将 Map 调用产生的中间 key 值分成 R 个不同分区（例如，hash(key) mod R），Reduce 调用也被分布到多台机器上执行。分区数量（R）和分区函数由用户来指定。

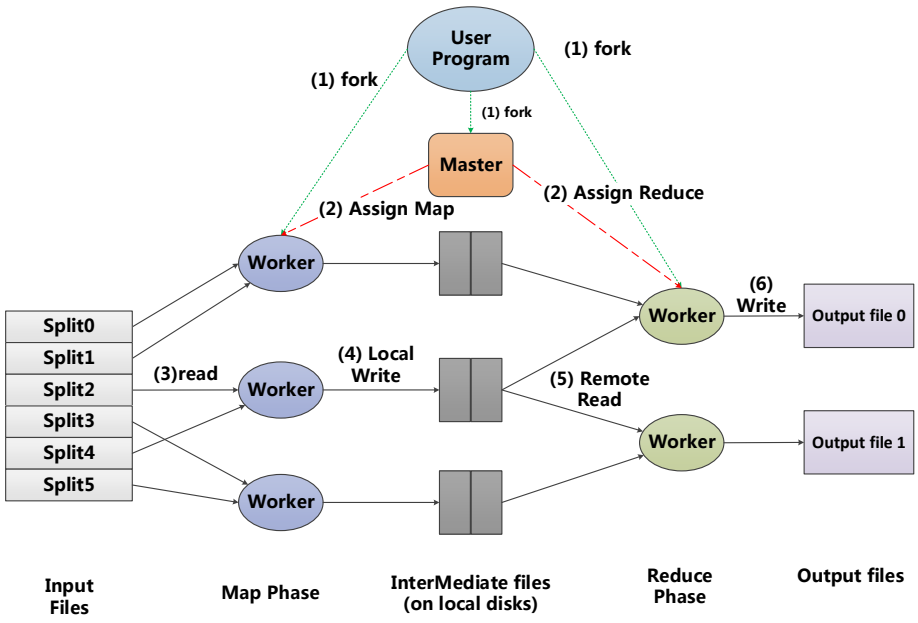


图 4-2 MapReduce 实现流程

图 4-2 展示了 MapReduce 实现中操作的全部流程。当用户调用 MapReduce 函数时，将发生下面的一系列动作（下面的序号和图 4-2 中的序号一一对应）：

1.用户程序首先调用的 MapReduce 库将输入文件分成 M 个数据片度，每个数据片段的大小一般从 16MB 到 64MB(可以通过可选的参数来控制每个数据片段的大小)。然后用户程序在机群中创建大量的程序副本。

2.这些程序副本中的有一个特殊的程序－master。副本中其它的程序都是 worker 程序，由 master 分配任务。有 M 个 Map 任务和 R 个 Reduce 任务将被分配，master 将一个 Map 任务或 Reduce 任务分配给一个空闲的 worker。

3.被分配了 map 任务的 worker 程序读取相关的输入数据片段，从输入的数据片段中解析出 key/value pair，然后把 key/value pair 传递给用户自定义的 Map 函数，由 Map 函数生成并输出的中间 key/value pair，并缓存在内存中。

4.缓存中的 key/value pair 通过分区函数分成 R 个区域，之后周期性的写入到本地磁盘上。缓存的 key/value pair 在本地磁盘上的存储位置将被回传给 master，由 master 负责把这些存储位置再传送给 Reduce worker。

5.当 Reduce worker 程序接收到 master 程序发来的数据存储位置信息后，使用 RPC 从 Map worker 所在主机的磁盘上读取这些缓存数据。当 Reduce worker 读取了所有的中间数据后，通过对 key 进行排序后使得具有相同 key 值的数据聚合在一起。由于许多不同的 key 值会映射到相同的 Reduce 任务上，因此必须进行排序。如果中间数据太大无法在内存中完成排序，那么就要在外部进行排序。

6.Reduce worker 程序遍历排序后的中间数据，对于每一个唯一的中间 key 值，

Reduce worker 程序将这个 key 值和它相关的中间 value 值的集合传递给用户自定义的 Reduce 函数。Reduce 函数的输出被追加到所属分区的输出文件。

7.当所有的 Map 和 Reduce 任务都完成之后，master 唤醒用户程序。在这个时候，在用户程序里的对 MapReduce 调用才返回。

例如，计算一个大的文档集合中每个单词出现的次数，下面是伪代码段：

```
map(String key, String value):
// key: 文档名称
// value: 文档内容
for each word w in value:
EmitIntermediate(w, "1");

reduce(String key, Iterator values):
// key: 一个单词
// values: 一个计数列表
int result = 0;
for each v in values:
result += ParseInt(v);
Emit(AsString(result));
```

4.3.2 多核模拟退火算法模型

很显然，第一，本算法的数据并没有达到 PB 量级，甚至不到 GB 级别，所以本文并没有必要直接使用一些开源的 MapReduce 框架；第二，本算法主要是在多核平台上运行，而 MapReduce 框架主要是在计算机集群上实现，其中包括了容错处理、容灾实现和本地优化等，这些本文都是不需要的；第三，在 Google 的 MapReduce 框架中，要求划分的各任务之间完全独立，各任务完成的先后顺序并不影响程序运行的结果，而本文的算法中，一次被接受的交换会影响整个 FPGA 的布局信息，从而对后面的交换造成影响，因此也无法做到完全的独立。

综合上述考虑，本文在 MapReduce 模型的基础上实现了一个简单的适合于多核平台的并行模型。整个算法的模型如图 4-3 所示。

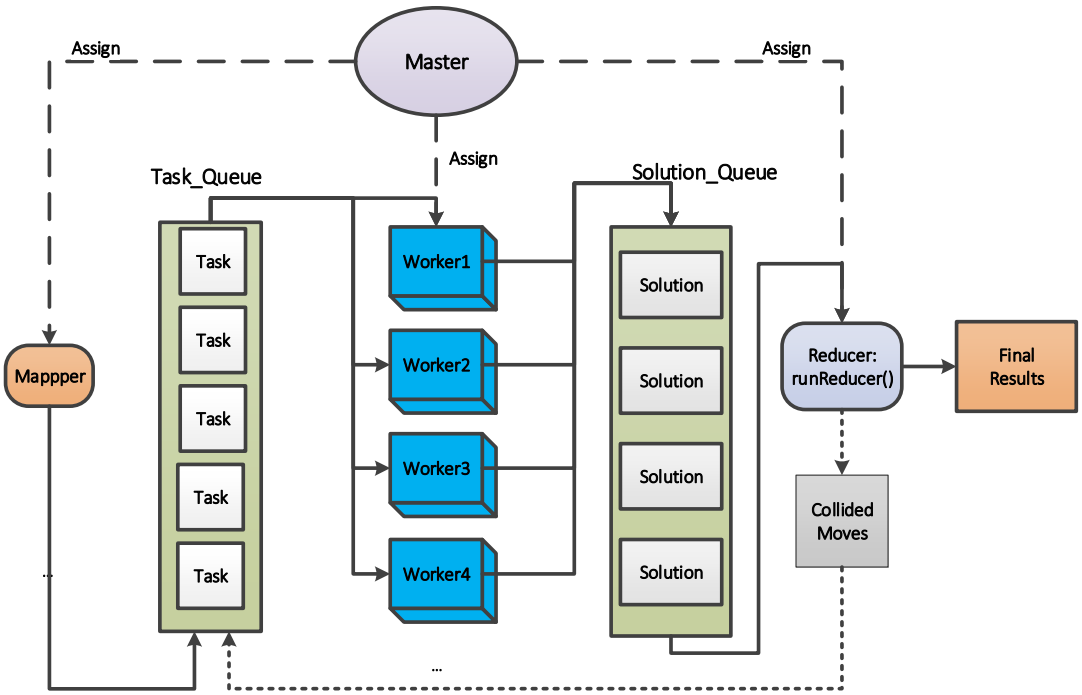


图 4-3 多核模拟退火算法实现框架

首先，主线程在开始退火时，创建一个 Mapper 线程、若干 Worker 线程和一个 Reducer 线程。

Mapper 线程负责分配退火的任务，它将需要进行退火的 Move 放入任务队列；

Worker 线程负责执行每个 Move 的 Processing 阶段，它从任务队列中取出任务，然后计算该次 Move 的位置和代价信息，并将计算结果放入结果队列中，然后继续取出 Move；

Reducer 从结果队列中取出 Move 执行完的结果，然后先判断该 Move 是否与发生冲突，如果不发生冲突的话，那么继续判断其能否被接受，如果被拒绝，那么 Reducer 舍弃该次 Move 并继续取出下一次 Move，如果接受的话，那么 Reducer 要更新相应的数据信息；如果发生冲突的话，那么 Reducer 将此次 Move 放回任务队列重新处理。

如果所有 Worker 是共享相同的数据信息的话，那么 Reducer 对数据信息所做的修改将会及时被所有 Worker 接收到，而如果此时该 Worker 正在提出一次 Move 的话，那么该 Worker 可能会提出很荒谬的 Move，例如一个逻辑块与自己进行交换，更严重的情况会导致程序崩溃。因此为了避免这种情形，本算法中让所有的 Worker 都单独保存自己必需的数据结构，只在每次 Move 结束时才进行更新。

Reducer 产生最终的布局结果。整个算法依据 MapReduce 的框架进行了简单

的处理，使其适应于多核平台。

4.3.3 保证串行等价性

前面提到，本文的算法必须保证确定性和串行等价性。然而如果上述模型的流程不经过任何约束的话，并不能保证串行等价性。保证串行等价性的难点主要在两个方面：一是随机数的串行等价性；二是 Move 之间的不独立导致冲突的可能；

4.3.3.1 新的随机数算法

在串行的 VPR 模拟退火算法中，采用的是线性同余法来产生随机数。线性同余法产生随机数的原理为：

$$X(n+1) = (a \times X(n) + c) \% m \qquad \text{式(4-1)}$$

其中在 VPR 算法中：$a = 1103515245$，$c = 12345$, $m = 2147483648$；$X(n) = 0$；可以看到，线性同余法产生的随机数必然不是真正随机的，它总是在一个大的周期中进行循环，但是只要上述参数选择得当，是可以得到满足给定精度范围的足够随机性，这样的随机数一般也称为“伪随机数”，是在计算机科学中常用的。

在串行模拟退火算法中，所有随机数在同一个种子下生成，每一个随机数都由前一个随机数计算而来，因此每次布局的结果是确定的。

但是在并行架构中，许多 Move 是同时进行的，而在一次 Move 中，需要产生随机数的地方很多，比如起点位置，终点位置，判断是否接受的随机数等等，因此这些同时进行的 Move 不可能像串行那样保证相同的随机数产生顺序。因此不能再使用同一个种子来生成所有的随机数。一个简单的想法就是将每次 Move 的顺序作为随机数的种子，这样每次 Move 中产生的随机数的只决定于 Move 的顺序，这样能够保证 Move 的执行顺序相比串行还是一样的，那么就可以保证串行等价性。

但是 Move 的顺序是连续的，而线性同余随机数发生器在连续的种子下产生的随机数绝对不是随机的，而是线性的，因此必须更换随机数发生器。Coveyou 提出了一种非线性的随机数发生器，其产生随机数发生器的原理如下：

$$X_{n+1} = X_n(X_n + 1) \bmod 2^k \ , \ X_0 \bmod 4 = 2 \qquad \text{式(4-2)}$$

高德纳在《计算机程序设计艺术》第二卷中证明了中算法的随机性[13]。

经过一些测试，发现这种随机数发生器可以在连续的种子情况下产生较好的随机数。本文将 Coveyou 随机数发生器和线性同余随机数发生器进行了比较。图

4-4 中使用固定的种子随机产生了 5000 个随机数，并以（X(n)，X(n+1)）做点图得到如下的随机分布效果。线性同余随机数发生器的分布效果比较均匀，而且随机性较好，点图中基本无规律性。

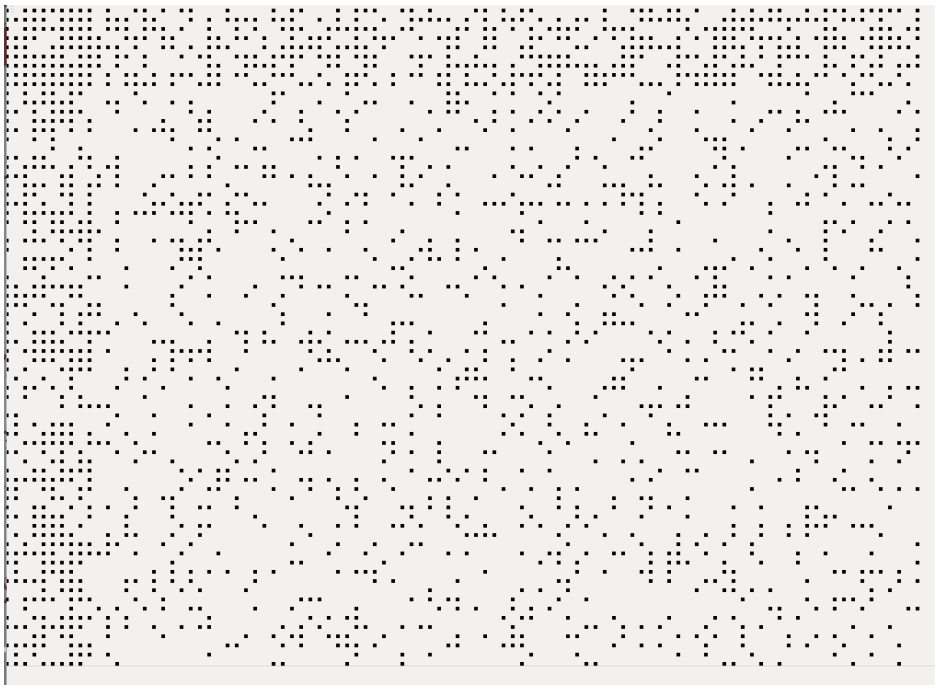


图 4-4 线性同余随机数发生器的测试结果

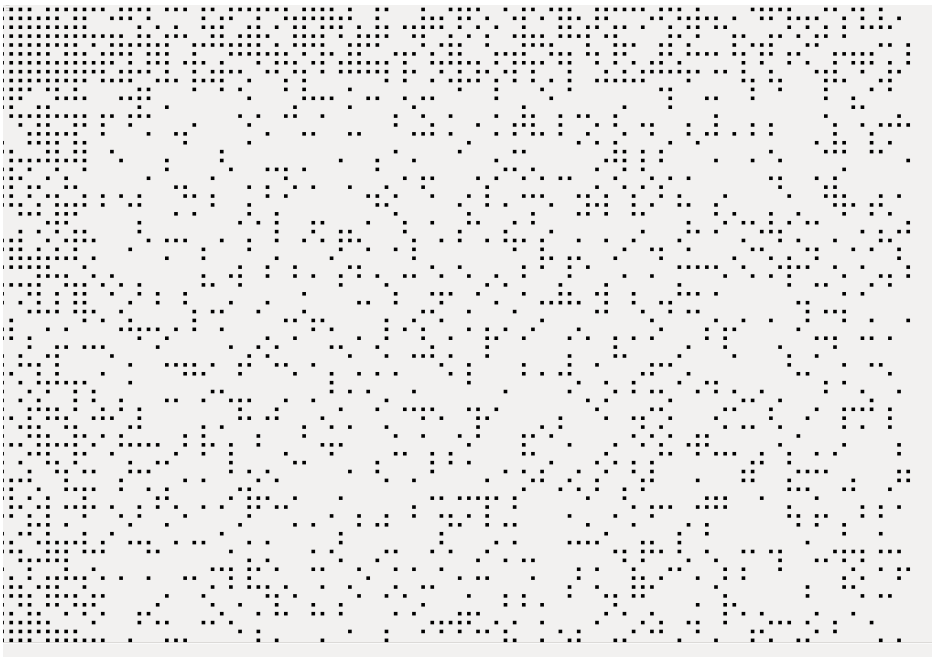


图 4-5 Coveyou 随机数发生器的测试结果

同样，在图 4-5 中，以连续的自然数作为随机数的种子，在每个种子中产生 5 个随机数，同样进行 1000 次的测试得到 5000 个随机数，并做点图得到如上的

随机分布效果。得到的分布图随机效果同样很好，而且分布均匀。

在实际的测试中，使用 Coveyou 随机数发生器替代串行的线性同余随机数发生器，得到的布局质量的下降不超过 1%，因此在并行退火算法中，本文使用 Coveyou 随机数发生器来产生布局中需要的随机数，并使用 Move 的顺序作为随机数的种子。

4.3.3.2 检查冲突的移动

如果能够保证 Move 的执行顺序与串行保持一致，那么上述的随机数也就能够与串行保持一致。因此需要采取措施来保证 Move 的执行顺序与串行一致,需要解决的问题是 Move 的冲突。

因为有多个 Worker 在同时提出 Move，而一个 Move（记为 Move1）在提出之后，需要产生随机交换的两个逻辑块或者需要移动的一个逻辑块和移动的目的地，并找到与这些逻辑块相关的线网，计算代价。如果在 Move1 被提出之后到被 Reducer 处理之前有其他的 Move 被接受并且修改了与 Move1 相关的位置或者线网，那么 Move1 的计算结果是无效的，因为它使用的数据信息没有及时被更新，这样 Move1 需要放回任务队列重新处理。

发生冲突主要有以下几种情况，如：在图 4-6(a)中，两次 Move 同时将选定同一个逻辑块进行交换；在图 4-6(b)中，两次 Move 同时将不同的逻辑块移到相同的位置；在图 4-6(c)中和图 4-6(d)中，两次 Move 选取的逻辑块和中终点位置都不相同，但是两个 Block 处于同一条线网上，如果前一次 Move 被接受势必会影响线网的代价值，从而影响后一次 Move 的代价差计算，从而影响后一次线网是否被接受。

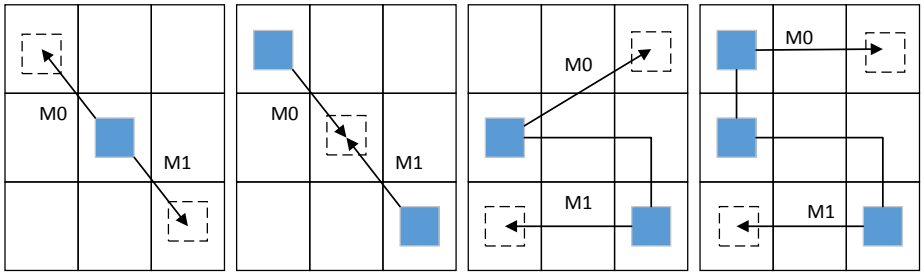


图 4-6 四种可能的冲突情形

为了解决冲突，本文给每条线网和 FPGA 的每个逻辑坐标位置都贴上一张标签，记录这条线网或这个位置最近被更新的时间。同时每个 Move 在被提出时需要去获取当前的时间并保存下来。

这样当这个 Move 进入结果队列后，Reducer 取出该 Move 进行判断时，首先需要将此 Move 涉及到的坐标位置和线网最近的更新时间和 Move 被提出的时间进行判断，如果 Move 被提出的时间早于与 Move 相关的坐标或者线网被更新的时间，那么说明该 Move 的计算式无效的，此 Move 就需要重新处理，因此可以将相应 Move 的序号重新插入到任务队列的前端，等待 Worker 进行处理。

通过这种方法，本文保证了 Move 的顺序执行。同时，应该注意到，由于 Reducer 的处理速度有限，因此如果每个 Move 都持续的向结果队列中添加 Move，并且 Reducer 来不及处理的话，那么冲突的可能性就会增加，而冲突就需要重新处理，从而因此降低并行的效率，因此需要限制每个时刻正在处理或者已完成但未结束的 Move 的数量。

4.3.4 程序的优化与改进

4.3.4.1 冲突的分类解决

首先需要介绍一下 VPR 产生随机交换的两个 Block 的方法。伪代码如下：

```
from_block_id = random(movable_blocks.size());
rlx = ( block[from_block_id].x() + r_lim + x_max ) % x_max;
x_to = random(rlx);
rly = ( block[from_block_id].y() + r_lim + y_max ) % y_max;
x_to = random(rly);
```

其中，起点逻辑块的 id 是在可以移动的所有逻辑块中随机产生的，因此只要随机数的种子相同，起点逻辑块就不会变。终点位置的产生是在起点位置周围 r_lim 的范围内随机产生的(r_lim 是交换的半径，是根据退火温度变化的)，因此如果起点逻辑块的位置没有变的话，那么终点的位置也是相同的。

不是所有冲突的 Move 都需要重新处理。本文前面介绍了冲突的种类，对于(a)、(b)两种冲突，可以并称为位置冲突，这种冲突主要是因为第二次 Move 产生的时候前一次 Move 的结果还没有及时更新，导致采用了错误的位置信息来产生随机数，因此必须重新置随机数的种子，并产生新的起点和终点位置。但是对于(c)、(d)两种冲突，可以并称为线网冲突，此时由于随机数是以 Move 的序号作为种子的，因为产生的种子相同，而在没有位置冲突的情况下，重新产生的起点和终点是相同的，因此没有必要重新去执行产生 Move 的阶段，只需要重新计算线网的边界盒和代价函数值。

更进一步的，即使产生了线网冲突，也不代表线网的边界盒和代价需要重新计算。可以考虑如下图 4-7 所示的情形：图(a)和(b)中的边界框主要是由 E、F、G、H 和 I 这五个模块决定的。在图(a)中，交换 M0 将 B 移动到边界内的位置，交换 M1 交换 C 和 D，这两个 Move 虽然存在线网冲突，但是由于两次 Move 都没有影响线网边界框的改变，因此冲突的 M1 并不需要重新计算线网的边界框；而在(b)图中，M0 将 I 与 B 交换，M1 将 H 移动，两次移动的对象都是都处在边界框上，假设 M0 和 M1 是同时执行的，由于数据结构没有及时更新，因此两次 Move 执行完毕时，得到的边界框是相等的，但是如果不重新计算 M1 的边界框的话，就会在 M1 被接受时导致错误，因为两次 Move 都改变了边界框边界上的逻辑块数目。综上知，在边界框的大小没有发生变化并且边界框上的逻辑块数目都没有发生变化的时候，Move 的边界框是不需要重新计算的，而线网的代价函数直接决定于线网的边界框，因此线网代价也是不需要重新计算的。

这样做的好处是显而易见的，在模拟退火的早期，交换被接受的概率相当高，如果每次 Move 发生冲突都重新处理的话，将浪费很多的时间。另外在线网扇出比较大的时候，发生冲突而不导致边界框发生改变的情况也是频繁发生，因此此时的优化处理也可以大大节省运行时间。

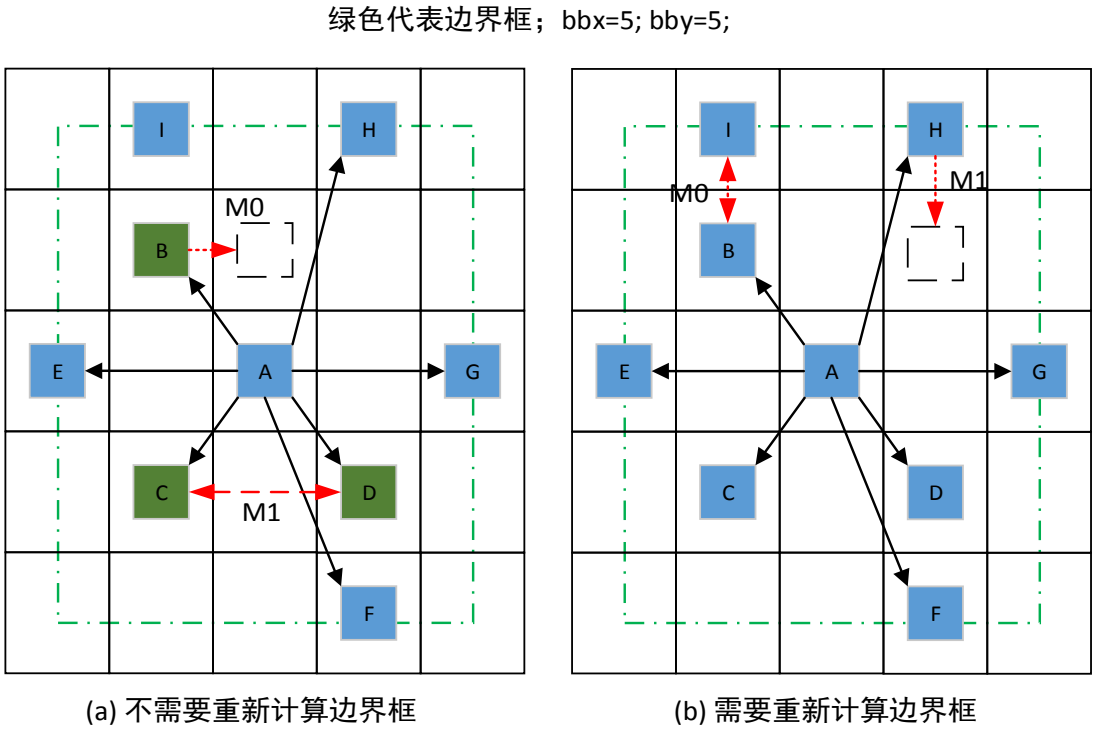


图 4-7 线网冲突但边界不冲突的情况

4.3.4.2 检查冲突的时刻

前面介绍的算法都是在一次 Move 被提交的时刻才检测是否冲突，这就意味着每次遇到冲突，Reducer 线程都需要去解决 Move 的冲突或者让 Worker 重新处理该次 Move，而每次的等待都需要大概 Worker 处理两次 Move 的时间，在扇出很高的线网中，冲突发生的概率很高，如果每次都等到提交时刻才去检查，那么在存在多个 Worker 的情况下，Reducer 基本上在解决冲突，而 Worker 却一直在继续执行新的 Move，冲突的 Move 就越来越多。

因此新的解决方案将在每次 Move 被提交之后采取向后检查的步骤。每次 Move 被提交的时刻，如果此时队列中存在多个 Move，那么 Reducer 检查这些 Move 是否有冲突，如果有冲突，将它们放回任务队列让 Worker 重新处理。

4.3.4.3 使用内存管理池取代 New 和 Delete

在本论文提出的基于并行移动并集中提交的布局算法中，由于每次 Move 的结果都要存储到结果队列中去，因此在每次 Move 执行过程中的中间变量都要存储到相应的数据结构中去。本文定义了一个 Move 类来存储受影响的逻辑块和线网，并在每次执行 Move 前新建一个 Move 类，在 reducer 处理完毕后释放掉相应的 Move。而在一次模拟退火过程中，大概要进行一百万次的 Move，频繁的申请和释放内存浪费了很多的时间。

C++语言中默认的内存管理器是通用的，是多线程安全的，灵活性很强，但这种灵活性是以牺牲速度为代价的，需要的计算越多，消耗的时钟周期就越多。new 和 delete 操作首先会调用 malloc 和 free 操作，它会执行一个系统调用，从用户态转到内核态，该系统调用会锁住内存硬件，然后通过链表的方式查找空闲内存，如果找到大小合适的，就把用户的进程地址映射到内存硬件地址中，然后释放锁，返回用户态；delete 是一个反过程。频繁的进行系统内核的调用会浪费很多的时间，可以通过内存池模式来提高这两个操作的效率：内存池就是预先分配好，放到进程空间的内存块，用户申请与释放内存其实都是在进程内进行，只有当内存池空间不够时，才会再从系统找一块很大的内存。

因此为了提高程序的执行效率，本文定义了一个内存的管理池，用于收集被释放的空间和重新分配这些已经申请的空间。在程序的初始阶段，申请了一部分的连续空间，类似于将这些空间挂载到一条链上，每次需要使用的时候从链上取下一个节点使用，使用完毕将节点归还到链上，从而节省大量的空间分配和释放时间。

4.4 本章小结

本章介绍了并行化模拟退火算法的研究目标和国内外研究现状，并在此基础上提出了基于 MapReduce 框架实现的通过多核并行移动的并行化布局算法，详细阐述了该算法的实现原理和保证串行化的方法。

第五章 基于并行模拟退火算法的布局工具的实现

5.1 布局工具的说明

该布局工具移植了以串行模拟退火算法为架构的 VPR 算法，实现了基于本文阐述的并行化算法。该算法采用 C++编程语言描述，因为 C++面向对象的编程方法易于实现本文中提出的 MapReduce 框架，同时 VPR 是采用 C 语言开发的，而 C 与 C++之间移植比较方便。

同时为了实现工具的实用性，本文不再直接采用通用的 FPGA 结构来进行测试，而是直接使用 Xilinx 的 Vertex-4 器件作为布局的测试平台。

5.2 VPR 软件运行介绍

VPR 软件基于 Linux 平台运行，软件运行界面如图 5-1 所示.界面中间是岛式 FPGA 芯片的基本结构，中间是 CLB，外围是 IOB。底下的状态栏显示了初始随机布局的代价信息。点击左侧的“Proceed”按钮，可以开始模拟退火的布局过程。

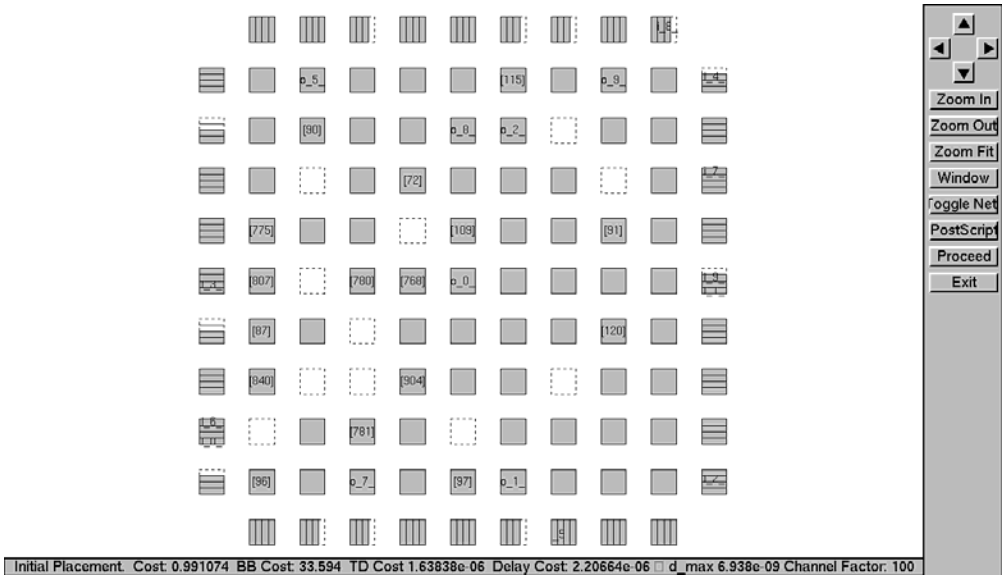


图 5-1　VPR 软件初始布局后界面

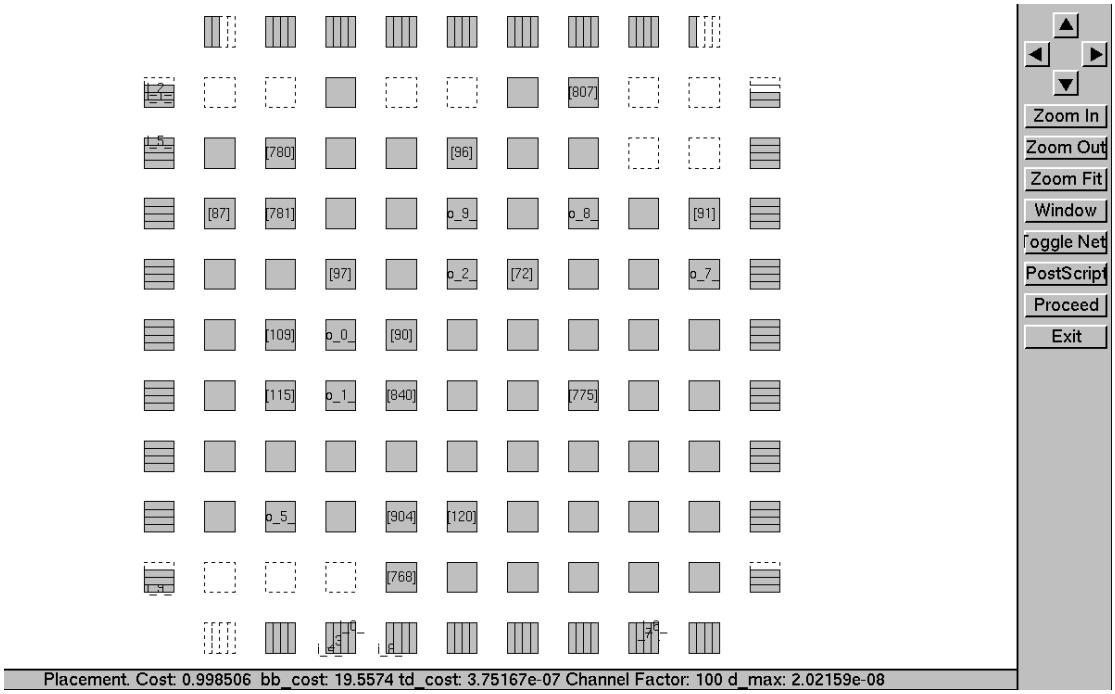


图 5-2 VPR 完成模拟退火之后的界面

图 5-2 是完成了模拟退火之后的布局界面，可以从状态栏看到 BB_Cost 从 33.594 降到 19.5574。

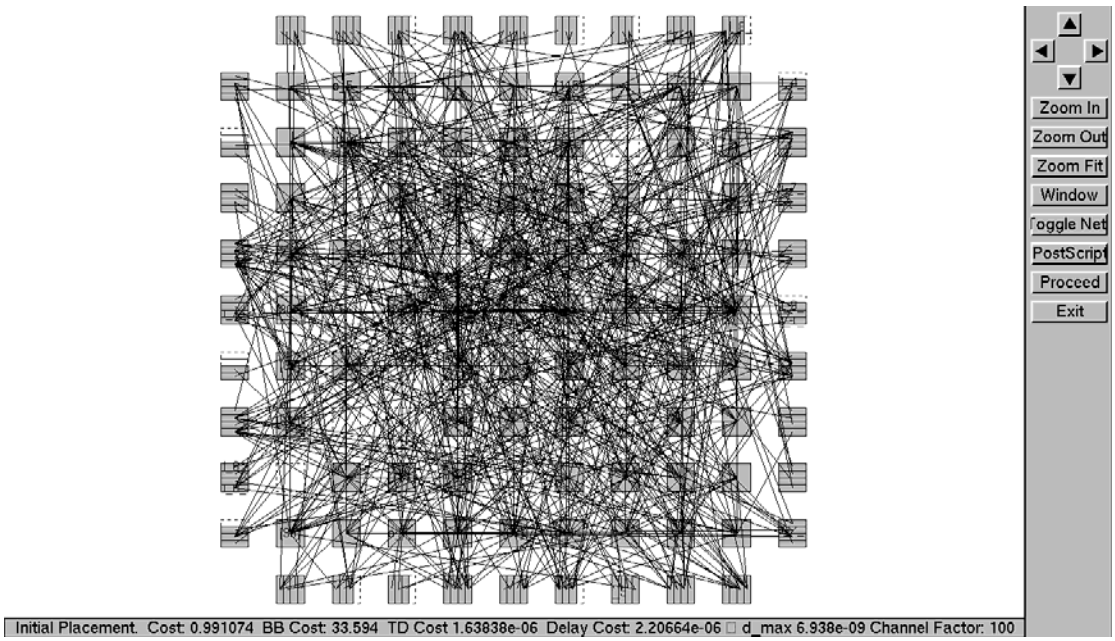


图 5-3 VPR 初始随机布局之后的线网分布图

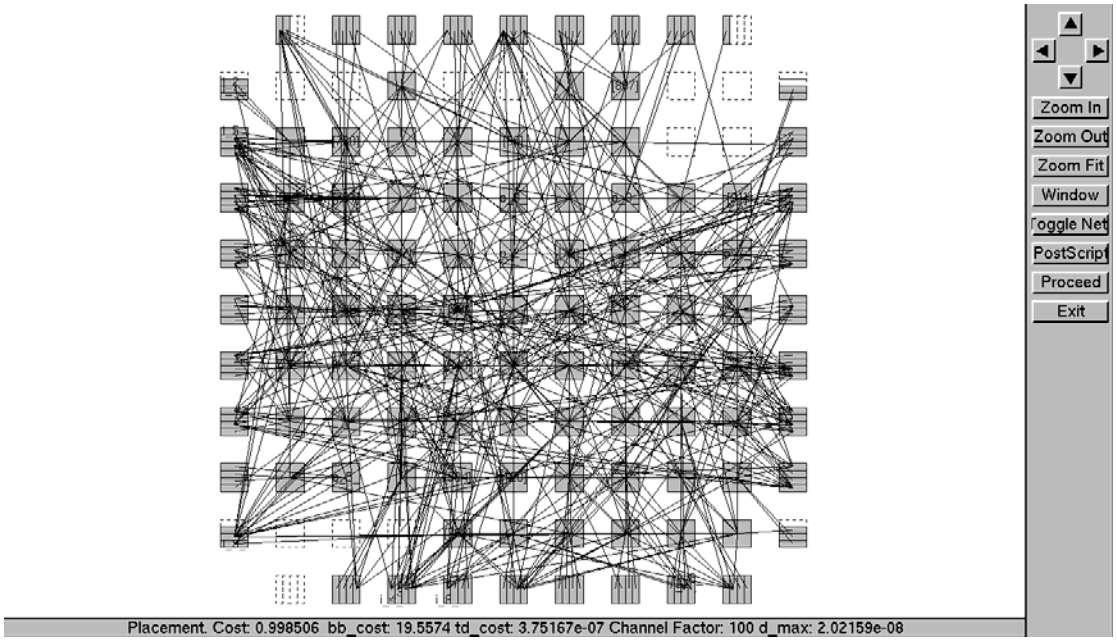


图 5-4 VPR 完成模拟退火布局之后的线网分布图

VPR 也可以查看布局的线网连接情况，图 5-3 是 VPR 在随机布局之后的线网分布图，而图 5-4 是 VPR 在完成模拟退火布局之后的线网分布，可以在布局完成之后线网的密度得到了很大改善。

5.3 多核编程语言的选择

本算法采用 POSIX 线程来实现并行的程序设计，原因有三：第一，本算法工具基于 Linux 平台开发，而 POSIX 线程已经集成到系统中，使用相当便捷；第二，POSIX 线程作为系统调用相当简单，而且在可以在 C++中直接调用和封装，执行效率也相当高；第三，POSIX 线程可移植性好，可以很方面的移植到 Windows 平台。

5.4 布局工具的测试方法

5.4.1 XDL 文件介绍

XDL（Xilinx Design Language）是一种人为可读的 ASCII 格式的网表文件，和另一种二进制格式的网表文件 NCD（Native Circuit Description）相等效，XDL 和 NCD 之间可以通过 Xilinx 提供的命令 xdl –ncd2xdl 和 xdl –xdl2ncd 进行相互转换。

NCD 文件是二进制格式的，所以无法直接编辑，但是可以用 Xilinx 的 FPGA Editor 打开查看电路，并直观地对电路进行修改，而 XDL 作为文本文件，可以

对其进行方便、快速的编辑，这是 NCD 文件所不及的地方。

在 Xilinx 的 CAD 工具中，XDL 发挥了重要的作用，基本作为综合和映射完成后的设计表现形式，可以从图 5-5 看出，Map 阶段、布局阶段、和布线阶段产生的 NCD 文件都可以通过 XDL 工具转化为 XDL。

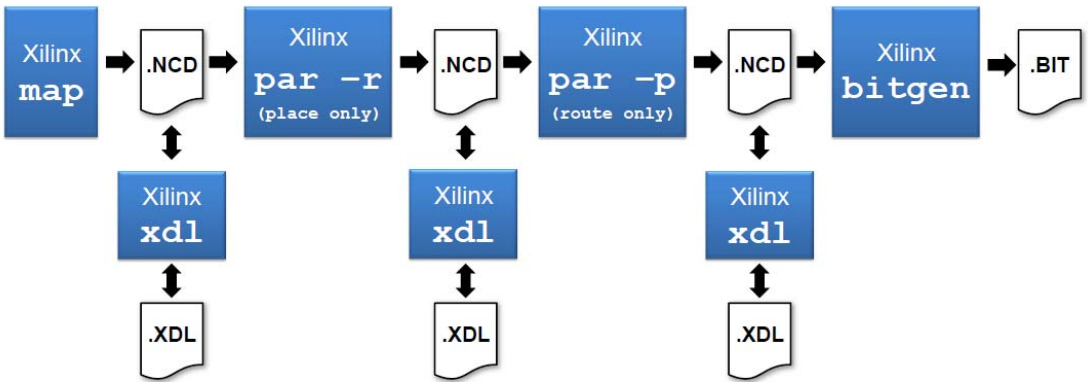


图 5-5 Xilinx 在各阶段产生的设计文件示意图

XDL 可读性的描述了 FPGA 中电路实现的信息，主要包含 design、module、instance 和 net 四类语句，下面对这 4 部分分别进行介绍。

(1) design 语句

```
# ============================================================
# The syntax for the design statement is:
# design <design_name> <part> <ncd version>;
# or
# design <design_name> <device> <package> <speed> <ncd_version>
# ============================================================
design "top" xc2s300epq208-6 v3.1 ,
   cfg "
        _DESIGN_PROP::PIN_INFO:p1<0> p1<0> INOUT 7 p1<7\:0>
        _DESIGN_PROP::PIN_INFO:p1<1> p1<1> INOUT 6 p1<7\:0>
        _DESIGN_PROP::PIN_INFO:p1<2> p1<2> INOUT 5 p1<7\:0>
        ……
        _DESIGN_PROP::BUS_INFO:8\:INOUT\:p0<7\:0>
        _DESIGN_PROP::BUS_INFO:8\:OUTPUT\:p2<7\:0>";
```

Design 部分描述了该电路设计的属性和芯片信息，如：电路设计的名字，器件的名字，器件的封装等。

(2) module 语句

```
# ===========================================================
# The syntax for modules is:
# module <name> <inst_name> ;
# port <name> <inst_name> <inst_pin> ;
# .
# instance ... ;
# .
# net ... ;
# .
# endmodule <name> ;
# ===========================================================
module "moduleName" "anchorInstanceName" , cfg "_SYSTEM_MACRO::FALSE" ;
port "portName1" "anchorInstanceName" "F2";
port "portName2" "anotherInstanceInTheModule" "F4";
...
inst "anchorInstanceName" "SLICEL", placed CLB_X14Y4 SLICE_X23Y8 , …
...
net "aNetInsideTheModule" , …
...
```

module 语句类似于宏，有一系列 port 来定义该宏的接口，还有一系列 instance 和 net 语句描述内部的逻辑。

(3) instance 语句

```
inst "u1/U_CHIP_RAM/MUX_BLOCK__n0014<1>_MUXF5225" "SLICE",placed R4C28 CLB_R4C28.S0 ,
  cfg " BXMUX::BX BYMUX::#OFF CEMUX::#OFF CKINV::#OFF COUTUSED::#OFF CY0F::#OFF
       CY0G::#OFF CYINIT::#OFF CYSELF::#OFF CYSELG::#OFF DXMUX::#OFF DYMUX::#OFF
       F:u1/U_CHIP_RAM/_n0014<0>450:#LUT:D=((~A3*A1)+(A3*A4)) F5USED::0
       FFX::#OFF FFY::#OFF FXMUX::#OFF
```

```
G:u1/U_CHIP_RAM/_n0014<0>451:#LUT:D=((~A2*A4)+(A2*A3))
        GYMUX::#OFF INITX::#OFF INITY::#OFF RAMCONFIG::#OFF REVUSED::#OFF
        SRFFMUX::#OFF SRMUX::#OFF SYNC_ATTR::#OFF XBUSED::#OFF
XUSED::#OFF
        YBMUX::#OFF YUSED::#OFF
```

以关键字“inst”开头，该语句描述了电路设计中使用到的逻辑模块的位置和配置信息。每个 instance 语句表示一个实例化的逻辑模块，包括逻辑模块名字、类型、位置和配置信息。如上实例所示，
“u1/U_CHIP_RAM/MUX_BLOCK__n0014<1>_MUXF5225”是该逻辑模块的名字，在这个电路设计中这个名字必须是唯一的，该模块的类型是 SLICE，R4C28 中的 S0 是该模块所在的位置，配置信息由 cfg 语句来定义，其中列出了一系列属性来描述该模块的功能，“BXMUX::BX”表示 BXMUX 这个配置项被配置为了配置值 BX。

(4) net 语句

```
net "u1/U_CHIP_RAM/_n0479<3>" ,
   inpin "u1/U_CHIP_RAM/MUX_BLOCK__n0014<1>_MUXF5217" G1 ,
   inpin "u1/U_CHIP_RAM/_n0479<3>" F3 ,
   outpin "u1/U_CHIP_RAM/_n0479<3>" XQ ,
   pip R15C19 S0_XQ -> OUT1 ,
   pip R15C20 OUT_W1 -> S0_G_B1 ,
   pip R15C19 S0_XQ -> OUT7 ,
   pip R15C19 OUT7 -> W23 ,
   pip R15C19 W23 -> W_P23 ,
   pip R15C19 W_P23 -> S0_F_B3 ,
   ;
```

net 语句定义了逻辑实例间的布线资源，包含以下内容：实例化名字，输入引脚（inpin），输出引脚（outpin），互连可编程点（pip）。上述实例中，“u1/U_CHIP_RAM/_n0479<3>”是该 net 实例化的名字，每个 inpin 和 outpin 后都跟有所属的逻辑模块的实例化名字和对应的端口名字，pip 后则跟着 tile 名字和连接起来的两条连线，“pip R15C19 OUT7 -> W23 ,”中的 OUT7 和 W23 都是 tile R15C19 中的两条连线，该 pip 表示一条从 OUT7 到 W23 的单向互连线。

Pip 中连接的符号有 4 种：

==：双向导通，无缓冲器

=>：双向导通，单向有缓冲器

=-：双向导通，双向都有缓冲器

->：单向导通，有缓冲器

5.4.2 布局工具的输入输出文件

为了便于结果的对比，同时也便于与其他 CAD 流程的兼容，本文采用上述的 XDL 文件作为布局工具的输入和输出文件。

布局工具读取 Xilinx 自动化工具产生的完成布局的 XDL 文件以及器件的库文件，因为布局后的 XDL 含有位置信息，因此可以计算出 Xilinx 的布局最终代价。同时因为 XDL 文件中含有详细的逻辑块和线网信息，因此使用这些信息进行布局，布局完成可以得到最终代价信息，从而可以与 Xilinx 比较布局的质量和布局花费的时间。

布局工具的输出文件也定义为与 Xilinx 自动化工具产生的相同的 XDL 文件，主要是因为让布局工具的结果具备通用性，因为布线阶段也可以使用 XDL 文件进行输入。而且布局的结果可以输入 Xilinx 自动化工具继续生成码流验证设计功能的正确性。

5.5 并行化测试平台

5.5.1 硬件测试芯片的选取-Virtex-4

Virtex-4 FPGA 是 Xilinx 公司推出的第四代 Virtex 系列 FPGA 产品。该 FPGA 可提供高达 20 万逻辑单元和 500 MHz 的性能。该产品采用先进的深亚微米设计技术、集成硬 IP 模块以及三次氧化 90 nm 工艺技术, 因而其器件成本和功耗降低了 50%。Virtex- 4 FPGA 系列中, 目前推出的三个平台的型号为 Virtex- 4 LX、SX 和 FX, 主要针对逻辑应用、DSP、高速互连功能和嵌入式处理四个应用领域, 每个平台都有多款规模不同的器件, 而同一平台中的器件具有大致相同的功能特性组合比例。

Virtex-4 系列 FPGA 器件主要由阵列式的可配置逻辑块(CLB)构成，在 CLB 的周围环绕着可编程的输入输出逻辑块(IOB)，另外还包括片上集成随机存储器模块(Block RAM)和一些时钟管理模块(DCM)以及其他功能模块等等，这样丰富的布线资源使它能够完成更大规模的、更加复杂的设计。

Virtex-4 是由基于 Tile 阵列的结构组成。每个 Tile 由一个通用布线矩阵

（General Routing Matrix，GRM）与一个可配置逻辑块（Configurable Logic Block，CLB）或输入输出逻辑块(Input/Output Block，IOB)或其他功能模块组成。

Virtex-4 系列芯片的结构示意图如图 5-6 所示(以 xc4vlx15sf363 芯片为例)。其中大面积的蓝色方格代表 CLB 模块，占据了 FPGA 的主要部分，外围的绿色方格代表 IO 模块，中间几行浅蓝色的方格代表与时钟相关的 HCLK 模块，还有几列紫色的方块代表 Block RAM 模块。另外正中间分隔的一部分方格代表的逻辑块功能在 Xilinx 的官方手册中给出了解释，如图 5-7 所示，从上到下依次为数字时钟管理(DCM)、相位匹配的时钟分频器(PMCD)、IOB、全局时钟(BUFGCTRL)，然后下面是与上面对称的结构。

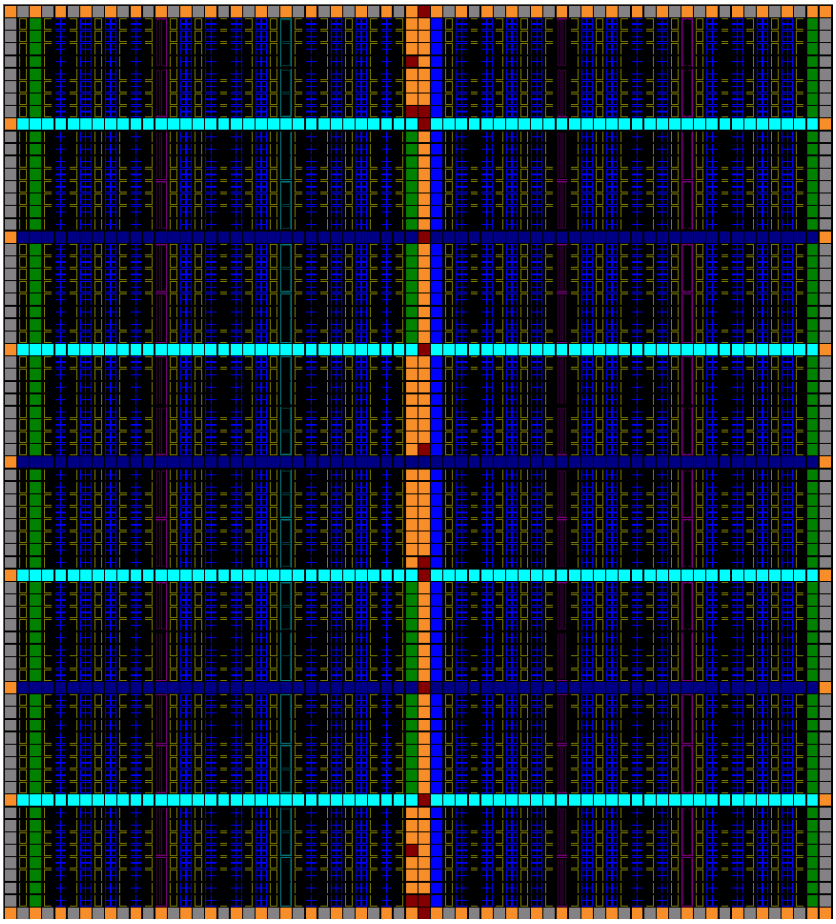


图 5-6 Xilinx 公司 Virtex-4 系列 xc4vlx15sf363 芯片结构示意图

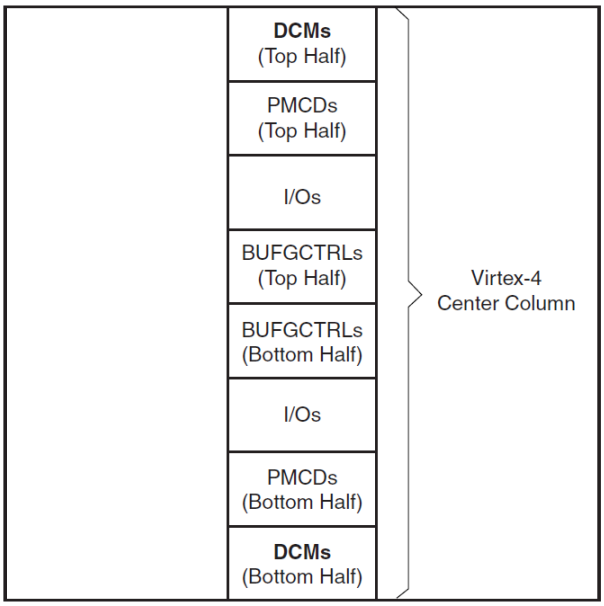


图 5-7 Xilinx Central Column 的模块示意图[14]

5.5.1.1 可编程逻辑块(CLB)

CLB 用于实现时序逻辑和组合逻辑,Virtex-4 FPGA 的 CLB 在面积和速度上都进行了优化,以适用于小型高性能设计。

每个 CLB 模块由 4 个相同的 Slice 和附加逻辑构成,其拓扑结构如图 5-8 所示，可以看出,四个 Slice 被分成了左右两列,各含两个 Slice,由 X 标志不同的列,左边的为 SliceM，右边的 SliceL。他们都有单独的进位链,但只有 SliceM 有移位链，两列 Slice 各自包含两个 4 输入查找表(LUT)、两个存储单元、算术逻辑门、多路复用器和快速超前进位链。

SliceM 能实现的功能要多于 SliceL。SliceM 可组成多种分布式 RAM,包括 16x4 bit 的单口 RAM、32x2 bit 的单口 RAM、64x1 bit 的单口 RAM 和 16x1 bit 的双口 RAM。SliceM 也可配置成 16 位的移位寄存器。

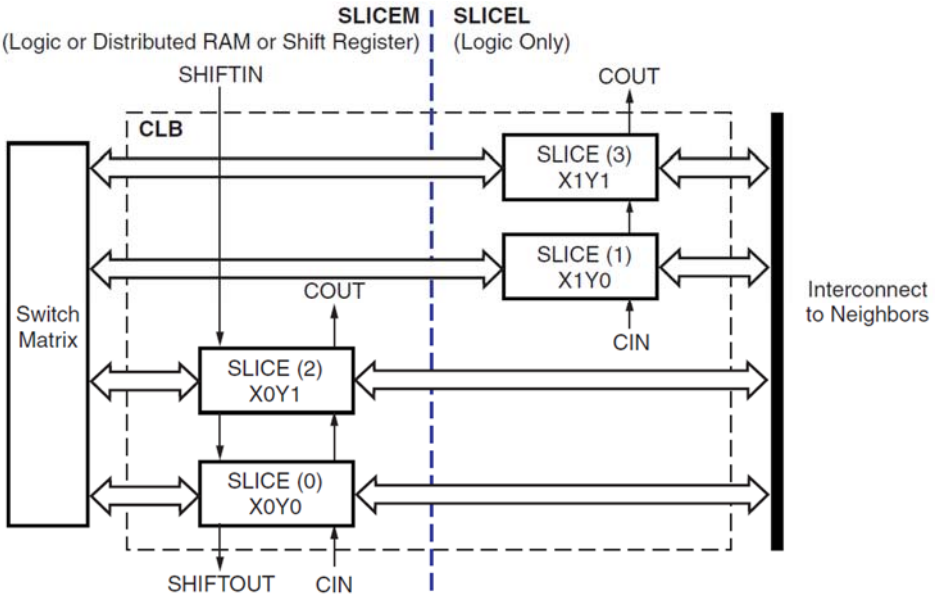


图 5-8 Virtex-4 芯片 CLB 结构示意图

5.5.1.2 可编程输入输出单元（IOB）

可编程输入/输出单元简称 I/O 单元，是芯片与外界电路的接口部分，完成不同电气特性下对输入/输出信号的驱动与匹配要求，其示意结构如图 5-9 所示。FPGA 内的 I/O 按组分类，每组都能够独立地支持不同的 I/O 标准。通过软件的灵活配置，可适配不同的电气标准与 I/O 物理特性，可以调整驱动电流的大小，可以改变上、下拉电阻。目前，I/O 口的频率也越来越高，一些高端的 FPGA 通过 DDR 寄存器技术可以支持高达 2Gbps 的数据速率。

外部输入信号可以通过 IOB 模块的存储单元输入到 FPGA 的内部，也可以直接输入 FPGA 内部。当外部输入信号经过 IOB 模块的存储单元输入到 FPGA 内部时，其保持时间（Hold Time）的要求可以降低，通常默认为 0。

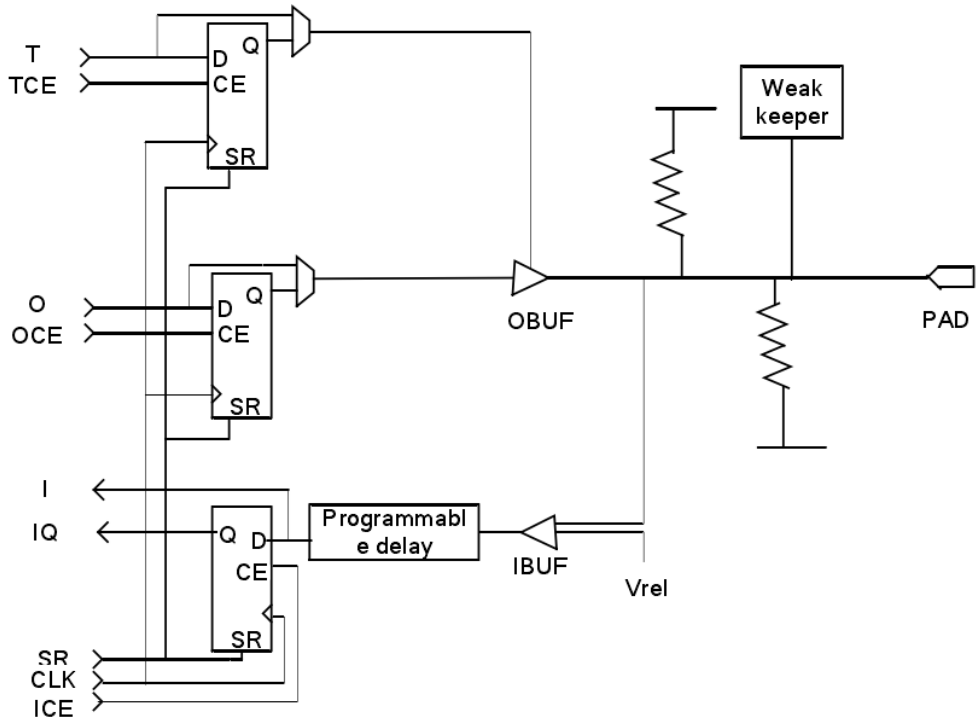


图 5-9 Virtex-4 IOB 结构示意图

为了便于管理和适应多种电器标准，FPGA 的 IOB 被划分为若干个组（bank），每个 bank 的接口标准由其接口电压 VCCO 决定，一个 bank 只能有一种 VCCO，但不同 bank 的 VCCO 可以不同。只有相同电气标准的端口才能连接在一起，VCCO 电压相同是接口标准的基本条件。

5.5.1.3 Block SRAM 模块

每个 Virtex-4 Block RAM 可存储 18Kb 的数据,也可以通过级联实现更大的 RAM，并支持多种纵横比、数据宽度变换和校验。每个 Block RAM 可以配置成多种形式,包括单口 RAM 和双口 RAM，以及各种不同的数据字长,如 16Kx1 bit，8Kx2 bit，512x36 bit。

Virtex-4 与上一代产品相比，新 Block RAM 增强的特点有：

(1) 新型内置流水线,具有新增的数据读出端寄存器，保持 500MHz 时钟；

(2) 两个相邻 Block RAM 可组合成一个更大的 32Kx1 bit 的存储器,而这不需要外部逻辑,也不会发生速度的下降；

(3) 能进行以字节为单位写入；

(4) 独特的内置多速率 FIFO 模式提供了地址排序以及控制电路,并且无需额

外的逻辑资源。FIFO 模式提供满输出和空输出,以及可编程的近满(Almost Full)和近空(Almost Empty)标志，并可保证这些标志无毛刺的生成；

(5) 内置纠错控制(ECC)功能。Block RAM 提供了可选的汉明码纠错及检查能力，以改善系统性能，提高效率并节省空间。

5.5.1.4 可编程布线资源

Virtex-4 的布线资源采用第四代可编程高性能的分段路由结构，特殊的交换矩阵保证了内部路由，使延时可预测。基于矢量的内部互连和分段连接特性，使得 CLB 模块与任何方向相邻的 CLB 之间都具有相同的延时和性能。实现主动内部互连和高性能路由的关键部件是可编程的通用布线矩阵(General Routing Matrix)。交换矩阵在每一个可编程部件之间提供了布线开关的阵列。Virtex-4 FPGA 的组件连接到布线矩阵,并且所有的组件均采用相同的互连机制。[15]

Virtex-4 提供了丰富的布线资源，这些资源具有不同的功能、互连特性和延时。这些布线资源包括：

(1) 长线资源,包括 24 条水平方向长线和 24 条垂直方向长线。这些长线资源是双向的,并贯穿整个器件；

(2) 智能型(Hex)长线资源，在水平和垂直方向各有 120 条，与长线的长度相同。智能型就是说在所有的 4 个方向，每间隔 3 个或 6 个 CLB(包括其它模块)相连，并彼此交错排列；

(3) 双长线资源，包括在水平和垂直方向的各 40 条。在所有 4 个方向上，每间隔 1 个或 2 个 CLB(包括其它模块)相连，并彼此交错排列；

(4) 直连互连线资源，每个交换矩阵提供 16 条直连互连线资源。该布线资源实现相邻共 8 个方向逻辑块之间的互连，该资源也称为“分段互连”结构，是 Xilinx FPGA 系列器件一贯采用的专利技术；

(5) CLB 内部的局部互连资源,每个 CLB 内部共有 8 条快速的局部互连线，这些互连资源实现 CLB 内部逻辑资源的高速互连和共享，如 LUT 的级联、输入输出等；

(6) 其它高速互连资源。这些布线资源为专用信号通道，包括全局的时钟缓冲器、CLB 中的进位链、水平级联链等。

5.5.1.5 Virtex-4 芯片与通用岛式芯片在布局上的区别

(1) 交换的对象

在 VPR 中。采用通用的岛式结构作为 FPGA 的架构描述，将基本的 CLB 和

IOB 作为交换的基本单元，因此每个逻辑块的坐标是二维的。

但在 Virtex-4 中，情况不太相同。第一，一个 CLB 中包含两个 SliceL 和两个 SliceM，用户电路如果分布在同一个 CLB 里面的不同 Slice 中，虽然可以实现相同的功能，但是会产生不同的时延，导致不同的布线结果；第二，前面说过 SliceM 和 SliceL 能实现的功能不太相同，SliceL 能实现的功能 SliceM 也能实现，但 SliceM 能实现的功能 SliceL 不能实现；第三，由于将 XDL 文件作为本文工具的输入输出文件，而 XDL 文件的描述信息是具体到 Slice 级别的。综上知，对于 Virtex-4 平台，本文将采用 Slice 作为布局的基本交换单位。

在 IOB Tile 中也是如此，一个 IOB Tile 中包含两个 IOB 模块、两个 ILogic 模块和两个 OLogic 模块，因此 IOB 的交换也不能以 IOB Tile 为单位，而需要以内部的 IOB 为交换单位。

(2) Chain

Xilinx Virtex-4 FPGA 芯片的一个重要特征是，他们具有实现快速进位链所要求的专用逻辑和互联，在 CLB 中，Slice 可以实现快速超前进位链。每个 Slice 中的两个 LC 之间，每个 CLB 中的 Slice 之间和 CLB 本身之间的专用互联可以相互补充配套。这些专用进位链和专用路由大大提升了一些逻辑功能（例如计数器）和算术功能（例如加法器）的性能。（需要修改）

在 VPR 中没有考虑 Chain 的影响，因此每次交换都是一个逻辑块移动到另一个地方或者两个逻辑块进行交换，但在 Virtex-4 中，需要考虑 Chain 的移动，Chain 的移动是整体的，因此如果选定的交换 Block 位于一条 Chain 上，那么需要判断能够将这条 Chain 完全交换或移动，如果不同，那么这次交换也就不能发生。

(3) 产生交换 Block 的算法

在 VPR 中，外围全部是 IOB，中间全部是 CLB，布局时在随机选定了起点 Block 之后，先判断当前 Block 是 CLB 还是 IOB，然后分别针对不同类型的 Block 在当前范围内随机选取 Block 进行交换。而在 Virtex-4 平台中，从图 5-6 中可以看到，不同类型的逻辑块的分布并不均匀，例如 IOB 并不仅仅分布在外围，Block RAM 列穿插在 CLB 中间。因此在 Virtex-4 中，由于分布的不同，在选定终点位置时，交换的范围也是不相同的，需要有目的性的选取，因此交换的算法不相同。

(4) 定向移动(Directed Move)

传统的模拟退火算法完全靠随机在一个由退火温度决定的窗口范围内来提出

交换的逻辑块。然而在实际的 FPGA 中，更多的是定向的移动，即有方向性的移动，而不是盲目的随机查找。定向移动意味着交换的提出是部分随机的，起始逻辑块和终点位置都是在给定优化条件的部分区域内随机产生的。定向移动可以更有效的完成布局和节约布局的时间。

5.5.2 软件测试环境的选取

(1) 测试平台

前面介绍过，本文的并行算法需要在保证质量不下降、满足确定性和串行等价性的前提下实现加速。因此本文测试的主要关注点有几个方面：布局质量，布局时间，多次布局的确定性，不同平台布局的等价性。

本文选定的测试计算机配置如表 5-1 所示。

表 5-2 测试计算机详细配置

CPU 型号	Inter Core(TM) i5-2500k@3.30GHz	Intel Xeon （R） E5-2643 V2@3.50GHz
物理 CPU 数目	1	2
逻辑 CPU 数目	4	24
单个 CPU 核数	4	6
超线程	否	是
L1 Cache	各核心独立 I 32K，D32K	各核心独立 I 32K，D 32K
L2 Cache	各核心独立 256K	各核心 256K
L3 Cache	共享 6MB	共享 25MB
内存	16GB	500GB
Linux 内核	linux 2.6.32 ubuntu x86_64	Linux 2.6.18 redhat x86_64

以下为了简化，本文称采用 Inter Core(TM) i5-2500k@3.30GHz 型号 CPU 的计算机为计算机 A，称采用 Intel Xeon（R）E5-2643 V2@3.50GHz 型号 CPU 的计算机为计算机 B。

(2) 测试案例-MCNC 基准电路：

MCNC 基准电路是北卡罗来纳微电子中心(美国)提供的一系列标准电路,它们已成为关于 FPGA 架构和布局布线方面研究的标准单元,基本上这方面研究 FPGA 低功耗布局布线算法的研究和改进都是基于这些标准电路的,我们在研究中利用这些标准电路来测试改进后系统的性能。表 5-3 描述了常用的 MCNC 电路内部资源,包括逻辑块数、网络数、CLB 数、输入端口数和输出端口数等信息。

表 5-3 常用 MCNC 电路资源统计

Circuit	Num of Blocks	Num of Nets	Num of CLBs	Num of Inputs	Num of Outputs
9symml	107	106	97	9	1
alu2	213	207	197	10	6
apex7	188	151	102	49	37
e64	404	339	274	65	65
example2	289	223	138	85	66
k2	609	564	519	45	45
terml	132	122	88	34	10
too-lrg	228	225	187	38	3
vda	374	308	291	17	39

5.6 布局工具的关键函数设计

(1) runMapper()

在主线程创建了 Mapper 线程之后，Mapper 开始执行 runMapper()函数，如图 5-10 布局工具中 Mapper 函数执行流程图所示。Mapper 首先判断退火过程是否结束，如果退火没有结束，那么 Mapper 判断当前温度是否更新，如果温度更新了，说明模拟退火算法刚刚完成某个温度上的一次迭代过程或者刚刚启动退火过程，此时 Mapper 分发一次迭代过程中的所有 Move；如果温度没有更新，Mapper 线程进入阻塞状态。所有迭代完成后，退火结束后，Mapper 会往任务队列中放入一个空的任务，目的是为了让 Worker 也能随后结束。

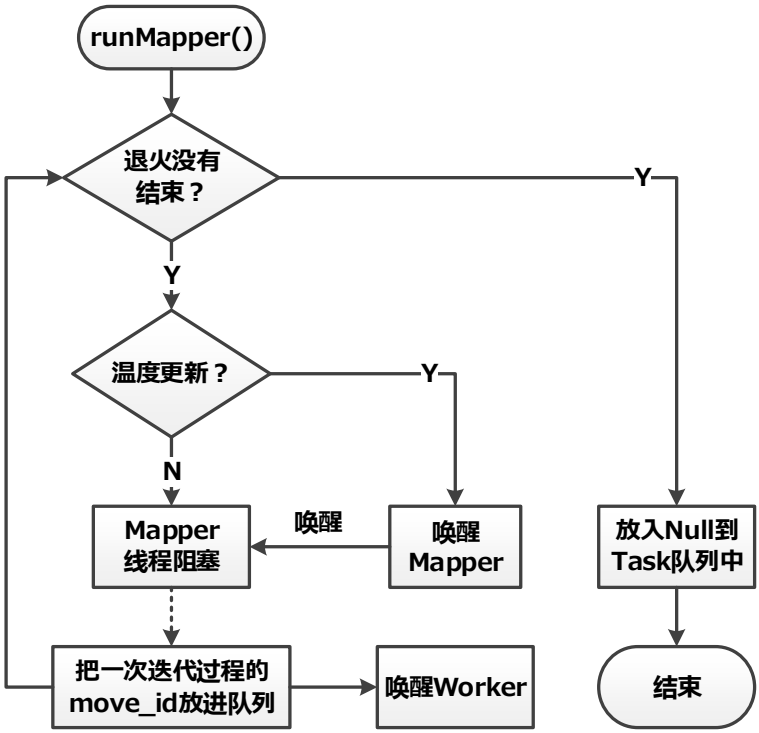


图 5-10 布局工具中 Mapper 函数执行流程图

(2) runWorker()

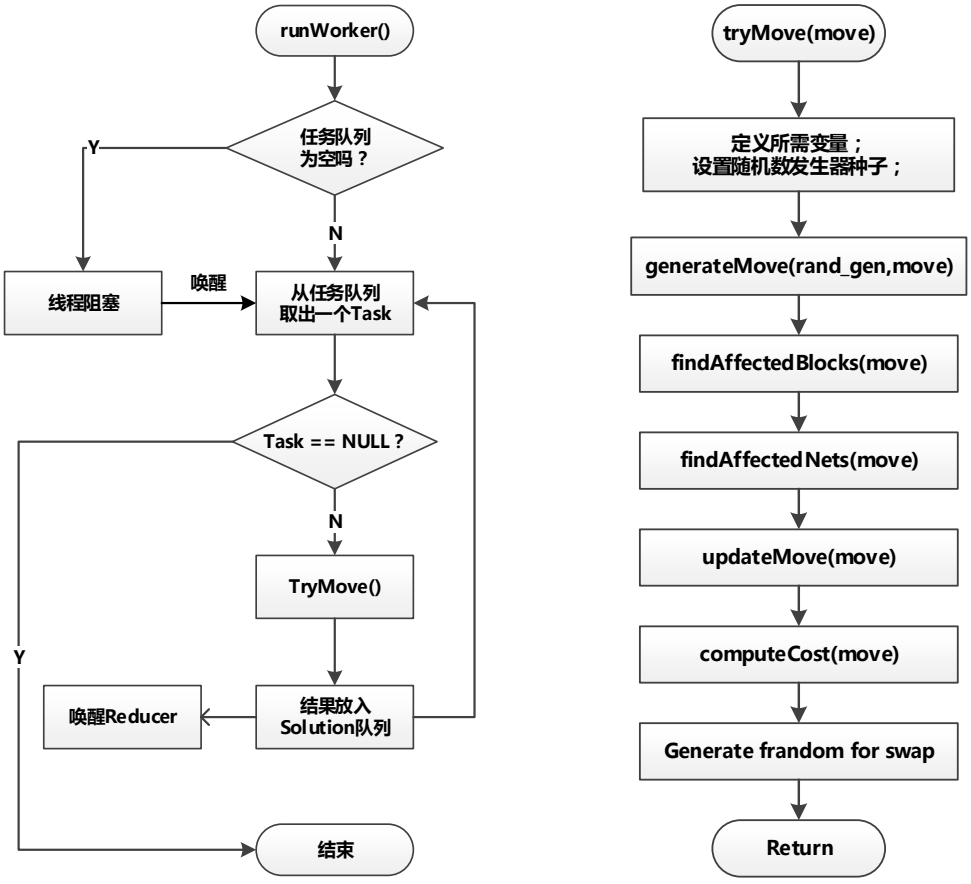


图 5-11 布局工具中 Worker 函数执行流程图

Worker 工作的内容的主要是源源不断的从任务队列中取出任务并执行，将执行结果放入结果队列，如图 5-11 所示。在从任务队列中取出任务时，如果任务队列为空，则该 Worker 线程将进入阻塞状态，直到 Mapper 或者 Reducer 向任务队列中放入新的任务时将其唤醒。

Worker 线程执行过程中最核心的部分是 tryMove 函数。tryMove 函数主要分为六步：产生随机移动、找到受影响的逻辑块、找到受影响的线网、计算交换导致的线网边界框变化、计算交换导致的线网代价变化和判断该交换是否被接受。

(3) runReducer()

Reducer 的执行过程如图 5-12 所示。Reducer 在退火没有结束的情况下按序（为了保证串行等价性）从 Reducer 中取出已完成的 Move 的，首先检查该 Move 是否冲突，并判断出冲突的类型。如果是位置或者逻辑块冲突，那么 Reducer 将该 Move 进行标记后重新放入任务队列等待 Worker 执行；如果是线网冲突，那么 Reducer 只需要重新处理计算冲突线网的边界框和代价，然后继续提交。如果交换被接受，那么更新全局的数据结构，并继续取出下一个 Move；如果交换被拒绝，不对当前数据做任何改动，继续取出下一次 Move。另外，如果 Reducer 需要处理的下一个 Move 还没有完成，那么 Reducer 将进入阻塞状态，在 Worker 完成了 Reducer 需要的 Move 之后由 Worker 唤醒 Reducer。

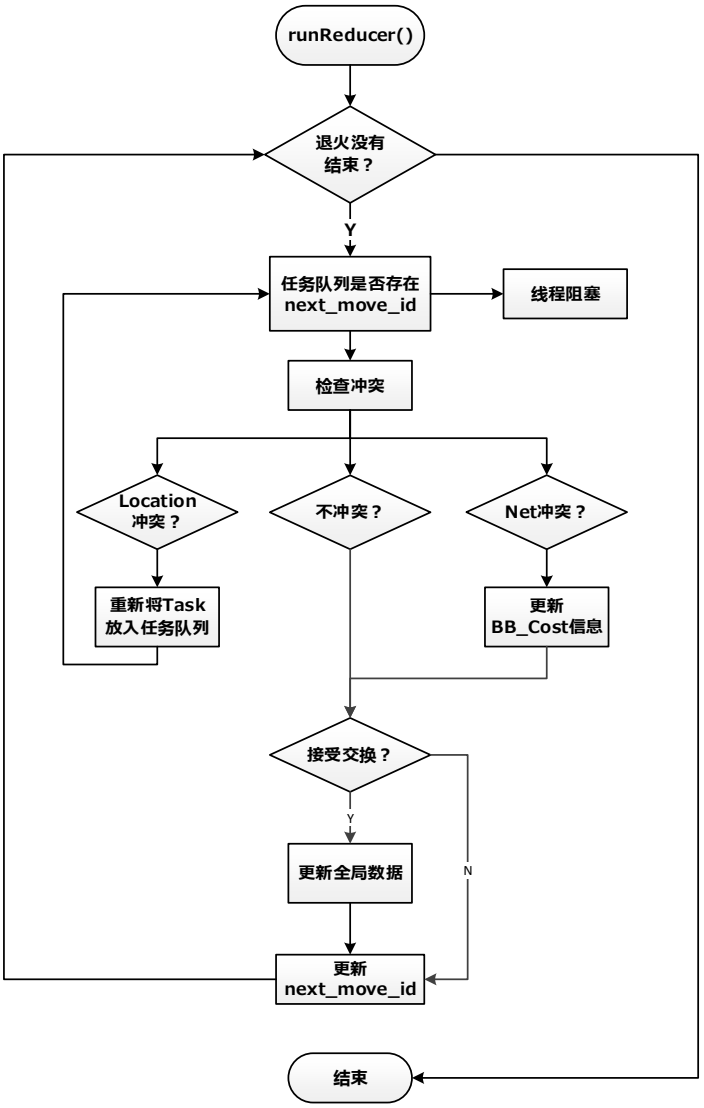


图 5-12 布局工具中 Reducer 函数的执行流程图

5.7 布局工具的测试结果

5.7.1 随机数性能的测试

本文将 VPR 中采用的线性同余随机数发生器换成了 Coveyou 随机数发生器并以 Move 序列的 ID 作为随机数发生器的种子，并进行了相应测试。

测试过程中，采用 VPR 随机数发生器的程序和仅仅采用了 Coveyou 随机数发生器并没有采用多核布局的程序分别对 MCNC 基准电路布局，得到的布局质量和运行时间结果对比如下：

表 5-4 分别采用 Coveyou 随机数和线性同余随机数布局结果比较

case\bb_cost	**PRNG**	**Coveyou**	**比较**
b09_opt	0.738801	0.72232	-2.2308%
b09	0.706576	0.640673	-9.3271%
b10	1.29084	1.3557	5.0246%
alu2	1.57072	1.607	2.3098%
lal	2.11636	2.11529	-0.0506%
cbp_16_4	2.20923	2.38479	7.9467%
count	3.56489	3.56654	0.0463%
ttt2	3.17092	3.17686	0.1873%
b9	3.62517	3.64401	0.5197%
too_large	6.05597	5.98034	-1.2489%
apex7	8.86842	8.95517	0.9782%
ex5p_mapped	8.7855	8.77141	-0.1604%
ex5p	8.69116	8.69593	0.0549%
fsm8_8_13	8.46968	8.29092	-2.1106%
vda	12.9908	12.8748	-0.8929%
ex4p	10.7695	11.1328	3.3734%
example2	24.2209	24.2885	0.2791%
clma_mapped	17.9932	18.0328	0.2201%
x4	27.9401	28.2625	1.1539%
e64	31.4278	31.657	0.7293%
k2_2	33.1976	33.5477	1.0546%
misex3	31.125	30.6928	-1.3886%
apex2	55.0431	54.8854	-0.2865%
diffeq	37.4143	37.048	-0.9790%
frisc	204.112	205.204	0.5350%
b15_1_opt	205.712	209.188	1.6897%
平均误差			0.2857%

可以看到，平均误差约为 0.2857%，因此采用 Coveyou 随机数发生器之后布局质量下降了 0.2857%，这是在可以接受的范围内的。因为采用线性同余随机数发生器时，所有的随机数基于同一个种子而产生，而采用 Coveyou 随机数后，所有的随机数是基于一系列的连续的 Move Id 作为种子产生的，种子之间的相关性较弱，所以布局质量的下降是合理的。

5.7.2 确定性和串行等价性的测试

对于两台测试计算机，分别使用 1 个 woker，2 个 worker，4 个 worker 和 8 个 worker，对 MCNC 的部分例子，均运行 40 次得到的结果如下所示。其中，表

5-5 是在计算机 A 上的运行结果，表 5-6 是在计算机 B 上的运行结果。

表 5-5 使用计算机 A 测试串行等价性的结果

case\bb_cost	computer A				标准差
	1 Worker	2 Worker	4 Worker	8 Worker	
b09_opt	0.72232	0.72232	0.72232	0.72232	0%
b09	0.640673	0.640673	0.640673	0.640673	0%
b10	1.3557	1.3557	1.3557	1.3557	0%
alu2	1.607	1.607	1.607	1.607	0%
lal	2.11529	2.11529	2.11529	2.11529	0%
cbp_16_4	2.38479	2.38479	2.38479	2.38479	0%
count	3.56654	3.56654	3.56654	3.56654	0%
ttt2	3.17686	3.17686	3.17686	3.17686	0%
b9	3.64401	3.64401	3.64401	3.64401	0%
too_large	5.98034	5.98034	5.98034	5.98034	0%
apex7	8.95517	8.95517	8.95517	8.95517	0%
ex5p_mapped	8.77141	8.77141	8.77141	8.77141	0%
ex5p	8.69593	8.69593	8.69593	8.69593	0%
fsm8_8_13	8.29092	8.29092	8.29092	8.29092	0%
vda	12.8748	12.8748	12.8748	12.8748	0%
ex4p	11.1328	11.1328	11.1328	11.1328	0%
example2	24.2885	24.2885	24.2885	24.2885	0%
clma_mapped	18.0328	18.0328	18.0328	18.0328	0%
x4	28.2625	28.2625	28.2625	28.2625	0%
e64	31.657	31.657	31.657	31.657	0%
k2_2	33.5477	33.5477	33.5477	33.5477	0%
misex3	30.6928	30.6928	30.6928	30.6928	0%
apex2	54.8854	54.8854	54.8854	54.8854	0%
diffeq	37.048	37.048	37.048	37.048	0%
frisc	205.204	205.204	205.204	205.204	0%
b15_1_opt	205.712	205.712	205.712	205.712	0%

表 5-6 使用计算机 B 测试串行等价性的结果

case\bb_cost	computer B				标准差
	1 Worker	2 Worker	4 Worker	8 Worker	
b09_opt	0.72232	0.72232	0.72232	0.72232	0%
b09	0.64067	0.64067	0.640673	0.640673	0%
b10	1.3557	1.3557	1.3557	1.3557	0%
alu2	1.607	1.607	1.607	1.607	0%
lal	2.11529	2.11529	2.11529	2.11529	0%
cbp_16_4	2.38479	2.38479	2.38479	2.38479	0%

count	3.56654	3.56654	3.56654	3.56654	0%
ttt2	3.17686	3.17686	3.17686	3.17686	0%
b9	3.64401	3.64401	3.64401	3.64401	0%
too_large	5.98034	5.98034	5.98034	5.98034	0%
apex7	8.95517	8.95517	8.95517	8.95517	0%
ex5p_mapped	8.77141	8.77141	8.77141	8.77141	0%
ex5p	8.69593	8.69593	8.69593	8.69593	0%
fsm8_8_13	8.29092	8.29092	8.29092	8.29092	0%
vda	12.8748	12.8748	12.8748	12.8748	0%
ex4p	11.1328	11.1328	11.1328	11.1328	0%
example2	24.2885	24.2885	24.2885	24.2885	0%
clma_mapped	18.0328	18.0328	18.0328	18.0328	0%
x4	28.2625	28.2625	28.2625	28.2625	0%
e64	31.657	31.657	31.657	31.657	0%
k2_2	33.5477	33.5477	33.5477	33.5477	0%
misex3	30.6928	30.6928	30.6928	30.6928	0%
apex2	54.8854	54.8854	54.8854	54.8854	0%
diffeq	37.048	37.048	37.048	37.048	0%
frisc	205.204	205.204	205.204	205.204	0%
b15_1_opt	209.188	209.188	209.188	209.188	0%

可以从比较结果的标准差看到，首先在同一台计算机上无论采用多少个 Worker 所有的运行结果都是相同的，采用的 Worker 数目不同，意味着同时进行第一阶段交换计算的 CPU 核数不同；再比较两台不同电脑上的运行的结果，也是完全相同的，所以最终的布局结果是串行等价性的。

5.7.3 加速比的测试

对于两台测试计算机，分别使用 1 个 worker，2 个 woker，4 个 worker 和 8 个 worker，对 MCNC 的部分例子，运行程序并统计布局质量和布局时间。

表 5-7 使用计算机 A 测试加速比的结果

Case \runtime	**computer A**								
	串行	**1 Worker**		**2 Worker**		**4 Worker**		**8 Worker**	
b09_opt	0.79	1.48	188%	4.01	508%	5.16	654%	7.06	894%
b09	0.77	1.39	180%	3.83	497%	5.00	649%	6.79	882%
b10	0.83	1.22	147%	3.76	453%	4.86	585%	6.67	803%
alu2	0.88	1.57	179%	3.93	446%	5.20	591%	6.99	794%
lal	0.66	1.46	221%	3.97	601%	5.24	794%	7.28	1103%
cbp_16_4	0.92	1.36	148%	3.93	428%	5.11	556%	6.73	732%

count	0.7	1.39	198%	4.36	623%	5.59	799%	8.05	1150%
ttt2	0.71	1.25	177%	3.97	560%	5.03	709%	7.02	989%
b9	0.71	1.27	179%	4.16	586%	5.44	766%	7.47	1052%
too_large	1.06	1.39	131%	4.09	386%	5.24	495%	6.97	658%
apex7	0.83	1.44	173%	4.37	526%	5.68	685%	7.89	951%
ex5p_mapped	0.97	1.28	132%	4.17	430%	5.45	562%	7.44	767%
ex5p	0.94	1.37	146%	4.39	467%	5.57	592%	7.63	811%
fsm8_8_13	0.86	2.06	240%	6.64	772%	8.25	960%	10.98	1277%
vda	0.88	2.06	234%	6.10	693%	7.66	871%	10.30	1170%
ex4p	0.91	1.27	140%	4.01	441%	5.11	561%	6.88	756%
example2	0.91	1.36	150%	4.65	511%	6.04	664%	8.31	913%
clma_mapped	0.79	1.25	158%	4.41	558%	5.82	736%	8.21	1039%
x4	0.85	1.35	158%	4.65	546%	5.82	685%	7.96	936%
e64	0.64	1.49	233%	4.71	735%	5.89	920%	8.90	1391%
k2_2	2	4.42	221%	12.61	630%	15.46	773%	19.99	1000%
misex3	3.83	9.61	251%	26.71	697%	32.95	860%	43.89	1146%
apex2	5.68	14.58	257%	40.31	710%	48.56	855%	62.61	1102%
diffeq	7.04	18.06	256%	51.94	738%	63.61	904%	83.69	1189%
frisc	49.14	124.30	253%	308.75	628%	362.39	737%	443.37	902%
b15_1_opt	54.73	121.68	222%	301.16	550%	353.91	647%	424.78	776%
平均时间比			191%		566%		716%		969%

表 5-8 使用计算机 B 测试加速比的结果

Case \runtime	**computer A**								
	串行	**1 Worker**		**2 Worker**		**4 Worker**		**8 Worker**	
b09_opt	2.4	3.91	163%	9.48	395%	4.98	208%	6.78	283%
b09	2.4	4.71	196%	10.28	428%	12.83	535%	9.20	383%
b10	2.43	4.79	197%	9.07	373%	4.61	190%	11.08	456%
alu2	2.68	5.32	199%	9.53	356%	9.98	373%	9.50	355%
lal	2.26	5.59	247%	9.77	432%	7.28	322%	9.69	429%
cbp_16_4	2.53	3.77	149%	9.36	370%	7.41	293%	9.94	393%
count	2.37	4.59	194%	11.72	494%	6.08	256%	8.00	337%
ttt2	2.74	5.75	210%	9.31	340%	4.78	174%	11.37	415%
b9	2.29	5.20	227%	7.51	328%	5.43	237%	11.67	510%
too_large	2.78	5.09	183%	9.35	336%	6.63	238%	9.44	340%
apex7	2.64	5.40	204%	12.55	476%	6.05	229%	9.19	348%
ex5p_mapped	2.88	5.48	190%	9.74	338%	5.23	182%	9.81	341%
ex5p	2.71	5.29	195%	11.55	426%	11.04	407%	7.25	268%
fsm8_8_13	3.7	9.05	245%	17.44	471%	7.82	211%	15.14	409%
vda	3.42	7.30	213%	16.15	472%	7.81	229%	18.62	545%
ex4p	2.67	4.91	184%	6.49	243%	4.79	179%	10.96	410%

example2	2.86	5.95	208%	11.26	394%	5.56	194%	14.06	492%
clma_mapped	2.69	4.12	153%	11.35	422%	5.55	206%	13.51	502%
x4	2.73	2.59	95%	10.68	391%	11.63	426%	13.06	478%
e64	2.55	2.14	84%	11.40	447%	7.37	289%	13.55	531%
k2_2	7.22	7.14	99%	32.38	448%	21.94	304%	28.61	396%
misex3	15.21	27.0	178%	76.17	501%	74.93	493%	62.44	411%
apex2	22.47	31.7	141%	112.4	500%	64.31	286%	87.83	391%
diffeq	27.16	55.3	204%	134.2	494%	144.6	533%	114.33	421%
frisc	146.4	184	126%	532.7	364%	615.4	420%	664.30	454%
b15_1_opt	163.6	243	149%	558.5	341%	591.4	362%	616.55	377%
平均时间比			178%		407%		299%		410%

其中，表 5-7 是计算机 A 上运行的结果。表 5-8 是在计算机 B 上运行的结果。两者的结果统计如下图所示：

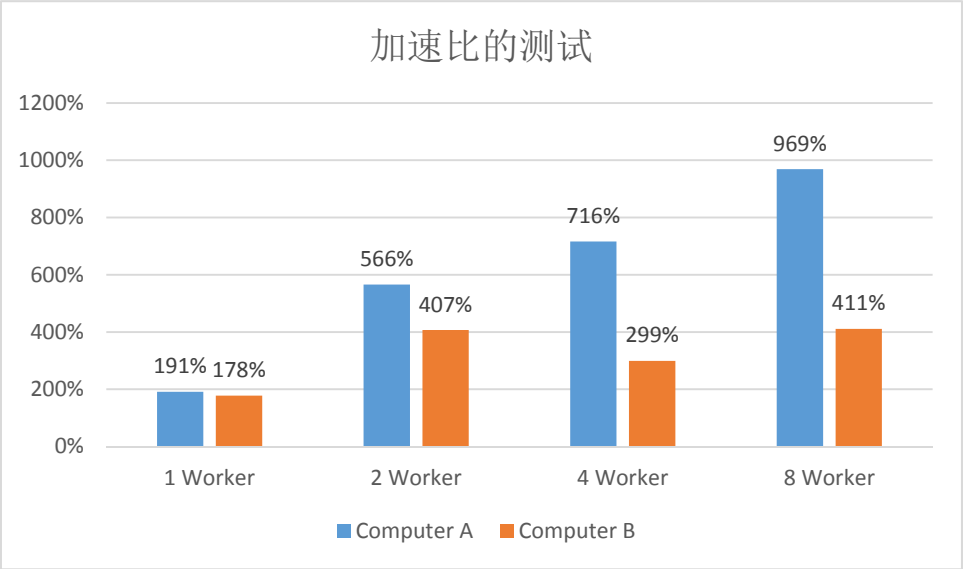


图 5-13 两台电脑上的加速比的测试结果

可以很明显的看出，目前的加速比测试结果很不理想。在一个 Worker 的时候，运行时间几乎达到了串行版本代码的两倍；而且随着 Worker 数目的增加，布局时间越来越长，在 8 个 Worker 的时候，计算机 A 上的运行时间几乎达到了十倍的时间。

经过一些调试之后，分析原因有以下几点：

（1） 代码还没有优化好。目前代码的测试结果中，Reducer 运行一次提交一个 Move 的时间与 Woker 提出并处理一个 Move 的时间大约相等，这意味着，

每次 Worker 提交一个 Move 之后，继续去执行下一次 Move 到完成，Reducer 刚刚才能对 Worker 这次提交的 Move 的处理。也就意味着这个算法目前与串行的版本效率几乎等价，而且由于并行结构中，每个 Worker 中不能更新全局的数据结构，因此 Worker 需要进行很多额外的存取操作，所以 1 个 Worker 版本的代码运行时间肯定要长于串行版本。更重要的一点是，在并行结构中，由于 Woker 处理 Move 的阶段还不能拿到真正正确的全局数据，因此还可能存储冲突，需要 Reducer 进行解决，这导致一个 Worker 的速度更慢了；

（2） 由于第一点原因的存在，单纯的增加 Woker 对问题的解决没有任何帮助，因为一个 Reducer 只能来得及处理一个 Worker 的工作，甚至可能还来不及，Worker 数目变多，在 Reducer 处理一个 Move 的阶段，其他 Woker 可能已经完成了很多次的 Move 计算，而这些 Move 中一大部分是冲突的，因为 Reducer 处理速度跟不上，特别是在存在大线网的时候。

（3） 由第二点知道 Worker 的增加只会导致冲突 Move 的增加。而且在 Worker 增加的，程序的线程数目也会增加，而线程的调度由操作系统完成，虽然操作系统会最优化调度，但是频繁的线程切换导致的时间消耗是不可忽视的。这几点原因导致了 Worker 越多，运行时间越长；

Reducer 的运行时间不应该比 Worker 的时间长，因为 Work 而需要完成随机产生 Move、查找受影响的逻辑块和线网、计算移动后的线网边界框变化和计算移动后的代码函数变化这些部分；而 Reducer 只需要取出 Move、判断冲突和更新数据结构这几步。所以理想的情况应该是 Worker 执行一个 Move 的时间里，Reducer 能完成 4 到 8 个 Move 的提交，这也是本文需要继续努力优化程序的目标。

目前 Reducer 程序的优化方案主要有两点：(1) 优化存取 Move 的时间。一方面前面已经通过内存管理器来优化 Move 的存取空间和时间。另外一方面本文采用堆队列作为 Worker 存放 Move 的容器，这是因为 Worker 不需要按序将 Move 放入队列，但是 Reducer 存取时需要按序存取，而堆插入过程可以实现自动调整，时间复杂度为 O(nlogn)，并且对于大文件的效果比较显著。但是在本算法中，需要优化堆队列的存取速度，可能考虑使用 Vector 取代堆；(2) 优化解决和判断冲突的时间。目前解决冲突需要判断逻辑块、逻辑坐标和线网的时间戳，浪费了许多时间，可以考虑使用内存跟踪来优化判断的时间，并且可以通过更加细化冲突的情况来减少重复的计算。

5.8 本章小结

本章首先简单介绍了基于并行交换的并行化算法实现工具，由于该算法基于VPR 改进，因此也简单介绍了 VPR 软件的运行界面。接着再介绍了该软件所采用的并行编程语言。由于软件是基于现代实用 FPGA 结构开发，因此本章介绍了算法的输入输出文件、Virtex4 芯片的结构和相应的测试方法。最后，本章对算法的关键实现部分进行了介绍并对该算法的性能进行了测试和比较。

第六章 总结与展望

6.1 工作总结

本文致力于基于模拟退火的 FPGA 布局算法的并行化，通过使用多核对布局过程的并行处理来达到加速的过程，并在算法的实现过程中保证布局的质量、算法的确定性和串行等价性。

本文采用了基于多核并行交换的模拟退火算法来进行 FPGA 布局，并使用 C++技术实现了基于该算法的软件工具。

本文详细分析了基于多核并行交换的 FPGA 布局算法的原理和步骤，同时对算法的性能和运行时间提出了相应的改进建议。本文将模拟退火算法中计算量庞大的交换过程分配给多核 CPU 完成，通过多线程的调度和资源分配实现了算法。同时，本文通过对随机数产生算法的改进保证了布局质量的稳定性，通过对交换冲突的解决保证了算法运行的确定性和串行等价性。

本文还基于多核并行交换布局算法开发了相应的并行软件布局工具。该工具能够读取 Xilinx 软件工具生成的 XDL 设计文件进行布局并生成对应的 XDL 文件进行测试。本文软件工具的实现难点主要有二：第一，为了保证串行等价性和确定性，算法需要对各种冲突情况进行判断，同时需要对线程之间的数据竞争进行调试，比较艰难；第二，布局算法每次交换达到一百多万次，如果中途运行出错，调试时基本上是无法跟踪的，而输出 log 文件也是达 GB 量级，处理时间很长，庞大的数据量导致程序调试的困难。

本文对基于多核并行交换的并行布局算法进行了相应测试。首先算法在更换随机数之后能继续保持良好的良好的布局质量。其次，算法经测试是确定性的和串行等价性的，并且保证了较好的布局质量。再次，算法实现的加速比不太理想，程序需要继续优化。

6.2 工作展望

因为时间有限，本文提出的基于多核交换的 FPGA 并行布局算法仍然存在许多需要改进的地方。

首先本文实现的加速比仍然有限，实际上这种算法理论上 N 核的加速比可以达到 N 倍，即使考虑到内存的有限带宽、拷贝数据的存取和冲突的重新处理等因素，也应该能达到 2-3 倍的加速比，因此本文的程序还有待继续优化。

另外，本文的程序中计算代价时没有考虑时序信息，而像 Xilinx，Altera 等公司的实用布局工具中都是基于时序驱动的模拟退火算法，因此时序信息在布局过程中是很重要的。而在考虑了时序信息后，本文提出的基于多核交换的并行布

局算法是需要修改的，需要考虑很多的细节，比如冲突的划分情况需要复杂一些。因此这是该工具需要继续改进优化的一个方面。

第三，在模拟退火的后期，交换被接受的概率很低，大部分交换都是被拒绝，但是在当前算法中，对于每次 Move，仍然需要检查冲突后按序出队，这需要浪费很多的时间，因此在退火后期对于被拒绝的 Move 其实可以不需要按序检测冲突，可以直接处理掉，等到后面检测到确实冲突时再重新处理即可，这也是算法可以优化的方面。

一个好的算法，总是在不断的测试和使用中不断改进，模拟退火算法是如此，并行化退火算法也是如此。算法到真正成熟还需要很多的努力。希望在接下来的时间里能够将算法优化到良好的性能。

参考文献

[1] 王伶俐，周学功，王颖. 系统级 FPGA 设计与应用.北京：清华大学出版社,2012.

[2] 王臻,来金梅. 一种基于并行模拟退火的 FPGA 布局算法[J]. 复旦学报(自然科学版),2012,06:716-721.

[3] Betz V, Rose J. VPR: A new packing, placement and routing tool for FPGA research[C]//Field-Programmable Logic and Applications. Springer Berlin Heidelberg, 1997: 213-222.

[4] Ludwin A, Betz V. Efficient and deterministic parallel placement for FPGAs. ACM Transaction onDesign Automation of Electronic System . 2011

[5] Cameron Hughes, Tracey Hughes. C++多核高级编程. 齐宁，译. 北京：清华大学出版社,2010: 19-22.

[6] Kravitz S, Rutenbar R. Placement by simulated annealing on a multiprocessor. IEEE Transaction onComputer-Aided Design of Integrated Circuits and Systems . 1987.

[7] Malay Haldar, Anshuman Nayak, Alok Choudhary, Prith Banerjee. Parallel Algorithms For FPGA Placement. Great Lakes Symposium on VLSI,Proceedings of the 10th Great Lakes symposium on VLSI . 2000.

[8] Wang C, Lemieux G. Scalable and Deterministic Timing-driven Parallel Placement for FPGAs. Proceedings of2011 International ACM/SIGDA Symposium on FieldProgrammable Gate Arrays . 2011.

[9] Witte,E.E., Chamberlain,R.D., Franklina,M.A. Parallel simulated annealing using speculative computation. IEEE Transactions on Parallel and Distributed Systems . 1991.

[10] J. A. Chandy, B Pfithviraj. Parallel simulated annealing strategies for VLSI cell placement. 9th International Conference on VLSI Design . 1996.

[11] Ludwin A, Betz V, Padalia K. High-quality, deterministic parallel placement for FPGAs on commodity hardware[C]//Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays. ACM, 2008: 14-23.

[12] Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters[J]. Communications of the ACM, 2008, 51(1): 107-113.

[13] Donald E.Kunth. 计算机程序设计艺术（第二卷）半数值算法. 苏运霖，译.

北京：国防工业出版社, 2002: 1-35.
[14] Xilinx Inc. Virtex-4 FPGA User Guide.
http://www.xilinx.com/support/documentation/user_guides/ug070.pdf
[15] 章玮. 基于 FPGA 的 64 位 CPU 验证平台的建立[D].同济大学,2007.

致谢

在行将毕业之际，首先要衷心感谢我的导师陈更生老师，他不仅仅为我指明了正确的研究方向，让我有了进入 EDA 领域锻炼的机会，还让我参与多个重要的工程项目，让我在学术、能力和项目管理上都得到了有益的锻炼，为以后的学习打下了坚实的基础。同时，陈老师在我的论文书写过程中给我很多切实的建议和帮助，陈老师认真严谨的学术态度和和蔼可亲的敦敦教诲给我留下了深刻印象。

同时要非常感谢复旦微电子公司软件组的 Michael、似飞、佐渭、小南、君君、林泉、卓远、吴老师、杨芳等，你们在我的毕业论文完成过程了给了我很多帮助，同时你们严谨认真的工作态度和卓越的编程能力等都给了我深深的印象和感染，跟你们在一起工作是件相当愉快的事情。

感谢跟我一起实习的连彩江、莫林峰、唐辉丰同学，你们在生活和学习上都给我很多帮助。同时向 CAD 实验室的所有师兄师姐师弟师妹表示感谢，有了你们让 CAD 实验室成为学习和生活的乐园。

还要感谢我的室友们，以及一起在张江度过三年本科生活的 10 微电所有同学，生命中有了你们的出现才更精彩。

最后我要深深感谢我的家人对我默默的支持！