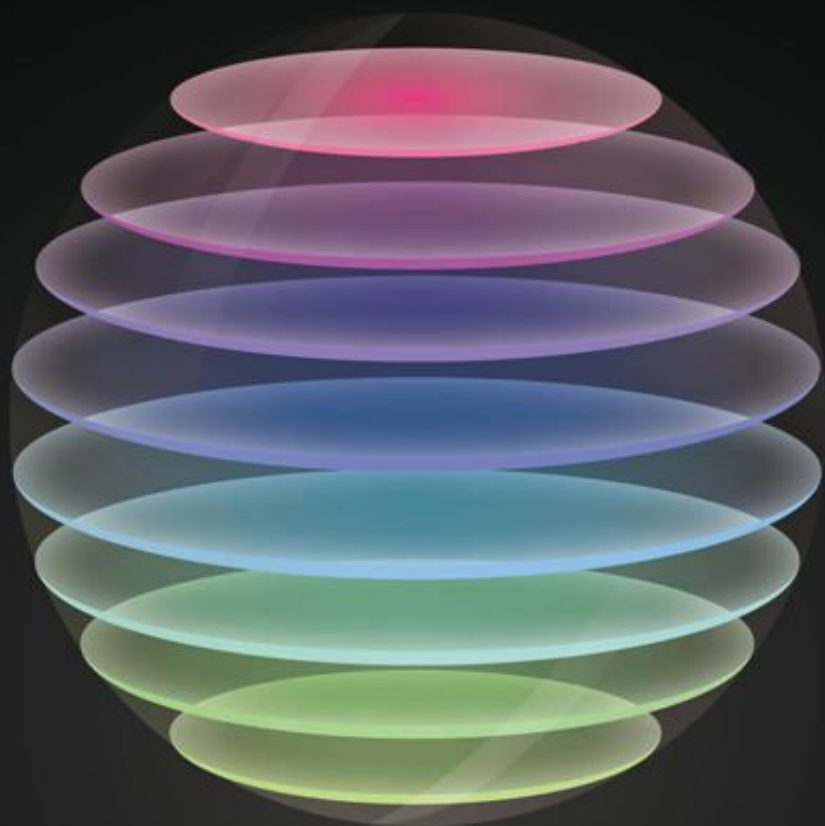


并行计算领域著名专家撰写，百度深度学习研究院“杰出科学家”吴勃鼎力推荐
结合大量示例和伪代码，全面讲解如何通过并行算法设计实现单核/多核处理器、GPU和移动处理器的性能优化及相关的并行化秘技，并首次提出实现复杂度的全新性能度量标准



Parallel Computing and Performance Optimization

并行算法设计 与性能优化

刘文志◎著



机械工业出版社
China Machine Press

高性能计算技术丛书

并行算法设计与性能优化

刘文志 著

ISBN: 978-7-111-50102-2

本书纸版由机械工业出版社于2015年出版，电子版由华章分社（北京华章图文信息有限公司，北京奥维博世图书发行有限公司）全球范围内制作与发行。

版权所有，侵权必究

客服热线：+ 86-10-68995265

客服信箱：service@bbbvip.com

官方网址：www.hzmedia.com.cn

新浪微博 @研发书局

腾讯微博 @yanfabook

目录

前言

第1章 绪论

- 1.1 并行和向量化的作用
- 1.2 为什么要并行或向量化
- 1.3 为什么向量化或并行难
- 1.4 并行的替代方法
- 1.5 进程、线程与处理器
- 1.6 并行硬件平台
- 1.7 向量化和多核技术不是万能的
- 1.8 本章小结

第2章 现代处理器特性

- 2.1 指令级并行
 - 2.1.1 指令流水线
 - 2.1.2 乱序执行
 - 2.1.3 指令多发射
 - 2.1.4 分支预测
 - 2.1.5 VLIW
- 2.2 向量化并行
 - 2.2.1 SIMD
 - 2.2.2 SIMT
- 2.3 线程级并行
 - 2.3.1 内核线程和用户线程
 - 2.3.2 多线程编程库
 - 2.3.3 多核上多线程并行要注意的问题
 - 2.3.4 多线程程序在多核和单核上运行的不同
- 2.4 缓存
 - 2.4.1 缓存层次结构
 - 2.4.2 缓存一致性
 - 2.4.3 缓冲不命中
 - 2.4.4 写缓存

2.4.5 越过缓存

2.4.6 硬件预取

2.4.7 缓存结构

2.4.8 映射策略

2.5 虚拟存储器和TLB

2.6 NUMA技术

2.7 本章小结

第3章 算法性能和程序性能的度量与分析

3.1 算法分析的性能度量标准

3.1.1 时间复杂度与空间复杂度

3.1.2 实现复杂度

3.2 程序和指令的性能度量标准

3.3 程序性能优化的度量标准

3.3.1 加速比与并行效率

3.3.2 Amdahl定律和Gustafson定律

3.4 程序性能分析实用工具

3.5 本章小结

第4章 串行代码性能优化

4.1 系统级别

4.2 应用级别

4.3 算法级别

4.4 函数级别

4.4.1 函数调用参数

4.4.2 内联小函数

4.5 循环级别

4.5.1 循环展开

4.5.2 循环累积

4.5.3 循环合并

4.5.4 循环拆分

4.6 语句级别

4.6.1 减少内存读写

4.6.2 选用尽量小的数据类型

4.6.3 结构体对齐

4.6.4 表达式移除

4.6.5 分支优化

4.6.6 优化交换性能

4.7 指令级别

4.8 本章小结

第5章 依赖分析

5.1 指令级依赖

5.1.1 结构化依赖

5.1.2 数据依赖

5.1.3 控制依赖

5.2 循环级依赖

5.2.1 循环数据依赖

5.2.2 循环控制依赖

5.3 寄存器重命名

5.4 本章小结

第6章 并行编程模型及环境

6.1 并行编程模型

6.1.1 指令级并行

6.1.2 向量化并行

6.1.3 易并行

6.1.4 任务并行

6.1.5 数据并行

6.1.6 循环并行化

6.1.7 流水线并行

6.1.8 区域分解并行

6.1.9 隐式和显式并行化

6.1.10 SPMD

6.1.11 共享存储器并行

6.1.12 分布式存储器并行

6.2 常见并行编程环境

6.2.1 MPI

6.2.2 OpenMP

6.2.3 fork/pthread

6.2.4 CUDA

6.2.5 OpenCL

6.2.6 OpenACC

6.2.7 NEON内置函数

6.2.8 SSE/AVX内置函数

6.3 本章小结

第7章 并行算法设计方法

7.1 划分

7.1.1 分而治之

7.1.2 划分原则

7.1.3 常见划分方法

7.1.4 并行性和局部性

7.2 通信

7.2.1 操作的原子性

7.2.2 结果的可见性

7.2.3 顺序一致性

7.2.4 函数的可重入与线程安全

7.2.5 volatile关键字

7.2.6 锁

7.2.7 临界区

7.2.8 原子操作

7.2.9 栅栏

7.3 结果归并

7.4 负载均衡

7.4.1 静态负载均衡

7.4.2 动态负载均衡

7.4.3 动态负载均衡算法的一般步骤

7.5 本章小结

第8章 并行算法缺陷

8.1 启动结束时间

8.2 负载均衡

8.3 竞写

8.4 锁

8.4.1 死锁

8.4.2 活锁

8.5 饿死

8.6 伪共享

8.7 原子操作

8.8 存储器栅栏

8.9 缓存一致性

8.10 顺序一致性

8.11 volatile同步错误

8.12 本章小结

第9章 并行编程模式实践

9.1 map模式

9.2 reduce模式

9.3 结合map和reduce模式

9.4 scan模式

9.5 zip/unzip模式

9.6 流水线模式

9.7 本章小结

第10章 如何并行遗留代码

10.1 找出软件的计算热点

10.2 判断是否并行化热点

10.3 设计算法并实现

10.3.1 选择何种工具进行向量化或并行化

10.3.2 重构热点代码

10.3.3 依据硬件实现算法

10.4 将实现后的代码嵌入原软件

10.4.1 混合编译

10.4.2 动态链接库

10.5 示例：如何并行化word2vec

10.6 本章小结

第11章 超级并行

11.1 超级并行方式编程

11.1.1 进程+线程

11.1.2 进程+GPU线程

11.1.3 线程+GPU线程

11.1.4 线程+向量指令

11.1.5 进程+线程+向量指令

11.1.6 进程+线程+GPU线程

11.2 矩阵乘法

11.2.1 多机CPU矩阵乘法

11.2.2 单机多GPU矩阵乘法

11.2.3 多机多GPU矩阵乘法

11.3 本章小结

第12章 并行算法设计的一般准则

12.1 并行算法设计14准则

12.2 本章小结

附录A 整型数据与浮点数据

前言

IT行业急需这本书

在解释为什么笔者认为IT行业急需这本书之前，先让笔者来介绍并行、并发和代码性能优化这三个概念，理解这三个概念是阅读本书的基础：

- 并行对应的英文单词是parallelism，是指在具有多个处理单元的系统上，通过将计算或数据划分为多个部分，将各个部分分配到不同的处理单元上，各处理单元相互协作，同时运行，以达到加快求解速度或者提高求解问题规模的目的。

- 并发对应的英文单词是concurrency，并发是指在一个处理单元上运行多个应用，各应用分时占用处理单元，是一种微观上串行、宏观上并行的模式，有时也称其为时间上串行、空间上并行。

- 代码性能优化是指通过调整源代码，使得其生成的机器指令能够更高效地执行，通常高效是指执行时间更少、使用的存储器更少或能够计算更大规模的问题。

从大的方面来说，并行和并发都是代码性能优化的一种方式，但是今天并行和并发已经是如此重要，需要“开宗立派”。为了清晰并行、并发和代码性能优化的边界，在本书中，代码性能优化特指除了并行和并发外的代码优化方法，比如向量化和提高指令流水线效率，在一些情况下，笔者也会将向量化独立来解说。

一般来说，并发是为了满足应用的功能需求，比如在计算的同时，用户界面能够响应用户；一个运行在16核处理器上的网络服务器需要同时支持64个用户而开启了64个线程。而并行更多的是为了提高速度或为了解决更大规模的问题。

人类生活的方方面面存在着并行或者并发，边吃饭边看电视，双手同时拔草，甚至吃饭时，嘴巴的动作和手的动作也是并行的。和人类社会广泛存在并行不同的是：计算机编程几乎一直都是串行的，绝大多数的程序只存在一个进程或线程（本书将它们统称为“控制流”）。对并行和向量化的研究可以追溯到20世纪60年代，但是直到近年来才得到广泛的关注，究其原因，主要是自2003年以来，能耗和散热问题限制了X86 CPU频率的提高，从而导致多核和向量处理器的广泛使用。

2003年以前，在摩尔定律的作用下，单核标量处理器的性能持续提升，软件开发人员只需要写好软件，而性能就等待下次硬件的更新。在2003年之前的几十年里，这种“免费午餐”的模式一直在持续。2003年后，主要由于功耗的原因，这种“免费的午餐”已经不复存在。为了生存，各硬件生产商不得不采用各种

方式以提高硬件的计算能力，以下是目前最流行的三种方式。

1) **让处理器一个周期处理多条指令**，这多条指令可相同可不同。如Intel Haswell处理器一个周期可执行4条整数加法指令、2条浮点乘加指令，同时访存和运算指令也可同时执行。

2) **使用向量指令**，主要是SIMD和VLIW技术。SIMD技术将处理器一次能够处理的数据位数从字长扩大到128或256位，从而提升了计算能力。

3) **在同一个芯片中集成多个处理单元**，根据集成方式的不同，分为多核处理器或多路处理器。多核处理器是如此的重要，以至于现在即使是手机上的嵌入式ARM处理器都已经是四核或八核。

目前绝大部分应用软件都是串行的，串行执行过程符合人类的思维习惯，易于理解、分析和验证。由于串行软件只能在多核CPU中的一个核上运行，这和2003年以前的CPU没有多少区别，这意味着花多核CPU的价钱买到了单核的性能。通过多核技术，硬件生产商成功地将提高实际计算能力的任务转嫁给软件开发人员，而软件开发人员没有选择只有直面挑战。

标量单核的计算能力没有办法接着大幅度提升，而应用对硬件计算能力的需求依旧在提升，这是个实实在在的矛盾。在可见的将来，要解决这个矛盾，软件开发人员只有代码优化和并行可以选择。代码优化并不能利用多核CPU的全部计算能力，它也不要求软件开发人员掌握并行开发技术，另外通常也无须对软件架构做改动，而且串行代码优化有时能够获得非常好的性能（如果原来的代码写得很差的话），因此相比采用并行技术，应当优先选择串行代码优化。一般来说采用并行技术获得的性能加速比不超过核数，这是一个非常大的限制，因为目前CPU硬件生产商最多只能集成十几、几十个核。

从2006年开始，可编程的GPU越来越得到大众的认可，GPU是图形处理单元（Graphics Processing Unit）的简称，最初主要用于图形渲染。自20世纪90年代开始，NVIDIA、AMD（ATI）等GPU生产商对硬件和软件加以改进，GPU的可编程能力不断提高，GPU通用计算比以前的GPGPU（General-Purpose Computing on Graphics Processing Units）容易许多，另外由于GPU具有比CPU强大的峰值计算能力，近来引起了许多科研人员和企业的兴趣。

近两三年来，在互联网企业中，GPU和并行计算越来越受到重视。无论是国外的Google、Facebook还是国内的百度、腾讯、阿里和360，都在使用代码优化、并行计算和GPU来完成以前不能完成的任务。

10年前，并行计算还是大实验室里面教授们的研究对象，而今天多核处理器和GPU的普及已经使得普通人就可以研究它们。对于软件开发人员来说，如果不掌握并行计算和代码性能优化技术，在不久的将来就会被淘汰。

代码性能优化和并行技术被许多顶级开发人员看成“不传之秘”或“只可意会，不可言传”的技术。本书将会把这些“不传之秘”一一展示在开发者面前，并且解释为什么。由于代码性能的具体细节非常难以解释清楚，笔者尽量在高层解释，避免陷入细节里。在写作此书时，我并没有查到世界上有类似的写给普通开发者的书籍，本书可算是第一本。

开发人员通常比较忙，因此本书力求简洁明了，点到为止即可。

读者对象

由于多核处理器和GPU已经非常便宜，而代码优化、向量化和并行已经深入IT行业的骨髓，所有IT行业的从业者都应当阅读本书，如果非要列一个清单，笔者认为下列人员应当阅读：

- 互联网及传统行业的IT从业者，尤其是希望将应用移植到多核向量处理器或GPU的开发人员。
- 大中专院校、研究所的学生和教授。

如何阅读本书

本系列包括三本书^[1]，此书是此系列的第一本，侧重于介绍与代码优化和并行计算相关的理论、算法设计及实践经验。

本书不但包括单核、多核代码的性能优化与并行化，还包括新出现的基于图形处理器（GPU）和移动处理器的代码性能优化及并行化。不但有实际的并行方式的介绍说明，还有理论的分析。笔者希望通过这种方式能够让阅读本文的软件开发人员掌握并行编程方法。

整体而言，本书分为如下几个部分：

- **理论基础**，本部分主要介绍并行软件和硬件基础，并行算法设计思想以及一些软件优化方法。主要包括第1章、第2章、第3章、第5章。
- **代码优化**，本部分主要介绍常见的串行代码优化手段（不包括向量化）。主要内容是第4章。
- **并行算法设计考量**，本部分主要介绍如何设计优良的并行算法并将算法映射到硬件上。主要内容是第6章、第7章、第8章、第9章、第11章和第12章。
- **如何将现有的串行代码并行化**，主要内容是第10章。

第1章 主要介绍并行化和向量化的相关概念，如并行和向量化的作用、为什么并行化和向量化、并行

或向量化面临的现实困难。另外还介绍了一些不写代码也能够利用多核处理器性能的一些方法。

第2章 介绍了现代处理器的特性，如指令级并行、向量化并行、线程级并行、处理器缓存金字塔、虚拟存储器和NUMA（非一致内存访问）。

第3章 介绍了算法性能和程序性能的度量与分析。算法性能分析和度量的主要标准是时间复杂度、空间复杂度和笔者自己提出的实现复杂度。程序性能的度量标准主要有：时间、FLOPS、CPI、指令延迟和吞吐量。用来衡量优化一部分代码对程序整体性能的影响主要有：Amdahl定律和Gustafson定律。本章最后介绍了常见的用于程序性能分析的工具。

第4章 介绍了常见的串行代码优化方法。依据优化所涉及的尺度将优化方法归类并分为：系统级别、应用级别、算法级别、函数级别、循环级别、语句级别和指令级别。

第5章 简单介绍了指令级依赖和循环级依赖，并给出许多如何去除依赖的示例，最后以简单介绍处理器硬件支持的寄存器重命名结束。

第6章 介绍了常见的并行编程模型和目前主流的并行编程环境。

第7章 详细介绍了并行算法设计的基本步骤和主要内容：①划分；②通信；③结果归并；④负载均衡。

第8章 介绍了并行程序相比串行程序具有的一些可能的、天然的缺点，并分析了如何缓解某些缺点的方法。

第9章 介绍了如何使用SIMD向量指令、多核多线程和GPU来实现map、reduce、scan和流水线等并行实践模式。通过这些并行实践模式的介绍、解说，希望读者能够通过模式解决一系列相关的并行计算问题。

第10章 介绍了并行化遗留代码的基本步骤，并指出每个步骤的常用方法和需要注意的事项，最后以如何并行化word2vec做为示例结束。

第11章 介绍了常见的几种超级并行方式，并且以矩阵向量乘为例展示在各种超级并行模式下如何划分数据和计算。最后介绍如何在几种超级并行方式下优化矩阵乘运算。

第12章 给出了设计并行算法需要注意的一些准则，以方便读者随时查阅。

附录A 介绍了整数的运算规则和浮点数据的IEEE-754表示。

本书希望通过这种方式能够让读者渐进地、踏实地拥有并行思维，并且能够写出优良的并行代码。

对对并行和代码优化不太了解的人员，笔者希望你们按章节顺序仔细阅读。而对对并行或代码优化非常了解的人员，可按照需求选择章节阅读。

勘误和支持

由于笔者的水平有限、工作繁忙、编写时间仓促，而并行和代码优化又是一个正在高速发展的、影响因素非常多、博大精深且具有个人特色的领域，许多问题还没有统一的解决方案，虽然笔者已经努力确认每个细节，但书中难免会出现一些不准确的地方甚至是错误，恳请读者批评指正。你可以将书中的错误或写得不好的地方通过邮件发送到ily152832912@gmail.com或微信联系“风辰”，以便再版时修正，另外笔者会尽快回复邮件。如果你有更多的宝贵意见，也欢迎发送邮件，期待能够得到你们的真挚反馈。

致谢

首先要感谢我的老婆，她改变了我的人生轨迹，让我意识到人生有如此多的乐趣。

感谢中国地质大学（武汉）的图书馆，那是我对并行计算产生兴趣的地方。感谢中国科学院研究生院和中国科学院图书馆，在那里我奠定了从事并行计算事业的基础。

感谢我的朋友陈实富、赖俊杰、高洋等，如果没有你们，我还需要更多时间来提升水平。感谢我的老板王鹏、吴韧和汤晓欧，在这些技术大佬和“人生赢家”的指导下，我才会成长得如此迅速。

感谢机械工业出版社华章公司的高婧雅和杨福川，是你们引导我将这本书付梓成书，是你们帮我修改书稿，让它变得可读，是你们的鼓励、帮助以及引导使我顺利完成全部书稿。

最后感谢我的爸爸、妈妈、姥姥、姥爷、奶奶、爷爷，感谢你们将我培养成人，并时时刻刻为我提供精神力量！

谨以此书献给我最爱的家人，以及众多热爱代码优化、并行计算的朋友们！愿你们快乐地阅读本书！

风辰

[1] 除本书之外，另两本书分别是《并行编程方法与优化实践》和《科学计算与企业级应用的并行优化》，均由机械工业出版社华章公司（www.hzbook.com）出版。——编者注

第1章 绪论

在2003年以前，计算机性能的提升主要依赖CPU主频的提升，科研人员只要写好程序，几乎用不着优化，因为下一代CPU主频的提升会轻易地提升软件的性能，这使得计算机行业进入一个良性循环：由于性能的提升，人们能够使用计算机做更多的事，当人们习惯当前计算机的速度后，又会提出新的性能要求，让计算机以更快的速度做更多的事；CPU生产商也乐于升级硬件以赚取更多的利润。计算机行业在这种良性互动下发展了几十年，但是由于CPU的功耗与频率的三次方近似成正比，无限制地提升频率已不可能。到2003年，CPU频率的提升接近停止，为了能够卖出自己的产品，各CPU生产商纷纷通过各种方式提升计算能力，如提高指令级并行能力、在一个时钟周期内执行更多指令、向量指令、多核和超线程技术等。从长远来看，最有可能引领未来的是向量化和多核技术：向量化是指使用同一条指令同时操作多个数据；多核技术是采用在同一个芯片上集成多个核心的办法。而高端的服务器版本则会集成多个多核处理器，这称为“多路”。相比CPU从单核到多核、多路，从标量到向量的发展，图形处理单元（GPU）一出世即通过将几百、几千核心集成在一块硅片上以满足图形图像及视频对性能的需求，这称为“众核”。一方面：众核处理器集成的核心数量远远超过多核；另一方面：众核处理器将更高比例的晶体管用于计算，因此其原生性能也超过多核。

作为本书的绪论，本章会从并行和性能优化的作用及为什么需要并行开始，既而介绍并行和代码性能优化面临的现实困难；然后针对某些特殊情况，介绍一些不修改代码也能利用多核处理器性能的几种替代方法；再介绍与多核技术息息相关的进程、线程等相关概念；最后介绍目前主流的向量和多核计算平台；最后本章以笔者对多核和向量化的点评结束。

1.1 并行和向量化的作用

并行和向量化的首要作用是尽量发挥硬件提供的全部计算能力，以减少延迟（更快地完成计算任务）或提高吞吐量（在相同的时间内完成更多任务）。目前绝大多数软件都是标量且串行的，虽然目前CPU使用乱序执行、指令流水线等指令级并行技术提升了处理器效率，这些技术的使用使得从指令级来看程序的执行并不完全和串行的代码系列一致，但是软件依旧只有一个标量控制流。以Intel Haswell I3-4130处理器为例，它使用了256位的向量，集成了2个核心，那么在其上运行的、使用32位浮点计算的标量串行代码最多能够发挥峰值性能的1/16，其中标量计算意味着只能发挥向量运算性能的1/8，而串行代码只能发挥2核中1核的性能。标量串行软件只能利用多核向量处理器提供的部分计算能力，为了利用多核向量处理器提供的全部计算能力，必须采用向量化和并行化的思维方式编写软件代码，而现有的串行软件必须修改才能利用多核向量处理器的全部计算能力。

并行/并发的另一个作用是实现功能和同时满足多个用户请求：比如需要软件在计算的同时能够响应用户的交互，此时就必须使用并行/并发，因为存在两个或多个控制流；假设一个用户请求需要100ms完成，那么如果使用单线程服务的话，1秒钟可以满足10个用户请求，但是如果使用20个线程的话，每个用户请求的满足时间可能只需要150ms，甚至更少。

在科学计算中，物理模拟需要长时间的运行以获得更精确的结果，因此并行和向量化技术应用比较广泛。在这个领域，并行和向量化主要提供以下两方面的作用：

- **让程序运算得更快，以节约时间**。假设程序原来需要一年才能模拟一次，通过使用并行和向量技术，现在一个月就能够完成一次模拟，缩短的11个月时间就可以多做实验、提前发论文、提前毕业或陪女朋友逛街等。

- **让程序能够计算更大规模的体系**。大规模体系要求更大的计算能力，因此对并行和向量化的需求更为迫切，而且大规模意味着可以更真实地模拟现实系统。假设一个二维域分解问题的计算量和区域的大小成正比，假设使用并行和向量化技术获得的计算能力是原来的36倍，那么可以将模拟区域两个方向都扩大为原来的6倍。

1.2 为什么要并行或向量化

从2003年开始，CPU频率的提升接近停止，那种每次硬件的更新都会提升标量串行软件的性能的“免费午餐”已经结束，为了提升软件的性能，软件开发人员不得不使用并行或向量化技术。除了现实世界中的硬件已经完全是并行硬件外，还有几个原因要求我们必须使用并行或/和向量化：现在的编译器不能很好地自动向量化或并行串行软件（一些编译器能够自动向量化和并行简单程序）；现实世界中人类对计算能力的需求永无止境。

有些编译器公司和组织（如Intel、GNU、PGI、CAPS）想让编译器自动向量化或并行串程序，这一直是人们奋斗的目标。20世纪80年代中期，基于依赖分析的自动向量化工具已经基本可用，可以帮助程序员将Fortran语言代码移植到向量计算机上进行并行计算。现代的GCC能够向量化和并行一些非常简单的代码，但是对于复杂一些的程序基本上无能为力。实际上，即使是简单的代码，自动向量化或并行的性能也很难做到完美（通常性能比手工编写的要差很多）。其他的一些相关的研究，包括共享存储的MIMD（Multi instruction Multi Data，多指令多数据）和分布式存储结构的自动并行化，到目前为止，这些方法都少有进展。实际上，自动向量化或并行的主要难点在于编译器没有办法收集/分析向量化或并行所需的数据相关性和控制相关性等信息，必须需要程序开发人员干预。现在，研究的重点转向基于编程语言的策略研究，即从开发人员那里获得更多有关逻辑控制和数据相关性的描述，同时利用自动向量化或并行技术来减轻程序设计的负担。OpenMP、OpenACC、CUDA和OpenCL就是其中的典型代表。

由于标量单核的性能已经不能大幅度提升，以及向量多核/众核技术的普及，只有向量化或并行才能充分利用多核/众核技术带来的性能提升，而研发人员总能发掘出要求更高计算能力的应用（希望程序能够运行得更快或者能够计算更大规模的问题），这些应用对计算能力的需求推动着硬件和软件技术向前发展。一旦软硬件技术的发展暂时满足了应用对计算能力的需求，科研人员又会提出更高的要求。这种良性互动推动着科学发展。只是其中的一方由硬件厂商变成了硬件厂商和软件开发人员的结合，而硬件厂商通常只管卖出产品，发挥多核向量处理器性能的实现责任落在了开发人员肩上，而开发人员没有选择。

1.3 为什么向量化或并行难

向量化或并行编程方式和目前通行的标量串行软件开发方式并不一样，它要求开发人员显式地编码以处理多核向量代码中向量内的多个元素、多个控制流之间的依赖关系，这使得向量化或并行软件的设计和开发难度远超标量串行软件，主要原因有人为的，也有技术方面的。本节将会介绍这些原因和可能的解决办法。

由于多核与并行技术的流行只是近十年的事情，虽然向量化技术已经使用了多年，但是过去软件开发人员没有动力去采用它们，因此目前的大多数软件开发人员没有足够的经验来应对向量化或并行的挑战。而初学者也没有很好的资料及成熟的项目代码学习，另外向量化或并行编译器的低能以及向量化和并行调试工具的匮乏也增加了并行化与向量化编程的难度，这种现象和计算机编程早期一样。在计算机编程的早期，只有科学家才能编程，一方面那个时代只有科学家才能接触到计算机，另一方面那个时代还没有高级程序设计语言，必须要使用机器语言编程，而今天几乎人人都能编程。随着未来向量化和并行化技术的流行，最终向量化或并行编程将会越来越简单，成为软件开发人员的必备技能。

综合分析，笔者以为向量化或并行化难的主要技术原因有以下几个。

- 没有很好的设计方法学：向量化和并行的本质在某种意义上和现行的面向对象程序设计方法学冲突。
- 遗留代码：过去几十年积累下来的代码是企业的巨大财富，没有人会放弃。但是向量化或并行它们将面临现实的挑战。
- 可扩展性：如果代码能够发挥双核的计算能力，那么4核、8核、16核呢？是否能够线性扩展？如果处理器拥有上百核心呢？何况并行程序在上百核心的处理器上会发生什么事情也是个未知数。
- 可维护性：向量化或并行代码的可读性通常不如标量串行代码，如何在原来的开发人员离开后，接管的开发人员也能够维护就变得非常重要。
- 任务/数据划分：并行意味着多个控制流同时执行，而向量化意味着同时操作多个数据，并行需要在各个控制流之间划分任务和数据并去除依赖，向量化则需要处理向量内要处理的数据的依赖关系。数据/任务的划分方式不但决定了编程时的难易程度，而且划分带来的负载均衡和通信问题往往也会对程序的最终性能产生决定性的影响。
- 并发访问控制：多个控制流需要访问不同的或相同的资源，如何协调对这些资源的访问就变得非常

重要，这也成为并行编程的一大难点。

- 资源划分：资源划分方法不但关系到编程的难易，还关系到最终的性能。
- 与硬件交互：为了最好地发挥性能，软件开发人员通常会应用硬件的特性。
- 对软件开发人员的过高要求，开发工具不够智能，且市场不愿付出相应的薪水。

下面笔者将详细解释各个方面。

1.并行程序设计方法学

面向对象设计方法主导了今天大型软件项目的开发，面向对象设计方法指导设计人员通过分离问题中涉及的对象将大问题分解成小问题，然后通过对对象编码以解决小问题，通过利用对象之间的通信来解决的小问题，进而解决大问题。面向对象设计方法学完全没有考虑向量化或并行必须考虑去除的数据和控制的依赖关系，而对象之间的通信本质上来讲是一种依赖关系，因此在某种程度上来说，面向对象设计方法学在本质理念上和向量化或并行冲突。

面向对象方法鼓励隐藏对象拥有的数据，而向量化或并行化需要分析作用在数据上的操作的依赖关系。分析一个对象内拥有的原生数据的依赖关系可能会比较容易，但是如果对象内调用了其他对象，那么可能还需要分析被调用对象的依赖关系，直到所有对象之间的依赖都已经弄清楚。

如果使用一张有向图来表示对象间的引用关系的话，分析对象间依赖的难易程度和有向图中非零元的数量近似成正比。读者可以想象一下：如果代码有成百个对象，且对象之间都有联系，那么分析它们之间的依赖关系将会相当复杂。

对于向量化或并行来说，最合适的设计方法应当是[过程化的设计方法加上以数据为中心](#)。面向对象方法习惯将数据隐藏在对象内部，而向量化或并行本质上是以数据为中心的，显式的数据传递对于向量化或并行来说更为适用。通过过程化的设计方法来设计程序流程，以数据为中心来向量化或并行已经是主流的设计方法。

在实践中，开发人员可以通过分析指导的方式来分析面向对象软件中的最耗时代码（也称为热点）。如果程序的热点很集中，那么只需要采用过程化的方法加上数据为中心的方式并行化热点即可；如果程序的热点很分散，那么可能需要重新设计软件。

2.遗留代码

一些大的软件项目拥有成千上万，甚至百万行代码，而通常大项目对性能的要求又更为急迫。如何向量化或并行这些代码却比较难，因为向量化或并行的难度和代码的长度呈线性关系，而且当前维护这些代码的人员通常并非原始的开发人员，这使得向量化或并行的代价和风险都很大。

对于遗留代码来说，基于编译制导的编译器（如OpenMP和OpenACC）会是一个比较安全的选择。编译制导方法基于原来的串行代码，加入指导向量化和并行的伪指令语句。在向量化或并行的过程中，整个代码还能够运行，允许软件开发人员逐步地向量化或并行现有代码，便于调试和验证正确性。

3.可扩展性

现在16核的机器已经开始普及，编写的程序在16核上可扩展性可能会比较好，但是如果把程序放到32核、64核上会发生什么事情，或许此时需要重新改写代码，甚至需要更改向量化或并行方法。

Amdal定律告诉我们：在计算规模一定的前提条件下，只要代码有不能并行的部分，程序是不太可能完全线性加速的。在处理器核心增多的条件下，并行性不好的部分代码可能会成为限制可扩展性的瓶颈。最终程序或者硬件会达到一个核心数量的极限，在这个极限上，再增加核心数量就不会再提升性能了。

可扩展性的问题基本上没有解决办法，因为开发人员不能完全正确地预测在目前还不存在的硬件上发生的事情。通常的缓解方法是要求在开发项目时，留下足够的设计文档，使源码有足够的、准确的注释。这样在核心数增多可扩展性出现问题时，开发人员能够尽快定位问题，找到可能的解决办法。

4.可维护性

由于在原有的标量串行逻辑中加入了向量化和多个控制流的调度内容，而人脑能够同时维护的状态数量是有限的，这使得向量化或并行代码比标量串行代码更加难以维护。

一些项目同时维护一个标量串行版本和一个向量并行版本，这带来一个问题：如何让两者保持一致。

5.任务/数据划分

由于并行需要将多个工作划分成几个小部分，然后每个控制流处理一个或多个部分。任务/数据划分时需要十分小心，划分方式不但影响编程的难易，还影响程序最终的性能。比如，不均匀的划分会导致负载不均衡。（负载均衡用于在控制流之间重新分配任务/数据，以获得更好的性能。）而某些划分方式会导致程序的很多代码顺序执行。另外，划分可能导致某些全局处理变得复杂，此时可能需要同步，以安全地处理这些全局数据。

依据对任务和数据的划分方式的不同，可将并行编程划分为不同的编程模式，本书会在第6章详细分

析。

通常划分后各个控制流之间需要一些通信（易并行可能无须通信）。由于通信会引起开销，不成熟的划分方式可能使得通信的开销过大而导致性能的极端下降。

基于CPU的并行编程中，控制流的数量必须加以控制，因为每个控制流都会占用一些资源，比如缓存、虚拟存储器。如果过多的控制流同时在一个处理器核心上执行，那么每个控制流使用的资源数量就会减少，这可能会引起缓存命中率过低，从而降低性能。另一方面，大量的线程可能会带来大量冗余计算和IO操作。

最后，并行会大量增加程序的状态空间，导致人脑难以理解，降低生产率。这一点通常可以通过采用成熟的软件工程方法予以克服。

6.并发访问控制

并行程序的多个控制流需要协调对某个资源的访问，比如打印机，如果不加以控制的话，并行程序打印出来的可能就是“天书”。

基于消息传递的编程模式允许各控制流拥有自己独立的存储器内容，此时数据的交流通过传递消息实现。需要注意由于资源访问导致的死锁、活锁、饿死等问题。

基于共享存储器的编程模式只有一个存储器空间，这样各个控制流访问同一存储器地址时就有可能产生冲突，常见的有“读后写”、“写后读”和“写后写”等问题。这类问题通常通过互斥（互斥是指某一时刻只允许一个控制流访问）资源的访问解决。

并行编程中，最常见的并发访问控制是文件，如果多个控制流同时读一个文件，那么就有可能读到错误的数据，常见的解决方法有：

- 由一个控制流读取文件，然后分发数据；
- 将文件分成多个子文件，每个控制流读取自己的子文件。

前一种方式编程简单，但是由于分发数据操作完全是串行的，有可能会造成过大的性能开销。而后一种则相反。

关于并发资源的访问，比较明智的做法是将访问分为写和读，由于不同的控制流可以同时读一个数据，因此此时无须访问控制。而多个控制流要写的数据必须要特殊处理。需要提醒读者的是，当对一个数据有些控制流读、有些控制流写时，也必须进行特殊处理。

7.资源划分

如何在不同的控制流之间分配计算资源一直是并行的难题，这往往和负载均衡关联，如果分配给某个控制流的资源多，就可能要让其他的控制流等待它计算完成。在基于X86的处理器上，这往往只涉及内存、共享文件的划分。在基于GPU的并行计算环境上，这个问题往往更加复杂。

资源划分与并发访问控制、通信密切相关，好的资源划分方式能够既减少通信又保证资源访问的局部性，这通常意味着优秀的性能和可扩展性。

资源划分通常依赖于应用。数值计算频繁地将矩阵按行、列或子矩阵进行划分。控制流可能会静态地分割数据，或者每个控制流处理的数据量会随时间改变。资源划分是常见的向量化和并行化方式，事实证明它非常有效，但是随之而来的复杂的数据结构的处理也非常有挑战性。

8.与硬件交互

向量化或并行编程要求软件开发人员对机器的配置比较了解，只有这样才能避开硬件的缺陷，编写出高效的代码。涉及新的硬件特性时，经常需要直接与这些硬件打交道。当需要榨取系统的最后一点性能时，通常需要直接访问硬件。

由于不同硬件的设计方法、发挥硬件性能的编程方式及硬件设计上容易造成性能瓶颈的地方都不相同，这些因素可能会导致在某一硬件上性能很好的算法，在另一硬件上性能却非常差。

基于多机系统编程时，网络的拓扑结构和网线的传输速度非常重要；基于多核编程时，核心和缓存之间的组织比较重要（这个方面经常出现的是伪共享问题）；基于GPU编程时，GPU硬件的组织更为重要，如核心之间缓存的组织、DRAM的组织、核心的组织以及程序如何映射到硬件上执行。在这些情况下，软件开发人员需要根据目标硬件，协调程序各个方面的设计。

9.对软件开发人员的要求

目前编译器及开发环境对向量化或并行的支持能力比较差，主要包括以下三个方面：

- 只能自动向量化一些简单的代码，即使能自动向量化代码，通常也不是最优的；
- 只能自动并行化简单代码，编译器在自动并行化方面做得通常比自动向量化还要差；
- 不能找出并行冲突的地方；
- 不能协调资源访问。

Intel的并行工具系列能够发掘出程序简单的并行性并识别读写冲突，另外其具有简单的自动并行化能力（能够自己决定是否使用SSE指令及OpenMP），但是对于优秀的开发人员来说，这远远不够。

由于编译器缺乏相应的功能，软件开发人员不得不自己来做。软件开发人员需要自己发掘应用的并行性，并且处理共享资源的访问冲突。另外由于不同的并行化方法可能利用了硬件/软件不同的特性，因此其性能更难以把握。

目前的调试器对向量化或并行的支持非常差且不可靠，软件开发人员缺乏工具导致生产率上不去，这就导致了雇主不愿使用并行开发。

由于硬件生产商极力地、不负责任地吹嘘向量化或并行编程是如此简单（实际上并行化和并行化代码这种责任也是硬件生产商转嫁给软件开发人员的），使得很多雇主认为只要付给并行软件开发人员和串行软件开发人员一样的工资就够了，而且一般而言并行软件的开发周期比串行软件开发要长得多，这也导致了软件开发人员不愿意使用向量化或并行技术。个人认为市场应当给经验丰富、能力强的并行软件开发人员2倍以上的工资（与同等能力的标量串行软件开发人员比），否则开发优秀的并行软件通常是一句空话。

1.4 并行的替代方法

由于向量化和并行编程的难度和软件开发周期长，很多人不愿意使用它们，但这并不意味着他们不能享有向量多核并行处理器的好处，另外一些方法也能够像并行一样发挥向量多核处理器性能。下面给出了几种简单方法：

- 运行同一程序的多个实例；
- 利用已有的并行库；
- 优化串程序序。

1.运行同一程序的多个实例

如果需要计算同一条件作用在不同的数据集上的效果，或者要计算同一数据集在不同的配置条件下的运行结果。软件开发人员可以在多核处理器上同时为每个数据集运行一次串行程序，这样多个进程可同时占用多个计算核心上的资源。或者为每一种配置条件运行串行程序的一个实例，这样运行每个配置条件的一个实例可占用一个计算核心上的资源，多个不同的实例就可以同时占用多个核心。这些方法并没有减少一次运行的时间（由于资源共享，甚至有可能增加一次运行的时间），但是系统整体的吞吐量会得到提升。

在进行基于深度学习的卷积神经网络实验优化时，经常需要在不同的配置条件下（假设有三个配置，记为A、B、C）学习参数模型，以从多个可选的模型中获得最优的模型。可以分别配置A、B、C运行三个优化实例a、b、c，那么a、b、c可分别在一个核心上运行，这样系统上便有三个核心在运行计算任务，相比只有一个核心在运算，这会提高吞吐量。

如果同时在一个单核机器上运行多个程序，或者在多核处理器上同时运行远超过核心数量的控制流，这可能会既增加单个程序的运行时间，又减少多核处理器整体的吞吐量。

2.利用已有的并行库

目前已经有许多函数库实现了并行，如Intel的MKL、IPP、TBB，以及NVIDIA开发的基于其CUDA计算环境的CUDNN、NPP、CUFFT、CUBLAS等，这些库简化了向量化或并行的设计，使用这些库能够方便地利用多核向量处理器的性能。

NVIDIA的nvblas库使用NVIDIA GPU加速计算密集的blas三级函数。对于原先使用CPU blas三级库函数

的应用来说，只需要在编译时链接nvblas库即可利用NVIDIA GPU加速。对于在Linux上运行的调用了blas的应用来说，还可以不用重新编译，只需要在运行程序前使用LD_PRELOAD环境变量让应用使用nvblas中的GPU实现代码即可实现加速。

使用已有的并行库通常比自行编写并行代码要便利，但是要避免由于使用不当，导致性能反而不如标量串行代码的情形。

3.优化串行程序

优化标量串行程序通常应当在向量化或并行之前进行，结果通常会更吸引人。

优化标量串行代码获得的性能提升与向量的长度和核心的数量没有直接的关系，其后还可以利用向量化和并行进一步提升性能。

相比向量化和并行，优化串行代码在操作上通常更容易一些，代码的可扩展性和可维护性通常也更好。

算法的改进获得的性能提升可能是指数级，而向量化或并行带来的性能提升通常和向量长度及核心的数量成正比。

1.5 进程、线程与处理器

现代处理器和进程、线程两个概念紧密关联。进程的概念简化了程序设计、存储器管理等，并且提供了一种大粒度并行的方法。线程存在进程之中，进程中的所有线程共享进程的资源并独享某些资源，因此更易于通信。本节介绍与此相关的进程、线程、超线程和处理器等概念及关系。

在实践中，进程可调度到一台机器中的一个或多个处理器核心上执行，而线程会调度到一个核心上执行，向量化的代码则会映射到一个核心内的向量单元上执行。由于操作系统调度策略的不同，并不能保证进程和线程会一直在相同的核心上执行。

通常基于进程的是像MPI一样的分布式存储器编程模式，基于线程的是像pthread、OpenMP等这样基于共享存储器的编程模式。由于分布式计算的各节点有其独立的存储器，因此基于进程的消息传递通信更合适，而多核等由于共享存储器，所以基于线程的共享存储器更易于通信。

1.进程

当程序在系统上运行时，操作系统会提供一种假象，就好像系统中只有这个程序在运行，只有该程序在使用核心、DRAM和设备。如果真的只有这个程序一直在使用核心的话，在单核心的系统上，用户就必须得等待当前正在执行的程序运行完才能输入下一条指令。现代操作系统并非这样，这是通过进程的概念实现的。

进程是对操作系统正在运行的程序的一种抽象。多个进程可以并发运行在一个核心上，通过时间片轮转，就好像进程一直在使用核心。系统内的多个进程通过时间片轮转并发执行，这就要求有一种机制能够在时间片到期时保存正在运行的进程的状态，并运行另一个等待运行的进程。这种保存一个进程的状态切换到另一个进程的过程称为“上下文切换”。

上下文是指保持进程运行所需要的寄存器、缓存和DRAM等资源。在任何一个无限精度的时刻，一个处理器核心最多只能运行一个进程或一个线程，当操作系统需要在某个核心上运行另一个进程时，就会进行上下文切换。

通过上下文切换进程获得了并发的特性，而运行在多个核心上的多个进程又获得了并行的特性，由于进程的上下文切换和通信比较耗时，因此基于进程的并发往往只适合于大粒度的任务并行。

进程和程序有关系也有不同，程序是静态指令的集合，进程是程序正在运行的状态。进程是资源拥有的独立单位，不同的进程拥有不同的虚拟地址空间，不能够直接访问其他进程的上下文资源。

2.线程

进程之中可以有許多线程，这些线程共享进程的上下文，如虚拟地址空间和文件，但是独立执行且可通过存储器进行通信。当进程终止时，进程内的所有线程也会同时终止。另外线程也有其私有逻辑寄存器、栈和指令指针PC。

由于线程共享进程资源，因此线程的建立、销毁比进程的建立、销毁更高效，在多核处理器上逐渐比进程更引人关注。

由于进程的存储器资源是独立的，而线程的存储器资源是共享的，因此通常基于进程的并行编程更简单，但是基于线程的并行在多核处理器上通常更高效。在多机系统中，不同的计算机天然适合多进程。因此在多核处理器上应优先选择线程级并行，而多机系统应选择进程级并行。实际上，许多现代系统及大规模程序充分利用这两种优势：[在节点间使用进程级并行，在节点内的多核上使用线程级并行，这称为混合或超级并行。](#)

目前基于GPU的并行编程也使用基于线程的开发环境，这是一种“硬件线程”，其线程的创建、调度和销毁开销接近0。

多线程程序在多核和单核上执行时具有明显的差别。由于在单核上多线程通过分时共享执行，这使得一些长延迟的操作如锁、IO访问不会导致核心空闲。事实上，网络服务器就是通过多线程技术来提升系统的吞吐量的。

3.超线程

Intel的一些高端机器支持称为“超线程”（Hyper-Threading，HT）的技术，超线程通过双倍增加一些资源（PC和寄存器）来减少线程的切换代价，但是只有一份执行单元，因此其峰值计算能力并没有提高。对于那些指令类型丰富且多的应用，超线程能够很好地提升性能。但是超线程不是万能的，在某些应用上性能可能会下降，而在绝大多数应用上提升不会超过20%。

超线程技术将一个物理处理器核模拟成两个逻辑核，可并行执行两个线程，能够在单个时钟周期内在两个线程间切换，让单核都能使用线程级并行计算，减少了CPU的闲置时间，提高CPU的运行效率。采用超线程，应用程序可在同一时间里使用芯片的不同功能单元。单线程核心在任一时刻只能对一条指令进行操作，而超线程技术可以让一个核心同时进行两个线程处理。

Intel表示，超线程技术让（P4）处理器在只增加5%芯片面积的情况下，就可以换来15%至30%的效能提升（实际上，对某些程序或非多线程程序而言，超线程反而会降低性能）。超线程技术需要主板芯片组

和操作系统的配合，才能充分发挥效能。

4.阻塞和同步

在并行编程中，进程或线程的阻塞和非阻塞、同步和异步是非常常见的名词。准确地说，这4个名词有非常明显的区别，但是在某些文献中，阻塞与同步、非阻塞与异步的含义是一样的。

具体来说，阻塞是相对于进程或线程本身而言，如果一个操作并不阻碍进程或线程，接着执行代码，称这个操作为“非阻塞”，反之则为“阻塞”。相对非阻塞来说，阻塞更为常见，因为非阻塞要求开发人员手动保证操作的完成，这可能会带来数据一致性问题。

同步或异步则是针对通信的多个进程或线程，如果一个进程或线程与其他进程与线程通信时，不需要其他线程做好准备，称之为“异步”，反之则为“同步”。

阻塞和同步的具体含义可能会依据不同的编程环境、语言等有微小的不同。比如对于某些MPI异步数据传输函数实现来说，数据传输时传输的缓冲区和结果在异步操作返回时是否立即可用。

1.6 并行硬件平台

不同的并行计算平台适合不同的并行编程模式，对于某个具体的并行应用来说，如果选对了并行硬件平台，实现后的性能通常会比较好，编程也会简单。下面列出一些常用的并行硬件平台，并说明其适合的编程方式及特点。

1. 机群

通过使用网线依据某种拓扑方式将多台微机互连以获取更大的计算能力，这种系统通常称为机群，而每台微机称为节点。目前几乎所有的超级计算机都是机群系统。

通常机群通过TCP/IP协议通信，使用物理网络互连，为了提高通信效率，一些超级计算机也会使用特有的协议和网络硬件。

目前常用于机群通信的网线主要有万兆以太网和InfiniBand网卡，其中万兆以太网的最高速度为1.25GB/s，而InfiniBand可达7GB/s，映射到处理32位浮点数据时的速度，这大约是现在的多核向量处理器速度的千分之一，极易在计算时成为瓶颈。

将多台机器联结在一起的方式称为网络互连，目前流行的互连方式有星形、环形、树形、网格和超立方。由于不同的互连方式可能导致程序运行时信息的路由路径不同，其对数据传输的延迟影响非常大，如网格就适合具有二维局部性的数据传输应用。

由于节点具有独立的处理器和存储器，在机群上编程需要使用显式或隐式的机制指定数据何时需要在不同的节点间传输和接收。在程序运行时，程序需要通过网络互连在各个节点间交换数据，那么数据的传输路径就会影响传输延迟和带宽，因此显式的消息传递编程接口MPI成为这类平台上的标准和首选。

使用MPI在机群系统上编程时，需要提前处理好输入数据和输出数据在节点间的分布，以利用各个节点能提供的带宽。

2. X86多核向量处理器

多核向量处理器指将两个或更多独立单核向量处理器核封装在一个集成电路（IC）芯片中。多核向量处理器既可以执行向量运算，又可以执行线程级并行处理。由于生产商大量生产这种集成多个向量核心的芯片，因此硬件随处可得，近年来越来越受到开发人员的重视。

（1）X86多核

相比超线程，多核是真正的线程级并行设备。多核与超线程技术相结合，可能会进一步提高系统的吞吐量。

多核的每个核心里面具有独立的一级缓存，共享的或独立的二级缓存，有些机器还有独立或共享的三级/四级缓存，所有核心共享内存DRAM。通常第一级缓存是多核处理器的一个核心独享的，而最后一级缓存（Last Level Cache, LLC）是多核处理器的所有核心共享的，大多数多核处理器的中间各层也是独享的。如Intel Core i7处理器具有4~8个核，一些版本支持超线程，其中每个核心具有独立的一级数据缓存和指令缓存、统一的二级缓存，并且所有的核心共享统一的三级缓存。

由于共享LLC，因此多线程或多进程程序在多核处理器上运行时，平均每个进程或线程占用的LLC缓存比使用单线程时要小，这使得某些LLC或内存限制的应用的可扩展性看起来没那么好。

由于多核处理器的每个核心都有独立的一级缓存，有时还有独立的二级缓存，使用多线程 / 多进程程序时可利用这些每个核心独享的缓存，这是超线性加速（指在多核处理器上获得的性能收益超过核数）的原因之一。

图1-1展示了某个AMD多核处理器的缓存组织结构。

硬件生产商还将多个多核芯片封装在一起，称之为多路，多路之间以一种介于共享和独享之间的方式访问内存。由于多路之间缺乏缓存，因此其通信代价通常不比DRAM低。一些多核也将内存控制器封装进多核之中，直接和内存相连，以提供更高的访存带宽。

多路上还有两个和内存访问相关的概念：UMA（均匀内存访问）和NUMA（非均匀内存访问）。UMA是指多个核心访问内存中的任何一个位置的延迟是一样的，NUMA和UMA相对，核心访问离其近（指访问时要经过的中间节点数量少）的内存其延迟要小。如果程序的局部性很好，应当开启硬件的NUMA支持。

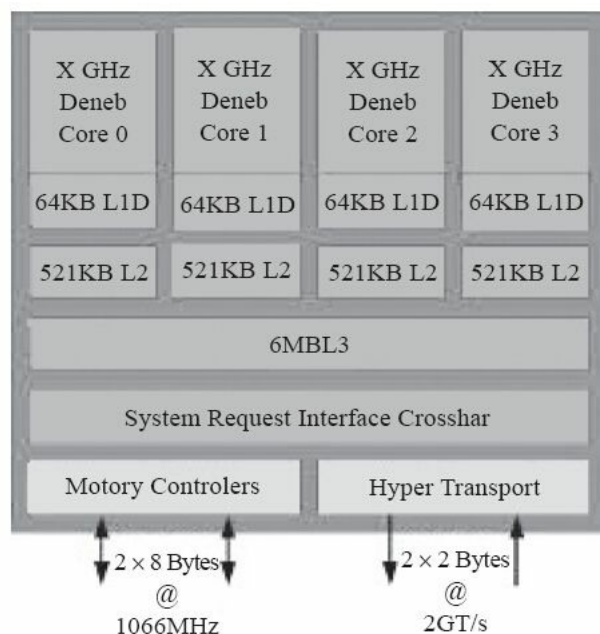


图1-1 多核结构示意图（图片来自Internet）

发挥多核处理器多个核心性能的编程方式通常是使用OpenMP和pthread等线程级并行工具，容易产生的性能问题主要是伪共享和负载均衡。

（2）X86向量指令

SSE是X86多核向量处理器支持的向量指令，具有16个长度为128位（16个字节）的向量寄存器，处理器能够同时操作向量寄存器中的16个字节，因此具有更高的带宽和计算性能。AVX将SSE的向量长度延长为256位（32字节），并支持浮点乘加。在不久的将来，Intel会将向量长度增加到512位。由于采用显式的SIMD编程模型，SSE/AVX的使用比较困难，范围比较有限，使用其编程实在是一件痛苦的事情。

MIC是Intel的众核架构，它拥有大约60个X86核，每个核心包括向量单元和标量单元。向量单元包括32个长度为512位（64字节）的向量寄存器，支持16个32位或8个64位数的同时运算。目前MIC的核为按序的，因此其性能优化方法和基于乱序执行的X86处理器核心有很大不同。

为了减小使用SIMD指令的复杂度，Intel寄希望于编译器，实际上Intel的编译器向量化能力非常不错，但是通常手工编写的向量代码性能会更好。在MIC上编程时，软件开发人员的工作由显式使用向量指令转化为改写C代码和增加编译制导语句以让编译器产生更好的向量指令。

要发挥X86向量处理器的向量计算能力，可以使用三种编程方式：

1）使用串行C语言，让编译器产生向量指令，或使用编译制导语句（如OpenMP 4.0）。这种方式最为

简单，代码的可移植性通常也最好，但是给予软件开发人员的控制力最差，通常能够发挥的性能也最差。

2) 使用Intel规定的内置函数。使用这种方式需要软件开发人员显式地、使用C函数来指定如何向量化操作，使用向量指令来加载数据。使用内置函数时，需要注意哪些内置函数被处理器支持，哪些不被支持（尤其是在开发机和线上机架构不同的情况下）。

3) 使用汇编语言。当编译器生成了不够优化或不需要的指令时，就需要使用汇编语言来榨取系统的最后一点性能。使用汇编语言编程相当不便，代码也难以调试，故应当作为不得已的选择。

另外，现代64位CPU还利用SSE指令执行标量浮点运算。

3.GPU+CPU

近年来GPU（Graphics Processing Unit，图形处理器）的晶体管集成度和（向量多核并行）处理能力的发展速度都远远快于CPU（Central Processing Unit，中央处理器），CPU与GPU的融合是芯片技术发展的一种大趋势。Intel和AMD都在其CPU中集成GPU，而NVIDIA和ATI（AMD）则增强其GPU的编程能力，使得GPU越来越易于满足通用计算的需求。

GPGPU是一种利用处理图形任务的GPU来完成原本由CPU处理（与图形处理无关的）的通用计算任务。由于现代GPU强大的并行处理能力和可编程流水线，令其可以处理非图形数据。特别在面对单指令流多数据流（SIMD），且数据处理的运算量远大于数据调度和传输的需要时，GPGPU在性能上大大超越了传统的CPU应用程序。

NVIDIA和AMD持续改进GPU的编程能力，尤其是CUDA和OpenCL推出后，基于CPU+GPU的异构并行计算越来越得到大家的重视。

GPU是为了渲染大量像素而设计的，并不关心某个像素的处理时间，而关注单位时间内能够处理的像素数量，因此带宽比延迟更重要。考虑到渲染的大量像素之间通常并不相关，因此GPU将大量的晶体管用于并行计算，故在同样数目的晶体管上，具有比CPU更高的计算能力。

CPU和GPU的硬件架构设计思路有很多不同，因此其编程方法很不相同，很多使用CUDA的开发人员有机会重新回顾学习汇编语言的痛苦经历。GPU的编程能力还不够强，因此必须要对GPU特点有详细了解，知道哪些能做，哪些不能做，才不会出现在项目开发中发觉有一个功能无法实现或实现后性能很差，从而导致项目中止的情况。

由于GPU将更大比例的晶体管用于计算，相对来说用于缓存的比例就比CPU小，因此通常局部性满足CPU要求而不满足GPU要求的应用不适合GPU。由于GPU通过大量线程的并行来隐藏访存延迟，一些数据

局部性非常差的应用反而能够在GPU上获得很好的收益。另外一些计算访存比低的应用在GPU上很难获得非常高的性能收益，但是这并不意味着在GPU实现会比在CPU上实现差。CPU+GPU异构计算需要在GPU和CPU之间传输数据，而这个带宽比内存的访问带宽还要小，因此那种需要在GPU和CPU之间进行大量、频繁数据交互的解决方案可能不适合在GPU上实现。

4.移动设备

目前高端的智能手机、平板使用多个ARM核心和多个GPU核心，运行在移动设备上的应用对计算性能需求越来越大，而由于电池容量和功耗的原因，移动端不可能使用桌面或服务器高性能处理器，因此其对性能优化具有很高需求。

现在移动设备主要使用基于ARM的处理器，目前市场上的高性能ARM处理器主要是32位的A7/A9/A15。ARM A15 MP是一个多核向量处理器，它具有4个核心，每个核心具有64KB一级缓存，4个核心最大可共享2MB的二级缓存。ARM支持的向量指令集称为NEON。NEON具有16个长度为128位的向量寄存器（这些寄存器以q开头，也可表示为32个64位寄存器，以d开头），可同时操作向量寄存器的16个字节，因此使用向量指令可获得更高的性能和带宽。

现在移动设备上的GPU主要是高通的Adreno系列和Imagination的PowerVR系列，这两家厂商的大多数移动GPU已支持使用OpenCL进行并行计算，笔者也使用OpenCL在两家的移动GPU上编写过一些并行代码。

1.7 向量化和多核技术不是万能的

在2003年，Intel曾经预测能够在2010年采用10纳米或更小的制作工艺开发出30GHz的计算机，实现万亿指令级别的性能（即每秒钟处理一万亿条指令）。但是现在Intel和其他的生产商在使用14nm技术生产主频低于5GHz的处理器，虽然Intel和AMD的超频技术使得计算机的瞬间主频远远超过5GHz，但是这不可持续且会降低硬件寿命。

由发热和能量消耗带来的问题，使得硬件生产商采用向量化和多核技术，并且宣称向量化和多核技术延续了摩尔定律。向量化和多核技术部分解决了发热和能耗问题，但是这种解决方案也引来一个棘手的软件问题：如何编程以发挥向量多核处理器的计算能力。

由于应用和硬件中串行运算部分的存在，随着处理器数量的增加和向量宽度的增加，并行程序的效率就会降低一些。对于某个应用，当使用的核心数量达到一定程度时，增加核心反而会减慢应用的速度，硬件上也存在这种问题。可能在制造几百个长向量核心的处理器前，多核向量处理器已经达到了实际限制。

对于软件开发人员来说，一些问题很容易向量化和并行，但是也有一些不行；有些问题适合向量化和并行，但是程序却不容易编写。很难将算法划分为几百、几千个控制流，因为人脑很难维护这些控制流的状态空间。实际上自动向量化和自动并行化是解决这些问题的首选（软件开发人员编写串行代码，编译器或硬件给多个处理器有效地分发指令），但是现在的自动向量化和自动并行化工具仍旧非常弱。

抽象是软件开发中的有效技术。但是，编写并行程序的时候，抽象就不太好。抽象要求隐藏程序处理的数据对象，而并行必须要处理数据的依赖关系，两者之间存在天然的冲突。

1.8 本章小结

本章是全书所涉及的基础知识的简单介绍，介绍了许多与并行和向量化相关的重要内容，包括但不限于本节内容。

并行和性能优化的两大作用：①发挥硬件的计算性能，提升程序的吞吐量、增加计算规模或减少计算时间；②为了满足某些功能性需求。

为什么需要向量化和并行：①依靠硬件厂商提升性能的“免费”时代已经结束；②自动向量化工具已经可用，但是只能向量化简单代码，并且性能往往并不理想；③应用对性能的需求依旧在提高，软件开发人员没有选择。

并行和代码性能优化面临的现实困难：①没有很好的向量化和并行设计方法学；②向量化和并行化遗留代码并不容易；③向量化或并行代码的可扩展性和可维护性差；④任务、数据划分及并发访问控制难；⑤对开发人员的过高要求。

一些不修改代码也能利用多核处理器的性能的几种替代方法：①同时运行程序的多个实例；②利用已有的并行库；③优化串行代码。

目前主流的向量和多核计算平台：X86 CPU、GPU和ARM。

处理器是程序的运行平台，不同的现代处理器具有许多相同和不同的特性，在编写代码时如果能够很好地利用这些特性，那么就可以很好地发挥硬件的性能，下一章笔者介绍现代处理器具有的一些典型特性。

第2章 现代处理器特性

现代处理器是向量化或并程序的运行平台，程序的最终性能由运行它的处理器实现，只有了解了目标处理器的特性，才能写出高效的代码。

从系统启动到终止，处理器一条接着一条地执行内存中的指令，就好像是前一条指令执行完之后下一条指令才开始执行。实际上，现代处理器利用了指令级并行技术，同一时刻存在着多条指令同时执行，并且处理器执行指令的顺序无须和汇编代码给出的指令顺序完全一致，编译器和处理器只需要保证最终结果一致，这类处理器称为“乱序执行处理器”。而严格按照顺序一次执行一条指令，只有前一条执行完才开始执行后一条指令的处理器，称为“按序处理器”。目前主流的CPU和GPU，无论是服务器端，还是移动端基本上都已经是乱序执行处理器。

处理器的处理速度远快于内存读写速度，为了减少访问数据时的延迟，现代主流处理器主要采用了两种方式：

- 利用程序的局部性特点，采用了一系列小而快的缓存以保存正在访问和将要被访问的数据，以近似于内存的价格获得近似于缓存的速度；
- 利用并行性，在一个控制流由于高延迟的操作而阻塞时，执行另一个控制流。

简单来说，前一种方法是将经常访问的数据保存在低延迟的缓存中，以减少延迟，主要由目前主流的CPU所采用。而后一种方法则尽量保证运算单元一直在忙碌，以提高硬件的吞吐量，这种方法目前主要由主流的GPU所采用。这两种办法没有天然的壁垒，无论是CPU还是GPU都采用了这两种的方法，区别只是更偏重于使用哪一种。

现代乱序执行多核向量处理器具有许多和代码性能优化相关的特点，本章主要介绍以下部分：

- 指令级并行，主要有流水线、多发射、VLIW、乱序执行、分支预测等技术；
- 矢量化，主要有SIMT和SIMD技术；
- 线程级并行，多核支持的线程级并行是目前处理器性能提升的主要手段；
- 缓存层次结构，包括缓存组织，缓存特点以及NUMA。

上述特点的存在增大了现代处理器的实际处理数据的速度。本章会详细解析这些技术，并分析其如何影响程序的性能。软件开发人员如果了解现代多核向量处理器的这些特性，就能写出性能效率超过一般开

发人员的代码。

2.1 指令级并行

表面上看，处理器是一条又一条串行的执行指令，实际上可以同时多条指令求值，这称为指令级并行。指令级并行要求同时执行的指令之间没有数据或控制依赖。指令级并行相关的技术主要有：指令流水线、乱序执行、多发射、VLIW和分支预测。

通过指令级并行，处理器可以调整程序中指令在该处理器上的执行顺序，能够处理某些在编译阶段无法知道的相关关系（如涉及内存引用时）；在指令集兼容的条件下，能够允许一个流水线机器上编译的指令，在另一个流水线机器上也能有效运行。

指令级并行能够利用处理器上的不同组件同时工作，如果程序具有类型丰富的运算，指令级并行能使处理器性能迅速提高。

2.1.1 指令流水线

处理器有许多不同的功能单元，如果能够利用它们可以同时执行的特点，就可以提高执行速度。现代处理器将指令操作划分为许多不同的阶段，每个阶段由某个单元执行，这样存在多个操作在处理器的多个单元上像多个水流一样向前流动，这称为“流水线执行”，而每个水流就称为一个流水线。流水线

(pipeline) 是一串操作的集合，其中前一个操作的输出是下一个操作的输入。流水线执行允许在同一时钟周期内重叠执行多个指令。图2-1是一个取指令、指令解码、数据加载、操作和写回的经典5阶段流水线示意图。

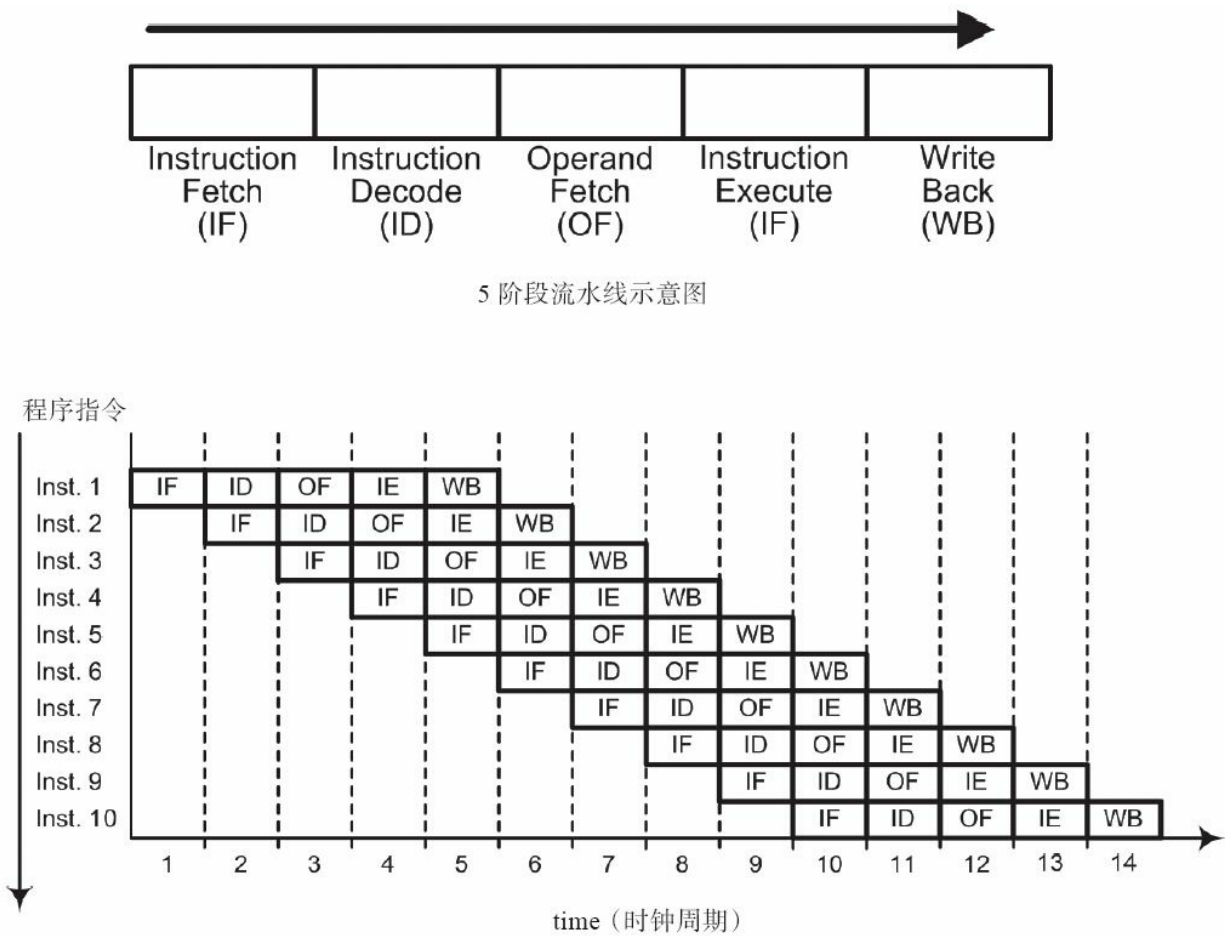


图2-1 流水线示意图

为了充分利用流水线的好处，一些处理器将一些复杂的指令划分为许多更小的指令以增加流水线的长度。流水线系统中存在许多正在执行且还没有执行完的指令，现代处理器能够允许上百条流水线指令同时执行，而每条指令的延迟可能长达几个甚至几十个时钟，而最终的结果是某些指令的吞吐量达到每时钟周

期几个。（如果能够达到每时钟周期超过一条指令的吞吐量，称之为超标量。）通常一条指令的计算和另一条指令的访存同时进行，这样能够更好地利用流水线的好处。

为了更好地利用指令流水线，现代处理器升级通常会增加指令流水线的长度，但是代码中指令级并行是有限的，一旦达到此限制，再增加流水线长度就不会有好处。如ARM A9的指令流水线长度为8，而A15为13。

带有长流水线的处理器想要达到最佳性能，需要程序给出高度可预测的控制流。代码主要在紧凑循环中执行的程序，可以提供恰当的控制流，比如大型矩阵或者在向量中做算术计算的程序。此时在大多数情况下处理器可以正确预测代码循环结束后的分支走向。在这种程序中，流水线可以一直保持在满状态，处理器高速运行。

2.1.2 乱序执行

对于按序处理器来说，一旦一条指令因为需要等待前面指令的结果，那么该指令之后的所有指令都需要等待。为了充分利用指令流水线或多个执行单元的好处，处理器引入了乱序执行的概念，乱序执行是指后一条指令比前一条指令先开始执行，但要求这两条指令没有数据及控制依赖。由于在很多情况下，处理器比较难以判断指令的数据相关性，编译器也引入了类似的功能，主要有指令重排和变量重命名。通常还需要软件开发人员以处理器和编译器友好的方式编写代码，以充分发掘应用具有的并行性，利用处理器的乱序执行功能。

乱序执行需要在执行指令前知道指令之间的依赖关系，如果两条指令之间有依赖，那么这两条指令就不能乱序执行。现代处理器对乱序执行有不同程度的支持，比如大多数的Intel X86桌面处理器和服务处理器上都具有重排缓冲区（ReOrder Buffer，ROB），并且具有远多于逻辑寄存器数量的物理寄存器以支持寄存器重命名。

乱序执行会重排指令的执行顺序，这要求处理器的发射能力大于其执行能力。如果处理器的发射能力和指令的执行能力一致，那么ROB中就不会有指令等待重新排列执行顺序。由于处理器执行不同指令的速度并不相同，因此其发射能力并不一定比执行最快的指令的吞吐量大，比如主流X86 CPU一个周期能够处理4条整数加法指令，但是其指令发射能力也是一个周期4条。

2.1.3 指令多发射

从乱序执行的角度来说，处理器的发射能力最好要大于指令执行能力。而从处理器具有多个执行单元、每个执行单元能够在周期内同时执行多条指令的角度来看，如果指令发射单元每个周期只能发射一条指令，那么必定有些单元空闲。从这个角度来说，只要是具有多个执行单元的处理器，无论是否支持乱序执行，都要求其指令发射能力大于执行能力。

指令发射单元一个周期内会发射多条指令，通常指令发射单元的发射能力会超过单一硬件单元的处理能力，如在NVIDIA Kepler GPU上，SMX一个周期可以发射8条指令，但是SMX本身却最多能消耗6条乘加指令；如在主流的Intel X86 Haswell CPU上，一个周期可发射4条指令，但是只能消耗2条乘加指令。

许多处理器支持一个周期发射2条或多条指令，但是多条指令要满足一条条的条件，比如有些处理器要求没有依赖关系、有些处理器只允许访存指令和计算指令同时发射，而Intel Xeon PHI处理器两个周期可以作为一个线程发射两条指令，但是这两条指令要没有背靠背的依赖。

指令多发射增加了硬件的复杂度，提高了处理器的指令级并行能力。

2.1.4 分支预测

当处理器遇上分支指令（判断指令，如if）的时候，有两种选择：

- 直接执行下一条指令，如果分支是循环的判断条件，则这很有可能造成流水线中断；
- 选择某条分支执行，一旦选择错误，处理器就需要丢弃已经执行的结果，且从正确的分支开始执行。

目前几乎所有的处理器都采用了后者，而选择哪个分支的过程称为“[分支预测](#)”，很多生产商宣称分支预测正确率达到90%以上（关于这一点，聪明的软件开发人员各有各的观点）。

如果程序中带有许多循环，且循环计数都比较小，或者面向对象的程序中带有许多虚函数的对象（每个对象都可以引用不同的虚函数实现），此时处理器很难或者完全不可能预测某个分支的走向。由于此时程序的控制流不可预测，因此处理器常常猜错。处理器需要频繁等待流水线被清空或等待被新指令填充，这将大幅降低处理器的性能。

关于分支预测，不同的处理器展现出完全不同的态度，比如X86就极力地优化其分支预测器的性能，而ARM和主流的GPU则要保守得多。

为了帮助处理器更好地进行分支预测，软件开发人员需要依据某些原则编写分支代码，可参考本书的第4章。

2.1.5 VLIW

乱序执行和分支预测增加了硬件的复杂性。在并行执行任何操作之前，处理器必须确认这些指令间没有相互依赖。乱序执行处理器增加了硬件资源用于调度指令和决定依赖。

而VLIW（Very Long Instruction Word）通过另外一种方法来实现指令级并行。VLIW的并行指令执行基于编译时已经确定好的调度。由于决定同时执行的工作是由编译器来完成的，处理器不再需要调度硬件。结果VLIW处理器相比其他多数的超标量处理器提供了更加强大的处理能力且更少的硬件复杂性。（硬件的复杂性降低了，但编译器的复杂性提高了。）

VLIW对于一些向量操作非常有效，并且能够组合某些不相关的指令以同时执行，但是VLIW对编译器提出了过高的要求，故实际性能通常并不是很理想。早期的AMD GPU大量使用了VLIW，现在已经全面转向使用SIMT。目前主要是移动端GPU和一些过时的AMD GPU/APU在使用VLIW。

2.2 向量化并行

向量化并行是现代处理器提升性能的重要方法，和多核一样，它通常要求软件开发人员显示编写向量化的代码。大多数编译器提供了内置函数来避免直接使用汇编指令编写向量化代码，如X86的SSE/AVX、ARM的NEON，它们都是SIMD（单指令，多数据）模式。而主流的GPU则采用了SIMT（单指令，多线程）。

向量化主要指一条向量指令操作向量寄存器中的多个元素，这是一种数据并行模式。在本书中，向量化并行是数据并行的一个子集。

2.2.1 SIMD

SIMD是指一条指令作用在多个数据上面，目前Intel X86提供了SSE/AVX指令，向量寄存器长度分别为128位、256位。利用SSE/AVX指令可显著提高处理能力，但是由于SSE/AVX指令缺乏灵活性和性能的可扩展性，开发人员需要比较长的时间才能熟练使用它们，因此目前应用范围有限。

目前在X86上有多种办法可以使用SIMD指令，比如汇编语言、内置函数（Intrinsic）和OpenMP 4.0。从编程难度来看，使用汇编语言最难，使用OpenMP 4.0最容易，但是大多数时候使用内置函数也可接近或者达到手工使用汇编优化的性能。

虽然SSE/AVX指令集是由Intel设计的，但是AMD的处理器也支持它们，因此在处理器都支持AVX指令集的条件下，使用SSE/AVX指令编写的程序在Intel和AMD的X86处理器之间可移植。

使用SIMD指令要求开发人员非常熟悉指令的类型、吞吐量和延迟。因为不同的处理器对SIMD指令的支持程度不同，这不但表现在指令类型很不相同，还表现在同一指令在不同的架构处理器上的延迟和吞吐量可能也不相同，或者某些指令存在某些未公开的性能缺陷。

2.2.2 SIMT

SIMT是NVIDIA GPU和AMD GCN GPU采用的向量化方法，SIMT也是数据并行的一个子集，但是去掉了SIMD的一些限制，具有先天的优势。相比SIMD所具有的限制，SIMT具有的优势主要有以下几个方面。

- SIMT指定了逻辑向量宽度，而隐藏了物理向量宽度，软件开发人员依据逻辑向量宽度来编写代码，这提高了代码性能的可移植性。比如AMD GCN，其逻辑向量宽度为256字节，而物理向量宽度为64字节，如果AMD为了性能将其物理向量宽度改为128字节（不改变其他条件），那么为AMD GCN架构编写的代码不用修改，在物理向量长度增加后的架构上其性能也会得到相应的提升。而在支持AVX指令集的硬件上运行使用SSE指令集编写的程序却不能得到相应的性能提升。

- SIMT无须显式地编写向量化代码，其向量化代码逻辑隐藏在了多核代码中。软件开发人员只需要编写一份CUDA或OpenCL代码就可以利用GPU这种多核向量处理器的全部性能，而X86多核向量处理器则需要同时使用向量指令和多线程两种编码方式才能发挥全部性能。虽然Intel也提供了基于其X86多核向量处理器的OpenCL编程环境，但是本质上硬件的向量化方式依旧是SIMD，而且软件开发人员需要自己保存编译器生成的向量指令。

- OpenCL和CUDA比内置函数（Intrinsic）要简单直接，更易于调试、验证和维护。

由于相比SIMD向量化方式具有以上优点，用于图形处理的GPU才能够在Intel X86处理器的围剿下杀出一条血路，并且其应用场景正在不断扩大。

2.3 线程级并行

线程级并行将处理器内部的并行由指令级和向量级上升到线程级，旨在通过线程级的并行来增加指令吞吐量，提高处理器的资源利用率。线程级并行的思想是：

1) 在多核处理器上，使用多个线程使得多个核心同时执行多条流水线；在多核处理器上使用多线程技术，使得多核处理器的多个核心上有多个流水线同时执行，以利用多个核心的计算能力；

2) 在单核处理器上，当某一个线程由于等待数据到达而空闲时，可以立刻导入其他的就绪线程来运行。在单核处理器上使用多线程技术，使核心能够始终处于忙碌的状态。

使用线程级并行使得系统的处理能力提高了，吞吐量也相应提升。除了提高处理能力和吞吐量，线程级并行也经常用来提高用户的使用体验，如果使用单线程，那么网络服务器在一段时间内只能满足一个用户的请求，而多线程则可能满足多个用户的请求，虽然这可能增加了单个请求的延迟，但是提高了整体的吞吐量。使用多线程技术，网络服务器能够同时满足多个用户的请求，也不会使得某个用户因等待过长而抱怨。

2.3.1 内核线程和用户线程

运行在用户空间（运行时）的线程称为用户线程，同理运行在内核空间（操作系统）的线程称为内核线程。用户线程由库管理，无须操作系统支持，因此其创建、调度无须干扰操作系统的运行，故消耗少。内核线程的创建、调度和销毁都由操作系统负责，操作系统了解内核线程的运行情况。操作系统不知用户线程的存在，因此无法将其映射到核心上，当一个用户线程由于资源分配而阻塞时，操作系统无法切换。

由于操作系统内核和物理硬件的限制，一个进程支持的线程数量是有限的，在Linux下可以通过`sysconf`函数查询得到，通常其值不小于64。

很明显的是，用户线程的缺点刚好是内核线程的优点，而内核线程的缺点又正好是用户线程的优点，如果能够合理地将用户线程映射到内核线程上，就有可能综合两者的优点，因此现代库和操作系统将用户线程映射到内核线程。存在4种映射方式：一对多，一对一，多对一，多对多。现在很多多线程库都使用多对多的映射方式，实际在核心上执行的线程数量可能远少于声明或创建的线程数量。

`pthread`是用户级线程库，但是Linux使用了内核级线程实现（一对一映射方式），因此`pthread`线程和GCC的OpenMP线程都是内核线程。

2.3.2 多线程编程库

目前硬件和编译器还无法自己产生高效的线程级并行代码（自动并行化），软件开发人员需要自己丰衣足食。由于线程级并行和传统的串行很不一样，因此很多机构和组织想设计一种全新的语言来满足线程级并行的挑战，但是几乎所有的努力都失败了，而还没有失败的那些正在苦苦挣扎。现在，通行的做法是在串行语言的基础上增加多线程库或通过编译制导语句给予编译器额外的并行信息，然后由编译器生成代码以达到并行目的。以下是目前常用的多线程库和编译制导语句标准。

- pthread，POSIX标准规定了UNIX及Linux实现的线程库接口pthread。pthread目前在所有的Linux下可用，在业界被广泛使用。
- win32 thread，是Windows系统内置的线程API。由于本书不包含Windows的内容，故也不包含win32 thread。
- OpenMP，是一个开放的、基于编译制导语句的API，目前在各大操作系统上均可用，由于其与具体系统平台无关，提供了比较好的可移植性，因此应当优先使用。
- C/C++标准线程，新的C/C++标准都已经引入了线程API，只是在本书编写时，编译器还不支持。
- OpenCL和CUDA，它们是基于GPU（OpenCL也正用于在FPGA上编程）的并行计算的架构、语言和API，由于在某些问题上能够比CPU更快地解决问题，目前正在被广泛使用，使用范围正在逐渐扩大。
- OpenACC是一个基于加速器的编译制导标准，通常用于GPU、FPGA等的并行编程，目前提供编译器的主要厂商有Cray和PGI。

2.3.3 多核上多线程并行要注意的问题

多核处理器上的所有核心共享内存，可以有各自的一、二级缓存和寄存器。由于资源共享，在多核处理器上使用线程级并行需要注意的问题有以下几个。

- 线程过多：如果系统上的线程数量远远超过核心的数量，那么就会导致频繁的上下文切换，进而降低性能，如缓存污染。通常支持超线程的多核处理器能够使用的线程数最多是物理核心数的2倍（即逻辑核心数），再增加就有可能降低程序的性能。

- 数据竞争：当多个线程读写同一共享数据时，便会产生竞争，需要同步，比如两个线程对同一个共享数据执行操作时，就需要同步以保证结果是一致的。一方面，同步通常会导致线程之间相互等待，潜在地降低了性能；另一方面，如果不使用同步程序可能无法并行，这可能提升程序的并行性。

- 死锁：线程发生死锁时，处理器都在操作（一直在询问需要的资源是否可用），但是线程都在相互等待其他线程释放资源，谁也无法前进一步，处于一种“僵死”的状态。

- 饿死：当一个或多个线程永远没有机会调度到处理器上执行，而陷入永远的等待的状态。

- 伪共享：当多个线程读写的数据映射到同一条缓存线上时，如果一个线程更改了数据，那么其他线程对该数据的缓存就要被失效，如果线程频繁地更改数据，硬件就需要不停地更新缓存线，这使性能从独享缓存的水平降低到共享缓存或内存的水平。

2.3.4 多线程程序在多核和单核上运行的不同

多线程程序在单核和多核上运行具有不同的特点。

- 锁：在单核上，多个线程执行锁或者临界区时，实际上只有一个线程在执行临界区代码，而核心也只支持一个线程执行，因此不存在冲突。如果某个线程持有锁，那么只是其他线程不会被调度到CPU上执行，影响的只是持有和释放锁的时间，处理器时刻在运行着。但是在多核上运行时，锁或临界区会导致其余处理器空闲而只允许一个处理器执行持有锁的那个线程，这是一个串行的过程，会影响性能。

- 负载均衡：在单核上不用考虑负载均衡，因为各个线程轮流执行，当一个线程执行完时，便会执行另一个线程，不存在线程等待问题。即使各个线程的任务非常不平衡，也不会影响总执行时间。而在多核上执行时，此时最终时间由运行时间最长的线程决定。

- 任务调度：单核上，任务调度完全是操作系统的工作，无须软件开发人员干预，通常有时间片轮转、优先级算法等。而在多核上运行时，软件开发人员要合理地在核心间分配任务，以尽量同时结束计算（此时任务调度的工作已经从操作系统转移到了软件开发人员）。

- 程序终止：在多线程环境中，何时终止程序就变得复杂，因为程序终止时需要确定各个线程都已经计算完成。幸运的是，多线程库通常都提供了对应的函数。在多机编程上，这个问题可能会变得非常复杂。

2.4 缓存

现代X86 Haswell处理器的吞吐量是一个时钟周期内4次整形加法、2次单精度浮点乘加。从延迟上看，做一次乘法也只要3个周期，一次除法也就十几个周期。而一次内存访问却需要200周期以上，并且随着技术的发展，内存容量变得越来越大，带宽也在增加（但是增加的幅度比处理器的吞吐量要小），但延迟并没有减少而是越来越大。处理器吞吐量与内存吞吐量和延迟的差异越来越大，这称为内存墙。现代处理器通过几种方式来减小这种差距实际产生的影响：

- 每次内存访问、读取周围的多个数据，因为这些数据随后极有可能会被用到；
- 采用容量小但更快的存储器（称为缓存）缓存内存访问，如果访问的数据在缓存中，那么就无须去访问内存；
- 支持向量访问和同时处理多个访问请求，通过大量并行访问来掩盖延迟。

程序在一段时间内访问的数据通常具有局部性，比如对一维数组来说，访问了地址x上的元素，那么以后访问地址x+1、x+2上元素的可能性就比较高；现在访问的数据，在不久之后再次被访问的可能性也比较高。局部性分为“时间局部性”和“空间局部性”，时间局部性是指当前被访问的数据随后有可能访问到；空间局部性是指当前访问地址附近的地址可能随后被访问。现代处理器通过在内存和核心之间增加缓存以利用局部性增强程序性能，这样可以用远低于缓存的价格换取接近缓存的速度。

现代处理器缓存的带宽很高，比如Intel Haswell一级缓存的带宽为每时钟周期每核心64字节，要发挥其峰值必须要使用向量指令。而内存的延迟很高，要隐藏内存的高延迟，则需要发起多个访问请求以让流水线始终在满负荷运行。

软件开发人员应当意识到：对于性能限制在内存/缓存上的程序来说，缓存能够显著增加程序的实际性能，因此要编写缓存友好的代码，同时多核的条件下要注意避免伪共享问题导致的性能损失。

2.4.1 缓存层次结构

为了更好地利用程序访问内存/缓存的局部性，现代处理器采用了多层次的、容量不同和性能不同的缓存，其中上一级缓存容量比下一级缓存小，但是延迟更小、带宽更大。比如Intel Haswell CPU一级缓存大小为32KB，延迟为3个周期，读吞吐量为每周期64字节；其二级缓存大小为256KB，延迟为11个周期，读吞吐量为每周期64字节。

现代处理器至少具有二级缓存，某些高端的多核处理器还具有三级缓存。通常都采用上一级缓存缓存下一级缓存的访问的方式。如寄存器缓存一级缓存访问，一级缓存缓存二级缓存访问，二级缓存缓存三级缓存访问，三级缓存缓存内存访问。当然一些处理器有微小的不同，比如AMD A10 APU中CPU的一级缓存就不缓存来自寄存器的写操作。

不同的缓存层次组织成一个类似于金字塔的结构，上一级缓存容量小、速度快，下一级缓存容量大、速度慢，上一级缓存下一级。如图2-2所示，寄存器缓存level 1缓存，level 1缓存level 2，而level 2缓存RAM，RAM缓存硬盘、网络等，而硬盘缓存远程请求等等。

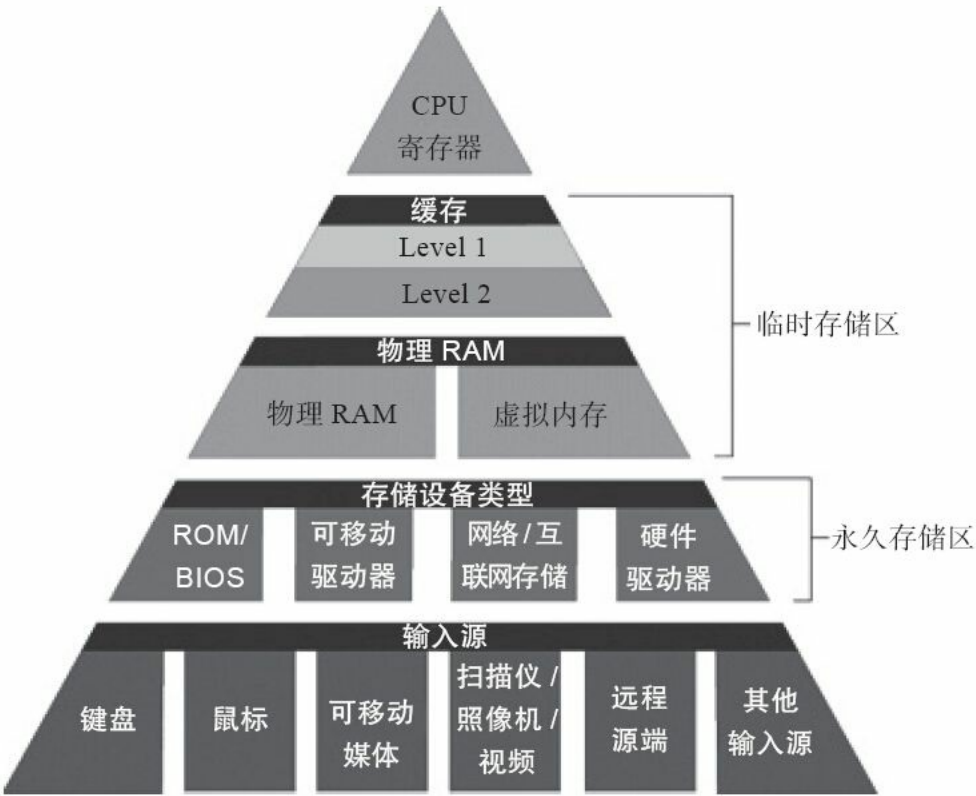


图2-2 缓存金字塔

由于这种金字塔的保存方式，某些数据可能同时被缓存在某个核心的多个缓存层次上（也有可能只缓存在某个层次上），如同时被L1和L2缓存。在多核处理器上，某个数据还可能同时被多个核心上的缓存所缓存。

一次内存访问，如果访问的数据在缓存中，则称为缓存命中，如果不在，则称为缓存不命中，为了衡量缓存命中的概率，提出了缓存命中率的概念，其指程序执行过程中缓存命中的次数占总访存次数的百分比。通常程序的缓存优化过程就是提高缓存命中率的过程。

2.4.2 缓存一致性

由于处理器具有多级缓存，那么如何保证缓存中的数据 and 内存中的数据是一致的，这由处理器的缓存一致性协议来保证。

对于单核处理器来说，其缓存一致性协议要保证，某个地址上读得到的数据一定是最近（一般指处理器视角）写进去的。比如，某条指令更改了地址0x0100上的内容，那么核心对0x0100的缓存需要失效。

多核处理器上还存在一个和单核处理器不同的缓存一致性问题，即多个核心缓存了同一个内存地址的数据，那么一个核心更改了某个地址的数据，其他的核心就需要对该地址数据的缓存失效。一些简单的多核处理器缓存一致性协议使得缓存一致性的代价和处理器的数目成正比，如Intel Xeon PHI。

为了正确性，一旦一个核心更新了内存中的内容，硬件就必须要保证其他的核心能够读到更新后的数据。目前大多数硬件采用的策略或协议是MESI或基于MESI的变种：

- M代表更改（modified），表示缓存中的数据已经更改，在未来的某个时刻将会写入内存；
- E代表排除（exclusive），表示缓存的数据只被当前的核心所缓存；
- S代表共享（shared），表示缓存的数据还被其他核心缓存；
- I代表无效（invalid），表示缓存中的数据已经失效，即其他核心更改了数据。

一旦某个核心更新了内存中的数据，那么硬件便会使其他核心对该数据的缓存失效。实际上更新的缓存数据何时被写回内存，各个处理器的策略也不相同。

多核系统的存储器具有缓存一致性并不代表多个控制流同时读写一个变量不会产生问题，主要是因为：①现在的编译器和编程语言几乎都采用弱一致性协议；②多个控制流执行的先后顺序通常没有办法控制；③缓存一致性并不保证顺序一致性。

对于开发人员来说，缓存一致性是透明的，就如同缓存一样，因此无须过多关注。

缓存失效是一个长延迟的操作，不能完全流水线化，故很多处理器都提供了失效队列来保存缓存失效操作。伪共享本质上也是一种缓存一致性问题。

2.4.3 缓冲不命中

如果处理器无法在某级缓存中得到需要的数据，那么就会向下一级缓存请求数据，这称为缓存不命中。不命中意味着从上一级缓存的性能下降到下一级缓存的性能，因此知道什么情况下会导致缓存不命中就非常重要。通常缓存不命中有以下几种情况。

- **冷不命中**，所谓“冷不命中”是指程序开始执行时，由于缓存里根本没有数据，因此无论请求任何数据，都会不命中。通常程序无须特别关注这种情况，但是在测试缓存性能、大小或结构时，要注意这点。

- **满不命中**，满不命中是指如果缓存已经完全被占用，那么请求的数据如果不在缓存中，就需要将其中的某个已缓存的数据x覆盖为新请求的数据。如果随后又要请求x，那么x不命中，这称为“满不命中”。如果数据存在局部性，但是访问的数据的大小超过了缓存大小就会存在满不命中的情况。

- **冲突不命中**，新读取的数据将会被放入缓存中的某个地址a，a并不是任意的，而是由某些算法决定（由于内存比缓存大得多，因此多个内存地址会被映射到同一个缓存地址）。如果a中原始数据在随后被访问，那么也不会命中，这称为“冲突不命中”。冲突不命中和缓存的组织有关，比如缓存线长度，每组里面有多少缓存线。

实际上，如果只是为了正确性，软件开发人员并不需要关注这几种不命中的情况是否发生。然而为了提高性能，软件开发人员就必须知道硬件缓存不命中的机制和原因。

2.4.4 写缓存

当程序将数据从寄存器写入内存时，这涉及一个是直接写入内存，还是写入到下一层缓存中的选择问题，依据写是否命中来说，各有两种基本策略。

如果要写的数据已经存在缓存中，即写命中时，有如下两种策略。

- 写回（write back），指仅当一个缓存线需要被替换回内存时（缓存已满，或者多线程时，其他线程需要访问这个数据），才将其内容写入内存或下一级缓存。如果缓存命中，则总是不用更新内存。为了减少耗时的内存写操作，缓存线通常还设有一个脏位（dirty bit），用以标识该缓存线在被载入之后是否发生过更新。如果一个缓存线在被替换回内存之前从未被写入过，就不用写回内存。

- 写直达（write through），指每当缓存接收到写数据指令，都直接将数据写回到内存或下一级缓存。如果此数据也在缓存中，则必须同时更新缓存。

无论是写回还是写直达，如果更新的地址被其他核心缓存，那么其他核心对此地址的缓存必须失效。写回的优点是节省了大量的写操作。因为，对一个缓存线内多个不同地址的更新只需一次写操作即可完成，这节省了内存带宽和功耗。写直达比写回更易于实现，并且能更简单地维护数据一致性。

无论是写回还是写直达都需要多个周期，不能完全流水线化，因此现代处理器核心上通常都有一个缓冲区分用于临时保存等待写回内存的数据，这称为写缓冲。只要写缓冲中的数据不在随后被访问或更新，那么写操作就可以完全流水线化。但是某些写后读或写后写会导致串行化问题（读时必须等待前面的写操作完成），如代码清单2-1的前缀和代码：

代码清单2-1 需要检查写缓冲的前缀和代码

```
for(int i = 0; i < n; i++){
    a[i+1] += a[i];
}
```

由于下一次循环需要使用前一次循环写入的数据，因此硬件必须检查写缓冲中的操作是否完成，这导致延迟增加，不能完全流水线化，聪明一些的编译器会将其改写为代码清单2-2的代码：

代码清单2-2 不需要检查写缓冲的前缀和代码

```
tmp = a[0];
for(int i = 0; i < n; i++){
    tmp += a[i];
    a[i] = tmp;
}
```

如果需要被写的数据不在缓存中，即写不命中的话，那么现代处理器会执行两种处理方法中的一种：

- 写分配 (write allocate)。写分配是指，如果要写的数据没有被缓存，那么就在缓存中分配一条缓存线，这类似于大多数处理器对读的处理。如果被写入的数据的局部性很好（在随后会被读），那么写分配就很适合。

- 写不分配 (write no-allocate)。写不分配是指，如果要写的数据没有缓存，那么数据就直接写入内存，而不占用缓存线，这有点类似于越过缓存。如果被写入的数据的局部性很差（在随后不会被再次使用），那么写不分配就很适合。

2.4.5 越过缓存

对于某些应用来说，其代码只具有空间局部性而没有时间局部性，即某个数据只被访问一次，其后其相邻的数据会被访问，但是其本身不会被再次访问。如果在访问这类数据的时候，能够做到既不占用缓存，又能够利用到其空间局部性，那么就能够将缓存留给更需要缓存的数据访问。实际上只需要硬件为读写分配几条长度和缓存线长度相同的缓冲区即可。某些现代X86处理器使用流加载或流存储的概念，如SSE/AVX中的__mm_stream_load、__mm256_stream_load和__mm_stream_store、__mm256_stream_store。

通常使用流加载和流存储指令要求数据访问的步长为1，如下面的代码所示：

```
for(int i = 0; i < n; i++){
    b[i] += a[i];
}
```

由于现代X86处理器为编程的常见情况做了许多优化，流加载和流存储并不总是能够提升性能。

2.4.6 硬件预取

为了利用空间局部性，同时也为了掩盖传输延迟，可以投机性地在数据被用到之前就将其取入缓存。这一技术称为预取（Prefetch）。本质上讲，处理器每次加载一条缓存线即是一种预取，即预取了这条缓存线上的其他数据。

预取可以通过硬件或软件控制。典型的硬件指令预取会在缓存因失效而从内存载入一个缓存线的同时，请求紧随其后的另一个缓存线。某些Intel的处理器需要通过启用BIOS的选项来启用硬件预取，因为如果程序的局部性不好，则硬件预取反而会降低性能。

如果程序访问数据能够很好地满足局部性要求，硬件和软件预取几乎总能提高性能，但是如果程序的局部性很差，则预取反而会降低性能。预取降低性能的原因主要有以下几个：

- 如果预取的数据在随后没有被访问，那么预取的数据就不会被使用（完全浪费掉了），还不如不预取；
- 如果预取过早的话，可能会出现冲突不命中（预取的数据有可能会被替换出缓存）；
- 有可能导致满不命中，如果缓存已满，那么预取的数据需要换出缓存中数据，如果被换出缓存的数据在随后被访问，那么就增加了缓存访问次数。

X86和ARM处理器支持软件预取技术，如SSE和NEON中的prefetch指令。

2.4.7 缓存结构

缓存以缓存线为基本单位读写，每条缓存线可保存L（现代机器上L一般为64）个字节。数据在缓存间上下移动时以缓存线为单位，即不可能出现只加载半条缓存线的情况。对于软件开发人员来说，缓存的总量和缓存线的大小相当重要，另外缓存的层次结构、缓存映射策略也需要了解。

目前内存的容量越来越大，根据机械定律，内存容量越大则其访问延迟就越长，为了弥补访问时间的损失，缓存会越来越大且层次会越来越多。目前大多数机器具有三级缓存，不久的将来，四级缓存也会出现。

1.缓存线

缓存线是缓存中数据交互的最小单位，即读写数据时，每次会读写一个或多个缓存线，不会存在读取半个缓存线（不包括寄存器从一级缓存中读）的情况。

通常缓存线的长度是2的幂，主流的CPU上缓存线长度为64B；主流的GPU上，缓存线长度为128B。缓存线会映射到连续的地址，通常读取数据时，如果能够对齐到缓存线长度，可以有效减少访存的次数。

在缓存总量一定的条件下，增加缓存线长度会增加访问延迟，但是有可能减少访存次数以提高带宽。

2.缓存组

为了减少读取多个映射到同一个缓存的内存地址时造成的存储器访问“抖动”的影响（即冲突不命中），通常将多个缓存线组成一组，每个对齐到缓存线的内存地址可映射到一组中的某一条缓存线。在读取时，具体读入到组中的哪一条缓存线则由其替换策略决定，常见的替换策略是最近最少使用、随机策略等。通常组的大小为8或16条缓存线。

在缓存容量固定且缓存线大小固定的前提下，如果增加缓存组内缓存线的数量，那么缓存组的数量就会减少。缓存组数量越少，出现满不命中的可能性就越大。

2.4.8 映射策略

当缓存被数据占满时，哪些缓存的内容要被替换、从下一级缓存取出的数据要放到上一级缓存的什么地方等，这些行为都和映射策略相关。

常见的缓存映射策略是：直接相联、组相联和全相联。

组相联是指缓存组中的缓存线数量大于1，即一个内存地址可映射到缓存组中的多条缓存线。而直接相联是指每个缓存组中只存在一条缓存线；而全相联是指所有的缓存线都属于同一个组；直接相联和全相联都可视为组相联的特殊情况。

以Intel X86 CPU的一级数据缓存为例，通常其大小为32KB，缓存线长度为64B，每个缓存组中有8条缓存线，故共有64组，这样地址相差 $64 \times 64 = 4\text{KB}$ 的数据就会映射到同一组中，如果该组中的所有缓存线都已经被使用，那么就需要将某条缓存线的数据空出来（即冲突不命中）。

2.5 虚拟存储器和TLB

虚拟存储器是对内存和IO设备（包括硬盘）的抽象，通过将内存中的数据切换到硬盘，它使得进程好像拥有了比整个内存容量大得多的内存空间。虚拟存储器的设计同时简化了操作系统和编译器设计，比如可以假设进程的可执行代码的起始地址都是0x00400000，而具体的物理地址是多少则由映射机制解决，那么操作系统只需要从地址0x00400000处加载程序代码，而不必关心代码保存在哪个位置。通过虚拟存储器这种抽象，Windows和Linux都将存储器抽象为按字节寻址的线性存储器，这称为平面存储器模型。在虚拟存储器抽象下，软件开发人员无须了解存储器组织细节就可以编写高性能程序。

虚拟存储器使用虚拟地址寻址，而物理存储器使用物理地址寻址，因此硬件执行访存操作时需要将虚拟地址翻译成物理地址，操作系统和存储器管理单元（Memory Management Unit，MMU）硬件配合来完成这一工作。由于从虚拟地址到物理地址的转换非常耗时，处理器和操作系统主要使用了两个优化来减少转换次数。

- **分页机制**，虚拟存储器和物理存储器被划分为大小相等的页，虚拟存储器和物理存储器之间的每次交换都以页为单位，通常页大小为4KB、64KB或4MB（Linux下页的默认大小即为4KB），这远大于缓存线的大小，处理器通过每次载入一页的方式（相当于一次转换可完成一页而不是一个数据）减少载入单位数据时从虚拟地址到物理地址转换的消耗。

- **TLB**，TLB用于缓存已经翻译的虚拟地址，通过利用访问页的局部性来减少翻译次数。由于虚拟地址和物理地址之间的转换非常耗时且TLB不命中的代价很大，因此TLB的映射策略通常设计为全相联。实际上，在一些硬件上，缓存层次中也使用多层TLB来减少地址翻译代价。一些需要对多维数据进行访问的程序在大数据量的情况下，通常存在TLB不命中的情况。

解决程序TLB不命中而导致的性能问题的方法主要有如下几个：

- 增大页的大小。比如以前使用了4KB大小的页，现在改成使用64KB、甚至2MB大小的页；
- 对于重复多次使用且局部性很好的多维数据，临时分配数据空间来保存数据的一部分，然后重复使用该临时数据空间，此时TLB只需要保存这一部分临时数据空间的地址即可；
- 对数据进行分块优化。如果数据被多次使用，只是由于其大小超过TLB能够缓存的范围，即满不命中。此时可将数据分块，操作完一块后接着操作另一块。

对代码优化人员来说，无须了解虚拟存储器和TLB的实现细节，只需要依据其抽象就可以编写高性能

的代码。

2.6 NUMA技术

随着多路系统中多核处理器数目的增多，内存带宽和多路系统计算能力之间的差距越来越大。为了提高多路系统中多核处理器之间通信的带宽，NUMA应运而生。

对Intel的处理器来说，多路处理器之间通过QPI总线通信；而AMD的处理器则通过HT总线通信。QPI和HT的带宽比内存带宽小，而延迟则大于访问内存的延迟，这是NUMA存在的根本原因。

对NUMA架构上的每个处理器来说，访问各个存储器的速度是不一样的。访问“靠近”处理器自身的存储器速度要比访问“远”的存储器快，如图2-3所示。

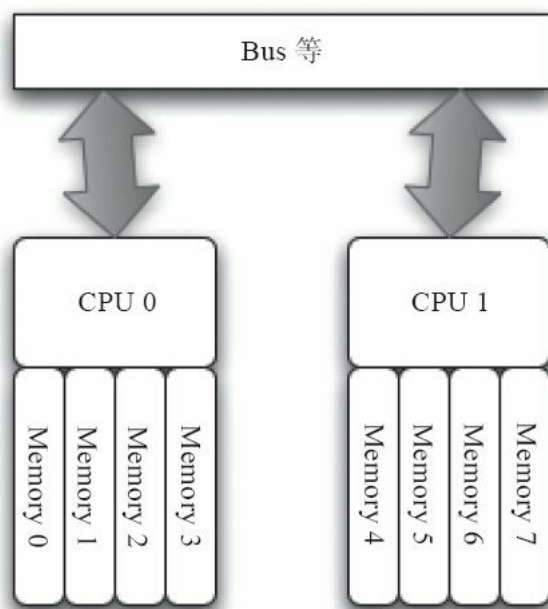


图2-3 处理器和存储器连接

要利用NUMA的优点，控制流分配存储器时要保证分配的存储器离自身所在的处理器比较“近”，这需
要保证两点：

- 控制流分配存储器时分配在离自己近的物理内存上，这可以通过在线程内使用malloc分配办到，如代码清单2-3的OpenMP代码所示。但是如果控制流在运行时迁移到另一个物理核心上，这点就可能失效。这引出了下一点：

代码清单2-3 分配

```
#pragma omp parallel  
{
```

```
int myid = omp_get_thread_num();
data[i] = (float*) malloc(size);
init(data[i]);
}
```

· 控制流不能在核心间迁移。为了使得处理器核心能够公平地得到任务，操作系统会在核心间迁移控制流，一旦出现线程迁移到其他处理器上的情况，那么此线程访问存储器就可以不满足NUMA要求。可以通过线程亲和性保证线程不会在核心间迁移，如gcc环境变量GOMP_CPU_AFFINITY。

GOMP_CPU_AFFINITY使用（s-e: span）格式表示如何将线程固定在CPU上，其中s表示起始线程固定到的CPU编号，e表示最后一个CPU编号，span表示相邻线程间隔的CPU编号距离，需要提示的是：CPU从0开始编号。如GOMP_CPU_AFFINITY="0-34-10: 2"表示：线程0固定到0号CPU、线程1固定到1号CPU、线程2固定到2号CPU、线程3固定到3号CPU、线程4固定到4号CPU、线程5固定到6号CPU、线程6固定到8号CPU、线程7固定到10号CPU。

异构并行计算引入了GPU和FPGA等加速器，这些加速器目前通过PCI-E连接到IOHub，IOHub再连接到CPU。如果主板支持多个socket，那么GPU和GPU之间，GPU和CPU之间也存在NUMA。

Linux系统提供了numactl工具来设置NUMA特性，而pthread和OpenMP实现提供了设定线程亲和性的函数。代码清单2-4就使用pthread将线程t1固定到处理器0上。

代码清单2-4 numactl使用示例

```
cpu_set_t cpu_info;
__CPU_ZERO(&cpu_info);
__CPU_SET(0, &cpu_info);
if (0!=pthread_setaffinity_np(t1, sizeof(cpu_set_t), &cpu_info)){
    printf("set affinity failed");
}
```

cpu_set_t是一个掩码向量（目前长度为1024位），每一位对应着一个CPU核心，__CPU_ZERO宏将掩码向量cpu_info全部清零，__CPU_SET宏将掩码向量的某一位设置为1，如代码清单2-4就将掩码向量cpu_info中第一位设置为1，而函数pthread_setaffinity_np则将线程固定到掩码指定编号的CPU上，如代码清单2-4就将线程t1固定到CPU 0上。

2.7 本章小结

本章主要介绍了现代处理器的主要特性，包括但不限于如下内容：

现代处理器使用的指令级并行技术有：指令流水线、乱序执行、指令多发射、分支预测和VLIW。

现代处理器主要使用两种向量化并行技术：SIMD和SIMT，其中CPU主要使用SIMD，而GPU主要使用SIMT。

本章还介绍了线程级并行的基础知识：用户线程和内核线程、常见的多线程编程库等。

关于现代处理器的存储器系统，本章介绍了：缓存层次结构、缓存一致性、缓存不命中的分类、缓存结构及缓存的映射策略，另外还介绍虚拟存储器和TLB及NUMA技术。

现代处理器特性是代码优化和并行的基础之一，而另外一个基础是如何衡量算法和程序的性能，只有知道如何衡量算法和程序的性能才能评价算法和程序的优劣。下章笔者将会介绍目前常见的衡量算法和程序性能的一些标准和概念。

第3章 算法性能和程序性能的度量与分析

通过分析算法或程序的性能，能够说明优化是否产生效果。最重要的是分析程序的性能，告诉代码优化人员程序发掘了计算机的多少计算能力，即程序还有多少潜在的优化空间。算法的性能分析能够指导算法设计，而算法或程序性能的度量是分析的基础。目前常用的表达算法性能的量有时间复杂度和空间复杂度，而表达程序性能的量有：时间、FLOPS、CPE、IPC等。

由于时间复杂度和空间复杂度无法准确衡量算法实现的复杂度，因此本章提出了[实现复杂度](#)的概念。时间复杂度和空间复杂度关注抽象算法的性能，而[实现复杂度更关注于如何估计算法的具体实现性能](#)。即使是同一个算法，不同的开发人员、基于不同的平台实现，其最终的性能通常也差别巨大，相比时间复杂度，实现复杂度提供了一种更好的比较算法实现后优劣的途径。

本章首先介绍算法的性能度量标准，通过算法度量标准可以大致分出算法的优劣；其次介绍如何表达一个程序在某个平台上运行时的性能；再次介绍对程序的一部分代码向量化或并行化对整个程序而言的性能收益；最后介绍一些常见的程序性能分析工具。笔者希望通过这种方式让读者了解：如何衡量算法和程序的性能，如何通过工具获得程序的性能，为性能优化打好基础。

3.1 算法分析的性能度量标准

对于代码性能优化和并行化来说，如何衡量算法的性能非常重要。如果能够在编写代码前就依据某些标准选择最优算法，那么就会节约大量算法选择时间。

通常使用时间复杂度来度量算法运行的性能，使用空间复杂度来度量算法要使用的存储器空间大小。时间复杂度和空间复杂度用得如此广泛，几乎每一本关于算法的教科书都会使用一定的篇幅来介绍它们，故本章只予以简单介绍。

对于代码性能优化来说，时间复杂度和空间复杂度存在很多缺陷。为了更好地指导代码性能优化和并行化，本章提出了实现复杂度的概念，并将其进一步细分为计算复杂度、访存复杂度和指令复杂度。由于初次提出实现复杂度的概念，故在一些细节上可能存在不足，还请读者原谅。

3.1.1 时间复杂度与空间复杂度

时间复杂度和空间复杂度是常见的衡量算法性能的度量标准。由于大学教科书中花大量的篇幅介绍，故本节只简单介绍其作用。

1.时间复杂度

通常使用时间复杂度来衡量算法需要的大致计算时间尺度，比如算法需要操作 n 个数据，每个数据大约要运算 $2n$ 次，则时间复杂度为 $O(n^2)$ 。

以下列伪代码所示的矩阵向量乘法为例：

```
for(int row = 0; row < nRows; row++){
    T temp = (T) 0;
    for(int col = 0; col < nCols; col++){
        temp += matrix[row*nCols + col]*vector[col];
    }
    product[row] += temp;
}
```

其中外层循环执行次数为 $nRows$ ，内层循环执行次数为 $nCols$ ，循环内计算次数为4，外加对累加器 $temp$ 赋值为零和更新结果 $product$ 的两次计算，故其总的计算量为 $nRows \times (4 \times nCols + 2)$ ，假设矩阵为方阵，即 $nRows == nCols = n$ ，则总的计算量为 $4 \times n^2 + 2 \times n$ ，时间复杂度忽略低阶和常数的影响，故上述算法的时间复杂度为 $O(n^2)$ 。

从上面计算矩阵向量乘法的时间复杂度的过程我们可以得知，时间复杂度有以下几个方面需要特别注意：

- 1) 只关注计算对性能的影响，而不关注访存；
- 2) 只关注最高阶计算量对性能的影响，而忽略低阶计算量对算法性能的影响；
- 3) 只关注运算量的阶，而忽略阶的比例常数对算法性能的影响；
- 4) 不关注不同的计算对性能的影响。

再比如如下的矩阵加法算法代码， $nRows == nCols = n$ ，其计算量为 $5n^2$ ，和矩阵向量乘的例子相比，它们的时间复杂度均为 $O(n^2)$ ，即时间复杂度分析的结果是一样的，而在性能分析实践中，这两者的性能必然存在比较大的差距。

```
for(int row = 0; row < nRows; row++){
```

```
        for(int col = 0; col < nCols; col++){
            int index = row*nCols + col;
            result[index] = matrix1[index]+matrix2[index];
        }
    }
```

虽然本节后面会解释两者性能差别的原因，但是感兴趣的读者可以先自行分析一下。

时间复杂度是一个很好的大致评价算法优劣的标准，但不是一个很好的性能优化度量标准，主要原因如下。

- 时间复杂度只考虑计算对性能的影响，而不考虑数据读写对性能的影响。从冯氏架构来看，数据读写为计算提供运算所需的输入/输出，不考虑数据读写明显忽略了影响性能的一个重要方面；从计算机架构的进展来说，目前数据读写越来越成为多核向量处理器性能的瓶颈（内存墙）。这意味着大部分时间花费在访问存储器上面的算法都不适合使用时间复杂度来估计性能，比如上面提到的矩阵加法。

- 不同的算法具有不同的限制因素。有些算法的大部分性能限制在读取网络数据上，有些算法限制在TLB^[4]上，对于这些算法来说，只有时间复杂度是不够的。一个典型的极端例子就是：从来没有优秀的开发人员使用时间复杂度来衡量网络服务器或数据库服务器的性能。

- 不考虑处理器执行不同指令的速度差异。处理器生产商需要考虑处理器将来要运行哪些软件，这些软件的主要运算是什麼，如何能够在不大量增加晶体管的情况下提升这些软件的性能，故并不是对所有指令一视同仁。通常生产商将更多的晶体管用于更常见指令，以提高这些指令的性能，这导致同一处理器上不同指令的性能并不相同。比如Intel Haswell处理器一个周期可执行4次整数加法，而对于浮点加法一个周期只能执行1次。

- 忽略常数和低阶运算对性能的影响。由于时间复杂度忽略常数和低阶运算对性能的影响，因此常常出现一些时间复杂度为 $O(n)$ 的算法，在某些问题规模上，性能比时间复杂度为 $O(n^2)$ 的算法要差的情况。比如计算复杂度为 $O(n \log n)$ 快速排序算法在实践中通常比时间复杂度为 $O(n)$ 的基数排序要快。

- 不能很好地度量并行算法。并行算法需要的数据/任务划分、通信等都超出了时间复杂度的范围。一旦并行算法的这些特性成为了影响性能的重要因素，时间复杂度分析通常会得出与实际不符的结论。

在大的尺度上，用时间复杂度分析抽象算法的性能非常有用，但是某些具体算法的实现则需要特别留意是否有上面列出的情况。

2.空间复杂度

空间复杂度用来表示算法运行需要的存储器空间，比如某算法需要使用一个长度为 n 的空间作为输入，

使用另一个长度为 n 的空间作为输出，则空间复杂度为 $O(n)$ 。

和时间复杂度类似，空间复杂度也忽略低阶和常数的影响。如上一小节的矩阵向量乘法算法，输入矩阵占用空间大小为 n^2 ，输入向量和输出向量大小俱为 n ，其使用的空间大小为 $n^2+2 \times n$ ，其空间复杂度为 $O(n^2)$ 。而上一小节所示的矩阵加法算法，输入矩阵和输出矩阵大小俱为 n^2 ，其使用的空间大小为 $3 \times n^2$ ，其空间复杂度为 $O(n^2)$ 。两个算法的空间复杂度俱为 $O(n^2)$ ，但是很明显它们运行时需要占用的空间大小并不相同。

空间复杂度能够衡量程序运行时需要的大致数据量，因此与性能优化没有过多联系。表面上看空间复杂度与程序的性能关系很大，如果算法设计不好，运行时需要消耗大量内存，这样就会影响整个系统的性能。持有这种观点的人简单地把程序在真实系统上运行时需要的空间大小等价于算法分析时的空间复杂度。另外决定现实系统运行速度的是算法在现实系统上运行时的瓶颈，这并不由算法本身独立决定。

空间复杂度也忽略常数的影响，且无法衡量程序运行时要访问多少次存储器，更没有考虑缓存层次结构的影响。

空间复杂度的定义侧重于表示算法要使用的存储器空间，其相对意义大于绝对意义，即使其相对意义也并不非常实用。我们无法估计一个空间复杂度为 $O(n)$ 的算法其在某个平台上实现时需要的内存空间大小，也无法说空间复杂度为 $O(n)$ 的算法在某个平台上实现时需要的具体空间就一定比空间复杂度为 $O(n^2)$ 的算法小。

[1] TLB (Translation Lookaside Buffer, 传输后备缓冲器) 是一个内存管理单元，用于改进虚拟地址到物理地址转换速度的缓存。TLB作为一个小的、虚拟寻址的缓存，其中每一行都保存着一个由单个PTE组成的块。如果没有TLB，则每次取数据都需要两次访问内存，即查页表获得物理地址和取数据。

3.1.2 实现复杂度

无论时间复杂度还是空间复杂度，都是理论上或“纸面上”的东西。在某个特定的处理器上，程序的实际性能基本上由运行时的计算、访存和指令决定，因此本文提出了计算复杂度、访存复杂度和指令复杂度的概念。为了便于区分时间复杂度和空间复杂度，本文将计算复杂度、访存复杂度和指令复杂度统称为实现复杂度。

相比时间复杂度和空间复杂度，实现复杂度更贴近硬件，对程序性能的度量通常也更为准确。将实现复杂度作为优化代码性能的标准是一个更好、更实际的选择。

对于具体的代码而言，依据在某种处理器上运行的性能瓶颈不同而采用实现复杂度的不同方面来度量。对于一个计算限制的算法，应当使用计算复杂度和指令复杂度；而对于一个存储器限制的算法，应当使用访存复杂度来分析。

1. 计算复杂度

计算复杂度可用来衡量每一个控制流的计算数量，不但计算加载存储指令，还需要考虑指令的吞吐量或延迟，通过加权平均来表示计算性能。由于各种现代向量多核处理器具有不同的特点，比如X86 CPU是为延迟优化的，在考虑计算复杂度时权重应该以延迟为准；而GPU是为吞吐量优化的，考虑计算复杂度时权重应当以吞吐量倒数为准。但这并不绝对，一些优化良好的X86 CPU代码有很好的指令级并行能力，也应当从吞吐量倒数考虑。

以输入两个长度为 n 的32位浮点数组 a 、 b ，对两个数组中的每个元素 $a[i]$ 、 $b[i]$ 做加法，并以保存结果这一简单运算为例（为叙述方便，标记为例1）。其计算复杂度公式为：

$$O(2n \times \alpha + \beta \times n + \gamma \times n)$$

其中：

- α 表示加载的吞吐量倒数，即加载指令的吞吐量倒数；
- β 表示加法指令的吞吐量倒数；
- γ 表示存储指令的吞吐量倒数。

如果代码运行在一台为延迟优化的处理器上，只需将 α 、 β 、 γ 替换成延迟时钟周期数即可。以Intel

Haswell架构为例，假设数据都在一级缓存中，此时 α 为3，表示从一级缓存中加载数据的延迟为3个周期； β 为0.25，表示加法指令的吞吐量为4； γ 为3，表示从存储数据到一级缓存的延迟为3个周期，故计算复杂度为 $O(9n+0.25 \times n)$ ，从中可以看出：绝大部分计算复杂度是由访存带来的，这意味着这个算法的瓶颈是访存。

计算复杂度还和硬件的架构相关，如果算法的某种指令可以被SIMD单元处理且程序的确会使用SIMD指令，那么其吞吐量倒数就会考虑一次可SIMD处理的操作数量。以例1为例，如果使用128 SSE指令的话，其计算复杂度为 $O(0.5n \times \alpha + 0.25\beta \times n + 0.25\gamma \times n)$ 。如果某条分支指令需要被SIMD单元执行两次，则需要对相应的吞吐量倒数除以2。

在并行算法上应用计算复杂度分析时，如果控制流之间的计算量并不均衡，那么需要同时考虑控制流之间的最大计算复杂度和最小计算复杂度，它们的比例就可大致估计负载不均衡的程度，进而估计优化负载后，最大可提升的性能比例。假设一个有两个线程的并行算法，两个线程的计算复杂度分别为 $O(0.5n)$ 和 $O(2n)$ ，故计算时间由 $O(2n)$ 决定，应用负载均衡算法后，最好的性能提升是 $2 \times 2 / (2 + 0.5) = 1.6$ 倍。

2. 访存复杂度

访存复杂度衡量算法访问缓存层次的字节数量，通常不是直接地加和，而是由瓶颈在缓存层次的哪一级决定。比如，如果算法是内存密集型的，只需要分析其内存访问字节数量；如果算法是一级缓存密集型，则需要分析一级缓存访问字节数量；如果算法是多核心共享缓存密集型，则需要分析所有处理器核心访问共享缓存的总数量。

计算访存复杂度时，由于缓存容量大小、替换策略和缓存线大小导致的额外开销也应当计算在内。

对于访存复杂度，通常使用缓存层次的带宽来表示，比如算法发挥的一级缓存的带宽是多少，二级缓存带宽多少。

由于访存复杂度要考虑存储器层次结构，因此对于一些复杂的算法需要处理器提供硬件计数器支持，可喜的是，现在大多数主流的处理器都已经提供了该项支持。

对于某一个具体的程序来说，通常有以下几种带宽定义：

1) 存储器的峰值带宽：该值由硬件规格决定，通常可由硬件参数计算出来。如双路双通道DDR3-2333内存，其峰值带宽为： 2 （双路） $\times 2$ （双通道） $\times 2.333$ （内存等效频率） $\times 8$ （64位内存控制器） $= 74.6\text{GB/s}$ 。

2) 可获得的存储器带宽：该值亦由硬件决定，不过其大小由程序测试获得。由于硬件设计方面可能存

在不足，可获得的存储器带宽通常要小于存储器的峰值带宽。如NVIDIA K20c GPU的峰值显存带宽大约200GB/s，但是可获得的显存带宽大约175GB/s。

3) 程序发挥的存储器带宽：此值是硬件为某个特定程序发挥的带宽，一般而言，此值可通过硬件计数器获得，但是某些顶级的软件开发人员也能够手工计算出来。

4) 程序的有效访存带宽：此值由算法计算获得，其表示程序最小要访问的数据量与程序计算时间的比值。其还表示访存优化能够达到的上限，对程序访问带宽的优化不可能比程序的有效访存带宽还小。

访存复杂度分析的是程序发挥的存储器带宽，因为其值是程序在硬件的存储器上执行时的实际访存字节数。

可获得的存储器带宽与存储器的峰值带宽的比例可以反映存储器处理存储器访问的效率，比如Intel Xeon Phi处理器的峰值显存带宽大约350GB/s，而可获得的显存带宽大约170GB/s，这就是一个典型的存储器硬件存在不足的例子。对于主流的GPU而言，此值在70%~80%，而在Intel X86处理器上，此值通常能够超过90%。

程序发挥的存储器带宽与可获得的存储器带宽的比例表示程序已经发挥了硬件能够提供的带宽的比例，它意味着存储器还能够提供的带宽的比例，如果值接近1，则表示存储器的带宽已经被耗尽，下一步的优化应当考虑如何减少访存次数和浪费的存储器带宽。

程序的有效访存带宽和程序发挥的存储器的比例表示程序访问存储器的效率，本书称之为访存效率。程序的访存效率越高表示浪费的带宽越小，即访存模式越优秀。比如访存效率为50%，那么表示有一半的带宽访问是程序不需要的。

3.指令复杂度

相比计算复杂度和访存复杂度而言，指令复杂度则分析实现算法的某个具体程序生成的指令数量、类型、是否可以同时执行，等等。

从某种程度上说，指令复杂度的分析是人的大脑对程序指令在具体处理器上执行的分析，因此非常复杂和困难，也更容易出错，笔者通常使用一些编译器工具协助分析。

通常使用执行代码关键路径所需要的时钟周期数来表示指令复杂度。指令复杂度是计算复杂度在具体计算平台上的表现。由于指令复杂度和处理器、编译器及运行时环境密切相关，故并没有一个统一的标准来度量指令复杂度。不同算法的指令复杂度具有可比性，但是没有意义，因为对于性能优化没有参考价值。

由于处理器能够同时执行不同的指令，计算复杂度没有考虑这一情况，而指令复杂度考虑，因此指令复杂度分析的结果通常比计算复杂度分析的结果要小。

由于同一代码在不同硬件、不同编译器、不同编译选项配置条件下，生成的指令系列可能并不相同，因此指令复杂度可能不适合交流。但对于顶级代码优化人员来说，指令复杂度是优化的终极法宝。代码优化的过程，就是指令复杂度降低的过程。

指令复杂度分析通常关注耗时最长的循环，通常分析此循环内部的指令数量，不同指令是否能够同时发射、执行等，通过这些信息来重构代码，以便生成的指令能够更好地在处理器上执行。

3.2 程序和指令的性能度量标准

目前常用的度量程序、代码段和指令性能的标准有时间、FLOPS、吞吐量和延迟等。对于不同类型的内核（指程序耗时的核心计算部分），应当选用不同的评价标准。如果在一个延迟优化的机器上使用吞吐量来评价性能的意义就不会比延迟好。对于一个存储器限制的内核使用FLOPS就可能得出错误的结论。

本节将介绍常见的程序代码和指令的性能度量标准，并分析其适用场景和优缺点。

1.时间

判断程序优劣的最简单方式就是计算程序的运行时间，在同一台机器上，运行时间短的程序一般来说是更优的。当然不能一概而论，毕竟决定程序运行速度的因素很多，比如算法、机器的指令集、使用的语言，编译器是否优秀，等等。

计时的方法很多，如标准C库的time、clock系列函数，CUDA的事件，Linux下的gettimeofday，Windows下的GetTickCount等。

- time系列函数可以返回从1970年1月1日到现在过的秒数，另外它还提供了比较、格式化及本地化功能。

- clock函数返回从进程开始执行到程序调用clock时处理器运行的时钟计时数目（此计时周期并不等于处理器时钟周期），两次调用之差即是调用之间程序片段运行时占用的处理器时钟计时数目。宏CLOCKS_PER_SEC表示一秒的时钟计时数目。

在Linux系统下，笔者习惯使用gettimeofday函数，其原型如下：

```
gettimeofday(struct timeval*, struct timezone*);
struct timeval{
    long tv_sec;
    long tv_usec;
}
struct timezone{
    int tz_minuteswest;
    int tz_dsttime;
}
```

为了方便，笔者通常使用下面的函数来计算当天系统时间的毫秒数表示：

```
long getTimeOfMSecond(){
    struct timeval tv;
    gettimeofday(&tv,NULL);
    return tv.tv_sec*1000+tv.tv_usec/1000;
}
```

通常使用clock和getTimOfMSecond函数就已足够，如果需要纳秒级的计时，可以使用Linux的clock_gettime函数，clock_gettime支持多种计时模式，感兴趣的读者可以使用Linux系统的帮助命令man查询。

需要提醒开发人员的是：clock函数并不适合准确的测试程序的运行时间，更不应当用来测试多核并行程序运行的时间。对于多线程程序来说，由于多线程的影响导致最后的计时结果通常是实际执行时间的倍数，这个倍数会大致等于核数。

2.FLOPS

FLOPS表示硬件每秒执行的浮点运算的数量，通常使用的单位有M（ 10^6 ）、G（ 10^9 ）。以两个大小为N的方阵相乘（计算量为 $2 \times N^3$ ）、耗时t秒为例，其FLOPS为 $2 \times N^3/t$ 。

从定义来看，要提高一个程序的FLOPS有两种办法：

- 1) 在时间不增加的前提下，增加程序的浮点运算数量；
- 2) 在程序的浮点运算数量保持不变的情况下，减小程序的运行时间。

现代处理器都有专门用来处理浮点运算的“浮点运算器”（FPU），FLOPS所测的实际上就是FPU的执行速度。最常用来测量超级计算机性能的基准程序Linpack就采用FLOPS来表示性能。由于FLOPS不但不区分运算指令，还不区分IO、缓存与计算，因此大多数程序的实际性能通常远小于硬件的FLOPS峰值性能。

FLOPS并不适合用来表达常用程序的性能，因为很多能够提升程序性能的算法级优化会减少程序的计算数量，这可能会导致FLOPS减小。对一些计算数量固定不变的应用来说，FLOPS能够很好地指导算法设计和处理器设计，比如一些科学计算算法、矩阵乘、NBody等。

FLOPS通常用来表示系统跑某个固定计算的性能，大量用于大型机群性能的特征。FLOPS促进了高性能机的研究，但是也使得高性能机的设计与现实脱节，目前已经有许多学者和研究人员建议废弃FLOPS。

3.CPI

通常程序中最耗时的部分就是小循环，因此如何表征对循环的优化效果就非常重要。循环内部通常操作某些数组中的元素，这样平均作用在每次循环上的时钟周期数量和作用在这次循环上的操作就直接相关。CPI（Clock Per Iteration）表示每次循环耗费的时钟周期数，能够表示每次循环的平均代价。

笔者以下的计算向量平方和代码段为例来展示如何计算一段代码的CPI：

```
float result = 0.0f;
for(int i = 0; i < num; i++){
    float temp = data[i];
    result += temp*temp;
}
```

假设测试发现上面的代码段在某个具体配置下耗时 T 个时钟周期，那么其CPI为： T/num 。

非常常用的性能优化手段循环展开的效果却无法用CPI表示，而且CPI无法判断循环的优化是否已经到了系统的界限或还有多少优化可能。如对上面的代码循环展开两次，如下所示：

```
float result0 = 0.0f;
float result1 = 0.0f;
for(int i = 0; i < num; i += 2){
    float temp0 = data[i];
    float temp1 = data[i+1];
    result0 += temp0*temp0;
    result1 += temp1*temp1;
}
float result = result0+result1;
```

假设其运行时间不变，但此时其CPI变化： $T/(\text{num}/2)$ 。此时CPI是未循环展开时CPI的2倍，这和运行时间不变的假设不一致。但是如果循环次数定义为执行循环展开前的循环次数就可以解决这个问题。

性能优化通常要减小程序的CPI，但是这并非绝对。在循环次数不变的前提下，要减少CPI，只能是减少执行循环的周期数。通常使用的办法有：减少循环内指令数，使得时钟周期数更小的指令等。

循环的CPI通常由其依赖链（指整个循环中相互依赖的运算或访存，也称为关键路径）决定，减少循环CPI的过程就是减小或去掉循环内依赖链的过程。

4.延迟

延迟是指一条指令、一个操作从开始发射到完全结束所经历的时间。通常说一条指令需要几个周期执行完成，这个周期数就是延迟。延迟的一个形象的比喻是走扶梯，假设走一级扶梯需要一个周期，从你踏上扶梯开始到你离开扶梯时结束，你走过的长度（也就是扶梯的级数）即为扶梯的延迟。

Intel在其编程手册中给出许多指令的延迟，但是像NVIDIA和ARM并不会给出其处理器上指令的延迟。读者也可以通过软件方法测试某个硬件平台上具体指令的延迟，这个方面就不展开讲了。

对于某条具体的指令来说，减少其延迟是处理器生产商的责任；对于代码段来说，通常可以通过减少指令数量、使用延迟更小的指令等方法来减少代码段的整体延迟。

相对指令的整体延迟来说，还需要关注指令发射时间，因为在现在的乱序执行的硬件上，发射时间也是指令性能能够达到的一个上限。

现代X86 CPU是基于延迟优化的架构（虽然有乱序执行、流水线和SIMD等指令级并行技术）。对于运

行在X86 CPU上的程序来说，减少其获取操作数和执行指令的延迟是非常重要的，这导致了现代CPU的缓存越来越大，指令集越来越复杂。

现代CPU采用的延迟优化技术有：乱序执行、缓存和多线程等。

5.吞吐量

对于计算来说，吞吐量是指在单位时间内，硬件能够完成的操作数；对于访存来说，吞吐量表示在单位时间内，硬件能够读写的字节数量。通常有两种方式表示吞吐量：①每时钟周期指令数量（IPC）；②每秒执行指令数量，对于浮点数据就意味着FLOPS。

主流的X86处理器和GPU生产商都在其手册中给出了一些指令的吞吐量。读者也可以通过软件的方式测试某个硬件平台上某个具体指令的吞吐量。

从表面定义看来，吞吐量和延迟是倒数关系，实际上并非如此。由于现代CPU和GPU有非常长的流水线和多个执行单元，因此硬件平均完成一个操作的时间可能远远小于其延迟。虽然每个指令需要多个周期才能完成，但是能够达到在一个周期内完成多个指令的效果。

由于对吞吐量优化的技术不要求尽快得到数据，而要求在单位时间做更多的运算，因此硬件往往采用更多频率更低的计算单元，采用“人海战术”取胜。由于这种小而简单的单元工作频率和电压都较低，因此耗能少，能够在给定的能耗下集成更多的核心，进而在不增加功耗的前提下提高整体的吞吐量。目前无论是X86 CPU还是GPU都采用了这种思路来提高性能。

吞吐量倒数表示在某个架构上，某种指令的吞吐量的倒数。吞吐量倒数表示平均下来某种指令最少需要的时钟周期数，是一种下限，而延迟则是一个上限。在不考虑其他指令影响的前提下，使用吞吐量倒数和延迟即可基本估计运行一段代码所需要的时间范围。假设一段代码中有500条整数加法指令，在Intel Haswell处理器上运行时，由吞吐量倒数决定的下限是125个周期，由延迟决定的上限是500个周期，故这500条指令耗费的时钟周期数范围是[125, 500]。对于性能优化来说，使用瓶颈指令（耗时最多的指令）的吞吐量倒数乘以数量，就可大致估计出一段代码的最低运行时钟周期数。

6.IPC

IPC（Instruction Per Clock）表示每时钟周期执行的指令数目，它是指令吞吐量的一种具体表示。对于某个具体的程序来说，处理器的运行频率和指令的IPC决定了其性能。处理器执行某种指令的吞吐量（FLOPS）的计算公式是：指令吞吐量=IPC×处理器频率。

对于某条具体的指令来说，IPC表示处理器每时钟周期执行多少条此种指令。而对于一个代码段来说，也可以算出一个平均IPC，但是这个IPC没有意义，因为平均IPC大的程序执行时间并不一定小。当然主流的处理器生产商Intel、AMD和NVIDIA都提供了其处理器上某些指令的IPC，读者也可以通过编程的方式测试其值。

在使用IPC时，应该注意以下几点：

1) **不同的程序不能使用IPC来比较性能**。即使同一算法的不同实现或同一程序的不同优化版本也不能使用IPC来比较性能，因为不同算法或同一程序的不同优化版本的性能瓶颈可能并不相同。对没有改变性能瓶颈的同一程序的不同优化版本，通常IPC越大，程序性能越好。

2) **IPC很难衡量一个复杂程序的性能**。因为在不同的硬件上，不同指令的IPC不同，硬件的时钟频率也不相同，因此执行程序时IPC最大的硬件并不一定运行时间最少；同一程序在不同的机器上、不同的编译器上被翻译成多少种、多少条指令也不一定相同，因此执行程序时IPC最大的硬件可能是因为生成的指令数量最多。

处理器的指令发射IPC（处理器每秒发射多少条指令到流水线上）也非常重要，因为一个或多个核心共享发射单元，因此在指令种类丰富的情况下，发射IPC很容易成为瓶颈。

IPC还表示了一种架构执行某种指令的峰值性能，例如，在一个执行某种指令IPC为2的处理器架构上，由该指令限制的程序不能获得大于2的IPC。

7.little定律

little定律认为要发挥硬件执行某种指令的计算能力所需要的并行度等于指令的延迟乘以指令的吞吐量。并行度指没有依赖的指令数量，比如：如果循环内有8条不相关的乘加指令，那么此循环内乘加指令的并行度为8。并行度大意味着需要更多的不相关的指令，这要求软件开发人员发掘算法的并行性。Little定律是并行化和向量化最重要的理论工具，没有之一。

假设现代双路X86 CPU（每路6核）的内存带宽大约是50GB/s，内存的延迟大约是200个处理器时钟周期，在一个主频是3GHz的机器上，这意味着需要有 $50 \times 10^9 \times 200 \times 0.333 \times 10^{-9} = 3333$ 字节数据正在进行流水线操作才能发挥内存的带宽。假设一条缓存线长度为64字节，这意味着大约需要53条缓存线同时在流水线上执行，平均对一个核心而言，大约需要5条无依赖的缓存线访问。

Intel Haswell处理器的一级缓存读带宽为每周期每个核心64字节，延迟为3个周期，并行度为192字节。假设使用256位SIMD操作，则需要6条SIMD读指令同时在流水线上执行。

假设现代GPU的显存带宽大约是200GB/s，显存的延迟大约是700个时钟周期，在一个主频是700MHz的GPU上，这意味着需要同时读取 $200 \times 10^9 \times 700 \times 1.43 \times 10^{-9}$ 字节数据，即200200字节。假设一条缓存线长度为128字节，意味着需要1564条缓存线，Kepler K20 GPU拥有14个SM，这表示每个SM同时要有110条缓存线访问在流水线上执行才能完全利用显存带宽。

Intel Haswell处理器的浮点乘加指令吞吐量为每核心每周期2条，延迟为每条指令5周期，故需要的并行度为10，如果没有使用超线程技术，那么要达到浮点乘加的峰值，循环内需要10条没有依赖的乘加指令。NVIDIA Maxwell浮点乘加指令吞吐量为每周期每SM 128条，其延迟为6个周期（笔者测试获得），故并行度为768，如果每个SM上有128个线程正在执行，那么工作项内需要6条无依赖的乘加指令。

3.3 程序性能优化的度量标准

如何衡量代码优化获得的性能提升，最简单的办法就是比较优化前后的性能，即加速比。加速比是一种相对比值，给人一种优化后性能比之前快了几倍的印象，因此通常加速比越大，优化效果越显著。

对于并行来说，我们希望程序运行时间和核数成反比，即核数越多，运行时间越短。现实总是和理想有偏差，我们使用并行效率来衡量这种偏差。

3.3.1 加速比与并行效率

通常使用加速比来定义性能优化效果，其表示优化前程序的运行时间与优化后运行时间的比值，计算方式如下：

$$S = \frac{T_s}{T_p}$$

其中 T_s 表示串行程序的执行时间， T_p 表示优化后程序的执行时间。

假设串行一个程序执行时间是100s，在经过性能优化之后，其运行时间下降到20s，那么性能优化的加速比为：100/20=5。

加速比定理也可以用来衡量并行化的效果（因为并行化只是代码优化方法的一种），但是其无法说明并行优化的可扩展性，而并行效率表达了这一概念。并行效率定义为加速比与计算单元数的比例，计算公式如下：

$$P = \frac{S}{Mp}$$

其中 Mp 表示使用的核心数量。

假设串行一个程序执行时间是100s，在经过并行化之后，在8核处理器上其运行时间下降到20s，那么并行化获得的加速比为：100/20=5，而并行效率为：5/8=0.625。

一般而言，如果并行效率低于0.5就说明并行优化是失败的（这可能意味着双核的性能还比不上单核，当然如果你有几十个核，可能会认为并行效率为0.5以下也是成功的），通常此时应当减少核心数目而非相反。一般并行效率在0.75以上就已经非常好了。

并行效率和可扩展性紧密相连，并行效率越高，可扩展性通常就越好。

有时并行化获得的加速比会大于处理器数目，这称为“超线性加速”。超线性加速的原因之一是缓存，因为每个核心通常有自己的一级缓存，如果单个处理器没有办法将数据全部放到缓存，那么多个处理器划分后的数据有可能放入缓存中；如果缓存的效果超越多线程的开销时，那么就会出现超线性加速现象。

不同的优化人员获得的加速比之间通常没有可比性，因为基于的原始代码可能并不一样。简单的比喻就是：你的身高是一个身高为0.5米的人的2倍，另一个人A身高是笔者身高（1.8m）的0.8倍，这并不意味着

你比A高，倍数不是决定因素。

代码优化和并行化可能只适用程序中的一部分代码，那么获得某个加速比之后，对程序整体性能又会有怎样的影响，这个问题由Amdahl定律和Gustafson定律回答。

3.3.2 Amdahl定律和Gustafson定律

Amdahl定律描述了在固定问题规模的前提下，对某个模块获得了加速比S后对程序整体性能的提升，假设并行的部分在未优化前占整体比例为f，则程序的整体加速比S'为：

$$S' = \frac{1}{1-f + \frac{f}{S}} = \frac{1}{1-(1-1/S)f}$$

假设某个程序中，你优化了80%的代码，对这80%的代码你获得了加速比10，那么对整个程序而言，你的优化获得的加速比为： $1/(1-0.8+0.8/10) \approx 3.57$ ，这远小于10。

S'对f求导可知，在[0, 1]范围内，函数递增，这表明可并行的部分越多，整体加速比就会越大。当S无限增加时，S'逼近 $1/(1-f)$ ，这意味着程序最终可能获得的最大加速比由原始程序中串行代码运行时间的比例决定。故如果你只优化了程序的80%的代码，那么你最多可获得的加速比为5。对于这个准则的一个简明的解释就是如果一个程序用时10秒，无论如何优化耗时1秒的模块，总的时间不会少于9秒。而如果优化耗时9秒的模块，更容易优化到总耗时9秒以下。

依据Amdahl定律，要提高优化代码获得的加速比上限，需要优化更多的代码。从另一方面来说，Amdahl定律提供了另外一个事实上的性能优化上限，即如果你的优化产生的效果已经靠近Amdahl定律决定的上限，再优化就需要付出更多的努力（优化程序80%的代码，获得2倍加速，程序整体加速比为1.67；而获得4倍加速的话，程序整体获得的加速比为2.50。换句话说，你优化80%的代码又获得了2倍的加速，而程序整体获得的加速比为1.50）。

对于具有串行代码的程序来说，Amdahl定律是一个打击。而Gustafson定律则从增加问题规模方面回避了这个问题。

Gustafson定律描述了增加处理器数目的同时相应的增大问题规模对加速比的影响，Gustafson定律认为此时加速比是线性的（执行时间不增加），实际上这种情况只在解决问题的时间和规模之间存在线性关系的时候成立，如果其关系非线性，那么就不成立。

强扩展和弱扩展是与Amdahl定律和Gustafson定律相关的另外两个概念。强扩展是指在固定问题规模的前提下，随着处理器数目增加，其性能或加速比的变化情况。而弱扩展是指在固定每个处理器的问题规模的前提下，随着处理器数目的增加，其运行时间、性能和加速比的变化情况。大致来说，强扩展对应Amdahl定律，而弱扩展对应Gustafson定律。

3.4 程序性能分析实用工具

通过分析程序性能能够知道程序的热点在哪里，进而有针对性地优化程序，可以达到事半功倍的效果。对于某个确定的热点代码段，可以使用一些工具来分析硬件计数器，以获得更详细的性能数据，进而指导深度优化。本节简单介绍Linux下的剖分工具perf、gprof和valgrind，其他比较著名且简单的有VS 2010自带的剖分器和Intel Vtune工具等。

perf、gprof和valgrind对串程序的剖分相当有用，但是并不适用于并行程序。而TAU和Vampire可用来剖分MPI并行程序，其中TAU对MPI的支持比较好，而Vampire不但支持MPI，还支持OpenMP和pthread。

1.gprof

gprof是GNU编译器工具包提供的性能分析工具，gprof能够给出函数调用关系、调用次数、执行时间等信息。大多数Linux发行版默认安装了gprof。

gprof通常和gcc/g++配合使用，gcc/g++和gprof搭配使用非常简单。在使用gcc编译程序时，添加选项-pg-g（pg表示产生的程序可以使用gprof分析），gcc/g++会自动在程序中插入一些代码以保存函数执行时的一些信息（执行时间、执行次数、调用关系等）。然后在正常运行时，程序会自动将获得的程序执行时信息输出到gmon.out文件中，再应用gprof命令处理程序和其输出的gmon.out文件，便会输出人工可阅读的信息。以使用gprof分析一个简单的x.c程序为例，使用gprof分析时，其调用工具/命令如下：

```
gcc x.c -o x -g -pg
./x
gprof x gmon.out
```

gprof输出的信息包含每个函数被调用次数、每次运行的时间以及函数之间的调用关系。下面是笔者某次使用gprof分析分子动力学软件GROMACS产生的时间信息：

```
Each sample counts as 0.01 seconds.
%   cumulative   self           self         total
time  seconds    seconds    calls   s/call   s/call   name
21.40    31.83    31.83           11     1.93    2.98  nb_kernel010_x86_64_sse
14.30    53.09    21.26           11     1.93    2.98  nsgrid_core
5.07     60.63     7.54  35996400    0.00    0.00  nb_kernel1330
4.52     67.35     6.72          101    0.07    0.08  do_update_md
4.05     73.37     6.02  252823200    0.00    0.00  invsqrt
3.93     79.21     5.84  4950173     0.00    0.00  put_in_list
3.42     84.29     5.08          101    0.05    0.15  angles
2.88     88.57     4.28  72114000    0.00    0.00  cos_angle
2.64     92.49     3.92  35996400    0.00    0.00  do_dih_fup
2.47     96.17     3.68          101    0.04    0.15  do_listed_vdw_q
2.21     99.45     3.28  288456000    0.00    0.00  rvec_sub
2.14    102.64     3.19          101    0.03    0.07  bonds
2.04    105.67     3.03  252338400    0.00    0.00  rvec_inc
```

从中可以看出，gprof不但会列出各个函数耗时比例，还会列出调用次数和自身耗时。另外，gprof同样

会列出函数间的调用关系。

需要注意的是：gprof通过在编译时插入代码来分析程序，因此在一些情况下，其给出的结果会有不准确的地方。

为了简便起见，本节只简单展示了一下gprof的使用和功能，想了解更多功能的读者可参考gprof手册。

在大多数情况下，gprof适合粗略分析代码的性能瓶颈。程序在硬件平台上实际执行时的详细硬件计数器数据可以通过perf或Intel Vtune获得。

2.valgrind

valgrind原本用来查找程序的内存泄露问题，现在也支持剖分程序。比gprof更方便的是它不用重新编译程序，可直接分析可执行文件，当然编译时增加-g选项能够提供更多的信息。

使用运行以下命令程序，运行完便可得到剖分信息。

```
valgrind --tool=cachegrind 程序名 程序参数
```

然后使用可视化的工具kcachegrind便可看到分析结果。总体来说，valgrind的结果没有grpof可读性好。

3.nvprof

nvprof是NVIDIA开发的、用于分析运行在其GPU上的CUDA程序性能的工具。nvprof目前只支持对运行在NVIDIA GPU上的内核的分析，但是其特供了扩展工具函数来简单分析在CPU上运行的函数。

nvprof不但可以统计各个内核和CUDA函数的运行时间，还可以给出硬件计数器的值，比如某个SM上一共发出了多少指令，核心一共发射和执行了多少条指令。

使用nvprof非常简单，通常只需要在cuda程序前面加上nvprof即可，如下所示：

```
nvprof ./myprog
```

图3-1是一个由笔者开发的、基于nvprof输出的硬件计数器数据、分析GPU程序性能的VBA程序的一部分。从这个图中读者可以领略nvprof的强大功能。

inst_issued_1	1018503709
inst_issued_2	125729590
inst_executed	1055186560
inst_reply_overhead	16.91201616
active_warps	13585983802
active_cycles	424579363
elapsed_clocks	424817236
sm_efficiency	99.9440058
IPC	2.483860048
branch	530572
divergent_branch	
branch_effeciency	100

图3-1 基于nvprof的输出设计的一个工具

nvprof能够获得GPU指令发射器发射的指令数量，inst_issued_1表示GPU指令发射器一次发射一条指令的记数；inst_issued_2表示GPU指令发射器一次发射2条指令的记数（笔者当时使用的GPU支持每周期发射2条指令），故GPU指令发射器发射的指令总数量为： $2 \times \text{inst_issued_2} + \text{inst_issued_1}$ 。inst_executed表示GPU执行单元真正执行的指令数量的记数。因为硬件结构的原因（如全局存储器不合并访问），一些指令需要发射多次，故GPU发射的指令数量记数会大于GPU执行的指令数量记数，大于的比例大小笔者使用inst_reply_overhead表示。

4.vampire和vampireTrace

vampire的含义是“吸血鬼”，但是vampire程序的目的却是消除并行程序中的性能“吸血鬼”。vampireTrace是一个基于命令行的并行程序剖分工具，而vampire能够图形化地显示vampireTrace的结果。

vampireTrace支持MPI、OpenMP和pthread，另外也支持nvidia的基于GPU的并行环境CUDA，它能够在Linux、Windows上运行。OpenMPI已经自带vampireTrace工具。关于vampire的详细信息可参考<http://www.vampir.eu>。

图3-2是一张使用vampire显示vampireTrace的剖分结果图。

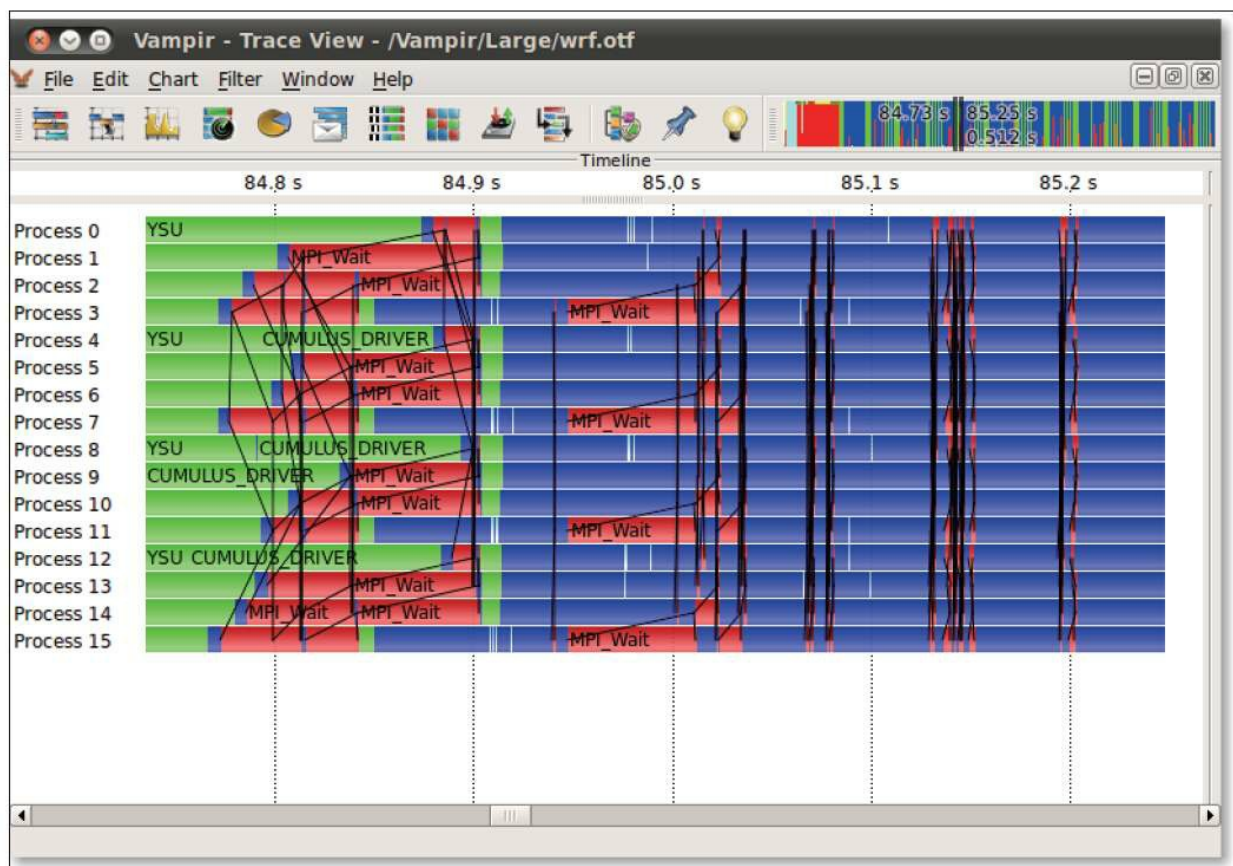


图3-2 vampire 效果

可以看出：不同进程的执行流程，进程什么时候在等待（MPI_Wait）。vampire和vampireTrace能够指导我们去掉进程级并行的性能瓶颈。

5.Intel VTune

Intel VTune工具能够分析程序的性能瓶颈、函数的调用关系、函数和语句的执行时间、每条高级语言代码对应的汇编代码，以及并发性和锁的消耗。

VTune能够分析程序在特定硬件架构上的带宽、端口使用、处理器前端和后端的使用情况。

由于Vtune是英特尔编译器套装的一部分，故笔者不打算介绍更多内容。

6.perf

perf是Linux内核2.6.31之后支持的、用户空间的（Linux内核分用户空间和内核空间，用户只能访问用户空间的数据、工具）、基于命令行的性能分析工具。perf提供了一系列子命令。perf能够统计剖分整个

Linux系统。

perf支持硬件和软件计数器，并且能够像strace一样跟踪内核调用。从Linux内核3.14开始，perf还支持功耗剖分。

perf支持下列子命令：

- stat：分析软件或硬件计数器。
- top：类似Linux top，可动态地观测热点函数。
- record：检测 and 采样一个程序的性能数据，并保存到文件中。
- report：分析perf record产生的文件，能够生成文本或图形分析。
- annotate：给代码或汇编加性能注释。
- sched：跟踪或分析调度行为和延迟。
- list：列出可用的事件。

下面是笔者使用perf分析一个程序所得到的输出（已经去掉了注释和不相关的内容）：

```
1,117,437,312 cycles
381,225,375 instructions
94,404,205 cache-references
1,696,035 cache-misses
37,909,352 branches
110,780 branch-misses
0 ref-cycles
525.486168 cpu-clock
525.516168 task-clock
2,042 page-faults
24 context-switches
0 cpu-migrations
2,042 minor-faults
0 major-faults
0 alignment-faults
0 emulation-faults
59,642,255 L1-dcache-loads
33,040,054 L1-dcache-stores
2,808,488 LLC-loads
2,733,084 LLC-stores
0.536186500 seconds time elapsed
```

perf给出的是原始的计数器数据，通过计算才能得到一级缓存带宽、缓存命中率、分支预测失败率和IPC等。

perf目前还不支持对代码文件中的某个指定函数进行分析，故要分析某个函数的性能时需要编写驱动函数。

3.5 本章小结

本章简单介绍了算法分析和程序分析的性能度量标准，以及如何度量一段代码优化后对整个程序的影响，最后本章简略介绍了一些常见的性能分析工具。

常见的衡量算法性能的标准有：时间复杂度、空间复杂度和实现复杂度。本章提出的实现复杂度可分为：计算复杂度、访存复杂度和指令复杂度。计算复杂度考虑了程序中不同的计算对性能的影响；而访存复杂度考虑了访存对性能的影响，这是时间复杂度和空间复杂度没有考虑的；而指令复杂度则考虑了不同指令在处理器上如何执行对性能的影响。

常见的用来衡量程序性能的标准有：时间、延迟、吞吐量、CPI、IPC和FLOPS等。

用来衡量优化一部分代码对程序整体性能的影响的主要有：Amdahl定律和Gustafson定律。Amdahl定律指出了在固定问题规模的前提下，增加处理器数量对程序整体性能的影响。而Gustafson定律指出了在固定每个处理器问题规模的前提下，增加处理器数量对程序整体性能的影响。

常见的用于程序性能分析的工具：gprof、valgrind、nvprof和perf等。不同的工具有不同的特点和限制。

在介绍完如何衡量算法和程序的性能后，下章将介绍如何优化串行代码。串行代码优化是并行化、向量化代码优化的基础。

第4章 串行代码性能优化

串行代码性能优化与并行代码优化同等重要，甚至更重要：一方面因为并行单独获得的加速通常有限（一般不大于处理器核心的数目），而串行代码优化有时能获得成千上万倍的加速；另一方面因为单个并行控制流的内部依旧是串行的，在这种情况下，一些串行优化措施依旧有效。

一般而言，不同算法上的优化是最有效的，要选择性能最优的算法并实现，但本章并不涉及。本章假设你已经有了一个可以运行并能够得到正确结果的程序，需要在此基础上进行优化。

本章将串行代码的优化分为以下几个层次。

- 系统级别：要求找出程序的性能控制因素以做针对性的优化。
- 应用级别：在程序编写前就要确定应用级别的配置，应用级别的优化相对比较简单，且实现后性能比较稳定。
- 算法级别：选择不同的数据组织方式，或者选择不同的算法，这两者对性能的作用非常大。
- 函数级别：函数级别的优化通常用来减少函数调用的开销或者减少函数调用带来的编译器性能优化阻碍。
- 循环级别：由于执行次数多，循环更易成为性能瓶颈，通常循环级别的优化能够发掘循环的并行性，减少循环内多余的运算。
- 语句级别：不同的语句产生的指令数量并不相同，应尽量使用产生指令条数少的语句。
- 指令级别：不同指令的吞吐量和延迟并不相同，应优先使用吞吐量大、延迟小的指令。

实际上以上分类可能并不合适，因为很多优化方法可以在多个层次上起作用，这需要读者融会贯通。

顶级的代码优化人员会以分析为导向，以程序热点为目标，以知识为后盾，一次一个目标，步步为营地把代码的性能做到极致。

本章所展示的技术是许多高级研发人员的“不传之秘”，如果读者理解了这些“不传之秘”就能够编写出比普通开发人员性能高的代码。

4.1 系统级别

优化程序的性能，首先需要在系统级别找出程序的性能控制因素，然后再做有针对性的优化。如果限制因素是处理器的计算能力，那么优化内存的访问就不会产生大的效果。通常在程序运行过程中，如果处理器的利用率一直是100%，那么限制因素就是处理器，需要减少计算。而如果处理器使用率不高但很平稳，此时就需要测试程序的内存带宽，比较程序的带宽和内存能够提供的带宽（指内存的有效带宽，而不是理论带宽，可以使用stream工具），以估计程序还有多少优化余地，此时优化应注意减少存储器访问。如果处理器利用率一会儿高一会儿低，就需要查找是什么因素导致了处理器长时间空闲，常见的因素是有其他进程在占用处理器。

如果应用通过网络互连交换数据或指令，那么网络速度、利用率及网络负载均衡也要考虑。

1.网络速度、利用率及负载均衡

如果应用经常等待网络数据的传输和到达，那么就得考虑网络的速度和利用率，如果是集群还得考虑网络负载均衡。如果网络的利用率很高，但是数据传输还是很慢，那么就得查看是否网线或网卡的带宽不够，另外网络拓扑或路由设置不好，导致数据传输经过的节点过多也有可能。如果网络中某些节点的负载特别不均衡，那么就可能需要更改网络拓扑结构或者为那几个节点选用更好的网线或网卡。

2.处理器利用率

Linux下的top命令可以查看计算机各个核心的负载。图4-1是笔者在某台工作站上运行top命令后输入1得到的输出：


```

top - 20:31:48 up 13 days, 16 min, 1 user, load average: 0.00, 0.01, 0.05
Tasks: 364 total, 1 running, 350 sleeping, 13 stopped, 0 zombie
Cpu0 : 0.0%us, 0.3%sy, 0.0%ni, 99.7%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu1 : 0.0%us, 0.0%sy, 0.0%ni, 100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu2 : 0.0%us, 0.0%sy, 0.0%ni, 100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu3 : 0.0%us, 0.0%sy, 0.0%ni, 100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu4 : 0.0%us, 0.0%sy, 0.0%ni, 100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu5 : 0.0%us, 0.0%sy, 0.0%ni, 100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu6 : 0.0%us, 0.0%sy, 0.0%ni, 100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu7 : 0.0%us, 0.3%sy, 0.0%ni, 99.7%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu8 : 0.0%us, 0.0%sy, 0.0%ni, 100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu9 : 0.0%us, 0.0%sy, 0.0%ni, 100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu10 : 0.0%us, 0.0%sy, 0.0%ni, 100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu11 : 0.0%us, 0.0%sy, 0.0%ni, 100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu12 : 0.3%us, 0.0%sy, 0.0%ni, 99.7%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu13 : 0.0%us, 0.0%sy, 0.0%ni, 100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu14 : 0.0%us, 0.0%sy, 0.0%ni, 100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu15 : 0.0%us, 0.0%sy, 0.0%ni, 100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu16 : 0.0%us, 0.0%sy, 0.0%ni, 100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu17 : 0.0%us, 0.0%sy, 0.0%ni, 100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu18 : 0.0%us, 0.0%sy, 0.0%ni, 100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu19 : 0.0%us, 0.0%sy, 0.0%ni, 100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu20 : 0.0%us, 0.0%sy, 0.0%ni, 100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu21 : 0.0%us, 0.0%sy, 0.0%ni, 100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu22 : 0.0%us, 0.0%sy, 0.0%ni, 99.7%id, 0.0%wa, 0.0%hi, 0.3%si, 0.0%st
Cpu23 : 0.0%us, 0.0%sy, 0.0%ni, 100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu24 : 0.0%us, 0.0%sy, 0.0%ni, 100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu25 : 0.0%us, 0.0%sy, 0.0%ni, 100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu26 : 0.0%us, 0.0%sy, 0.0%ni, 100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu27 : 0.0%us, 0.0%sy, 0.0%ni, 100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu28 : 0.0%us, 0.0%sy, 0.0%ni, 100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu29 : 0.0%us, 0.0%sy, 0.0%ni, 100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu30 : 0.0%us, 0.0%sy, 0.0%ni, 100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu31 : 0.3%us, 0.3%sy, 0.0%ni, 99.3%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu32 : 0.0%us, 0.0%sy, 0.0%ni, 100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu33 : 0.0%us, 0.0%sy, 0.0%ni, 100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu34 : 0.0%us, 0.0%sy, 0.0%ni, 100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu35 : 0.0%us, 0.0%sy, 0.0%ni, 100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu36 : 0.0%us, 0.0%sy, 0.0%ni, 100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu37 : 0.0%us, 0.0%sy, 0.0%ni, 100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu38 : 0.0%us, 0.0%sy, 0.0%ni, 100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu39 : 0.0%us, 0.0%sy, 0.0%ni, 100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Mem: 264121220k total, 261194424k used, 2926796k free, 504576k buffers
Swap: 0k total, 0k used, 0k free, 249200460k cached

```

图4-1 top示例

其中第二行的Tasks表示系统目前总共有多少进程、有多少进程正在执行、有多少进程在休眠、有多少进程已经结束、有多少进程是僵尸进程。

中间的各列表示各个核心上的占用情况：第一列表示逻辑核心（一个支持超线程的物理核心表示为两个逻辑核心）编号；第二列表示用户空间对逻辑核心的占用率；第三列表示系统空间对逻辑核心的占用率。

Mem开头的行表示系统内存信息。如图4-1所示，系统总共有256GB内存，其中绝大多数正在被使用。

处理器的利用率在两种情况下可能存在性能问题：

- 1) 处理器空闲较多：如果处理器大部分时间空闲，很明显程序不是最优的，因为程序无法发挥硬件的

计算能力。

2) 处理器利用率过高：如果核心的使用率为100%，或者单核程序中某个核的使用率为100%，这就意味着处理器成为了性能限制因素，通常也称为“计算限制”。对于大多数优化很好的程序来说，计算限制可能意味着优化已经到达极限；对其他的一些程序来说，可能意味着开发人员没有合适的编写代码，导致计算量增加。

依据处理器利用率优化程序的常见方法是：找出为什么处理器利用率比较低或比较高。比如，处理器空间较多可能是由于编程限制在访存上，或者处理器经常在等待数据到达、或其他进程或线程释放共享资源等。

优化计算限制的程序的性能并不意味着要降低处理器利用率，一般主要办法有减少计算数量、表达式移除、分支优化等，有时也可以优化算法。

3.存储器带宽利用率

如果处理器的利用率不高，这意味着存储器系统不能提供足够的带宽以满足处理器的需求。此时需要先测试存储器系统的可用带宽（可使用stream工具或自己编写代码测试）、程序的实际带宽（可使用硬件计数器的数据计算得到）和程序的有效带宽（从算法分析得到），如果程序的有效带宽远小于存储器系统的可用带宽，那么就存在优化的可能。

优化存储器访问性能的方法主要有：

1) 提高存储器访问的局部性以增加缓存利用。常见的有二维数据的行列访问顺序问题，对于C语言中的二维数组而言，要访问行，而对于Fortran语言中的二维数组而言，要访问列。

2) 将数据保存在临时变量中以减少存储器读写。由于临时变量通常占用空间更小，硬件能够将它们缓存到一级缓存，或编译器能够将它们分配到寄存器中，避免了访问更慢的存储器。

3) 减少读写依赖。读写依赖会导致流水线指令等待，减弱了流水线效率。

4) 同时读写多个数据以提高系统的指令级并行和向量化能力。

4.去阻塞处理器运算的因素

如果处理器利用率一会儿高一会儿低，通常这是因为一些因素阻塞了处理器运算，比如IO操作（如读文件）、其他用户的进程占用了处理器或内存带宽。可以通过观察处理器的使用率（或测试IO操作占用的

时间)来估计处理器有多少比例的运算时间是在等待IO操作完成,如果两者差别非常大就没有优化的必要了。如果IO操作和处理器计算时间比较接近,那么优化最多可使性能提升2倍。通常的解决方法是使用非阻塞的函数调用,如Linux或glibc的异步IO操作(感兴趣的读者可参考glibc文档和Linux手册),或使用独立的线程来处理IO操作(或计算)以使得IO和计算同时进行。

4.2 应用级别

应用级别的优化通常应当最先采用，一方面因为它可能关系到程序的各个部分，另一方面也因为软件开发越到后面，越不容易采用应用级别的优化。

1. 编译器选项

这是最简单的一种优化方法，一般而言使用高级的优化选项总比使用低优化选项的速度要快一些，如GCC就有O0（编译器不对程序做优化）、O1、O2、O3（编译器会为了性能做极致的优化）等优化选项。还有很多其他选项，如指定处理器架构、是否使用循环展开、是否使用SSE指令等，绝大多数程序使用O2优化选项。注意，有时因为程序编写的问题，O3优化选项会更慢，甚至产生错误的结果（大多数情况下这是由于程序编写的原因导致的）。

使用GCC时，建议使用如下编译选项：

```
-O3 -ffast-math -funroll-all-loops -mavx -mtune=native
```

其中：**fast-math**表示对超越函数使用更快但精度低一些的版本；**unroll-all-loops**表示使用循环展开；**avx**表示使用**avx**指令集向量化；**tune=native**表示为当前编译的处理器做优化。

2. 调用高性能库

直接调用高性能库，如优化的BLAS（Basic Linear Algebra Subprograms，基本线性代数子程序）和快速傅立叶变换自由软件库FFTW（Fastest Fourier Transform）等，可以减少大量工作。一方面因为它们通常已经比较成熟，另一方面也因为编写库代码的开发人员素质通常较高，性能和质量都比较有保证。

3. 去掉全局变量

全局变量尤其是多个文件共享的全局数据结构会阻碍编译器的优化，因为编译器需要在多个文件间分析其使用状态。为了保证不至于产生错误的结果，编译器就变得极为保守，对于全局变量优化就变得非常复杂，像移除子表达式、合并某些操作的结果这些常见的优化方式，编译器也必须小心翼翼。

比如类似代码清单4-1的代码，一些编译器可能就无法优化**g**函数，因为**f**函数修改了全局变量**x**。

代码清单4-1 全局变量阻碍优化示例

```
//global var
```

```
int x = 0;
int f(){
    return x++;
}
int g(){
    return f() + f();
}
```

另外，全局变量的使用也使得程序员不便追踪其变化，难以进行手工优化。假设例子中的x声明在另外一个头文件中，而被多个函数引用的话，谁又能保证在某个函数中对它的优化不会影响其他函数呢。

对于并行程序来说，全局变量（除非是不可变的）应当是绝对禁止的，因为多个控制流需要协调对全局变量的更改，如何保证全局变量在各个控制流的状态是一致的，这是并行编程的难点之一，而解决这个难点的最好策略是回避它。

4.受限的指针

C99为了减少存储器别名对性能的影响而引入了“受限的指针”这一特性，受限的指令表示该指针不会和其他指针存在存储器别名。存储器别名是指多个指针指向同一个内存地址或指向的内存地址有重叠，它会阻碍编译器对程序进行指令重排、表达式移除等优化。如下所示：

```
int f(int *a, int *b){
    *a += *b;
    *a += *b;
}
```

下面分别介绍上面代码可能存在的有存储器别名和没有存储器别名两种情况：

第1种：如果指针a、b指向不同的内存地址，也就是说a、b之间不存在存储器别名，那么函数f可简化为：

```
int f(int *a, int *b){
    int temp = *b;
    *a += 2*temp;
}
```

第2种：如果指针a、b指向同一个内存地址，即a、b之间存在存储器别名，那么函数f将变成0：

```
int f(int *a, int *b){
    int temp = *b;
    temp += temp;
    temp += temp;
    *a = temp;
}
```

进一步简化就变成了如下所示的代码：

```
int f(int *a, int *b){
    int temp = *b;
    *a = 4*temp;
}
```

因此在a、b之间存在存储器别名的情况下，第1种情况的结果是错误的。虽然大多数情况下，开发人员的意图是0，但是编译器并没有额外的信息来做出这个结论，因此保守起见可能选择不加以优化，而受限的指针可以解决这个问题。

C99标准对受限的指针使用的标识符是restrict，但是它没有被C++标准接受，不过G++编译器支持__restrict__限定符，其含义和restrict一样。

使用受限的指针能够允许编译器做更多的优化，潜在地提升性能。但是调用者自己要保证指针之间不存在存储器别名，而在一些复杂的程序中要保证不存在存储器别名是非常困难的。

5.条件编译

通常使用条件编译以满足可移植性。相比运行时检测，条件编译生成的代码更短，因此效率更高。C语言提供了#ifdef#else/elif#endif/ifndef等指令来支持条件编译。

如代码清单4-2和代码清单4-3所示，它们展示了条件编译和运行时分支判别的区别。由于宏条件在编译时就已经确定，故编译器可直接忽略不成立的分支；而分支判别的条件是在运行时求值的，故编译后的代码要长。不过如果要支持多个条件编译分支版本，条件编译的方式只能使用多个程序，而运行时分支版本却只需要一个。

代码清单4-2 switch条件分支

```
switch(mode){
    case ON_ARM_CPU:
        arm_f(); break;
    case ON_X86_CPU:
        x86_f(); break;
    default:
        /*...*/
}
```

代码清单4-3 条件编译

```
#ifdef ON_ARM_CPU
    arm_f();
#elif ON_X86_CPU
    x86_f();
#endif
```

由于可以在编译时指定宏ON_ARM_CPU或ON_X86_CPU，代码清单4-3生成的指令更短，对指令缓存等的利用也会更好。

4.3 算法级别

算法级别的优化通常涉及程序的一个部分，而这部分包含一个或多个函数，甚至只有一段语句。算法级别的优化主要涉及算法实现时要考虑的问题。通常算法要考虑数据的组织、算法实现的策略。缓存利用和数据的组织密切相关，软件预取能够隐藏访存延迟，而查表是一种有效的算法级优化。

缓存优化

发掘程序访存的局部性以合理地发挥缓存的带宽，通常这要求在设计算法时就要考虑数据结构。

其他一些优化缓存的方法，比如多维数组的分块等。缓存的优化，往往对程序的性能有成倍或数量级的改变。

1.索引顺序

访问多维数据时的局部性直接与各维数据在内存中存放的先后顺序相关，如C语言中数组是以行为主序存放的，在计算时尽量按行访问数据。而对于Fortran代码，则应当按列访问数据。

优化后的代码性能就应当比优化前的要好。

优化前的代码：

```
for(int i = 0; i < N; i++){
    for(int j = 0; j < M; j++){
        r[j] += a[j][i];
    }
}
```

优化后的代码：

```
for(int i = 0; i < M; i++){
    float ret = 0.0f;
    for(int j = 0; j < N; j++){
        ret += a[i][j];
    }
    r[i] = ret;
}
```

对优化前代码来说，在内层循环上，相邻循环体内访问a的地址相隔N个元素，在N比较大的情况下，可能会存在满不命中和冲突不命中的情况，故访问a的局部性很差；对优化后的代码来说，在内层循环上，相邻循环体内访问a的地址相隔1个元素，加载到缓存线中的数据会在随后使用，从这一点来说，访问a的局部性非常好。

理想状况下，编译器应当能够自己做这种代码形式变换，但是如果循环内代码很复杂，则再优秀的编译器可能也无能为力。

2.缓存分块

现代处理器都有多个层次的缓存，如果数据的大小超过了缓存的大小，那么就容易出现满不命中的情况，此时常见的减弱满不命中的代价的方法主要是缓存分块方法。

对于多维数组来说，在某一维上访问的局部性很好，并不代表着在其他维上的访问局部性也很好，因为缓存大小有限。以二维数据 $a[M][N]$ 为例，如果 N 比较大的话，那么在列方向缓存的数据便会不命中。

通常以核心间不共享的最低层缓存分块，这意味着为二级缓存分块，但是这也不绝对，在某些情况下，也许为L1分块更好，在某些情况下，可能会同时为多个缓存层次进行分块优化。但是在多线程的情况下，多个核心访问的数据都会被共享的缓存层次所缓存，多个线程会竞争共享缓存的带宽，故通常不会为共享的缓存分块。

以矩阵乘法为例，简单的基于缓存分块的代码如代码清单4-4所示：

代码清单4-4 基于缓存分块的示例

```
for(i=0; i<N; i+=NB)
    for(j=0; j<M; j+=NB)
        for(k=0; k<K; k+=NB)
            for(i0=i; i0<(i + NB); i0++)
                for(j0=j; j0<(j + NB); j0++)
                    for(k0=k; k0<(k + NB); k0++) {
                        C[i0][j0] += A[i0][k0] + B[k0][j0];
                    }
```

3.软件预取

预取是指在数据被使用之前，投机地将其加载到缓存中。预取通常有硬件和软件两种方式，软件预取指由编译器或软件开发人员将预取指令插入到代码的适当地方。指令集通常会提供预取指令供编译器优化时使用。编译器分析代码，并把预取指令适当地插入代码中。这类指令直接把目标预取数据载入缓存。在一些特殊情况下，为了隐藏访问内存的延迟，软件开发人员也会手动在代码中插入编译器厂商提供的预取内置函数。

在使用预取技术时，必须妥善考虑进行的时机和实施强度。如果过早地进行预取，预取的数据有可能在用之前就已经因为冲突置换、满不命中而被清除；通常会预取2到4个缓存线长度的数据，即在处理地址 x 的数据的同时，预取地址 $x+256$ 上的数据。如果预取得太多或太频繁，则预取数据可能会占用过多缓存，进而导致满不命中。循环内访问的数据越多，那么预取的数据量就应当越少。

4.查表法

查表法是这样一类方法：这类方法提前把数据组织成表格（一维、二维或多维），在真正需要使用数据时，只需要访问表格即可。

在一些应用中，查表法非常有用。查表法通常伴随着预先计算。比如有一批数据，要知道每个数据出现的次数，就没有必要每次都去统计，只要统计一次，然后记录结果，在每次查的时候，只要查记录的结果就行了。再比如查找，在一个有序系列中查找数据可以使用二分查找，这远比查找无序数据使用的线性查找要快，可以将待查找的数据排序，然后再二分查找（这要求数据内容不会变化或者很多次查找后数据才会变化，这样排序的代价会被查找的收益掩盖掉）。

在一些科学计算中，需要多次计算一些复杂函数的结果，此时也可利用查表法减少计算。比如在某个应用中需要计算十万次 $\exp(x)$ ，其中 x 范围为 $[0, 1]$ ，那么可查区间 $[0, 1]$ 不均匀分成1000份，靠近1时密，靠近0时稀，然后保存这1000个 $\exp(x)$ 的值来做查表。如果不采用查表法，那么需要计算十万次 \exp ，使用查表法后，计算 \exp 的次数减少为1000，不过在此例中使用查表法会导致结果精度的降低。

如代码清单4-5所示，使用了查表法计算 \exp 值，只是为了简单，对区间采用了均匀划分。

代码清单4-5 用查表法计算 \exp 值

```
void buildTable(int num, float *table){
    table[0] = 1.0f;
    for(int i = 0; i < num; i++){
        table[i] = i*1.0f/num;
    }
}

float computeExp(float x, int num, const float *table){
    int index = (int)(x*num);
    return table[index];
}
```

在需要大量计算 \exp 值的程序中，代码清单4-5所示的查表法能够极大地减少计算数量。它将不固定的 \exp 计算转化成了固定数量（ num ）的 \exp 计算和不固定数量的访存。

在实际项目中，通常将查表法和线性插值结合使用以减少计算精度的降低：对于每个要求值的元素，其返回值为多个结果（一维为2个：称为线性插值；二维为4个：称为双线性插值；三维为8个：称为三线插值）的线性插值。以一维查表为例，索引1.5的返回值等于索引1的返回值和索引2的返回值的和的一半；索引1.3的返回值等于索引1的返回值和0.7的积与索引2的返回值和0.3的积的和。

由于计算机图像学中需要对大量的像素做插值运算，故GPU的纹理设计时就支持硬件插值运算，在CUDA和OpenCL程序中可利用这一点加载某些运算。

4.4 函数级别

函数调用时，需要将调用参数通过寄存器或栈传递，且将函数返回地址入栈。函数级别的优化通常用于减少这部分的消耗及其导致的优化障碍。

建议函数只访问自己的局部变量和通过参数传入的值，这样编译器能够只依据函数内的信息就可以分析变量使用情况。

4.4.1 函数调用参数

在64位X86处理器上，函数的参数优先通过寄存器传递，超量之后才会通过栈传递。另外，如果一个函数的参数少，那么这个函数就更容易被记住和使用。

如果函数的参数是大结构体或类，应当通过传指针或引用以减少调用时复制和返回时销毁的开销，因为函数可能只会使用大的结构体中的一部分域。如代码清单4-6中函数getYByPtr就比getYByValue要好。

代码清单4-6 函数调用参数优化示例

```
struct BigStruct{
    int x[30];
    float y;
    float z;
}
float getYByValue(BigStruct bs){
    return bs.y;
}
float getYByPtr(const BigStruct *bs){
    return bs->y;
}
```

在调用getYByValue函数前，处理器会将结构体bs入栈，这意味着要复制128字节；而调用getYByPtr函数前，处理器只需要入栈一个指针。实际上，解决getYByValue函数调用时的入栈开销的一个有效办法是：内联函数。

如果函数使用了全局变量的话，那么就有可能阻止编译器优化（如某个函数更改了全局变量的值，那么编译器就难以将多次调用转化为一次调用）。因此要尽量不使用全局变量，就算要使用全局变量，也要通过参数传递。4.2.3节给出了一些使用全局变量阻碍优化的例子。

4.4.2 内联小函数

内联（inline）小函数能够消除函数调用的开销，且可能会提供更多的指令级并行、表达式移除等优化机会，进而增加指令流水线的性能。另外函数调用也可能阻止编译器优化，如果循环内有函数调用，则编译器很难进行向量化。对于这些情况，内联可以解决问题。

如果内联后的函数比较长，就可能会增加寄存器的压力。如果函数内有分支，内联后会对指令流水线产生不利的影响，因此通常对少于10行且其中代码没有分支的函数内联。

4.5 循环级别

循环如果执行次数多，更易成为性能瓶颈。通常循环级别的优化以发掘循环的并行性、减少寄存器和缓存的使用为主，以更好地利用硬件的资源。

4.5.1 循环展开

展开循环不但减少了每次的判断数量和循环变量改变的计算次数，更能够增加处理器流水线执行的性能。通常展开小循环且内部没有判断的会获益，展开大循环则可能会因为导致寄存器溢出而导致性能下降，而展开内部有判断的循环会增加分支预测的开销也可能导致性能下降。

如代码清单4-7用来给15000个数求和，代码清单4-8是循环展开4次的版本。

代码清单4-7 展开循环前

```
float sum = 0.0f;
for(int i = 0; i < num; i++){
    sum += a[i];
}
```

代码清单4-8 展开循环后

```
float sum = 0.0f, sum1 = 0.0f, sum2 = 0.0f, sum3 = 0.0f;
for(int i = 0; i < num; i += 4){
    sum1 += a[i];
    sum2 += a[i+1];
    sum3 += a[i+2];
    sum += a[i+3];
}
sum += sum1+sum2+sum3;
```

对于二层循环来说，通常建议优先展开外层循环，但是这不是一个普适的准则。

循环展开时需要注意处理末尾的数据，如代码清单4-7中的num大小不是4的倍数，则最后num%4次循环需要单独处理。

许多编译器可以自动展开循环，也有些编译器提供了控制循环展开的伪指令，开发人员可优先选择它们，在必要时再采取手动展开。

4.5.2 循环累积

循环累积主要和循环展开同时使用，在减少寄存器的使用量的同时保证平行度。如代码清单4-8的例子，循环展开6次、累积2次后代码如代码清单4-9所示：

代码清单4-9 循环累积

```
float sum = 0.0f, sum1 = 0.0f, sum2=0.0f;
for(int i = 0; i < num; i +=6){
    sum1 += a[i]+a[i+1]
    sum += a[i+2]+a[i+3];
    sum2 += a[i+4]+a[i+5];
}
sum += sum1+sum2;
```

如果直接使用循环展开6次，则总共需要至少6个临时变量，而现在只要3个，潜在地减少了寄存器的使用。在寄存器总量一定的前提下，减少寄存器的使用，就可以将循环展开更多次。

笔者在现代X86处理器上测试发现，累积的效果越来越不明显。

4.5.3 循环合并

如果多个小循环使用的寄存器数量没有超过处理器的限制，那么合并这几个小循环可能会带来性能好处（增加了乱序执行能力）。循环合并不但可以减少判断次数，还能够增加指令级并行能力（能够合并的两个循环内代码通常没有依赖）。代码清单4-10中的代码可合并为代码清单4-11所示代码。

代码清单4-10 循环合并前

```
for(int i = 0; i < len; i++){
    x1 += a[i];
}
for(int i = 0; i < len; i++){
    x2 *= b[i];
}
```

代码清单4-11 循环合并后

```
for(int i = 0; i < len; i++){
    x1 += a[i];
    x2 *= b[i];
}
```

循环合并的另一个常用场景是并行化，由于能够增加每个控制流的计算量，因此将多个循环合并再并行化能够更好地隐藏线程建立开销，获得更好的效果。

4.5.4 循环拆分

如果大循环（循环体内代码比较多、执行时间长）存在寄存器溢出的情况，那么将大循环拆分为几个小循环，就能够提高寄存器的使用，而且能够为循环展开等技术的使用提供条件。

循环拆分和循环合并是相对的技术，只是它们应用的对象类型不同。何时进行循环拆分必须就事论事，不能一概而论，通常与处理器的缓存容量、寄存器数量等相关。代码清单4-12的代码就可以拆分为代码清单4-13所示的代码。

循环拆分要求被拆分的代码之间没有依赖，如果有依赖的话，则不能拆分。循环拆分的一个简单规则是：尽量使得拆分后的循环在使用资源上面不出现重复，即尽量不要因为拆分循环导致访存或计算增加。

代码清单4-12 循环拆分前

```
for(int i = 0; i < len; i++){  
    doA();  
    doB();  
}
```

代码清单4-13 循环拆分后

```
for(int i = 0; i < len; i++){  
    doA();  
}  
for(int i = 0; i < len; i++){  
    doB();  
}
```

从表面上看，循环拆分增加了循环条件的执行次数，如果循环条件的计算量相比循环内代码计算量小的话（即大循环），那么循环拆分所导致的循环条件执行代价即可忽略。

4.6 语句级别

实现同一功能的不同语句系列编译后的指令数量不同、指令类型也不同，必然导致性能也不相同。对于实现同样功能的不同指令系列，指令系列中指令的延迟和吞吐量可能也不相同，故性能也不相同。对于语句级别的优化来说，需要尽量避免语句生成不需要的指令（生成更少数量的指令），或者让语句生成更加高效的指令（生成更快的指令）。从本质上来说，语句级别的优化和指令级别的优化比较相似，因为两者的粒度都非常小，而且没有本质的区别（从指令级别分析语句的结果就是指令级别的优化）。

4.6.1 减少内存读写

对于现代的DDR3内存来说，通常一次内存读写大约要200~400个时钟周期，相比之下处理器一个周期可完成多个浮点乘加计算，访问内存非常慢，因此应当优先使用数值（重用数据）而不是解引用。如果需要多次访问函数参数指针指向的值，应当先将其保存到寄存器中。

大多数情况下，编译器能够很好地解决这个问题，但是在具有存储器别名情况或读写依赖情况下，就需要开发人员手动处理。代码清单4-14所示为求前缀和代码，由于后一次循环需要使用前一次循环写入内存的结果，编译器无法很好地优化。

代码清单4-14 优化前的前缀和代码

```
for(int i = 1; i < n; i++){
    a[i] += a[i-1];
}
```

而代码清单4-15所示的版本，则通过保存中间计算结果减少了一些内存访问。

代码清单4-15 优化后的前缀和代码

```
temp = a[0];
for(int i = 1; i < n; i++){
    temp += a[i];
    a[i] = temp;
}
```

在某些情况下，可以重复计算某些值而不是计算后保存到内存中再读取，此时需要权衡计算的代价和读写内存的延迟。

4.6.2 选用尽量小的数据类型

由于能够在缓存中放更多的数据，小尺寸类型数据的访问速度比大尺寸类型数据要快（指以数据的个数统计，而非传输带宽）。在使用SSE/AVX指令时，一个SSE/AVX寄存器能够存放更多的小尺寸类型数据，这也能够提升某些程序的性能。

如代码清单4-16所示的代码是将RGB图像转换成灰度图像，但是代码使用了int来保存中间结果，而由于图像像素每通道大小为1字节，权重也可以使用1字节来表示，故使用short来保存就已经足够，优化后的代码如代码清单4-17所示。

代码清单4-16 RGB彩色图像转灰度图代码

```
for( int i = 0; i < n; i++){
    int r = r_buf[i]; // load red
    int g = g_buf[i]; // load green
    int b = b_buf[i]; // load blue
    int r_ratio = 77;
    int g_ratio = 151;
    int b_ratio = 28;
    int y = r * r_ratio;
    y += g*g_ratio;
    y += (b*b_ratio);
    dest[i] = (y>>8);
}
```

代码清单4-17 优化后的RGB彩色图像转灰度图代码

```
for( int i = 0; i < n; i++){
    short int r = r_buf[i]; // load red
    short int g = g_buf[i]; // load green
    short int b = b_buf[i]; // load blue
    short int r_ratio = 77;
    short int g_ratio = 151;
    short int b_ratio = 28;
    short int y = r * r_ratio;
    y += g*g_ratio;
    y += (b*b_ratio);
    dest[i] = (y>>8);
}
```

以ARM NEON 128位向量指令集为例，其寄存器为128位，若使用int型，则每个向量只能保存4个数据，但是使用short的话，每个向量可保存8个数据，这意味着使用short时潜在的性能是使用int时的两倍。

4.6.3 结构体对齐

声明结构体时，尽量大数据类型在前，小数据类型在后，一方面这样会节省一些空间，另一方面可以更好地满足处理器的对齐要求。

不同的硬件平台和编译器对结构体对齐的要求各不相同，这要求编码时尽量使用sizeof运算符。但是几条规则是一致的：

- 结构体占用总字节数尽量是2的幂；
- 每个域的开始地址是它的大小的整数倍；
- 编译器提供了按字节对齐的编译原语，如GCC的__attribute__ ((aligned (x))) 和 __attribute__ ((packed ())) 。

如下面的两个结构体，其元素相同，但是大小不同。

```
struct s{
    char x;
    double y;
    int z;
    short w;
}
```

对结构体s中的域按照大小排序得到结构体t。

```
struct t{
    double y;
    int z;
    short w;
    char x;
}
```

处理器和编译器对齐的要求导致结构体s占用24字节（浪费了9字节），而结构体t占用16字节（浪费了1字节）。无论是在32位还是64位系统上，无论是Windows还是Linux，它们占用的字节数不会变。

本质上来说，结构体s和t可用来保存相同的数据，但是由于域声明顺序导致结构体s占用的空间是结构体t的1.5倍，这意味着访问相同数量的结构体s和t，s对存储器的带宽要求更多。

有些结构体声明使得它们在不同的硬件上或不同的操作系统上大小并不一致，基于这一点，软件开发人员要记住永远都使用sizeof运算符，而不是确切的字节数。

数据占用的字节越小，在同样大小的缓存线中存放的数据数量就越多，缓存线的利用就越好。但是轻

易不要使用编译器的packed选项将结构体的存储空间压缩到最小，因为这可能会导致不对齐的读写存储器。

4.6.4 表达式移除

表达式移除是指去掉重复的、共同的计算或访存，这能够减少计算或内存访问次数。如代码清单4-18和代码清单4-19所示，前者每次循环都需要比较索引以检查访存是否越界，而后者只比较一次即可。

代码清单4-18 每次访存都检查是否越界

```
void readVI(VI *vi, int id){
    int len = vi->size();
    if(id < len) return vi[id];
    else return ERROR;
}
for(int i = 0; i < len; i++) {
    readVI(a, i);
}
.....
}
```

代码清单4-19 只检查一次索引

```
if(len >= a->size) return ERROR;
for(int i = 0; i < len; i++) {
    a[i];
}
...
}
```

很多开发人员害怕不检查索引导致的访存越界等错误。笔者建议：在开发阶段对所有的访存都检查索引是否越界，程序都验证正确性之后，再将性能相关部分代码的索引检查代码去掉。

4.6.5 分支优化

为了提高CPU的性能，现代CPU都设计了多级流水线，有的甚至有20级以上。当CPU遇到条件指令的时候，为了不中断流水线（某些处理器流水线中断的代价大约五六十个时钟），硬件会做一个预测，把预测的分支代码载入流水线，当发现预测错误的时候，需要清空流水线，重新载入正确的分支到流水线，此时实际损失的周期数通常是流水线长度的几倍！因此分支的优化非常重要。

通常一些简单的分支语句，分支预测的效果可能比较好。而对于随机的分支模式，再好的预测器也不可能做出好的预测。要优化条件分支，通常这些分支代码应该满足：该分支是热点代码的一部分，并且分支预测错误率较高，这样才能得到好的优化收益。

本节给出一些常见的分支模式的优化方法和建议。

1. 尽量避免把判断放到循环里面

由于分支预测失误会对流水线产生非常不利的影响，因此要避免循环里面有判断语句，此时尽量改成判断里面有循环（这要求循环和判断之间没有依赖）。

代码清单4-20 循环内有分支

```
for(int i = 0; i < len; i++){
    if(x > y) dosomething();
}
```

代码清单4-20可优化为代码清单4-21，在不考虑编译器优化的前提下，代码清单4-21只需要计算一次分支，而代码清单4-20要计算len次。

代码清单4-21 分支内有循环

```
if(x > y){
    for(int i = 0; i < len; i++){
        dosomething();
    }
}
```

在理想情况下，编译器应当能够自动做这种转换，实际上对于一些条件非常简单的（如本节的示例），编译的确能够完成；但是对于一些复杂的代码（如条件参数在循环内发生更改），编译器往往无能为力。

2. 拆分循环

某些循环中分支非常多，这可能会导致处理器分支预测失败率增加，把它拆分成几个小循环有可能改善处理器的分支预测正确率；在另外一些循环代码中，循环内分支条件依赖循环变量（如代码清单4-22中的i），这种情况可通过拆分循环去掉分支，如常见的奇偶分支模式，如代码清单4-22所示：

代码清单4-22 奇偶分支模式

```
for(int i = 0; i < len; ++i){
    if (i&1 == 0) do0();
    else do1();
}
```

此时可以将循环展开两次，一次处理奇数循环，一次处理偶数循环，自然就不需要分支判断。修改后代码如代码清单4-23所示：

代码清单4-23 循环展开优化奇偶分支

```
for(int i = 0; i < len-1; i += 2){
    do0();
    do1();
}
if(0 != len%2) do0();
```

如果do0和do1使用的临时变量比较多的话，代码清单4-24的性能可能更好（因为每个循环使用的寄存器可能更少）。

代码清单4-24 循环拆分优化奇偶分支

```
for(int i = 0; i < len; i += 2){
    do0();
}
for(int i = 1; i < len; i += 2){
    do1();
}
if(0 != len%2) do0();
```

使用循环拆分去掉分支时，由于去掉了分支路径执行的先后顺序，故要求分支路径之间没有相关性；而使用循环展开去掉分支则无此要求。

3.合并多个条件

一些分支条件是包含多个比较操作的复杂表达式，编译器通常将它们编译成嵌套的多个if语句代码，如果能够将其修改成一个运算，那么只需要一个分支，就可以提高分支预测成功率。比如：

```
if((a0==0) && (a1==0) && (a2==0)){
    .....
}
```

某些编译器将生成3个条件跳转指令，而使分支可预测性降低，可以改写为：

```
int x = (a0|a1|a2);
if(0 == x){
    ...
}
```

从而同时改进代码质量和分支预测率。

合并分支条件要求在分支执行前计算出分支的结果，在某些分支条件计算量非常大的情况下（如函数调用），并不应当使用这一技术。

4.使用条件状态值生成掩码来移除条件分支

很多时候可以利用C中判断式的结果为1（真），0（假）来去掉分支。比如下面的代码：

```
if(a > 0){
    x = a;
}else{
    x = b;
}
```

可优化为：

```
x = a > 0;
a = a*x + b*(1-x);
```

由于增加了计算量，在分支预测失败概率很小或编译器能够将其优化为条件复制指令的情况下，可能会导致程序性能更差。因此此优化应当只在分支预测失败率比较高的情况下使用。

5.使用条件复制指令以移除分支

在64位的X86 CPU上，可以使用条件复制指令（`cmov`）来移除分支。编译器会将C语言中简单的三目运算符编译成条件复制指令。移除分支条件中的第一段代码也可优化为：

```
x = (a>0 ? a : b);
```

由于64位X86处理器都支持条件复制指令，因此推荐使用。

6.查表法移除分支

查表法是指提前计算一些结果，将结果放到一张具有索引的表中（如C++`std`标准库中的`map`），在实际使用时，只需要依据索引从表中取得计算好的结果即可。如果能够将各分支路径的计算结果放到一张表

中，并将分支条件转化为表中值对应的索引，那么就可以将分支跳转转化为访问表中元素，这是查表法移除分支的主要思想。实际上编译器在优化一些分支模式代码时也使用了类似的想法。

常见的许多分支模式都可以使用查表法去除，比如：

```
if(score >= 90) //score属于[0...100]
    return 'A';
else if (score >= 80)
    return 'B';
else if (score >= 70)
    return 'C';
else
    return 'D';
```

可改写为：

```
char s[] = {'D', 'D', 'D', 'D', 'D', 'D', 'D', 'D', 'C', 'B', 'A'};
return s[score/10];
```

使用查表法去除分支有两点要求：①分支路径很容易转化成索引；②分支结果能够提前计算出来。

7.分支顺序

C中判断式求值是短路的，也就是说如果现在的信息已经能够决定整体的结果，后面的就不用算了。如if（a&&b），如果a为假，那么if语句就一定不能执行，故b不用再求值。例如，a、b中a是计算量相当大的，就应当将它放在后面，即（b&&a），如果计算量差不多，就把a、b中为假概率大的放在前面。对||运算可以类推。当两者有冲突的时候就需要使用剖分软件来分析数据的特征，然后依据数据特点来改变判断式中计算的顺序。

4.6.6 优化交换性能

交换变量值的操作在编程中经常出现（主要是排序相关代码），故优化其性能比较重要，为此ARM处理器在指令上提供了支持。

在实际使用排序类算法代码中，经常发现如下交换模式代码：

```
unsigned char tmp = a[ji];  
a[ji] = a[jj];  
a[jj] = tmp;
```

可优化为：

```
unsigned char aji = a[ji];  
unsigned char ajj = a[jj];  
a[ji] = ajj;  
a[jj] = aji
```

前一段代码只需要使用一个临时变量，而后一段代码需要使用2个，但是后一段代码两次读之间和两次写之间都没有依赖关系，因此并行性要高。

4.7 指令级别

不同的指令具有不同的吞吐量，在实现相同功能的前提下，使用高吞吐量的指令能够明显提升程序的性能。例如，计算某个数的平方采用自己写或pow2函数而不是pow，在可能的情况下使用移位运算计算乘除法。

许多编译器都提供了一些数学函数的快速实现，有些还提供了更快但精度低一些的实现。在应用条件允许的前提下，可优先使用。

1.减少数据依赖

数据依赖会减弱处理器的乱序访问性能，另外读写依赖会极大地减弱内存性能。代码清单4-25和代码清单4-26是计算前缀和的两种形式，代码清单4-26的每一次循环读地址i的操作都依赖前一次循环写入的地址i的数据；而代码清单4-25的循环之间只依赖临时变量temp，编译器很可能将其放入寄存器中，因此依赖更少。

代码清单4-25 寄存器依赖

```
float tmp = 0.0f;
for(int i = 0; i < len; i++){
    tmp += a[i];
    a[i] = tmp;
}
```

代码清单4-26 数据依赖

```
for(int i = 1; i < len; i++){
    a[i] += a[i-1];
}
```

减少数据依赖能够提升代码的指令级和向量级并行能力。在代码清单4-25中，读取数组a就具有非常好的指令级并行能力，而在代码清单4-26中，因为a[i]是由前一次循环写入的，所以写入数组a也一样。

2.注意处理器的多发射能力

在很多处理器上，单指令发射能力不能保证所有的执行单元都可同时运行，因此引入了双发射和多发射能力。但是实际上一些指令系列即使不存在依赖也不能多发射，因此应尽量将能够同时发射的指令安排在一起，不过这可能要求使用内置函数或汇编语言编写代码。

3.优化乘除法和模余

一般整数的位运算最多只要一个周期，而乘法要三个周期，除法十几个，模余需要几十甚至上百个，而通常移位运算只要一个周期。

某些情况下，可以将除法转化为乘法，或者保存某些除法的中间结果，必要时甚至可以使用某些近似算法。如下面的代码中的除法可转化为乘法：

```
for(int i = 0; i < numRows; i++){
    for(int j = 0; j < numCols; j++){
        int index = i*numCols + j;
        r[index] = a[index]/b[j];
    }
}
```

将除法转换成乘法后代码如下所示：

```
for(int j = 0; j < numCols; j++){
    rb[j] = 1.0f/b[j];
}
for(int i = 0; i < numRows; i++){
    for(int j = 0; j < numCols; j++){
        int index = i*numCols + j;
        r[index] = a[index]*rb[j];
    }
}
```

为了将多次除法转成一次除法加多次乘法，笔者先使用一个临时数组rb保存中间结果。

整数乘以一个整数可以转变成左移操作，如果乘数是常量，编译器会自动执行这种转换。整数除法和模余是非常耗时的，要少使用，如果是除以或模2的幂，则可转变为右移或位与运算。

4.选择更具体的库函数或算法

如2的整数次方可以采用移位运算实现，而且很多语言的数学库内有特殊操作的快速实现。如1加某个极小数的对数有log1p，如求2的n次方可以用函数pow2（n）而不是pow（2，n）。

5.其他

如声明float时加f后缀，使用const、static，少用虚函数等。尽量给编译器更具体更多的信息，以便编译器能够做出更好的优化决定。

4.8 本章小结

很多时候编译器能够很好地帮助我们优化代码，本章介绍的优化方法在某些编译器下，或者某些编译器开启了某些选项的情况下，并不一定有效。编译器也是一种好的优化工具，通过查看编译器生成的汇编代码就可以知道它做了哪些优化。

本章依据优化所涉及的尺度将优化方法归类分为：系统级别、应用级别、算法级别、函数级别、循环级别、语句级别和指令级别。

在介绍了如何优化串行代码的性能后，我们开始为代码的向量化和并行化做准备，要向量化或并行化代码，必须分析代码的依赖关系。

第5章 依赖分析

本章的依赖是指程序代码必须按照某种顺序执行，如果不依据这种顺序就会产生不同的运行结果。依赖分析是指分析代码中的这种依赖关系，以更好地理解代码行为。


依赖分析能够找出程序代码中哪些部分必须串行执行，哪些部分可以并行执行。这可以依据粒度分为两个层次：

- 指令级依赖，分析相邻的几条指令间的依赖以决定这几条指令是否能够利用流水线执行，指令级依赖分析可以帮助开发人员进行指令级并行优化。

- 循环级依赖，分析循环间是否存在指令级依赖，这是循环并行化的基础工作。

指令级依赖分析是优化流水线性能的基础，而循环级依赖分析则是向量化和数据并行的基础。如果能够确定循环中不存在依赖，那么该循环便可由多个控制流同时执行。细粒度的循环依赖分析可以确定代码是否能够被向量化。

理想地说，依赖分析应当是编译器的工作，实际上目前指令级依赖分析主要由编译器承担，但是一些依赖是编译器无法发现的，这就必须要由人脑来解决。

 **注意** 通常两条代码之间无依赖是指两条代码的任意输入或输出之间没有依赖，只要有一对输入或输出有依赖，那么这两条代码之间就有依赖。

本章从指令级依赖（控制依赖、数据依赖和结构化依赖）与循环依赖这两方面出发，总结一些典型的依赖关系，并给出一些典型依赖的特征，以及一些具体典型的去除依赖的方法。

5.1 指令级依赖

指令级依赖主要有以下几个方面：

- 结构化依赖：结构化依赖是指由于处理器本身的某些限制，导致本来没有依赖的指令有了依赖。处理器具有多级流水线，流水线的每一级负责不同的工作，如果多条无依赖的指令都需要流水线的同一级，那么这些指令之间存在结构化依赖。

- 数据依赖：数据依赖是指下一条对数据操作的指令必须等待上一条操作该数据的指令完成。常见的数据依赖有读后写、写后读、写后写等。由于不依赖于具体的硬件实现，数据依赖比结构化依赖更易于发现。

- 控制依赖：控制依赖指由于条件执行导致的依赖，由于某些语句只有在控制条件成立才能够被执行，因此这些语句依赖于控制条件的成立与否。

5.1.1 结构化依赖

如果一条指令执行所需要的资源都满足的话，这条指令就会被发射到处理器上执行。这些资源包括：

- 寄存器，用来保存运算的结果。如果有寄存器依赖的话，那么这条指令就不会被发射。
- 存储缓冲区，用来保存等待写入缓存/内存的运算结果。使用存储缓冲区的原因在于，如果不使用存储器缓冲区的话，那么只有结果写入缓存后才能执行下一条指令，写缓存的延迟通常比较大，会导致流水线停顿。如果存储缓冲区被用完，那么这条指令就必须等待直到有存储缓冲区可用，即指令无法发射。
- 读取缓冲区，用来保存读取的缓存/内存数据。由于读取内存的延迟比较大，读取缓冲区的存在使得流水线不会停顿。如果读取缓冲区不可用，那么指令将不会被发射。
- 分支缓冲区，用来保存分支预测的结果和等待提交（commit）的指令。
- 流水线，如果指令需要执行的流水线正在被其他指令占用，那么指令就不会被发射。

结构化依赖除了和上面提到的具体的处理器设计的细节密切相关，还和缓存层次结构、指令发射、寄存器数量等有关，这增加了结构化依赖分析的难度，但是有一些设计是相同或相似的，因此有一些常见的模式可供参考。

假设某个处理器上多条L1访问映射到同一条缓存线，那么即使这多个访问没有依赖，也需要串行处理，这就是由缓存的结构导致的依赖。

为了既提高硬件利用效率又保证提供必需的带宽，许多处理器的缓存层次（从寄存器到内存）都采用了存储体（Bank）的方式组织，其中每个存储体可同时、独立地提供带宽。这种组织方式要求有一种机制，将下一个缓存层次的地址映射到上一缓存层次（如从L2到L1，不包括寄存器）。如果程序一段时间访问的地址在缓存层次上被映射到某一两个存储体，那么程序所能够获得的带宽就是那一两个存储体能够提供的带宽。举个典型的例子，Intel X86处理器的寄存器或一级缓存存储体冲突会导致串行，使得程序无法发挥寄存器或一级缓存的带宽；NVIDIA Kepler GPU的共享存储器冲突导致结果带宽与存储体数量和每个存储体带宽密切相关，AMD GCN GPU的存储器控制器冲突会导致一些存储器控制器空闲进而无法发挥存储器的所有带宽，等等。

一些处理器的指令发射单元一个时钟周期只能发射一条指令，那么对于某个具体的指令，它很难获得接近1的IPC。

在ARM A15处理器上，其NEON指令集支持的32位浮点乘加向量指令延迟大约为8个周期，吞吐量为每核心每周期1个，故需要的并行度为8；如果处理器不能提供足够的寄存器，那么就需要从一级缓存中读取，此时寄存器的数量和一级缓存的带宽就有可能成为结构化的依赖。

发现程序遇到结构化依赖既是不幸，也是幸运，说其不幸是因为要去掉结构化的依赖通常需要细心的调整算法，难度可想而知；说其幸运是因为能够发现程序的性能是由结构化依赖导致的需要对处理器有足够深入的了解。

5.1.2 数据依赖

从数据的观点看，对同一数据进行操作的两条指令之间的关系有4种情况。

- 读后读：如果后一条指令的输入是前一次指令的输入，这称为读后读。读后读的两条指令可以并行操作。因为读不会改变值，因此可以交换两条指令的顺序。

- 读后写：如果后一条指令的输出是前一次指令的输入，称为读后写，也称为“反依赖”。读后写的两条指令不能并行操作，一旦交换它们的顺序，将会产生不同的结果，如下所示：

```
S1 area = PI * r * r;  
S2 r = 0.2;
```

在上面的代码中，S2更改了r的值，如果更改S1和S2的顺序将会产生不正确的结果。

解决读后写依赖的方法非常简单：如上例只需要将S2中的r变量重新命名为p即可。现在的编译器能够自动进行这种优化，而且一些处理器（如Intel X86）硬件上提供了寄存器重命名机制来帮助处理器在运行时处理这种依赖关系。

- 写后读：如果后一条指令的输入是前一次指令的输出，称为“写后读”，也称为“流依赖”。写后读的两条指令不能并行。如下所示为一个典型的写后读依赖：

```
S1 PI = 3.14  
S2 r = 5.0  
S3 AREA = PI * r * r
```

很明显，只有S1和S2的写完成，S3才能计算，这说明S3依赖于S1和S2，即只有S1和S2完成，S3才能开始执行。同样可以看出，S1和S2之间没有依赖，可以并行执行。

- 写后写：如果后一条指令的输出是前一次指令的输出，称为“写后写”，也称为“输出依赖”。写后写的两条指令不能并行执行，因为结果不确定，示例代码如下所示。但是如果使用一些像原子指令、临界区、锁等同步机制，也有可能能够并行。

```
S1 PI = 3.14  
S2 r = 5.0  
S3 AREA = PI * r * r  
S4 r = 2.0;  
S5 AREA = PI * r * r;
```

S3和S5都写了AREA变量，如果调换两者顺序，最终的AREA值将会不正确。

解决写后写依赖的方法非常简单：如上例只需要将S5中的AREA变量重新命名为ar，同时将S4的r变量重新命名为p即可。现在的编译器能够自动进行这种优化，而且一些处理器（如Intel X86）硬件上提供了寄存器重命名机制来帮助处理器在运行时处理这种依赖关系。

5.1.3 控制依赖

由于某些语句只有在控制条件成立才能够被执行，因此这些语句依赖于控制条件的成立与否。示例代码如下所示：

```
s1 if(y > 3){  
s2   x += 5;  
s3 }
```

很明显，s2依赖于s1，因为只有s1完成后，s2才能执行。实际上为了降低控制依赖导致的损失，许多现代处理器并不要求s2等待s1执行完，而是采用分支预测。分支预测失败的代价很高，因此分析并去除控制依赖仍然很有意义。

5.2 循环级依赖

(1) 常见循环依赖

循环级依赖也可以更细致地分为两个方面：

- 循环内部依赖，这和指令级依赖一致，可参考5.1节。
- 循环间依赖，指下一次循环的计算依赖上一次循环的计算结果。循环间依赖会阻碍编译器做循环变换优化、影响循环的流水线执行效率。

(2) 常见的去除依赖方法

通过去除代码中的依赖，可以增强代码的流水线执行、指令级并行或线程级并行能力。循环级依赖的粒度大于指令级依赖，因此去除循环级依赖通常更重要。本节以几个例子展示简单的依赖去除技术。

常见的去除依赖的技术有以下几种：

- 使用寄存器来保存有依赖的存储器访问，将存储器依赖转化成寄存器读写依赖，降低依赖对性能的影响。
- 使用临时存储器来保存有依赖的读写，在必要时再合并。
- 重新组织代码顺序，尽量使得对每个元素的处理在某一次循环内完成。

现在让我们看一下如何使用这几种技术来去除依赖。

5.2.1 循环数据依赖

如果下一次循环所读写的数据是前面几次循环的写入数据，那么就存在依赖。示例代码如下所示：

```
for(int i = 0; i < n; i++){
s1      a[i+1] = b[i] + c;
s2      d[i] = a[i] + e;
}
```

对于上面的例子，假设*i*=5，此时s1写入a[6]，而s2要读a[5]，可见s2和前一次的循环的s1相互依赖，即s1中写a中元素和s2中读a中元素存在写后读依赖，可以使用一个寄存器变量来保存s1计算的a[i+1]，故代码可转化成：

```
abefore = a[0], aafter;
for(int i = 0; i < n; i++){
s1      aafter = b[i] + c;
s2      a[i+1] = aafter;
s2      d[i] = abefore + e;
s3      abefore = aafter;
}
```

此转化把循环内4次访存转化成3次，但是并没有去掉写后读依赖，更进一步，如果将s2前提一次循环，那么代码可转化成如下形式：

```
d[0] = a[0] + e;
for(int i = 1; i < n; i++){
s1      x = b[i-1] + c;
s2      a[i] = x;
s3      d[i] = x + e;
}
a[n] = b[n-1] + c;
```

优化后，从原来代码中两次内存读、两次写减为现在的两次写、一次读，且去掉了循环间依赖，即s2和s3可以并行。去除循环间数据依赖后，在支持不对齐的向量加载指令的处理器上（如Intel Haswell），s1、s2和s3还可以向量化。

下面是一个更为复杂的例子：

```
for(int i = 0; i < n; i++){
S1      a[i] = b[i] + 1;
S2      c[i] = a[i] + 2;
S3      d[i] = c[i+1] + 1;
}
```

假设*i*=6，此时S2写入c[6]，S3读c[7]，而*i*=7时，S2写入S[7]，故存在读后写的依赖。

对于上面的示例代码，可以使用一个临时变量保存a[i]值，这可将代码变换成：

```
for(int i = 0; i < n; i++){
```

```
s1      x = b[i] + 1;
s2      a[i] = x;
s3      c[i] = x + 2;
s4      d[i] = c[i+1] + 1;
}
```

从上面的代码中看出，s4操作的c是原始的c数组中值，和s3中对c的更新无关，但是虽说无关，却降低了流水线执行能力，并且阻止了向量化和并行化的可能。我们可以将原始的c复制到另一个数组cb，以增加流水线效率，如下所示：

```
memcpy(cb, c+1, n*sizeof(c[0]));
for(int i = 0; i < n; i++){
s1      x = b[i] + 1;
s2      a[i] = x;
s3      c[i] = x + 2;
s4      d[i] = cb[i] + 1;
}
```

虽然这种做法提高了代码的乱序执行能力和使得循环能够被向量化，但是却增加了一次内存分配、一次释放和一次拷贝的开销，实际应用中需要权衡。

一个更好的方法是循环拆分，拆分为如下所示的两个循环：

```
for(int i = 0; i < n; i++){
s4      d[i] = c[i+1] + 1;
}
for(int i = 0; i < n; i++){
s1      x = b[i] + 1;
s2      a[i] = x;
s3      c[i] = x + 2;
}
```

上面的代码已经可以接着执行向量化、循环展开等优化，但是对c[i]的读写还可以进一步优化以提升流水线效率，如下所示：

```
x = b[0] + 1;
a[0] = x;
c[0] = x + 2;
for(int i = 1; i < n; i++){
s4      d[i-1] = c[i] + 1;
s1      x = b[i] + 1;
s2      a[i] = x;
s3      c[i] = x + 2;
}
d[n-1] = c[n]+1;
```

实际上上面的代码还可以接着进行一些可能的优化，这个就留给读者了。

5.2.2 循环控制依赖

如果控制条件依赖于前几次或后几次循环写入的值，那么就存在循环间控制依赖，示例代码如下所示。

```
for(int i = 1; i < n; i++){
S1    a[i] = b[i] + c[i];
S2    if(a[i-1]){
S3        c[i] = a[i] + 2;
    }
}
```

S2所依赖的a[i-1]由前一次循环的S1写入，两者之间存在依赖。

一些常见的分支优化办法同时也可以用于去除循环间控制依赖，如上所示的代码，如果使用一个变量来保存前一次写入的a[i]值，就去掉了一次存储器读取，如下所示：

```
prev = a[0];
for(int i = 1; i < n; i++){
S1    next = b[i] + c[i];
S2    if(prev){
S3        c[i] = next + 2;
    }
S4    a[i] = next;
    prev = next;
}
```

再使用一个临时变量保存c[i]的值，代码转化成：

```
prev = a[0];
for(int i = 1; i < n; i++){
    tempc = c[i];
S1    next = b[i] + tempc;
S2    if(prev){
S3        tempc = next + 2;
    }
S4    c[i] = tempc;
S5    a[i] = next;
    prev = next;
}
```

在X86处理器上，此时可以使用条件转移指令来去除控制依赖，如下所示：

```
prev = a[0];
for(int i = 1; i < n; i++){
    tempc = c[i];
S1    next = b[i] + tempc;
S2    c[i] = prev ? next+2 : tempc;
S3    a[i] = next;
S4    prev = next;
}
```

进行此转换之后，还可以进行循环展开、向量化等优化。

5.3 寄存器重命名

理想状态下，依赖分析应当是编译器的工作，但是一些依赖是编译器阶段无法发现的，还有一些是编译器即使发现了也无能为力的。现代的许多高性能处理器（如Intel X86）都提供了支持去除依赖的硬件。

如本章前面所描述，许多依赖关系都可以通过变量重命名来解决或缓解。这种简单的技术，现代的编译器也可以利用，只是受限于处理器指令集体系结构规定的逻辑寄存器数量的限制，汇编程序能够使用的寄存器是非常有限的，因此编译器很多时候只能是无能为力。在许多高性能的处理器实现中，物理寄存器的数量远大于逻辑寄存器的数量，如Intel IA64指令集只有16个逻辑寄存器，而物理寄存器近200个。如果处理器在执行时能够把一个逻辑寄存器映射到多个物理寄存器就可以实现变量重命名，实际上，在物理寄存器数量远大于逻辑寄存器数量的处理器上，这种能力是必需的，通常称这种处理器硬件支持变量重命名的机制为“寄存器重命名”。

寄存器重命名能够更好地发挥处理器流水线的性能，但是由于以下一些现实的原因，代码优化人员并不能很好地利用它们：

1) 处理器厂商没有提供其硬件寄存器重命名算法的详细细节。寄存器重命名算法比较复杂，而且这些算法也是处理器设计中的重要技术，处理器厂商并不愿意公开太多细节。

2) 行为由处理器运行时决定。由于不能直接在汇编指令中看到物理寄存器，因此代码优化人员无法确定他的优化的确能够帮助处理器更好地进行寄存器重命名工作。

5.4 本章小结

依赖分析应当是编译器的工作，实际上指令级依赖分析的工作主要由编译器承担，但是一些依赖是编译器无法发现的，这就必须要由人脑来解决。

本章简单介绍了指令级依赖和循环级依赖，并给出许多如何去除依赖的示例，最后笔者以简单介绍处理器硬件支持的寄存器重命名结束。

指令级依赖主要分为结构化依赖、数据依赖和控制依赖。循环级依赖主要分为循环数据依赖和循环控制依赖。

第6章 并行编程模型及环境

和串行编程类似，并行编程对于不同应用场景也有不同的解决方法。由于并行的特殊性，串行的解决方法不能直接移植到并行，因此需要重新设计。并行编程模型大多数以数据和任务（过程化的操作）为中心来命名，也有一些是以编程方法来命名。

一个具体的并行应用往往使用了多种并行编程模型，通常对于应用的某个小模块而言只使用一种并行编程模型。并行编程模型是并行算法的基础，并行算法的具体实现依赖于软硬件支持的并行编程模型。和串行编程类似，并行编程也表现出模式的特征，并行编程模式是对某一类相似并行算法的解决方案的抽象。本章将会介绍主流的并行编程模型，第7章和第8章将介绍并行算法设计相关的内容，第9章则会介绍如何实现常见的并行编程模式。

本章主要说明一些常见的、在实践中行之有效的并行编程模型，另外还会介绍目前非常常用的并行编程环境及这些环境所支持的编程模型。

6.1 并行编程模型

一方面，并行编程模型是建立在硬件体系结构模型之上的并行程序实现逻辑的抽象，它定义了并行算法设计与其实现之间的一种隐形的协议，为并行算法的设计者提供了一个简洁而清晰的并行软硬件系统结构的概念模型。从另外一个方面说，并行编程模型是指并行算法设计时对模块间通信方式的抽象。经过几十年的发展，人们已经总结出一系列有效的并行模型，这些模型的适用场景各不相同。本节将简要说明一些常用并行模型的特点、适用的场景和情况。

需要说明的是：从不同的角度看，一个并行应用可能属于多个不同的并行模型，本质原因在于这些并行模型中存在重叠的地方。由于模型并非正交，因此适用于一种模型的办法可能也适用于另一种模型，读者需要举一反三。

6.1.1 指令级并行

如果多条指令之间既没有数据和控制依赖，也没有结构化依赖，那么它们可以同时在处理器的多个流水线上同时执行，这称为指令级并行。超标量和VLIW处理器通常都拥有多个执行单元，这些单元可能是相同的，也可能是不同的。这些执行单元可以以并行或流水线的方式执行指令。通常处理器的指令级并行技术要求能够动态地进行指令依赖分析、顺序或乱序发射以及顺序或乱序执行，编译器也承担了部分相关的工作。

指令级并行优化粒度太细，需要深入了解处理器的流水线延迟和吞吐量以及编译器的能力，因此不易于使用。笔者认为，执行指令级并行优化首先是处理器的责任，其次是优化编译器的责任，最后才是代码优化人员的责任。

进行指令级并行优化比较好的实践模式是修改源代码以便让编译器生成需要的指令系列，这通常比直接使用汇编和编译器内置函数（Intrinsic）要易于使用，代码也易于维护。但是编译器可能不支持处理器提供的一些最新特性，或者支持但是存在缺陷，这个时候汇编可能是最好的选择。

指令级并行的重点是去除指令间的依赖关系。代码清单6-1所示的循环展开代码，循环内语言都依赖变量sum。可将其优化为代码清单6-2的版本。

代码清单6-1 循环展开代码

```
sum=0;
for(int i = 0; i < len; i += 4){
    sum += a[i];
    sum += a[i+1];
    sum += a[i+2];
    sum += a[i+3];
}
```

代码清单6-2 优化后的代码

```
sum=sum1=sum2=sum3=0;
for(int i = 0; i < len; i += 4){
    sum += a[i];
    sum1 += a[i+1];
    sum2 += a[i+2];
    sum3 += a[i+3];
}
sum += sum1;
sum2 += sum3;
sum += sum2;
```

对于代码清单6-1来说，循环内的4条语句都相互依赖于sum命令的更新，因此循环中的4个加法操作需要串行执行（读a操作可并行）。优化后，4条语句不再存在任何依赖，因此无论是加法运算，还是访存都

可以并行操作。

在需要使用汇编语言进行指令级并行优化时，笔者建议读者先试图使用内置函数生成想要的代码，如果不能，然后再在内置函数版本产生的汇编代码上修改。

6.1.2 向量化并行

向量化并行是一种细粒度的并行计算，通常是对不同的数据执行一条同样的指令，或者说一个指令作用于一个数组/向量。向量化并行是数据并行的一个细粒度子集，在某些情况下，可认为向量化并行是指令级的数据并行。主流的处理器的都支持向量指令集，比如X86处理器的SSE/AVX指令，ARM的NEON指令。编译器通常提供编译指导（通过给编译器一些“指示”，让编译器生成向量指令，如OpenMP 4.0）和内置函数（对汇编指令进行简单的C封装）的方式对这些向量指令提供支持。NVIDIA的CUDA和开放的OpenCL标准通过层次化的编程模型（SIMT）来使用一份代码既支持多核并行又支持向量化。

以求两向量的距离为例，假设不存在存储器别名，其串行代码如代码清单6-3，而对应的NEON代码如代码清单6-4所示。

代码清单6-3 求两向量的距离

```
float distance(int len, float* restrict a, float* restrict b){
    float dis = 0.0f;
    for(int i = 0; i < len; i++){
        float diff = a[i] - b[i];
        dis += diff*diff;
    }
    return sqrt(dis);
}
```

从代码中可以看出：i循环更新dis存在依赖，因为都更新同一个dis变量。可以将dis分割成几个变量，但是这意味着运算顺序发生了变化，变换运算顺序会使得浮点运算的结果产生差别。本节笔者假设变换运算顺序产生的误差是可以容忍的。

NEON向量寄存器的长度为128位，故可以同时处理4个32位浮点数。

代码清单6-4 代码清单6-3优化后的代码

```
float distance(int len, float* restrict a, float* restrict b){
    float32x4_t dis = vdupq_n_f32(0.0f);
    for(int i = 0; i < len; i+=4){
        float32x4_t a_vec = vld1q_f32(a+i);
        float32x4_t b_vec = vld1q_f32(b+i);
        float32x4_t diff = vsubq_f32(a_vec, b_vec);
        dis = vmlaq_f32(dis, diff, diff);
    }
    return sqrt(dis[0]+dis[1]+dis[2]+dis[3]);
}
```

为了示例方便，在NEON代码中，笔者并没有考虑len不是4的整数倍的情况。

向量化适合处理图形图像、音频视频、游戏等应用程序，因为这些程序通常需要对大量数据做相同的处理，并且对延迟敏感。科学计算和模拟中也大量使用向量化并行，如分子动力学、计算金融、石油探测

等，目前也有很多开发人员在互联网应用程序中使用向量化。

6.1.3 易并行

易并行计算是指并行执行的多个控制流之间没有通信的并行，英文为embarrassingly parallel，直译则为令人尴尬的并行。比如图像的二值化操作中，对每个像素的操作都和其他像素无关，是完全独立的，一些蒙特卡罗算法^[1]也有这种特点。

由于控制流之间没有通信，易并行计算算法的设计通常比较简单，属于比较笨的办法。对于计算限制的算法，易并行意味着通常可获得近线性的性能提升。

易并行计算是非常适合并行的，也是目前应用最好的一类并行。笔者认为并行算法设计的目标就是向易并行迈进。

基于进程的和基于线程的编程环境，甚至指令级并行环境都可以很好地利用易并行计算的特点，决定使用哪种环境的关键通常是计算问题的规模。必要时可同时使用这三种编程环境，在进程中分配线程，在线程中使用指令级并行。

[1] 统计模拟方法，是20世纪40年代中期由于科学技术的发展和电子计算机的发明，而被提出的一种以概率统计理论为指导的一类非常重要的数值计算方法。是指使用随机数（或更常见的伪随机数）来解决很多计算问题的方法。

6.1.4 任务并行

任务并行是指每个控制流计算一件事或者计算多个并行任务的一个子任务，通常其粒度比较大且通信很少或没有。

由于和人类的思维方式比较类似，任务并行比较受欢迎，且又易于在原有的串行代码的基础上实现。

代码清单6-5的invoke函数分配并初始化了d1和d2两个数组，然后调用f1函数来处理d1数据，调用f2函数来处理d2数据。使用任务并行的思想和pthread线程来改进的话，其代码如代码清单6-6所示。

代码清单6-5 任务并行逻辑示例

```
void* f1(void* data){
    computineOnData1(data);
    return NULL;
}
void* f2(void* data){
    computineOnData2(data);
    return NULL;
}
void invoke(){
    float* d1;
    ...
    f1(d1);
    float* d2;
    ...
    f2(d2);
}
```

由于函数f1和f2作用在不同的数据集上，因此这两个函数可以并行，可以使用一个控制流处理函数f1和d1，另外一个控制流处理函数f2和d2。使用pthread并行的版本如代码清单6-6所示：

代码清单6-6 pthread任务并行代码

```
void* f1(void* data){
    computineOnData1(data);
    return NULL;
}
void* f2(void* data){
    computineOnData2(data);
    return NULL;
}
void invoke(){
    pthread_t t1, t2;
    float* d1;
    ...
    pthread_create(&t1, NULL, f1, d1);
    float* d2;
    ...
    pthread_create(&t2, NULL, f2, d2);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
}
```

主线程使用pthread_create函数创建一个线程t1来执行函数f1，创建另外一个线程t2来执行函数f2，然后主线程再调用pthread_join函数来等待线程t1和t2执行完成。

任务并行的通信通常比较少，故易于实现，但是负载不均衡的可能性非常大，经常采用任务队列的方式来解决任务并行的这个问题。如果任务之间的计算量差别比较大，那么就需要为计算量大的任务分配比较高的优先级以先计算。另一种解决任务并行的负载均衡问题的方法是尽量将任务划分得比较小，每个控制流就可以分得许多小任务，再加上任务队列的作用，负载不均衡就很小了（至多一个小任务）。

由于计算粒度比较大，基于进程的编程环境和基于线程的编程环境都适用，通常使用哪种环境取决于各个任务是否需要共享资源或同步，及整个任务的规模。如果要共享资源或同步，则线程比较好；如果任务规模比较大，超过节点内存大小时，则需要使用基于进程的编程环境。

事件驱动并行广泛应用于GUI，通常用户对GUI的菜单、快捷键操作都会生产一个事件，对于每一个事件，系统都会使用一个单独的控制流来处理。事件驱动并行的好处是能够及时地响应用户的请求，提高使用体验。

6.1.5 数据并行

数据并行是指一条指令同时作用在多个数据上，那么可以将一个或多个数据分配给一个控制流计算，这样多个控制流就可以并行，这要求待处理的数据具有平等的特性，即几乎没有需要特殊处理的数据。如果对每个数据或每个小数据集的处理时间基本相同，那么均匀分割数据即可；如果处理时间不同，就要考虑负载均衡问题。通常的做法是尽量使数据集的数目远大于控制流数目，动态调度以基本达到负载均衡。

（1）数据并行的类型

以输入数据为主，每个控制流处理一个或多个输入数据，对于这种模型来说，对输入数据的处理通常无须同步，但是由于有可能多个控制流会更新同一个输出数据（因为输入/输出数据存在多对一的关系），因此对输出数据可能需要同步。

以输出数据为主，每个控制流处理一个或多个输出数据，对于这种模型来说，对输出数据的处理通常无须同步。通常输入数据都是只读的（如果不是只读的，通常也可以通过引入中间数据变成只读的），因此对输入数据的处理通常也无须同步，故应优先选择这种方式。如果一个数据既是输入又是输出，那么此时就需要仔细分析。

（2）数据并行的应用

数据并行应用最广的是图形学算法，因为像素的渲染通常是相互独立且要做的处理是相同的，故天然适合数据并行。

数据统计学要求分析大量数据，得到其统计特征，对大量数据的处理通常满足数据并行要求，如从大量数据中筛选出满足要求的数据。

一些模拟计算，如分子动力学、天体物理、计算流体力学算法也具有数据并行的特点。

目前异常火热的深度学习研究和应用也表现出数据并行的特点，如神经网络的主要运算是矩阵乘法，它也满足数据并行的特点；如卷积神经网络的主要运算是计算卷积，也满足数据并行的特点。

数据并行对控制的要求比较少，因此现代GPU利用这一特性，大量减少控制单元的比例，而将空出来的单元用于计算，这样就能在同样数量的晶体管上提供更多的原生计算能力。

基于进程的、基于线程的环境，甚至指令级并行环境都可以很好地应用在数据并行上。必要时可同时使用这三种编程环境，在进程中分配线程，在线程中使用指令级并行处理多个数据，这称为混合计算，后

面会详细说明。

6.1.6 循环并行化

很多算法迭代地处理大量数据，这表现为循环。如果某次循环必须等待其前面的循环执行完成才能够执行，这称为**串行循环**。如果各次循环之间不存在依赖可以并行执行，称之为**并行循环**。

通常而言，串行循环不能并行化，但是某些串行循环可变换成并行循环，而某些看起来不能并行的串行循环已经有很好的并行算法，如stream compaction、reduction和scan等。

一些看起来是串行循环，但是经过循环拆分可变成并行循环。假设不存在存储器别名，如代码清单6-7所示的代码不能并行。可将其更改为代码清单6-8或代码清单6-9，就都可以并行。

代码清单6-7 原始代码

```
for(int i = 0; i < n; i++){
    a[i+1] = b[i] *2;
    c[i] = a[i]/3;
}
```

如果直接并行循环，那么对数组a的读写会出现问题，因为从依赖分析的角度看，循环存在写后读依赖。考虑到对于 $i > 0$ 来说，循环内第二条语句中的 $a[i]$ 可转化成 $b[i-1] \times 2$ ，故可获得如下所示的代码。

代码清单6-8 优化后的代码

```
c[0] = a[0]/3;
a[1] = 2*b[0];
for(int i = 1; i < n; i++){
    a[i+1] = b[i] *2;
    c[i] = 2*b[i-1]/3;
}
```

修改后循环内代码不存在依赖，故可以并行。

如果对代码清单6-7执行循环拆分，即将一个循环拆分成两个循环，则代码如下：

代码清单6-9 循环拆分后的代码

```
for(int i = 0; i < n; i++){
    a[i+1] = b[i] *2;
}
for(int i = 0; i < n; i++){
    c[i] = a[i]/3;
}
```

很明显可以看出，拆分后的两个循环可以并行化。

对于并行循环来说，向量化、线程级并行和多机器并行都可能适用。

6.1.7 流水线并行

如果把任务分成多个阶段，各个阶段相互依赖，只能串行，而不同任务的阶段可以并行，这非常类似现代的指令流水线。流水线能够让不同的硬件单元同时运行以提高计算能力，流水线并行就利用了这一特性。

常用队列来保存在某一个阶段可以并行执行的任务，由于队列中的任务可能是前一个阶段生成的，因此队列可能必须支持并发操作。对一个四级流水线来说，可由4个队列来保存每一级流水线上操作的数据。

和指令级流水线并行类似，应用编程通常使用流水线并行以同时使用多个功能硬件，如异步编程同时计算和读取/存储数据，代码清单6-10是一个使用Linux AIO来重叠计算和写入数据的简单示例。在基于GPU的异构并行编程中，数据传输和运算同时进行。

代码清单6-10 用Linux AIO来重叠计算和写入

```
aio_write(&acb);
for( int i = 0; i < 4096; i++){
    count += rand()%255;
}
struct aiocb* list[1];
list[0] = &acb;
if (-1 == aio_suspend(list, 1, NULL)){
    perror("aio suspend failed");
}
```

由于需要对计算进行精密的控制，且很少可以实现为其他的并行模型，目前软件上完全使用流水线并行的并不多见，但是许多算法或实现中（如Ajax技术）都有它的影子。

6.1.8 区域分解并行

很多科学计算需要计算某个大区域的某些特性，如流场、密度等。如果能够将大的计算区域划分成多个小的区域，然后由一个控制流计算一个小的区域，那么计算大区域的所有控制流便可同时进行，这称为区域分解并行。

区域分解通常和网格联系在一起，将多维相邻的多个网格分配给某个控制流计算。由于相邻控制流负责计算的网格也相邻，故多维相邻控制流之间的通信和控制流内的缓存利用都可以比较高效地实现。

假设使用16个节点来计算16×16的流场，那么可以将16个节点组织成4×4的二维网格，然后再16×16的流场均匀划分给节点，即每个节点计算二维相邻的4×4的流场，如图6-1所示。

自适应网格适用于收敛速度不同的区域，通常在某些区域达到计算精度后停止计算，而将计算能力全部分配给那些精度还达不到要求的区域，以此来提高计算速度。

多层次网格不断将网格细分，并插值新产生的网格来加速计算，比如对一个9×9的网格，如果已知2×2的值，求3×3，其中中间格子的值由插值得到，求得3×3后，再求5×5，然后是9×9。

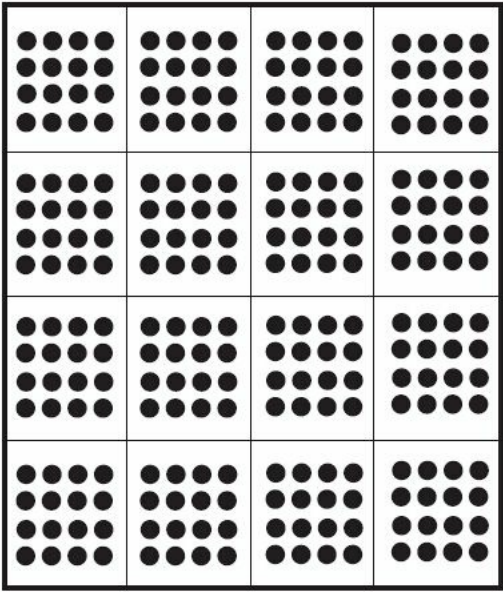


图6-1 16×16的流场示意图

区域分解并行计算算法通常要求计算相邻区域的控制流交换某些信息，此时要考虑控制流的组织和硬件处理器的拓扑结构，以更好地利用存储器层次结构和网络互连。

区域分解并行通常和具体的物理现象联系在一起，但是一些其他的运算也和区域分解类似，比如分块矩阵转置和分块矩阵乘法。

6.1.9 隐式和显式并行化

如果编写显式的并行代码的工作由开发者承担，这称为显式并行。显式并行通常使用线程或进程库以简化工作，如使用pthread开发并行代码就是一种显式并行方式。隐式并行通常是指开发人员通过给编译器添加某种标记，指定程序的并行性，而实际的并行代码由编译器生成，如OpenMP和OpenACC。在OpenMP和OpenACC中，开发人员不用显式地编写线程代码，只需要通过编译制导语句指导编译器，让编译器生成线程代码。

相对来说，显式并行易于控制，易于获得高效率，但移植性不好，工作量大，且难以调试。而隐式并行刚好相反。现在的编程环境通常基本上都允许这两种方式混合使用。可以首先使用隐式并行，再在必要的地方使用显式并行。

6.1.10 SPMD

SPMD (Single Program, Multiple Data), 表示单程序多数据, 是MIMD的子类, 是指将多个控制流的计算编写在同一个源代码文件的一种编程方式。由于不需要为各个控制流单独编写程序, 这会大量节约开发时间 (这一点使得SPMD成为目前最为流行的并行程序编写方法, Cell就是因为没有采用这种模型导致难以使用)。SPMD本质上是因为同一应用的不同控制流之间通常要有许多共同的/相似的操作。

现实中, 几乎所用的并行编程语言都采用SPMD模型。

6.1.11 共享存储器并行

共享存储器并行是指所有控制流都能够访问一个共同的全局存储器，通过这个存储器来交换数据。目前基于多核并行处理器的并行环境基本上都是共享存储器并行环境，比如pthread、OpenMP，等等。

由于所有控制流共享同一个地址空间，对共享数据的访问协调需要开发人员显式处理。

由于多核和多路处理器的所有核心都运行同一个操作系统，访问一个统一的物理地址空间，通信代价相对较小，因此它们天然适合使用共享存储器并行的编程环境。

6.1.12 分布式存储器并行

通过网络互连的多机系统、存储器分布在不同节点机上，每个节点机运行各自的操作系统，拥有独立的物理地址空间。由于通过网络互连，控制流之间的通信只能通过网络数据分发实现。分布式存储并行模型通过显式的消息传递来交换数据，天然适合这一类型应用。目前常用的分布式存储并行环境是MPI（消息传输接口），MPI已经是事实上的标准。

由于每个控制流拥有独立的存储器地址空间，因此不存在存储器访问冲突，但是多控制流之间需要协调通信。从编程上来看，分布式存储并行程序比共享内存并行程序要易于编写，但是通过网络的消息传递的带宽和延迟通常难以预测。

6.2 常见并行编程环境

常见的并程序序设计环境的分类有不同标准，主要是从[编程方式](#)和[通信方式](#)上来分类。

(1) 以编程方式分类

如果从编程方式上讲，大致分为隐式并行环境和显式并行环境：

- 隐式并行 (Implicit Parallel) 环境：用串行语言编程并使用某种标记指明并行性和控制流调度，编译器或运行时环境自动将其转化为并行代码。相比显式并行环境，隐式并行环境具有语义简单、可移植性好、易于调试和可验证正确性等优点。但是相比显式并行环境，其编程能力往往也更受限。

- 显式并行 (Explicit Parallel) 环境：编写代码显式地指明程序的控制流的创建、执行、调度和退出。在实现相同功能的前提下，使用显式并行环境开发的程序代码通常要比使用隐式并行环境开发的程序代码要长。相比隐式并行编程环境，显示并行环境要难使用，但显示并行环境给予了软件开发人员更多的自由，且现有的编译器可通过库实现支持。

(2) 以通信方式分类

如果从通信方式来说，有共享存储器并行环境和基于分布式存储的消息传递并行环境：

- 共享存储器并行 (Shared Memory) 环境：共享存储器并行环境支持多线程并行、使用统一虚拟地址空间通信、线程间使用函数显式同步。共享存储器并行环境天生适合多核处理器。运行在多机上的共享存储器并行环境，其线程并不知道数据分布在哪些机器上，因此其数据访问及迁移路径都是隐式的，因此性能可能不好且不利于优化。

- 消息传递并行 (Message Passing) 环境：消息传递并行环境支持进程并行、使用独立的地址空间、支持异步数据传输、显式同步、显式负载分配、显式通信。消息传递并行环境更适合分布式多机环境。消息传递并行环境也可以运行在多核处理器系统上。

如果两种分类组合起来就产生了4种小分类，目前常见的并行编程环境都可以归类到4种小分类中。

- 支持隐式共享存储器并行的环境：OpenMP、OpenACC、SSE/AVX和NEON内置函数。
- 支持显式共享存储器并行的环境：pthread、cilk、CUDA、OpenCL。
- 支持显式消息传递并行的环境：MPI。

本节会简略介绍这些编程环境的历史、功能和优缺点，使得读者有一个大致的印象，以便在实际项目中知道应该选择哪种并行环境。

6.2.1 MPI

MPI (Message Passing Interface, 消息传递接口) 是一种消息传递编程环境。消息传递指用户必须通过显式地发送和接收消息来实现处理器间的数据交换。MPI定义了一组通信函数, 以将数据从一个MPI进程发送到另一个MPI进程。在消息传递并行编程中, 每个控制流均有自己独立的地址空间, 不同的控制流之间不能直接访问彼此的地址空间, 必须通过显式的消息传递来实现。这种编程方式是大规模并行处理机 (MPP) 和机群 (Cluster) 采用的主要编程方式。实践表明MPI的扩展性非常好, 无论是在几个节点的小集群上, 还是在拥有成千上万节点的大集群上, 都能够很好地应用。

由于消息传递程序设计要求用户很好地分解问题, 组织不同控制流间的数据交换, 并行计算粒度大, 特别适合于大规模可扩展并行算法。MPI是基于进程的并行环境。进程拥有独立的虚拟地址空间和处理器调度, 并且执行相互独立。MPI设计为支持通过网络连接的机群系统, 且通过消息传递来实现通信, 消息传递是MPI的最基本特色。

MPI是一种标准或规范的代表, 而不特指某一个对它的具体实现, MPI成为分布式存储编程模型的代表和事实上的标准。迄今为止, 所有的并行计算机制造商都提供对MPI的支持, 可以在网上免费得到MPI在不同并行计算机上的实现, 一个正确的MPI程序可以不加修改地在所有的并行机上运行。

MPI标准定义了一个库, 共用300多个函数调用, 支持Fortran 77/90和C/C++调用, 从语法上说, 它遵守所有对库函数/过程的调用规则, 和一般的函数/过程没有什么区别。目前MPI最新的标准是3.0版。

MPI只规定了标准并没有给出实现, 目前主要的实现有OpenMPI、Mvapich和MPICH, MPICH相对比较稳定, 而OpenMPI性能较好, Mvapich则主要是为了Infiniband而设计。

MPI主要用于分布式存储的并行机, 包括所有主流并行计算机。但是MPI也可以用于共享存储的并行机, 如多核微处理器。编程实践证明MPI的可扩展性非常好, 其应用范围从几个机器的小集群到工业应用的上万节点的工业级集群。MPI已在MS Windows上、所有主要的UNIX/Linux工作站上和所有主流的并行机上得到实现。使用MPI作消息传递的C或Fortran并行程序可不加改变地运行在使用这些操作系统的工作站, 以及各种并行机上。

MPI既可用于功能分解, 也可以用于数据分解, 是一个比较通用的基于CPU的并行编程模式, MPI 3.0加入了异构并行计算的内容。MPI支持SPMD编程模式, 在MPI程序中定义的变量, 只要不是在进程id块内定义, 它在每个进程内都有一份副本。在运行时, 由于具体的运行进度不一致, 使得各个进程存储空间内的变量值可能不一致, 在必要的时候要使用存储器栅栏来保证存储器的一致性, 这点要特别注意。

6.2.2 OpenMP

OpenMP是Open Multi-Processing的简称，是一个基于共享存储器的并行环境。OpenMP支持C/C++/Fortran绑定，也被实现为库。目前常用的GCC、ICC和Visual Studio都支持OpenMP。

OpenMP API包括以下几个部分：一套编译器伪指令，一套运行时函数，一些环境变量。OpenMP已经被大多数计算机硬件和软件厂商所接受，成为事实上的标准。

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插入各种pragma伪指令来表明自己的意图，编译器据此可以自动将程序并行化，并在必要之处加入同步互斥等通信。当选择告诉编译器忽略这些pragma或者编译器不支持OpenMP时，程序又可退化为串行程序，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。OpenMP提供的这种对于并行描述的高层抽象降低了并行编程的难度和复杂度，这样程序员可以把更多的精力投入到并行算法本身，而非其具体实现细节。对基于数据并行的多线程程序设计，OpenMP是一个很好的选择。同时，使用OpenMP也提供了更强的灵活性，可以适应不同的并行系统配置。线程粒度和负载均衡等是传统并行程序设计中的难题，但在OpenMP中，OpenMP库从程序员手中接管了这两方面的部分工作。

OpenMP的设计目标为：标准、简洁实用、使用方便、可移植。作为高层抽象，OpenMP并不适合需要复杂的线程间同步、互斥及对线程做精密控制的场合。OpenMP的另一个缺点是不能很好地在非共享内存系统（如计算机集群）上使用，在这样的系统上，MPI更适合。

6.2.3 fork/pthread

`fork`是类UNIX系统的一个调用，它调用一次，返回两个进程，一个是调用父进程，另一个是新产生的子进程。通过返回值可以区分父子进程，对于子进程，`fork`返回值为0，对于父进程，返回值为子进程的id。

由于子进程要继承父进程的虚拟内存空间、打开的文件等，因此`fork`执行的代价非常大，`pthread`缓解了这个问题。

`pthread`是一个基于线程的库，提供了创建、回收线程的函数。`pthread`创建的线程和父线程共享内存和指令，但有其独立的指令指针（PC）。

6.2.4 CUDA

CUDA是NVIDIA于2006年11月推出的，用于发挥NVIDIA GPU通用计算能力的编程环境，目前支持CUDA C和OpenCL（Open Computing Language）语言，计算效率高，常可加速十几倍到几十倍。相比OpenCL和Brook+，CUDAC更易于使用。另外NVIDIA对CUDA的大力支持是其他厂商所不能比拟的。

CUDA认为系统上可以用于计算的硬件包含两个部分：一个是CPU（称为主机），一个是GPU（称为设备），CPU控制/指挥GPU工作，GPU只是CPU的协处理器。目前CUDA只支持NVIDIA的GPU，而CPU由主机端编程环境负责。

CUDA是一种架构，也是一种语言。作为一种架构，它包括硬件的体系结构（G80、GT200、Fermi、Kepler）、硬件的CUDA计算能力及CUDA程序是如何映射到GPU上执行；作为一种语言，CUDA提供了能够利用GPU计算能力的方方面面的功能。CUDA的架构包括其编程模型、存储器模型和执行模型。CUDA C语言主要说明了如何定义计算内核（kernel）。CUDA架构在硬件结构、编程方式与CPU体系有极大不同，关于CUDA的具体细节读者可参考CUDA相关的书籍。

CUDA以C/C++语法为基础而设计，因此对熟悉C系列语言的程序员来说，CUDA的语法比较容易掌握。另外CUDA只对ANSI C进行了最小的扩展，以实现其关键特性：线程按照两个层次进行组织、共享存储器（shared memory）和栅栏（barrier）同步。

目前CUDA提供了两种API以满足不同人群的需要：运行时API和驱动API。运行时API基于驱动API构建，应用也可以使用驱动API。驱动API通过展示低层的概念提供了额外的控制。使用运行时API时，初始化、上下文和模块管理都是隐式的，因此代码更简明。一般一个应用只需要使用运行时API或者驱动API中的一种，但是可以同时混合使用这两种。笔者建议优先使用运行时API。

6.2.5 OpenCL

OpenCL (Open Computing Language, 开放计算语言), 先由Apple设计, 后来交由Khronos Group维护, 是异构平台并行编程的开放的标准, 也是一个编程框架。Khronos Group是一个非盈利性技术组织, 维护着多个开放的工业标准, 并且得到了业界的广泛支持。OpenCL的设计借鉴了CUDA的成功经验, 并尽可能地支持多核CPU、GPU或其他加速器。OpenCL不但支持数据并行, 还支持任务并行。同时OpenCL内建了多GPU并行的支持。这使得OpenCL的应用范围比CUDA广, 但是目前OpenCL的API参数比较多 (因为不支持函数重载), 因此函数相对难以熟记。

OpenCL覆盖的领域不但包括GPU, 还包括其他的多种处理器芯片。到现在为止, 支持OpenCL的硬件主要局限在CPU、GPU和FPGA上, 目前提供OpenCL开发环境的主要有NVIDIA、AMD、ARM、Qualcomm、Altera和Intel, 其中NVIDIA和AMD都提供了基于自家GPU的OpenCL实现, 而AMD和Intel提供了基于各自CPU的OpenCL实现。目前它们的实现都不约而同地不支持自家产品以外的产品。由于硬件的不同, 为了写出性能优异的代码, 可能会对可移植性造成影响。

OpenCL包含两个部分: 一是语言和API, 二是架构。为了C程序员能够方便、简单地学习OpenCL, OpenCL只是给C99进行了非常小的扩展, 以提供控制并行计算设备的API以及一些声明计算内核的能力。软件开发人员可以利用OpenCL开发并行程序, 并且可获得比较好的在多种设备上运行的可移植性。

OpenCL的目标是一次编写, 能够在各种硬件条件下编译的异构程序。由于各个平台的软硬件环境不同, 高性能和平台间兼容性会产生矛盾。而OpenCL允许各平台使用自己硬件的特性, 这又增大了这一矛盾。但是如果不允许各平台使用自己的特性, 却会阻碍硬件的改进。

6.2.6 OpenACC

OpenACC编译器依据C/C++/Fortran编写的编译制导语句，将并行区域的代码翻译成另一种语言的表示，如CUDA、OpenCL等。软件开发人员只要在不同的操作系统和主机设备之间编译OpenACC代码，就可以实现可移植性。

OpenACC适合建立高层次的异构程序，不需要显式地初始化加速器，不需要显式地在主机和加速器之间传送数据或程序，只需要通过编译制导语句指定即可。

OpenACC允许程序员从重要的地方开始，一步步在已有程序的基础上将程序移植到加速器上，这既减少了工作量，又易于调试。

使用OpenACC编译指令，性能调优的工作重点主要在指出代码并行性，使编译指令可以更好地将这些代码翻译，而软件开发人员不用纠缠在语言的实现细节上。

OpenMP 4.0已经加入了类似OpenACC的编译制导语句支持，虽然具体的语法规则有所不同，但是基本思想完全一致。

6.2.7 NEON内置函数

NEON是ARM处理器上的SIMD指令，由于ARM在移动端得到广泛应用，目前NEON的使用也越来越普遍。

NEON支持数据并行，一个指令可同时对多个数据进行操作，同时操作的数据个数由向量寄存器的长度和数据类型共同决定。

ARMv7具有16个128位的向量寄存器，命名为q0~q15，这16个寄存器又可以分成32个64位寄存器，命名为d0~d31。其中qn和d2n、d2n+1是一样的，故使用汇编写代码时要注意寄存器覆盖。

使用NEON指令读写数据时需要对齐，ARM v8处理器将会支持不对齐的存储器读写，这将会扩大其应用范围。

6.2.8 SSE/AVX内置函数

SSE/AVX是Intel推出的用以挖掘SIMD能力的汇编指令。由于汇编编程太难，后来Intel又给出了其内置函数版本（intrinsic）。

SSE/AVX指令支持数据并行，一个指令可以同时多个数据进行操作，同时操作的数据个数由向量寄存器的长度和数据类型共同决定。如SSE4向量寄存器（xmm）长度为128位，即16个字节。如果操作float或int型数据，可同时操作4个，如果操作char型数据，可同时操作16个，而AVX向量寄存器（ymm）长度为256位，即32字节。

虽然SSE4/AVX指令向量寄存器的长度为128/256位，但是同样支持更小长度的向量操作。在64位程序下，SSE4/AVX向量寄存器的个数是16个。

SSE指令要求对齐，主要是为了减少内存或缓存操作的次数。SSE4指令要求16字节对齐，而AVX指令要求32字节对齐。SSE4及以前的SSE指令不支持不对齐的读写操作，为了简化编程和扩大应用范围，AVX指令支持不对齐的读写。

6.3 本章小结

本章介绍了常见的并行编程模型和目前主流的并行编程环境。了解了这些，我们就可以知道如何在具体的条件限制下选择合适的编程模型和对应的编程环境。

由于本章各小节内容相对比较独立，不易关联起来，因此在小结中笔者简单罗列主流的并行编程模型和并行编程环境的特点。

由于分类方式并非正交，很多并行编程模型之间都有重叠的部分，以下是常见的并行编程模型。

- 指令级并行：如果多条指令之间既没有数据和控制依赖，也没有结构化依赖，那么它们可以同时处理器的多个流水线上同时执行。

- 向量化并行：向量化并行是一种细粒度的并行计算，通常是对不同的数据执行一条同样的指令，或者说一个指令作用于一个数组/向量。向量化并行是数据并行的一个细粒度子集，在某些情况下，可认为向量化并行是指令级的数据并行。

- 易并行：易并行计算是指并行执行的多个控制流之间没有通信的并行，英文为embarrassingly parallel，直译则为令人尴尬的并行。

- 任务并行：任务并行是指每个控制流计算一件事或者计算多个并行任务的一个子任务。

- 数据并行：数据并行是指一条指令同时作用在多个数据上，那么可以将一个或多个数据分配给一个控制流计算，这样多个控制流就可以并行。

- 流水线并行：把任务分成多个阶段，各个阶段相互依赖，只能串行，而不同任务的阶段可以并行，这非常类似现代的指令流水线。

- 区域分解并行：将大的计算区域划分成多个小的区域，然后由一个控制流计算一个小的区域，那么计算大区域的所有控制流便可同时进行。

- 共享存储器并行：共享存储器并行是指所有控制流都能够访问一个共同的全局存储器，通过这个存储器来交换数据。

- 分布式存储器并行：对于通过网络互联的多机系统、存储器分布在不同节点机上，每个节点机运行各自的操作系统，拥有独立的物理地址空间。由于通过网络互联，控制流之间的通信只能通过网络数据分发实现。分布式存储并行模型通过显式的消息传递来交换数据。

常见的并行编程环境有：

- MPI：MPI（Message Passing Interface，消息传递接口）是一种消息传递编程环境。消息传递指用户必须通过显式地发送和接收消息来实现处理器间的数据交换。

- OpenMP：OpenMP是Open Multi-Processing的简称，是一个基于共享存储器的并行环境。OpenMP支持C/C++/Fortran绑定，也被实现为库。

- CUDA：CUDA是NVIDIA于2006年11月推出的，用于发挥NVIDIA GPU通用计算能力的编程环境，目前支持CUDA C和OpenCL（Open Computing Language）语言。

- OpenCL：OpenCL全称为Open Computing Language（开放计算语言），先由Apple设计，后来交由Khronos Group维护，是异构平台并行编程的开放的标准，也是一个编程框架，OpenCL的设计借鉴了CUDA的成功经验，并尽可能地支持多核CPU、GPU或其他加速器。OpenCL不但支持数据并行，还支持任务并行。同时OpenCL内建了多GPU并行的支持。

- OpenACC：OpenACC编译器依据C/C++/Fortran编写的编译制导语句，将并行区域的代码翻译成另一种语言的表示，如CUDA、OpenCL等。软件开发人员只要在不同的操作系统和主机设备之间编译OpenACC代码，就可以实现可移植性。

- NEON：NEON是ARM处理器上的SIMD指令，由于ARM在移动端得到广泛应用，目前NEON的使用也越来越普遍。

- SSE/AVX：SSE/AVX是Intel推出的用以挖掘SIMD能力的汇编指令。由于汇编编程太难，后来Intel又给出了其内置函数版本（intrinsic）。

第7章 并行算法设计方法

一个问题经常可以设计多个并行算法来解决，不同并行算法的性能可能差别很大；串行性能最佳的算法并行化后效果可能不理想；某些看来并行时间复杂度非常好的算法实现后的性能可能一般。软件开发人员可以利用硬件的特性来设计算法，也可以利用应用的某些特性来优化算法等，这些在某种程度上能够提升程序性能。如何设计一个实现后性能会非常好的并行算法，这区分了并行算法开发人员的能力层次。一个好的并行算法通常具有以下特点：

- 具有并行性的恰好是热点；
- 可扩展性好；
- 易于实现。

为性能考虑，应当让所有的控制流尽量自由地运行。除非必要，尽可能不要对控制流的执行顺序作限制。

通常并行算法的设计涉及几个部分：划分、通信、结果合并和负载均衡，本章将详细分析这几个部分，并简要说明并行算法设计的步骤、方法和一般准则。

7.1 划分

划分的目的是将计算任务分成多个部分，以便多个控制流同时处理。通常划分的对象有两种：一是计算任务，比如如何划分看电视和吃饭这个计算任务，以便能够同时处理，人类的做法是眼睛看电视的同时嘴巴在吃饭，这称为任务划分；另一种是计算数据，比如如何同时收割一亩地里的稻谷，可将一亩划分为十份，由十台机器同时收割，这称为数据划分。通常两者分别对应着任务并行和数据并行。任务划分和数据划分没有天然的鸿沟，很多任务既可以采用任务划分的方式并行处理，也可以采用数据划分的方式并行处理。

划分时，要注意和应用的特性相结合，这既可以降低编程难度，又能够减少通信，提升并行效率。比如通常将存储位置相邻的多个数据划分给同一个控制流处理，以利用现代处理器缓存对局部性的利用。再比如许多物理模拟算法将模拟区域划分成二维、三维网络，此时通常以分块的方式划分网格，网格可以是均匀的也可以是不均匀的，也可以是曲面，如图7-1所示。

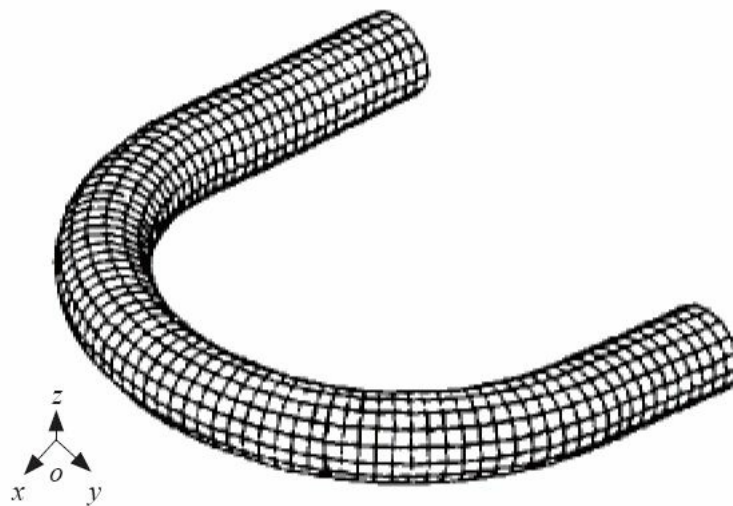


图7-1 网格

7.1.1 分而治之

分而治之是指将大问题分解成小问题，通过求解小问题，再将小问题的结果组合起来以解决大问题。串行编程中，分而治之经常导致递归算法，而递归算法通常还有优化空间。由于可使用多个控制流并行地解决小问题，因此分而治之和并行天生相符合。

串行算法设计中的最大连续子段和就可以使用分而治之的方法解决。最大连续子段和问题可描述为求一个长度为 n 的数组中所有连续 m ($0 \leq m \leq n$) 个元素和的最大值。如果将数组平均划分成两个子半段，那么其最大连续子段和必定是这三者的最大值：前半段最大子段和、后半段最大子段和、前半段后面和后半段前面部分之和，递归划分下去，直到数组中只有一个元素，问题就迎刃而解了。

许多支持并行编程的语言或库与分而治之有着天然的联系。在Linux系统编程中，`fork`函数会产生子进程，之后子进程和父进程共同工作，父进程调用`wait/waitpid`函数来等待回收子进程。在pthread_create函数产生子线程，子线程执行指定的工作直至结束，在主线程调用`pthread_join`来回收完成工作的子线程。OpenMP的`#pragma omp parallel`构造表示下面的一个代码块是由多个线程同时执行的，到块的结束处，主线程又回收所有其他的线程。在基于加速器的编程语言CUDA和OpenCL中，CPU线程启动一个内核意味着一个拥有大量硬件线程的网格创建并开始执行，CPU线程可以接着执行，也可以等待加速器硬件线程网格执行完成。

7.1.2 划分原则

对于一个问题，可能存在多种不同的划分方法，如何评价划分方法的优劣就变得非常重要，本节简要的说明一些划分原则。

- 尽量使算法映射到硬件上后，各控制流处理的数据或任务不相关。通常这一原则需要依据硬件条件做一些转变。比如CPU上线程在运行时会占据一个完整的执行单元，拥有自己的指令指针，因此应尽量使每个线程处理的任务不相关；而在GPU上执行时，由于多个线程共享一套执行单元，因此应尽量使得在一套执行单元上工作的多个线程和其他线程不相关；而在设计运行在硬件可编程（FPGA）处理器上的算法时，要优先使用流水线并行。

- 减少通信消耗。通信通常使得某些执行单元等待其他某些执行单元的操作，这引入了串行，会带来性能损耗，因此应尽量减少通信消耗。比如既可以使用临界区，也可以使用原子函数求一个键值对数组中的键相同的值的和，此时应当优先使用原子函数。

7.1.3 常见划分方法

一个问题可能有多个不同的划分方法可以解决，这些不同的划分方法可能会导致不同的实现方法，映射到硬件上后性能也会有所差别。对于一个具体的计算任务通常需要依旧划分原则来确定采用哪种划分方法。常见的划分方法主要有：均匀划分、递归划分、指数下降划分。本节将以网页服务器并行处理2万个访问请求为例解释这几种不同的划分方法。

1.均匀划分

均匀划分是指一次把计算任务或计算数据依据控制流数量划分成相等的几份，然后每个控制流处理一份。比如使用10台服务器来并行处理2万个访问请求，那么均匀划分2万个访问请求是指每台服务器都处理2千个访问请求。

对于任务划分来说，均匀划分通常使用在计算任务消耗的时间近似相同的情况下。对于数据划分来说，均匀划分通常使用在对每个数据的操作近似相同的情况下。

2.递归划分

递归划分指每次递归的将任务划分成几个相等或不相等的子任务，多次划分直到每个子任务都可以被简单处理。通常每次将任务划分成相等的两个子任务。

以使用10台服务器来并行处理2万个访问请求为例，先将2万个请求划分成两份，每份1万个请求，然后再递归的划分两个1万个请求的子任务，直到每份子请求都可以被1台服务器处理。实际上对于这个例子，更常见的做法是每次将任务划分成10个子任务，即第一次划分后共有10个子任务，每个子任务有2000个请求；第二次划分后共有100个子任务，每个子任务有200个请求；以此类推。

递归划分的优势在于和分而治之的算法设计思想联系非常紧密，许多分而治之设计的算法都导致最终递归划分计算或数据。

3.指数下降划分

指数下降划分是指后一次划分使用的任务数量都是前一次的一半，如果每一次划分的数据大小为 N ，那么下一次划分的数据大小为 $N/2$ ，再下一次为 $N/4$ ，以此类推。

以使用10台服务器并行处理2万个访问请求为例，假设第一次每台服务器处理1024个请求的子任务，那

么第二次每台服务器处理512个请求的子任务，直到任务被处理完。

7.1.4 并行性和局部性

现代处理器系统可分为计算系统和数据访问系统两个基本部分。为了并行算法性能好，应用应当使得各控制流的计算互不相关，这通常可以使得计算达到最优，却忽略了数据访问。现代处理器使用大量缓存来利用数据的局部性，并行算法的设计也应当考虑数据的局部性。

以计算为中心，设计的算法可能会出现数据访问局部性不好的情况，此时不妨从数据的角度来重新设计。由于现代处理器计算的速度比数据访问速度要快很多，因此设计算法时以数据为中心通常会是一个更好的选择。

数据划分时，要避免划分后，控制流相互访问彼此拥有的数据。对于一维数据划分来说，通常均匀划分即可；对于二维数据划分来说，可按行、按列或使用区域分解划分。

并行算法设计时，要兼顾计算的并行性和数据访问的局部性，而如何兼顾这两者考验着设计者的理论和实践功力。

7.2 通信

现实中易并行计算非常少，大多数并行算法都需要在控制流之间进行通信，这通常是因为要对某些计算步骤的结果进行合并处理。

相比串行算法来说，通信是并行算法引入的额外消耗，如果能够减少这种消耗就能够提高并行效率，获得更好的可扩展性。下面列出一些笔者常用的减少通信消耗的方法。

- 如果通信都具有局部性，那么应当在设计互连网络的时候就考虑这一点，以减少相邻控制流之间的通信代价。
- 减少通信次数，可通过将更多的代码并行化来实现，比如我们使用OpenMP并行了一个大循环里面的几个循环，那么每个小循环之间都会存在线程的建立、回收开销，如果把大循环内部的所有代码全都移到线程内部了，那么只需要创建、回收一次线程即可。依据Amdahl定律，串行比例决定了并行能够达到的最高加速比，将更多比例的代码并行化还潜在地提升了并行能够获得的性能提升上限。
- 使用异步通信算法，可以边计算边通信。比如在MPI程序中，可以边给其他进程传输数据边计算；在CUDA程序中，可以在GPU进行计算的同时，在CPU和GPU之间传输数据。
- 通信时需要注意粒度，通常要尽量使用大粒度的通信，即每次通信要传输尽量多的数据，以减少通信的准备时间。在某些情况下，可以通过合并多次小数据量的通信为一次大数据量的通信来减少这种损耗，小数据量的通信通常由延迟限制，而大数据量的通信由带宽限制，因为大数据量的传输，可以通过流水线并行掩盖内容传输之外的消耗（如打包、解包）。
- 将通信分散到计算系统的多个组件上避免某个组件成为瓶颈，也可能提升性能。如在机群环境下使用MPI编程时，使用异步数据传输且多个节点同时通信以避免某几个节点间的带宽成为瓶颈；在使用CUDA在NVIDIA GPU上编程时，CPU和设备间通信时使用CPU到GPU和GPU到CPU双向通信同时进行的方式。
- 尽量使用轻量级的通信方式。不同的通信方式的损耗不同，如原子函数的损耗就比锁和临界区小。在可以满足应用要求的前提下，应优先使用损耗小的通信方式。一些大数据算法通过迭代逐步更新权重，此时在更新不是很频繁的条件下可以允许某些对权重的更新失败，这样可以避免因通信导致性能大幅度下降。
- 减少由于通信导致的执行流等待，如使用多个局部锁，这样多个锁可并行执行。
- 某些看似必须通信的函数可能已经有了更好的实现版本，如并行前缀和，并行归约，此时应优先采

用。

如果在问题规模保持不变的情况下，通信时间没有随着处理器数目的增加而增加，那么扩展性就非常好。否则应当相应的增加问题规模，可避免通信成为瓶颈。

在MPI中，通信的方式就是消息传递，而在基于共享存储器的系统中，通信的方式相对比较多，常用的有：锁、临界区、原子操作、栅栏等。本节笔者将详细讨论与通信相关的操作的原子性、结果的可见性、函数的可重入性、线程安全及各种并行通信方法的优缺点，并尽量给出在何时应当选择何种通信办法的简单准则。

7.2.1 操作的原子性

原子性意味着操作必须作为一个整体执行，没有中间状态。如果多个控制流同时执行同一操作时能够产生确定的结果，通常也说这个操作是原子的。很难对原子性下一个确切的定义，因为不同领域、不同应用和不同人对“正确性”定义并不完全相同。

原子是一个来自物理学的概念，通常意味着它不可再分。但是在计算机高级编程语言中，几乎没有什么操作是不可再分的。在高级语言中，看起来很简单操作通常都可能会被映射到硬件的多个指令系列，因此通常应当假设“普通”计算机操作不具有原子性。

如下所示的代码在某些硬件和/或软件系统上也不是原子的。

```
int ret = a++;
```

编译后的代码可能大致如下所示：

```
int ret = a;
a = a+1;
```

为了提供并行需要的原子性，硬件会提供一些指令，比如锁总线、同步执行，等等。而操作系统也会提供一些底层的“原语”，这些“原语”可能是硬件指令的封装，也可能是一些高级算法。

如果一个操作不是原子的，那么多个控制流对其的操作会产生不同的结果，这称为“竞写”，在大多数情况下，竞写意味着算法设计出了问题，如代码清单7-1的pthread代码所示，最终count的结果可能是5个线程id的一个，而且每次运行结果也不一样。在某些其他情况下，竞写可能是性能优化的一种方式。

代码清单7-1 pthread代码

```
void* work(void* w){
    ((int)(*w)) = pthread_self();
}
int main(int argc, char *argv[]){
    pthread_t t[5];
    int count = 0;
    for(int i = 0; i < 5; i++){
        pthread_create(t+i, NULL, work, &count);
    }
    for(int i = 0; i < 5; i++){
        pthread_join(t+i, NULL);
    }
    return 0;
}
```

POSIX提供了原子函数，最新的C11和C++11也提供了原子操作的封装；OpenMP通过atomic构造对原子操作提供了支持，而CUDA和OpenCL则提供了一些整数和浮点原子函数。需要强调的是：无论哪一种语言

或库，对原子操作的支持都有限，通常是一些基本操作，比如整数和浮点加减，比较且设置（Compare and Set, CAS）等。

7.2.2 结果的可见性

在多个控制流同时执行的系统中，一个控制流产生的结果对自己总是可见的，这由处理器的缓存一致性协议保证。缓存一致性还保证如果一个控制流操作多个空间相邻的变量，无论底层的存储器组织如何，最终的结果都会得到保证。缓存一致性是如此的重要，以至于几乎所有的处理器都会在硬件上提供支持。

缓存一致性保证多个控制流操作不同的变量，其结果有保证，但是如果多个控制流操作一个或多个共享变量，那么结果的一致性都得不到保证。简单的说一个控制流写入的结果不能够保证被另一个控制流“看到”，这称为可见性。

现代处理器和并行编程语言通过“存储器屏障”（**memory fence**）来保证结果的可见性。只有在屏障调用后，调用屏障的控制流写入的结果才能够被其他控制流看到。

结果的可见性的细节还与高级语言的内存模型息息相关，由于目前常见的并行编程采用的都是弱一致性，因此可以假设：一个控制流对共享变量的更改不能保证被其他控制流看到。

7.2.3 顺序一致性

顺序一致性表示无论多个控制流以何种交错顺序执行代码，最终的执行结果都是确定的。如果一段并行代码满足顺序一致性，那么它就是“正确”的代码。顺序一致性包含原子性和可见性。开发人员可以认为：缓存一致性是保证在利用缓存性能的前提下，保证不因缓存的存在而产生不一致的结果；而顺序一致性保证，多个控制流交错执行的结果是一致的。缓存一致性和顺序一致性有个本质的区别：缓存一致性由处理器保证，而顺序一致性由开发人员保证。

为了优化程序的性能，处理器和编译器完全没有必要严格按代码规定的顺序来执行每一条指令，只要保证程序最终的结果和代码指定的一致即可，编译器使用的这种技术称为指令重排，而处理器采用的是乱序执行技术、流水线执行等。串行代码中这些优化对软件开发人员来说都是透明的，开发人员不用关心代码最终到底是按照什么顺序执行，因为处理器会保证优化后的程序的运行结果和原始程序的预期结果是一致的。在多核或多机上编译和运行时，由于种种原因，处理器和编译器还会继续做那些串行代码的优化，更重要的是编译器的这些优化遇到了并行程序需要对共享变量进行保护这一问题，编译器并没有能力完成这一工作，因此编译器无法保证顺序一致性，从而简单地忽略共享变量的保护的移植方法，使得并行程序的实际运行结果与期待的运行结果不一致，软件开发人员需要自己保证顺序一致性。

存储器屏障可以保证一个线程写入的结果会被其他线程看到，但是由于线程的进度可能不同，因此它也不能保证完全的顺序一致性。如代码清单7-2的示例代码就是一个初学者常见的错误，它并不能保证所有线程都输出6。其中threadId表示线程编号，fence（）调用存储器屏障函数。

代码清单7-2 错误示例

```
void memfenceExample(void* arg){
    float* data = (float*) arg;
    //every thread handle their own data
    ...
    if(threadId == 0){ data[0] = 6; fence();}
    printf("I am %d, my data[0] is %f\n", threaded, data[0]);
}
```

对于开发人员来说，顺序一致性更多的只是一种概念，它的实践意义远不如函数的线程安全和可重入。

7.2.4 函数的可重入与线程安全

在编写并行程序时，需要确保所有的库函数调用都是线程安全的，线程安全是指当多个线程并发/并行反复调用该函数时，它依旧能够产生正确的结果。对于软件开发人员来说线程安全意味着你可以并行地调用它而无须同步机制。一些编程语言的库，如Java中有明确的线程安全说明，而在C/C++库中通常缺少这一点，因此软件开发人员需要自己保证这点。如果不能确保的话，你应该给共享的资源加上同步机制。

通常线程安全的函数具有如下几个特征：

- **为共享变量的读写提供保护**。通常可以使用的机制是原子函数、锁和临界区。一般使用了C中的static变量的函数就不是线程安全的。标准C库中的errno变量就不是线程安全的，因为后一次函数出错的错误码会覆盖前一次的。
- **没有多个函数共享的状态**。C标准库中的rand函数不是线程安全的，因为它需要在多次调用间保持中间状态，一旦并行调用时，某些线程就有可能读到了另一个线程的中间运算数据。
- **没有返回指向静态变量的指针**。从并行线程中调用这类函数时，一个线程使用的结果很有可能被另一个线程悄悄覆盖。
- **没有调用非线程安全的函数**。非线程安全函数是可以传递的，如果函数a调用了非线程安全的函数b，那么a就不是线程安全的，同样调用a的函数也不是线程安全的。

通常使用位域来高效地存储位数固定且有限的变量，常用于操作系统内核/底层开发。一般来说，对同一个位域结构体内的不同位成员的访问不是线程安全的，因为它们往往不能对齐到能够原子读写的位置。

和线程安全类似的一个术语称为可重入，可重入的要求比线程安全高，它不但是线程安全的，还要求没有引用共享数据，这要求编程时不使用全局变量和static变量，没有存储器别名（这可以通过C99中受限的指针实现）。

7.2.5 volatile关键字

volatile的意思是“易变的”，C语言中的**volatile**效果是让编译器不要对这个变量的读写操作做任何优化，即不将其保存在缓存中，每次读的时候都直接去该变量的内存地址中去读，每次写的时候都直接写到该变量的内存地址中去，不做任何缓存优化。其典型的应用有下面几种：

- 避免寄存器或缓存对内存读写的优化。编译器常把频繁读写的变量保存到寄存器或缓存中，不用时或被替换出寄存器或缓存时再保存到内存中。如果某个内存地址中的值可能被另一个线程或是另一个设备读写，此时需要**volatile**关键字以保证能够读写到最新值。

- 同一个物理内存地址可能有两个或多个不同的引用（存储器别名问题，如*x=malloc (size) ; y=x; ）。例如两个控制流同时对同一个物理地址进行读写，那么编译器就不能假设这个地址只会有一个控制流访问而做缓存优化，所以程序员在这种情况下也需要把它定义为**volatile**的。

实际上，**volatile**是一种非常“鸡肋”的技术，普通的软件开发人员不应当使用它们。另一种鸡肋的技术是**memory fence**，但是这两者有一个共同的特点，那就是顶级的并行开发人才能够使用它们开发出许多高效的算法。

假设某个函数可以粗略地统计某个函数的访问数量，或当访问请求数量达到阈值，就将其值清零。当多个线程并行调用时，如果要得到精确的结果，需要使用锁或原子函数，此时可以使用**volatile**得到粗略的结果，如代码清单7-3所示：

代码清单7-3 使用**volatile**得到粗略的结果

```
volatile int counter = 0;
void access(){
    counter++;
    if(counter >= threshold) counter = 0;
}
```


7.2.6 锁

锁是指在使用时对持有它的线程有数量限制的对象。通常使用锁来实现排它性的操作，主要有排它锁和读写锁。排它锁是指只允许一个线程持有它，通常使用排它锁来达到不允许多个线程同时执行同一段代码的机制。读写锁是指只允许一个线程持有写锁而允许多个线程同时持有读锁的对象，如果有一个线程持有写锁，那么其他线程就既不允许持有写锁也不允许持有读锁。

由于排它锁一次只允许一个控制流执行代码，因此在多核或多处理器上可能会导致某些处理器或核心空闲，使得性能下降。解决这个问题通常有错开多个控制流加锁的时间，保证不会有多个控制流同时争夺锁，另一种方法是减少锁内的代码。

优化读写锁的一个有效方法是复制资源以分开读写，然后在必要的时候合并读写结果，如RCU等。RCU是Read Copy Update的缩写，它并不对资源的使用加锁，而是由写线程拷贝一份，然后在适当的时候将原来的资源替换成写线程拥有的那个版本。RCU在科学计算中使用并不多，但是其读写分离、适时合并的思想非常流行。

很多语言或库如OpenMP不支持读写锁，但是几乎所有的线程库都支持排它锁。Pthread支持排它锁和读写，OpenMP只支持排它锁。

锁能够使得原来不能并行的程序可以并行，但是锁的使用也有缺陷，如死锁、活锁及由锁竞争引起的性能瓶颈。

1.用锁防止竞写

在进行并行编程时，常需要使用锁来保护共享变量，以防止多个线程同时对该变量进行更新时产生数据竞写。例如，代码清单7-4中的两个线程可能同时执行“++”从而产生数据竞写，造成count最终值并不为5。

代码清单7-4 有数据竞写的代码

```
void* work(void* w){
    ((int)(*w))++;
}
int main(int argc, char *argv[]){
    pthread_t t[5];
    int count = 0;
    for(int i = 0; i < 5; i++)
        pthread_create(&t[i], NULL, work, &count);
    for(int i = 0; i < 5; i++)
        pthread_join(t[i], NULL);
    return 0;
}
```

出现结果不为5的原因主要是++操作不是原子的（实际上++由多个操作组成，这样多个线程就可能操作到中间数据，所以结果不正确）。

为了防止竞写现象，可以使用锁来保证每个线程对++操作的原子性。代码如代码清单7-5所示：

代码清单7-5 pthread锁防止竞写示例

```
pthread_mutex_t mutex;
void* work(void* w){
    pthread_mutex_lock(&mutex);
    ((int)(*w))++;
    pthread_mutex_unlock(&mutex);
}
int main(int argc, char *argv[]){
    pthread_t t[5];
    int count = 0;
    pthread_mutex_init(&mutex, NULL);
    for(int i = 0; i < 5; i++){
        pthread_create(&t[i], NULL, work, &count);
    }
    for(int i = 0; i < 5; i++){
        pthread_join(t[i], NULL);
    }
    pthread_mutex_destroy(&mutex);
    return 0;
}
```

实际上锁不但保证了每次只允许一个控制流执行锁内代码，还保证了控制流离开锁后，其更新的内容会立刻写回内存并使各处理器对其的缓存失效，这样其他线程也能够读到更新后的结果。其他的如临界区、原子函数等都具有这个特性。

2.锁竞争

在多线程程序中锁竞争是最主要的性能瓶颈之一。通过使用锁来保护共享变量能防止数据竞写，保证同一时刻只能有一个线程访问代码区。但是也要注意正是因为锁的这种特性造成了代码区的串行执行从而有可能成为并行程序的性能瓶颈。

从Amdahl定律知道，并行程序的性能很大程度上被串行执行限制，而由锁竞争引起的串行执行正是主要原因之一，因此如何减少锁竞争就很有意义。

3.避免使用锁

为了提高程序的并行性，最好的办法是不使用锁。从设计角度上来讲，锁的使用是为了保护共享资源。如果可以避免使用共享资源，那么自然避免了锁竞争造成的性能损失。很多情况下可以通过使用局部变量（如果需要在多线程程序中使用计数器，可以让每个线程先维护一个自己的计数器，只在程序的最后将各个计数器两两归并）和RCU技术，从而最大程度地提高并行度，减少锁竞争。

下面列出一些常见的用于避免、优化排它锁使用的方法：

· 使用读写锁或RCU。如果对共享资源的访问多数为读操作，少数为写操作，而且写操作的时间非常短，可以考虑使用读写锁来减少锁竞争。读写锁的基本原则是[同一时刻多个读线程可以同时拥有读锁并进行读操作；另一方面，同一时刻只有一个写线程可以拥有写锁并进行写操作；但是不允许读写同时进行](#)。读锁和写锁各自维护一份等待队列。当拥有写锁的线程释放写锁时，所有正处于读锁等待队列里的读线程全部被唤醒并拥有读锁以进行读操作；当这些读线程完成读操作并释放读锁时，写锁的等待队列中的某个写线程被唤醒，并拥有写锁以进行写操作，如此循环重复。换句话说，多个读线程和一个写线程将交替拥有读写锁以完成相应操作。需要注意的是，并不是所有的场景读写锁都具备更好的性能，大家应该根据测试结果来判断使用读写锁是否能真的提高性能，特别是要注意写操作虽然很少但很耗时的情况，此时有可能不适合读写锁，而适合RCU。

· 减少锁内操作数量。在实际程序中，有不少程序员在使用锁时把一些不必要的操作放在锁中，完全可以将某些操作移到锁外单独执行，以减少串行代码比例，增加并行度。如例子代码中，计算index的代码就完全没有必要放到锁中。

代码清单7-6 锁内有多余代码

```
lock()
int index = ...;//compute index of array
a[index]++;
unlock();
```

· 使用原子操作替代锁。某些操作完全可以使用更轻量级的原子操作来实现，根本不需要使用锁（如简单的加减可使用原子操作，求和可使用归约等）。如代码清单7-6所示的代码完全可以使用原子函数，修改后如代码清单7-7所示：

代码清单7-7 原子函数取代锁

```
int index = ...;//compute index of array
atomicAdd(a+index, 1);
```

· 使用无锁算法、数据结构。笔者并不推荐大家自己去实现无锁算法，建议直接使用语言或库自带的无锁算法（如OpenMP的归约算法），因为高性能的无锁算法的实现实在是太难了。

7.2.7 临界区

临界区是指只允许单个控制流执行的代码段。在临界区的开始到结束之间的代码不允许多个线程并行执行，这就提供了一种类似排它锁的机制。

临界区的大小会影响程序的性能，因此必要的时候应当尽量减少临界区内代码数量。

临界区和排它锁非常相似，通常可用排它锁实现临界区，但是临界区远远没有锁灵活和功能强大。比如，使用锁可以很容易实现使多个变量或使用某个变量的多个代码段串行执行，而临界区却无法实现这点。由于临界区的功能可被排它锁取代，因此很多基于多线程的库/语言并没有实现这个概念。

对于代码清单7-4的例子，如果使用OpenMP的临界区，代码如代码清单7-8所示：

代码清单7-8 使用OpenMP的临界区

```
void* work(void* w){
    ((int)(*w))++;
}
int main(int argc, char *argv[]){
    pthread_t t[5];
    int count = 0;
    #pragma omp parallel for
    for(int i = 0; i < 5; i++)
        #pragma omp critical
        {
            work(&count);
        }
    return 0;
}
```

7.2.8 原子操作

原子操作允许多线程并行地操作某个地址上的数据，且能够保证每个线程的操作都被原子执行，且结果对其他线程可见，由于代价比锁和临界区都要小，故经常优先用来实现一些同步机制。

在不同的处理器上、不同的语言中，原子操作的定义并不相同，如在CPU上，一些基本的内存读写操作本身已经由硬件提供了原子性保证。而在GPU上，连赋值这类操作也不能保证原子性。由于在不同的处理器上能够原子读写的数据类型不固定，因此即使是对共享变量的读写，也应当使用memory fence或原子操作。

需要说明的是，如果需要原子读写结构体内某个元素，那么就无须原子操作整个结构体，只需要原子读写这个元素即可。

一般情况下软件开发人员无须跟CPU提供的原子操作汇编指令直接打交道，只需要选择语言或者平台提供的原子操作API即可。而且使用封装后的原子操作更容易实现复杂的操作。

对于代码清单7-4的例子，如果使用OpenMP中的原子函数改写的话，只需要使用一条#pragma omp atomic伪指令即可得到正确结果，如代码清单7-9所示：

代码清单7-9 使用OpenMP中的原子函数改写

```
int main(int argc, char *argv[]){
    pthread_t t[5];
    int count = 0;
    #pragma omp parallel for
    for(int i = 0; i < 5; i++)
    #pragma omp atomic
    {
        count++
    }
    return 0;
}
```

7.2.9 栅栏

通常使用栅栏保证各控制流执行到同一代码处，以确定所有控制流都已经实现了一些操作，或者保证各控制流看到的某些存储器内容一致。

从行为上看，除非指定的控制流组内的所有控制流都已经到达栅栏调用处，否则执行到此处的控制流必须等待其他的控制流，直到所有的控制流都已经执行到调用处，所有的控制流会被唤醒以向前执行。从可见性上看，所有线程都到达存储器栅栏但是还没有向前执行时，所有线程看到的存储器内容是相同的。

如果各控制流的负载均衡不好，那么就有可能有的控制流早就到达栅栏，而有的控制流还有很多任务等待执行，这会显著减弱性能。

MPI中通过MPI_Barrier（）函数调用栅栏；OpenMP中通过#pragma omp barrier伪指令调用栅栏，；pthread通过pthread_barrier_t支持栅栏（实际上是计数栅栏）；CUDA支持线程块内栅栏；OpenCL和CUDA类似，支持工作组内栅栏。

假设线程需要得到其前后的计算结果才能计算下一步的结果，此时需要使用栅栏，简单的OpenMP代码如代码清单7-10所示：

代码清单7-10 OMP栅栏示例

```
#pragma omp parallel for
for(int i = 0; i < n; i++){
    int myid = omp_get_thread_num();
    mytemp = ?/computing
    temp_data[myid] = mytemp;
#pragma omp barrier
    result += temp_data[myid-1] + mytemp + temp_data[myid+1];
}
```

7.3 结果归并

对于非易并行计算来说，每个控制流计算后得到的结果可能并不是最终需要的，不同的控制流计算的结果之间可能存在重叠、依赖，等等。为了获得最终需要的结果，就要在各个控制流计算得到的结果的基础上进行处理。

通常并行计算会存在一个数据本地化的过程，即数据划分。而很多并行算法的最终结果是各控制流本地结果的函数，这需要对各控制流的本地结果进行归并处理，以得到程序最终的结果。

如多个处理器求某个数组的和，各个处理器求一个部分，这样每个处理器的结果都不是最终的结果，此时需要将各个处理器计算的结果求和。

结果归并可被视为控制流之间的通信。和通信一样，结果归并相比串行代码，也是额外的部分。结果归并的代码占用时间越少，性能就越向线性扩展迈进，可扩展性也就越好。

7.4 负载均衡

负载均衡是指通过调整计算在各个处理器上的分配，以充分发挥系统内处理器的计算能力，通常这意味着各个处理器近似同时结束计算。

好的并行算法应当具有好的负载均衡，因为负载不均衡会导致效率的降低。一般而言，随着处理器的增多，处理器之间出现负载不均衡的可能性就越大，负载均衡也越来越重要。

负载均衡也会消耗时间，如果随着处理器的增多，负载均衡耗时保持不变，加速比会近线性；如果随着处理器的增多，负载均衡耗时也增多，此时就要考虑处理器的数目和问题的规模问题，而不是处理器数量越多越好。

负载均衡算法主要分为两种：

- 静态负载均衡，是指在程序运行前，软件开发人员已经将计算资源分割为多个部分并保证能够均匀地把各部分计算分配给各个控制流运行，通常在作用在各个数据上的操作或处理任务的时间近似相等时采用；

- 动态负载均衡，是指在程序运行过程中，显式地重新调整任务的分布以达到负载均衡的目的。

由于动态负载均衡和静态负载均衡具有其自身优点，实践中经常混合使用。任务队列作为一种常用的动态负载均衡算法得到了广泛使用，因此本节单独予以说明。

动态负载均衡要解决的核心问题有两个：①是何时进行计算迁移，这其实是一个系统负载不均衡评价的问题；②是怎样进行任务迁移，即确定哪些控制流传递负载及如何传递。这两个问题的答案都与应用的具体情况密切相关。

与负载均衡联系在一起的是终止检测，它确定何时程序可以结束计算。

7.4.1 静态负载均衡

如果负载分配策略在程序运行前就已经确定了，这称为静态负载均衡。静态负载均衡通过估计程序分段的执行时间来安排计算。

相比动态负载均衡，静态负载均衡往往耗时比较少，在负载不均匀的情况下其效果会差一些。静态负载均衡减少了负载均衡导致的同步等开销。另外静态负载均衡算法易于做时间复杂度估计。目前常用的静态负载均衡算法有下面几种。

- **循环算法**，是指按照控制流索引顺序，依次将计算任务分配给各个控制流，当所有控制流都分配任务后再将任务分配给第一个控制流。在实践中，经常看到循环算法的使用，OpenMP构造也提供了直接支持。

- **随机算法**，对于每个控制流随机地选择执行任务。实践中这种算法很少用。

- **递归二分**，通过递归的将问题划分为两个子问题，将子问题交给控制流，并且尽量减少通信。递归二分算法在区域分解并行中应用非常广泛。

7.4.2 动态负载均衡

在运行时才确定负载分配的称为动态负载均衡，它是一种在应用程序运行期间依据前一段时间内各控制流对资源的使用状况，对各控制流间负载进行平衡的调度算法。通过实时分析并行系统的负载信息，动态地将任务在各处理机之间进行分配和调整，以消除系统中负载分布的不均匀性。

虽然动态负载均衡会在执行期间产生额外的通信开销，但在运行时控制流负载变化比较大的情况下比静态负载均衡更有效。

动态负载均衡有集中式的和分散式的。在集中式负载均衡中，某个控制流作为任务分配中心，其他控制流从任务分配中心获得计算任务，具有清晰的主从结构。而分散式负载均衡中，控制流之间没有主次之分，一个控制流可以从其余任何一个控制流获得任务，也可以将任务发送给其他的任何一个控制流。

在集中式的负载均衡中，主控制流拥有要被执行的任务集。当从控制流完成一个任务，向主控制流请求另一个任务时，任务就由主控制流发给从控制流。

由于易于编程和控制，目前大多数系统都采用集中式的负载均衡策略。而具体实现时，主控制流可以执行计算任务，也可以不执行计算任务。如果其执行计算任务，那么其决定任务如何重分配时，其他控制流可能就必须得等待了。

1.任务队列

顾名思义，任务队列是指一个装载着计算任务的队列，系统内的控制流从任务队列中获得计算任务。通常在计算任务运行时间差别比较大的时候应用任务队列，各控制流从任务队列中依据负载均衡的要求，以不同的粒度获得计算任务。

由于可能有多个控制流同时请求任务，因此需要加锁。实现中通常可以使用原子操作替代。

任务分发时，需要注意任务的粒度，如果粒度太小，通信时间可能成为瓶颈。如果粒度太大，可能因为没有足够的任务而导致负载不均衡。

有时有些计算任务比其他任务更重要，此时就使用优先级队列，这样先到达的控制流获得的计算任务优先级高，优先级高的任务就可以先得到处理。

任务列队使得负载均衡比较容易解决，只需要将等待处理的任务放入任务队列，各控制流从队列中获取任务，或某个管理控制流负责从任务队列中取出任务分发给其他控制流。

由于所有控制流都向任务队列取得任务，因此任务队列极易成为瓶颈，分布式队列部分解决了这个问题。

2.任务偷取/分布式队列

每个控制流维护自身的一个任务队列，通常从自己的队列中查找任务，当自己的任务完成时，便从其他控制流的任务队列中获取任务，这称为任务偷取，也称为分布式队列。由于只有在自身没有任务可以执行时，才有可能发生锁竞争，因此相比任务队列，这种方式潜在地提升了性能。

7.4.3 动态负载均衡算法的一般步骤

动态负载均衡算法的实现一般包括负载信息收集、重分配和负载迁移三个基本步骤。

1. 负载信息收集

负载信息涉及一个问题是如何表达各控制流的负载，比如一个粒子模拟系统的负载可以以粒子数表示，也可以以粒子之间的作用数目表示，这通常和应用的具体情况有关，而且很多时候需要综合考虑许多因素。负载信息收集主要有以下几种策略：

- 周期性收集策略：每隔一定的时间周期（具体时间间隔通常由应用的具体情况决定），系统就收集各控制流的负载。

- 命令驱动策略：当某个控制流发现需要了解其他控制流的负载状况以决定是否需要调用负载均衡算法时，它就会向负载控制中心发送一条命令，由负载控制中心负责收集系统内的负载信息。

周期性收集策略对系统状态变化的适应能力很差，适合负载不可能大幅度改变的系統；而命令驱动策略在负载可能大幅度改变的状况下效果更好。

2. 负载的重分配

收集到负载后，系统需要决定下一次计算时各控制流的负载，对于负载需要发生变化的控制流，一个重要的问题是缺少负载的控制流需要从其他那些控制流中获得负载，这是一个应用相关的问题。此时通常要综合考虑各控制流目前拥有的负载，以尽量减少负载在各控制流之间迁移导致的通信损耗。

通常负载重分配的实践方式是：从节点将负载信息交给主节点，由主节点重新分配，然后把重分配结果交给各个从节点。

3. 负载的迁移

任务迁移是指依据负载的重分配结果将节点间任务进行传递。负载平衡中的任务迁移，一般分为抢占式和非抢占式两种。

抢占式迁移是指把一个已部分执行的计算迁移到其他控制流上去。这要求提供待迁移任务目前的状态信息，以便接受任务的控制流能够接着执行。通常信息包括进程的虚拟映像、进程控制块、待处理消息、IO缓冲区、文件指针，等等。由于收集进程的状态信息通常是比较困难的，因而这种迁移要很大的系统开

销，故实际中并不多见。

非抢占式迁移则只迁移还没有开始运行的计算。目前大多数的系统实现的都是这种负载迁移，因为它的代价小、效率高，且便于编程实现。

在实现任务迁移时，要注意几点问题：

- 如果迁移到新节点的计算执行时必须频繁访问原结点的资源，那么就不应当迁移以减少迁移代价。
- 要尽可能避免计算迁移的抖动问题，抖动是指前一次从A迁移到B的计算和下一次从B迁移到A的计算是一样的。

7.5 本章小结

本章详细介绍了并行算法设计的基本步骤：划分、通信、结果归并和负载均衡。

通常使用任务划分来划分计算任务，而使用数据划分来划分数据，它们分别对应着任务并行和数据并行。常见的划分方法有均匀划分、递归划分和指数下降划分。在划分时，需要特别注意保证算法的并行性和数据访问的局部性。

在设计并行算法时，需要注意几个重要的概念：操作的原子性、结果的可能性、函数的可重入性和顺序一致性。

常见的并行程序通信方式有：锁、临界区、原子操作、栅栏和volatile关键字。

由于并行算法设计引入了划分，故并行程序各控制流计算的结果可能并非最终结果，因此需要对各个控制流的结果进行归并处理。

常用的负载均衡方式有：静态负载均衡和动态负载均衡。

由于引入了多个控制流、任务划分和通信，并行算法具有许多串行算法所没有的特点，这些特点中的一些可能会使得并行算法的性能潜在的比串行算法的性能要差，下一章笔者将会介绍并行算法主要有哪些特点会导致性能变差。

第8章 并行算法缺陷

并行和串行之间本质的区别只是由单个控制流执行代码转向了由多个控制流同时执行代码，但这种简单的转变导致了许多并行不同于串行的问题和可能的性能缺陷。这些问题和缺陷中的大部分是为了提升并行算法的性能，或让不能并行的代码能够并行而引入的，但是使用不当的话，却可能使程序性能降低。并行引入的问题和缺陷常见的有以下几种：竞写、死锁、饿死、原子操作、内存栅栏、缓存失效和伪共享。本章将分析这几种缺陷产生的原因，并试图给出一些常见的防治策略。

8.1 启动结束时间

由于缓存的刷新，操作系统启动、调度和结束进程或线程都是耗时的操作，故通常不建议在CPU上频繁创建和结束进程、线程，而是一次创建，多次重复使用。

代码清单8-1是一段OpenMP伪代码，由于OpenMP `parallel`语句在循环内，因此需要多次创建、销毁线程，故代码引入了比较大的线程创建、销毁开销，故通常改成代码清单8-2所示的版本。

代码清单8-1 多次创建、销毁版本

```
for(in titer = 0; iter < num; iter++){  
    #pragma omp parallel  
    {  
        doSomething();  
    }  
}
```

由于`#prgma omp parallel`在循环内部，那么需要`num`次创建、销毁线程，故初始化的代价非常大，如果把`#pragma omp parallel`移到循环外，此时就减少了线程多次创建、销毁的代价，为了正确性，还需要在每次循环完成前栅栏同步。

代码清单8-2 一次创建、销毁版本

```
#pragma omp parallel  
{  
    for(in titer = 0; iter < num; iter++){  
        doSomething();  
    }  
    #pragma omp barrier  
}
```

但是GPU的线程是非常轻量级的，其创建、销毁代价很少，通常鼓励创建大量线程。

8.2 负载均衡

由于很难保证分配给不同的控制流的任务计算量完全相同，故其计算完成时间可能并不完全相同，这使得先完成任务的控制流必须等待。通常解决负载均衡问题的方法是减少任务的粒度，然后在各控制流间动态分配任务。

如代码清单8-3所示的代码，由于内层循环次数 j 依赖外层循环次数 i ，故不同外层循环层次 i 的计算量并不相同，故使用静态负载均衡策略会带来负载不均衡问题，改为动态负载均衡策略即可，如代码清单8-4所示：

代码清单8-3 负载不均衡代码示例

```
#pragma omp parallel for schedule(static)
for(int i = 0; i < n; i++){
    for(int j = 0; j < i; j++){
        float diff = pos[i] ? pos[j];
        ...//computing
    }
}
```

代码清单8-4 缓解负载不均衡示例

```
#pragma omp parallel for schedule(dynamic)
for(int i = 0; i < n; i++){
    for(int j = 0; j < i; j++){
        float diff = pos[i] ? pos[j];
        ...//computing
    }
}
```

8.3 竞写

多个控制流操作共享变量时，非常容易出现竞写问题，发生竞写时的结果和多个控制流的运气有关，决定运气的因素有：硬件和操作系统的调度策略、存储器子系统的特性等。

代码清单8-5所示为一段pthread竞写示例代码：

代码清单8-5 竞写示例

```
void* work(void* index){
    int id = *((int*)index);
    data[id]++;
}
int index = 3;
for(int i = 0; i < 5; i++)
    pthread_create(t+i, NULL, work, &index);
```

代码本意是5个线程，每个线程都给data[3]加1，期望最终的结果是加5，但是实际结果在大多数情况下少于5（在单核机器上运行很有可能刚好增加5）。

通常解决竞写问题有两个策略。

- **不共享数据**，尽量使用控制流私有的局部变量，对共享变量加const限制符（如果是函数参数时，对值类型不起作用），在使用OpenMP时，尽量将变量指定为private并声明局部变量；在显式的线程环境中，优先使用局部变量，不要使用全局变量。代码清单8-5所示的代码可以将每个线程对data的更新保存起来，然后再一次增加到data上。

- **对共享数据的读写使用锁或原子操作等**，如在更新data时使用锁或者原子操作加以保护。

为了获得好的性能和可移植性，应用应当优先采用不共享数据。

8.4 锁

锁常用来解决竞写问题，使得原本不能并行的算法能够并行。由于锁会导致竞争锁的多个控制流串行运行，且多个控制流竞争锁会引入更多内存访问，故会导致性能下降。通常要求锁住的临界区要尽量小，这意味着两个方面：

- 时间上，锁被调用的次数应当尽量少，比如可以缓存对共享变量的更新，然后一次处理完成；
- 空间上，锁内的指令数要少，比如只锁住对结构体的某个元素的访问而不是对整个结构体的访问加锁。

8.4.1 死锁

死锁是指多个控制流相互拥有对方请求的资源的同时请求对方持有的资源，如果没有外力改变，这种状态将永远持续下去。一个简单的比喻是吃饭问题，假设甲乙太穷，他们总共只有一双筷子，他们每人各拥有一支筷子，相互等待对方放下手中的筷子，如果甲乙都不愿意放下手中的筷子，那么他们都不能吃上饭。死锁是一种算法设计缺陷，目前只能依靠软件开发人员自身的智慧，要求软件开发人员在设计算法的时候排除死锁的可能（如果Linux系统检查到存在死锁，则会杀死进程）。

从甲乙吃饭的例子可以看出，很明显死锁的发生必须存在几个条件：

- **资源访问互斥**。如果某个资源允许多个控制流同时访问，那么就不会发生死锁（两个人不能同时使用相同的筷子）。
- **请求对方拥有的资源**。如果不是这样的话，那么它就能独立执行（甲乙都要求对方手里的一支筷子）。
- **资源持有不可剥夺**。如果控制流对资源的持有可以剥夺的话，那么死锁就不能存在（如果甲觉得他够强壮，从乙手里抢了另外一支筷子，甲就能够吃上饭了）。
- **循环等待**。如果甲决定现在不吃饭了，扔下筷子，乙就可以吃饭了，乙吃完后，甲就可以吃饭了。

资源互斥访问和请求对方持有的资源，这两条就使得多个控制流形成环状结构，环内的控制流相互请求对方的资源，等待对方不再使用资源。

通常只要使得死锁存在的4个条件中1个不成立就能够解决死锁问题，但是目前还没有很方便的算法能够在程序运行前判定程序存在死锁（程序当前没有发生死锁并不代表没有死锁的可能，也许下一次运行就死锁了），因此主要依靠软件开发人员解决。

在代码清单8-6所示的代码中，假设pthread 1在获取mutex_1后正在尝试获取mutex_2，而pthread 0此时可能已经获取了mutex_2并正在尝试获取mutex_1，两个pthread就会因为获取不到自己想要的资源，且正占有着对方想要的资源不愿意释放而产生死锁。

代码清单8-6 死锁示例

```
void* work(void *w){
    int flag = (int)w;
    if(flag){
        //pthread 1
        pthread_mutex_lock(&mutex_1);
```

```

        pthread_mutex_lock(&mutex_2);
        fork2();
        pthread_mutex_unlock(&mutex_2);
        pthread_mutex_unlock(&mutex_1);
    }else{
//pthread 2
        pthread_mutex_lock(&mutex_2);
        pthread_mutex_lock(&mutex_1);
        fuor1();
        pthread_mutex_unlock(&mutex_1);
        pthread_mutex_unlock(&mutex_2);
    }
}
int main(int argc, char *argv[]){
    pthread_t t[2];
    for(int i = 0; i < 2; i++){
        pthread_create(t+i, NULL, work, i);
        .....;
    }
    return 0;
}

```

实际上，代码清单8-6的例子不一定会发生死锁，因为只要两个pthread步调不一致就不会出现占有mutex且请求对方拥有的mutex的情况。但是可能性已经存在，不能因为它此时没有发生而假设它以后不会发生，这也是并行编程不同于串行编程的一个方面，也是并行编程相比串行编程更难以调试的原因之一，因为有时即使程序运行一万次没有死锁，也有可能在第一万零一次运行时发生死锁。

在串行编程中，通常鼓励开发人员封装，增加小的函数调用以增加代码可读性和使程序更易于理解。然而在并行编程中这使得死锁的检测更为困难。以上面的代码为例，如果将fork1（）和fork2（）前后的加锁语句放到fork1（）和fork2（）中，死锁还是那么容易看出来吗？如果觉得还是比较容易的话，那再在中间加上600条语句呢？结果会是什么样子。

实际上没有好且简单的方法避免死锁，但有一条基本原则是保证各个线程加锁的顺序是全局一致的。还是以代码清单8-6的代码为例，如果pthread 1和pthread 2都是先对mutex_1加锁再对mutex_2进行加锁就不会产生死锁。

有时可能不知道对方加锁的顺序，这就没办法保证加锁顺序的全局一致性，此时可以通过释放已经持有的锁来解决。假设要加两个锁，在获得第一把锁后接着获取第二把，但是此时获取失败，如果不释放已经获取的第一把锁就有可能导致死锁。如果能够释放已经获得的锁，那么其他线程就有可能获取锁。

对基于锁的并行程序设计来说，尽量减少锁的数量不但可能提高性能（减少锁的数量也有可能降低性能，如分布式锁和层次锁），也能减少死锁发生的概率。如果只有一把锁，那么死锁发生的概率就几乎不存在。

8.4.2 活锁

除死锁外，多个线程的加锁、解锁操作还可能造成活锁。在代码清单8-7中，为了防止死锁的产生而做了如下处理。

当pthread 1获取mutex_1后，再通过调用pthread_mutex_trylock函数尝试获取mutex_2。如果pthread 1成功获得mutex_2，则trylock（）加锁成功并返回true，如果失败则返回false。pthread 0也使用了类似的方法。

这种方法虽然防止了死锁的产生，却可能造成活锁。例如，pthread 1获得mutex_1后尝试获取mutex_2失败，释放mutex_1并进入下一次while循环；如果pthread 0在pthread 1进行pthread_mutex_trylock（&mutex_2）的同时执行pthread_mutex_trylock（&mutex_1），那么pthread 0也会获取mutex_1失败，并接着释放mutex_2及进入下一次while循环；如此反复，两个线程都可能在较长时间内不停地进行“获得一把锁、尝试获取另一把锁失败、再解锁已获得的锁”的循环，从而产生活锁现象。当然，在实际情况中，因为线程之间调度的不确定性，最终必定会有一个线程能同时获得两个锁，从而结束活锁。但是活锁会产生性能损耗。

代码清单8-7 活锁示例

```
void* work(void* w){
    int flag = (int)w;
    if(flag){
        //pthread 1
        int done = 0;
        while(!done) {
            pthread_mutex_lock(&mutex_1);
            if (pthread_mutex_trylock(&mutex_2)) {
                fuck2();
                pthread_mutex_unlock(&mutex_2);
                done = 1;
            }
            pthread_mutex_unlock(&mutex_1);
        }
    }else{
        //pthread 0
        int done = 0;
        while(!done) {
            pthread_mutex_lock(&mutex_2);
            if (pthread_mutex_trylock(&mutex_1)) {
                fuck1();
                pthread_mutex_unlock(&mutex_1);
                done = 1;
            }
            pthread_mutex_unlock(&mutex_2);
        }
    }
}

int main(int argc, char *argv[]){
    pthread_t t[2];
    for(int i = 0; i < 2; i++)
        pthread_create(t+i, NULL, work, i);
    .....;
    return 0;
}
```

实际上如果两者的加锁顺序一致的话，就不可能出现活锁。另外常见的避免死锁的机制要求释放所有锁，而不是只释放部分锁。

8.5 饿死

饿死是指某个控制流一直得不到计算，本质上是一种负载均衡问题。大多数情况下可能和优先级有关，另外就是软件开发人员的失误。饿死的直观表现就是控制流一直在等待，就好像进入了死循环。

比如系统以优先级调度控制流，软件开发人员错误地为某个控制流分配了一个非常低的优先级，那么可能只要系统有任务执行，这个控制流就不可能得到处理器。

另外一种饿死是从任务的角度来说，也即某个任务一直存在系统中，没有控制流来计算它。

8.6 伪共享

为了保证缓存一致性，如果某个核心更新了缓存线中的数据，那么其他缓存该数据的核心就必须使得其他缓存该数据的缓存失效。如果存放在一个缓存线上的数据被多个核心使用，那么就会出现一种情况：假设有两个核心A、B，它们访问的数据分别为a、b，数据a、b在运行时被缓存到同一条缓存线中（但是并非同一地址），那么一旦A更新了a，B核心缓存a的缓存线将会失效，如果随后B访问b，则需要重新从内存中加载数据，反之如果B更新了b，那么A中的缓存线也会失效。以此类推，如果运气不好的话，所有的访问都可能通过内存完成，这称为“**伪共享**”。伪共享是一种更严重的缓存失效。另外，控制流的上下文切换也会导致缓存失效，上下文切换时，系统将被切换下的控制流的缓存写回内存，同时将切换上的控制流的上下文加载入缓存，有人也称这为“缓存污染”。如果由于某种原因导致频繁地进行上下文切换，那么缓存污染导致的代价就非常大。

伪共享本质上是缓存一致性导致的，它使得程序的性能从缓存的性能降到内存或共享缓存的性能（从几个周期上升到几百个周期）。由于每个线程更新sum时索引相邻，代码清单8-8的代码存在严重的伪共享问题。解决这个问题通常只要改变数据类型的大小、更改控制流分配数据的粒度或对齐以使得它们能够不被保存到同一个缓存线就可以了。

代码清单8-8 伪共享示例

```
#define LEN 268
#define NUM_THREADS 4
volatile double sum[NUM_THREADS];
double *data;
void* workOn(void *w){
    int id = (int)w;
    for( int i = id*LEN; i < (1+id)*(LEN); i++){
        sum[id] += data[i]/5;
    }
}
pthread_t t[NUM_THREADS];
for(int id = 0; id < NUM_THREADS; id++)
    pthread_create(t+id, NULL, workOn, id);
```

在使用原子操作解决竞写时，由于多个控制流同时更新一个数据，原子操作会导致严重的伪共享，这是原子操作慢的原因之一。

8.7 原子操作

原子操作通常保证多个控制流同时操作的结果和每个控制流依次串行的操作的结果一致，其概念在某种意义上与CPU流水线上的一次执行一条指令的抽象相冲突。现代CPU使用了很多非常聪明的手段让那些实际上不是原子的操作看起来是原子的。不过即使如此，也还是有一些指令是流水线必须延迟甚至需要刷新，以便一条原子操作成功完成。

由于频繁需要刷新流水线，原子操作的延迟通常在几百个周期，而且通常随着处理器数目的增加，代价也成倍增大。

不幸的是，原子操作通常只用于数据的单个元素。由于许多并行算法都需要在更新多个数据元素时保证正确的执行顺序，所以大多数CPU都提供了存储器栅栏。存储器栅栏也是影响性能的因素之一。

8.8 存储器栅栏

为了缩小存储器访问和处理器之间速度的差异，现代计算机使用了大量多层次的缓存。某个核心大量使用的变量将可能被保存在缓存中，允许快速访问相应的数据。这就使得某个共享变量在不同的核心之间存在一个或多个不稳定的状态（不稳定意味着不能保证状态到底是什么，只能说它可能是几种可能状态中的某一种）。假设某个核心需要读取另一个核心刚写入存储器的共享数据，实际上此时该核心看到的存储器内容可能并不是写入后的值，甚至编译器可能将读写优化（根本不写）。要保证多个控制流看到的存储器地址空间上的数据是完全一致的，需要保证：①所有的控制流都执行到相同的代码；②所有控制流对存储器地址空间的更新都已经完成且对其他控制流可见。存储器栅栏提供了这两种保证。在所有控制流都完成存储器栅栏调用且没有向前执行时，各个控制流看到的存储器地址空间的内容是完全一样的。如果某个控制流需要访问不久前另一个控制流更新后的数据，那么就需要调用存储器栅栏。

由于需要保证各个核心中的缓存数据都已经写回到内存中，因此其延迟也就几百个周期，但是如果存在负载不均衡的情况，即各个控制流并不是同时到达栅栏调用处时，则会导致某些核心等待。

存储器栅栏引入了两种类型的消耗：①控制流之间相互等待；②减弱了缓存的作用。

8.9 缓存一致性

在处理器还只有单核心的时代，缓存一致性便已经存在，其目的是保证内存中和缓存中数据是一致的。多核CPU在硬件中同样实现了这种机制，它确保了在多核CPU中，已被缓存的一个地址的读操作一定会返回那个地址最新的（被写入）的值。虽然硬件实现了缓存一致性，但是这并不意味着一个控制流对某个地址的写操作会立刻被另一个控制流看到，除非程序显式调用存储器栅栏，这样就能够让编译器大胆地优化代码而无须保证多个控制流之间的缓存一致性（单个控制流内部的缓存一致性是有保证的）。而分布式网络中的缓存一致性需要软件人员自己保证，幸运的是，很多语言和库都会提供现成的存储器栅栏和存储器屏障函数以供调用。

8.10 顺序一致性

对于顺序一致性，计算机科学界并没有一个统一的定义，本书是指多个控制流运行同一段代码可以得到唯一的结果，无论各个控制流是以任何顺序运行代码，结果应当是一致的。实际上，这并不是一个必须严格遵守的条件，比如某些软件可能要求必须两个控制流运行，但是允许某些结果有一定的误差。

从某种意义上说缓存一致性和顺序一致性是交织在一起的，因为线程的执行顺序无法确定，因此其对缓存的更新顺序就无法确定。硬件和编译器出于性能的考虑会对程序做违反顺序一致性的优化，而这种优化可能会影响多线程程序的正确性。

假设有两个线程（线程1和线程2）分别运行在两个处理器上，有两个初始值为0的全局共享变量x和y，两个线程分别执行下面两条指令：

初始条件: x=y=0;	
线程 1	线程 2
x++;	y++;
y=x;	x=y;

因为多线程程序是交错执行的，所以程序可能有如下几种执行结果：

第1种：x=2；y=2；（执行顺序x++；y=x；y++；x=y）；

第2种：x=1；y=1；（执行顺序x++；y++；y=x；x=y）；

第3种：x=2，y=2；（执行顺序y++；x=y；x++；y=x）；

第4种：x=2，y=2；（执行顺序y++；x++；x=y；y=x）。

8.11 volatile同步错误

如果两个线程需要同时访问一个共享变量，为了让其中两个线程每次都能读到这个变量的最新值，就把它定义为volatile。这个想法是如此直观和理所当然，然而却可能是错误的，这是volatile之所以引起争议的最大原因。如代码清单8-9所示，软件开发人员认为每个线程执行work函数时，volatile变量x都是最新值，实际上x可能只是中间结果。要解决这个问题可使用锁或原子函数，使用原子函数的版本如代码清单8-10所示：

代码清单8-9 volatile同步错误

```
volatile int x = 0;
void* work(void*){
    x++;
}
```

虽然volatile意味着每次读操作和写操作都是直接操作内存，但是volatile在现有C/C++标准中不保证原子性，许多用volatile来进行多线程同步的方案都是错的。

代码清单8-10 使用原子函数同步

```
volatile int x = 0;
void* work(void*){
    atomicAdd(&x, 1);
}
```

8.12 本章小结

本章介绍了并行程序相比串行程序具有的一些可能的、天然的缺点，并分析了如何缓解某些缺点的方法。

以下是并行常见的缺陷引发因素。

- 线程启动、结束时间：如果频繁地建立、结束线程，那么线程的开销就会比较大。
- 负载均衡：在某些情况下，静态负载均衡方法使得不同控制流的计算量并不相同，故有些线程会提前计算完成，然后等待。此时使用动态负载均衡策略通常会更好。
- 竞写：如果多个控制流不加锁，同时更新一个内存地址，那么可能出现有些写操作没有体现在结果上，导致错误。
- 锁：锁竞争会导致串行执行，因此降低性能。
- 伪共享：会将程序访问数据的性能由缓存降低到内存的性能。
- 原子操作：会刷新缓存线。
- 存储器栅栏：如果控制流的执行步骤不一致的话，存储器栅栏会强制要求执行快的控制流等待。
- volatile：由于语义的问题，使用volatile来同步的方案大多数都是错误的。

并行代码开发人员需要注意，小心避免由于并行引入的消耗导致并行程序性能不如串行程序。

第9章 并行编程模式实践

和串行算法的编程模式^[1]一样，并行算法实践也有类似模式的特征。和串行算法的编程模式不同的是：为了挖掘硬件的性能，并行算法的实践模式还与具体的硬件有关。

模式的意义在于挖掘算法的相似性，以同样的方式解决类似的问题。并行实践模式的更大意义在于如何更好地挖掘算法的并行性，并将算法模式映射到硬件上，以获得高性能。

考虑到目前的并行硬件主要是向量多核处理器，如X86和ARM CPU、NVIDIA和AMD GPU等，本章也主要基于这几种硬件设计实现。在实现语言的选择上，本章优先使用OpenMP和OpenCL，NEON内置函数，在必要时使用pthread^[2]。

对于本章讨论的每一个具体模式，笔者会试图给出其串行、SIMD并行、多核处理器并行和GPU并行示例代码。本章主要以求一个长向量的2范数（即长度）为例展示常见并行模式的使用，选择2范数为例是因为其简单且用于展示大多数实践模式已经足够。

[1] “算法模式”指的是程序算法层面的模式。它们表达的是，为了解决某一类实际问题，形成的固定的解决方法。

[2] POSIX线程（POSIX threads），简称Pthreads，是线程的POSIX标准。该标准定义了创建和操纵线程的一整套API。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API。

9.1 map模式

map模式是一种非常简单的易并行的实践模式。map实践模式直观的表述是：对每个数据施加同样的运算。从这个表述来看，map模式操作的数据表现出了数据并行的特征。

在应用map实践模式时，需要注意算法的粒度和硬件的粒度。

算法的粒度是指：某些应用在一种粒度上看是map模式，而在另一种粒度上看却不是map模式。例如，需要对300块不同的数据排序，那么对于数据块来说，这是一种map模式，因为对每一个块的操作是一样的（排序）。而对于块内的每一个数据而言，这又不是map模式，因为对每个数据操作可能并不相同。

硬件的粒度是指：主流的处理器的编程都可分成线程化和向量化两个层次，map模式既可以映射到线程上，也可以映射到向量上，但是这两种映射可能会导致不同的性能，需要注意。

1. 串行实现

计算向量的2范数，要求先求向量每个元素的平方和，然后开方。对每个元素求平方这一步是一个典型的map模式，其串行代码如代码清单9-1所示：

代码清单9-1 串行代码实现

```
inline float map(float d){
    return d*d;
}
void computeSqure(int len, const float* __restrict__ in, float* restrict out){
    for(int i = 0; i < len; i++){
        out[i] = map(in[i]);
    }
}
```

代码遍历向量中的每一个元素，然后对每一个元素应用map操作（平方）。

2. SIMD指令实现

因为SIMD指令都是作用在固定长度的数据上的，因此作用在数据上的map模式会很适合。具体实现时，在X86 CPU上可以使用SSE/AVX，在ARM CPU上可以使用NEON，而在GPU上可以使用OpenCL和CUDA。

NEON的内置函数vld1q_f32从内存中加载一段16字节、对齐的、32位浮点数据到128位浮点寄存器中，而vst1q_f32则将128位浮点寄存器写到对齐的、32位浮点类型的内存；vmulq_f32将两个128位浮点寄存器以32位浮点为单位相乘。

对向量的每一个元素求平方这一步使用NEON实现，代码如代码清单9-2所示：

代码清单9-2 SIMD实现

```
void computeSquireNEON(int len, const float* __restrict__ in, float* restrict out){
    int end = len ? len%4;
    for(int i = 0; i < end; i += 4){
        float32x4_t a = vld1q_f32(in+i);
        a = vmulq_f32(a, a);
        vst1q_f32(out+i, a);
    }
    for(int i = end; i < len; i++){
        out[i] = map(in[i]);
    }
}
```

考虑到向量长度可能不是4的整数倍，需要对最后的几个特殊处理，使用end变量来表示可使用SIMD处理的向量长度。

在计算量比较小的情况下，map模式进一步的优化方式主要是循环展开。

3.多核处理器上实现

如果map模式的作用单位是数据块，那么多核会很适合，因为只需要将一个或多个数据块映射到一个核心上即可。具体实现时，线程库、OpenMP或MPI都可以。

如果map模式的作用单位是数据，多核也同样适合，因为只需要将多个数据映射到同一个核心上即可。

对于求向量元素平方这一步，只需要使用OpenMP在代码清单9-2上增加一条OpenMP语句即可，如代码清单9-3所示：

代码清单9-3 OpenMP实现

```
void computeSquireNEON(int len, const float* __restrict__ in, float* restrict out){
    int end = len ? len%4;
    #pragma omp parallel for
    for(int i = 0; i < end; i += 4){
        float32x4_t a = vld1q_f32(in+i);
        a = vmulq_f32(a, a);
        vst1q_f32(out+i, a);
    }
    for(int i = end; i < len; i++){
        out[i] = map(in[i]);
    }
}
```

#pragma omp parallel for表示紧跟其后的for循环由编译器生成多个线程并行处理的版本，操作系统会将线程映射到处理器上。使用OpenMP能够很方便地实现线程级并行。

4.在GPU上实现

如果map模式的作用单位是数据块，那么GPU可能适合，因为只需要将一个或多个数据块映射到GPU的一个计算单元（Compute Unit，CU）上即可。由于GPU的每个CU是SIMD处理器，因此数据块内应当还有适用于SIMD的并行性，当然数据块内并行性可以不呈现map模式。具体实现时，CUDA、OpenCL或OpenACC都可以。

如果map模式的作用单位是数据，GPU也同样适合，因为只需要将多个数据映射到同一个SM（流多处理器，Stream Multiprocessor）上即可。

关于求向量元素的平方这一步，使用OpenCL的代码如代码清单9-4所示：

代码清单9-4 OpenCL实现

```
inline float map(float d){
    return d*d;
}
kernel void computeSquareOCL(constant int len, global const float* __restrict__
                             in, global float* restrict out){
    int tid = get_global_id(0);
    if(tid < len){
        out[tid] = map(in[tid]);
    }
}
```

向量是一维的数据结构，因此OpenCL的工作项布局也采用一维。

9.2 reduce模式

reduce模式表示从多个输入中产生一个输出，在不考虑误差的前提下，输出与输入的多个数据的顺序无关。比如求多个数据的和、最大值等。

reduce模式的一个变种称为segment_reduce，表示输出的数据并非只有一个，可能有多个，比如求图像像素的直方图。

并行reduce时，因为数据的计算顺序发生改变可能会导致串行的结果和并行结果有微小差别，这主要是由于浮点运算不满足结合律和分配率。

对求向量的2范数而言，前面已经使用map求取了各元素的平方，此次可以使用reduce模式来对平方的结果求和。

1.串行实现

使用reduce模式来对平方的结果求和，只需要将各元素平方的结果相加即可，如代码清单9-5所示。

代码清单9-5 串行实现

```
float computeSum(int len, const float* restrict out){
    float sum = 0.0f;
    for(int i = 0; i < len; i++){
        sum += out[i];
    }
    return sum;
}
```

2.SIMD指令实现

如果reduce模式的操作（指求最大值、和等）得到处理器的SIMD指令支持，那么SIMD实现会比较直接。

与代码清单9-5对应的NEON SIMD代码如代码清单9-6所示，其中vaddq_f32内置函数将两个长度为128位的浮点寄存器以32位浮点为单位相加。

代码清单9-6 SIMD实现

```
float computeSumNEON(int len, const float* restrict out){
    int end = len - len%4;
    float32x4_t sum = vdupq_n_f32(0.0f);
    for(int i = 0; i < end; i += 4){
        float32x4_t a = vld1q_f32(out+i);
        sum = vaddq_f32(sum, a);
    }
}
```

```

        float ret = 0.0f;
        for(int i = end; i < len; i++){
            ret += out[i];
        }
        ret += (sum[0]+sum[1]) + (sum[2]+sum[3]);
        return ret;
    }

```

变量end表示小于且最接近len、是4的倍数的值，比如len等于22，那么end就是20；如果len等于32，那么end就是32。

3.多核处理器上实现

从定义来看，reduce模式的作用单位是数据块，因此多核会很适合。可以将一个或多个数据块映射到一个核心上，然后串行或并行地处理多个数据块的结果。

OpenMP提供了reduction来支持reduce，其他语言则需要手动实现。

对于代码清单9-6，其对应的pthread多线程实现版本如代码清单9-7所示（去掉了任务分配代码，即初始化data）：

代码清单9-7 Pthread实现

```

typedef struct{
    int len;
    float *addr;
    float partRet;
} ArgData;
void* computeSumPthread(void* data){
    ArgData* arg = (ArgData*)data;
    Arg->partRet = computeSumNEON(arg->len, arg->addr);
    return NULL;
}
float computeSumNeonPthreadMulti(int len, const float* __restrict__ out){
    int index[NUM_THREADS];
    pthread_t t[NUM_THREADS];
    ArgData data[NUM_THREADS];
    //init data
    .....
    for(int i = 0; i < NUM_THREADS; i++){
        pthread_create(t+i, NULL, computeSumPthread, data+i);
    }
    for(int i = 0; i < NUM_THREADS; i++){
        pthread_join(t[i], NULL);
    }
    float sum = 0.0f;
    for(int i = 0; i < NUM_THREADS; i++){
        sum += data[i].partRet;
    }
    return sum;
}

```

每个pthread线程负责计算一部分数据的和，pthread线程计算完成后，主线程再负责把各个pthread线程的结果归并。由于每个线程需要知道自己要计算的数据，而pthread_create建立的线程执行的函数只能有一个void*参数，故需要使用一个结构体打包数据，ArgData即是为了完成这一目的。

4.GPU实现

从定义来看，reduce模式的作用单位是数据块，因此GPU会很适合，实现上有许多库实现了GPU上的reduce模式，如boost.compute。可以将一个或多个数据块映射到一个CU上，然后串行或并行地处理多个数据块的结果。

OpenACC提供了reduction来支持reduce，其他语言则需要手动实现。

由于OpenCL使用了两层的工作项组织方式，且OpenCL并没有提供线程组间同步的手段，因此本文使用OpenCL实现reduce模式时采用全局存储器来通信。

代码清单9-8 OpenCL实现reduce

```
//WGS is the size of Workgroup
inline void computeSumWorkgroup(local float* restrict out){
    int lid = get_local_id(0);
    for(int i = WGS/2; i > 0; i = i/2){
        if(lid < i){
            out[lid] += out[i+lid];
        }
        barrier(CLK_LOCAL_MEM_FENCE);
    }
}

void kernel computSumOCLStage(const int len, global float* restrict out,
                             float* restrict temp, local float* local_out){
    int gid = get_global_id(0);
    int globalSize = get_global_size(0);
    int lid = get_local_id(0);
    float sum = 0.0f;
    for(int i = gid; i < len; i += globalSize){
        sum += out[i];
    }
    local_out[lid] = sum;
    barrier(CLK_LOCAL_MEM_FENCE);
    computeSumWorkgroup(local_out);
    if(0 == lid) temp[get_group_id(0)] = local_out[0];
}
```

在实际程序中，需要运行computeSumOCLStage两次，第一次使用多个工作组调用，并将各个工作组的部分和保存到temp中；第二次调用时，只使用一个工作组，且以第一次调用的结果temp为输入。

9.3 结合map和reduce模式

如果不保存map的结果到一个数组中，而是直接用于作为reduce的输入，那么就节约了内存的读写时间。

在本节，笔者还是以求向量的2范数为例展示如何结合map和reduce以提升性能。

1.串行实现

结合map和reduce后串行代码如代码清单9-9所示：

代码清单9-9 结合map和reduce串行实现

```
inline float map(float d){
    return d*d;
}
float computeSqureSum (int len, const float* restrict in){
    float sum = 0.0f;
    for(int i = 0; i < len; i++){
        sum += map(in[i]);
    }
    return sum;
}
```

使用map，再使用reduce时，需要使用一个长度和in相同的内存out来保存中间结果。结合map和reduce后，out可被省略。从内存使用上分析，原来需要一次读in、一次读out和一次写out，而现在只需要一次读in即可。

2.SIMD实现

NEON支持浮点乘加指令，这可减少循环内指令的数量，如代码清单9-10所示，使用mla而不是mul+add。

代码清单9-10 结合map和reduce SIMD实现

```
float computeSqureSumNEON(int len, const float* restrict in){
    int end = len - len%4;
    float32x4_t sum = vdupq_n_f32(0.0f);
    for(int i = 0; i < end; i += 4){
        float32x4_t a = vld1q_f32(in+i);
        sum = vmlaq_f32(sum, a, a);
    }
    float ret = 0.0f;
    for(int i = end; i < len; i++){
        ret += map(in[i]);
    }
    ret += (sum[0]+sum[1]) + (sum[2]+sum[3]);
    return ret;
}
```



注意 由于多线程任务分发没有不同，因此代码没有区别，故并不列出多核处理器上的实现代码。

3.GPU实现

对于GPU来说，硬件的计算访存比很高，并且这个例子是访存密集型的，因此减少访存数量对性能提升会非常明显，因此通常建议结合map和reduce使用。如代码清单9-11所示：

代码清单9-11 结合map和reduce OpenCL实现

```
inline void computeSquireSumWorkgroup(local float* restrict out){
    int lid = get_local_id(0);
    for(int i = WGS/2; i > 0; i = i/2){
        if(lid < i){
            out[lid] += out[i+lid];
        }
        barrier(CLK_LOCAL_MEM_FENCE);
    }
}

void kernel computSquireSumOCLStage1(const int len, global float* restrict
in, float* restrict temp, local float* local_out){
    int gid = get_global_id(0);
    int globalSize = get_global_size(0);
    int lid = get_local_id(0);
    float sum = 0.0f;
    for(int i = gid; i < len; i += globalSize){
        sum += in[i]*in[i];
    }
    local_out[lid] = sum;
    barrier(CLK_LOCAL_MEM_FENCE);
    computeSumWorkgroup(local_out);
    if(0 == lid) temp[get_group_id(0)] = local_out[0];
}

void kernel computSquireSumOCLStage2(const int len, const global float*
restrict temp, float* restrict out, local float* local_out){
    int localSize = get_local_size(0);
    int lid = get_local_id(0);
    float sum = 0.0f;
    for(int i = lid; i < len; i += localSize){
        sum += temp[i];
    }
    local_out[lid] = sum;
    barrier(CLK_LOCAL_MEM_FENCE);
    computeSumWorkgroup(local_out);
    if(0 == lid) temp[0] = local_out[0];
}
```

由于第二次求和并不需要对输入做处理，因此笔者另外编写了computSumOCLStage2来做第二个阶段的处理，实际上，可以通过宏定义来避免这种重复。

9.4 scan模式

scan实践模式通常也被称为“前缀和”。在串行编程中，scan基本上无用武之地；而在并行算法中，scan大放异彩。scan可以作为许多算法的基础，如排序、划分，等等。

scan的并行实现的访存量大约是串行实现的访存量的1.5倍，而并行实现的计算量大约是串行实现的计算量的2倍。

1. 串行实现

串行实现非常简单，由于第*i*个结果是第*i*-1个结果加上输入的第*i*个数据的和，因此可以使用一个临时变量temp来保存累加的结果，然后每次往后迭代一个数据，代码如代码清单9-12所示：

代码清单9-12 Scan串行实现

```
void scan(int len, float* __restrict__ data){
    float temp = data[0];
    for(int i = 1; i < len; i++){
        temp += data[i];
        data[i] = temp;
    }
}
```

2. 多核实现

在多核上实现scan时，可通过3步：

第1步：每个核心计算一个或多个数据块的和，这可使用reduce模式达到，逻辑可参考代码清单9-7。此步需要读原始数据空间一次，并且写与线程数目相同的数据。

第2步：对每个核心计算的结果串行的做scan。由于计算量不大，这一步串行处理即可。此步需要读写线程数目相同的数据。

第3步：每个核心计算一个或多个数据块的scan，这一步可多个线程并行操作。此步需要读写原始数据空间各一次，同时需要读与线程数目相同的数据（第二步的结果）。

除了第2步，其他两步都可以并行计算。从算法可知，此算法需要读原始数据两次，写原始数据空间一次，而串行算法只需要读写原始数据空间各一次，因此并行算法访存量是串行算法访存量的1.5倍。

3. GPU实现

实现方法基本上和多核一致，但是由于GPU的CU还需要进一步的并行性，因此需要并行计算一个数据块的scan。目前已经有许多算法库可用，如基于CUDA的Thrust和CUB，基于OpenCL的boost.compute。

在GPU上实现scan模式基本步骤和在多核上实现相同，只是在每个CU上需要依据GPU的特点实现，细节比较复杂，本节就不加展示了，感兴趣的读者可参考boost.compute中的代码。

9.5 zip/unzip模式

对于串程序而言，由结构体组成的数组（简称AOS）对缓存的利用更好，表达数据也更为直观。而数组组成的结构体（简称SOA）则更易于并行化和使用处理器支持的SIMD指令。zip模式用来将数组组成的结构体转换成结构体组成的数组，而unzip模式刚好相反。

ARM处理器的SIMD扩展指令neon对zip/unzip模式提供了直接支持，其vld系列指令会将AOS类型数据加载成SOA寄存器；而vst系列指令则将SOA类型数据寄存器存储为AOS类型数据。

本节以一个具有3个32位浮点元素（3个元素为x、y和z）的结构体为例，展示如何使用zip/unzip模式。

1.串行实现

本节的串行代码实现如代码清单9-13所示：

代码清单9-13 Zip/unzip串行实现

```
inline float3 make_float3(float x, float y, float z){
    float3 xyz;
    xyz.x = x;
    xyz.y = y;
    xyz.z = z;
    return xyz;
}
void zip(int len, const float *x, const float *y, const float *z, float3 *xyz){
    for(int i = 0; i < len; i++){
        float xt = x[i];
        float yt = y[i];
        float zt = z[i];
        xyz[i] = make_float3(xt, yt, zt);
    }
}
void unzip(int len, float *x, float *y, float *z, const float3 *xyz){
    for(int i = 0; i < len; i++){
        float3 xyzt = xyz[i];
        x[i] = xyzt.x;
        y[i] = xyzt.y;
        z[i] = xyzt.z;
    }
}
```

2.SIMD实现

在X86 SIMD上实现时，可使用shuffle指令，而ARM NEON则提供了直接的支持。在NVIDIA Kepler GPU上，可通过shfl指令实现。本节展示如何使用NEON SIMD指令实现，为了简单起见，假设len为4的倍数，实现代码如代码清单9-14所示：

代码清单9-14 Zip/unzip SIMD实现

```
void zipNEON(int len, const float *x, const float *y, const float *z, float3 *xyz){
    for(int i = 0; i < len; i += 4){
```

```

        float32x4x3_t xyzt;
        xyzt.val[0] = vld1q_f32(x+i);
        xyzt.val[1] = vld1q_f32(y+i);
        xyzt.val[2] = vld1q_f32(z+i);
        vst3q_f32(xyz+i, xyzt);
    }
}
void unzipNEON(int len, float *x, float *y, float *z, const float3 *xyz){
    for(int i = 0; i < len; i += 4){
        float32x4x3_t xyzt = vld3q_f32(xyz+i);
        vst1q_f32(x+i, xyzt.val[0]);
        vst1q_f32(y+i, xyzt.val[1]);
        vst1q_f32(z+i, xyzt.val[2]);
    }
}

```

vld3q_f32内置函数从一个具有3个32位浮点分量结构体的数组中加载数据到3个128位向量中，其中结构体的每一个分量加载到一个向量中，顺序和结构体的声明顺序对应；vst3q_f32含义可以类推。

3.多核实现

由于处理xyz中的每个数据都和xyz中的其他数据无关，因此多核实现时，可让每个线程处理连续的多个数据，OpenMP代码如代码清单9-15所示：

代码清单9-15 Zip/unzip OpenMP实现

```

void zipNEON(int len, const float *x, const float *y, const float *z, float3 *xyz){
#pragma omp parallel for
    for(int i = 0; i < len; i += 4){
        float32x4x3_t xyzt;
        xyzt.val[0] = vld1q_f32(x+i);
        xyzt.val[1] = vld1q_f32(y+i);
        xyzt.val[2] = vld1q_f32(z+i);
        vst3q_f32(xyz+i, xyzt);
    }
}
void unzip(int len, float *x, float *y, float *z, const float3 *xyz){
#pragma omp parallel for
    for(int i = 0; i < len; i += 4){
        float32x4x3_t xyzt = vld3q_f32(xyz+i);
        vst1q_f32(x+i, xyzt.val[0]);
        vst1q_f32(y+i, xyzt.val[1]);
        vst1q_f32(z+i, xyzt.val[2]);
    }
}

```

#pragma omp parallel for表示使用多个线程并行处理循环。

4.GPU实现

对OpenCL实现来说，每个工作项处理一个元素即可，代码如代码清单9-16所示：

代码清单9-16 Zip/unzip OpenCL实现

```

void kernel zip(const int len, global const float *x, global const float
*y, global const float *z, global float3 *xyz){
    int gid = get_global_id(0);
    if(gid < len){
        float xt = x[gid];
        float yt = y[gid];
        float zt = z[gid];
        xyz[gid] = make_float3(xt, yt, zt);
    }
}

```

```
}  
void kernel unzip(constant int len, global float *x, global float *y, global  
float *z, global const float3 *xyz){  
    int gid = get_global_id(0);  
    if(gid < len){  
        float3 xyzt = xyz[gid];  
        x[gid] = xyzt.x;  
        y[gid] = xyzt.y;  
        z[gid] = xyzt.z;  
    }  
}
```

为了方便和数据直接映射，使用了一维的工作项。

9.6 流水线模式

流水线模式与指令流水线类似，通过并行使用不同硬件资源的操作来获得高性能。本节以从文件中加载向量做2范数运算为例展示其使用。

假设每次从文件中加载`len`个数据，那么总的加载次数`numIter`等于数据总数除以`len`。

1. 串行实现

串行实现示例代码如代码清单9-17所示，总共迭代`numIter`次，每次迭代会加载数据，然后计算。

代码清单9-17 流水线模式串行实现

```
float sum = 0.0f;
for(int iter = 0; iter < numIter; iter++){
    loadDataFromFile(file, iter, len, data);
    sum += computeSquireSumNEON(len, data);
}
```

使用`loadDataFromFile`函数加载数据主要使用的是存储器子系统，而使用`computeSquireSumNEON`计算主要是使用计算组件，在计算系统上看，这两者使用的是计算机的不同组件，故潜在的是可以并行操作。

2. 异步实现

从串行版本可以看到 $n-1$ 次迭代时计算平方和和 n 次迭代时加载数据之间不存在相关性，因此可以并行来做。具体实现可使用多线程、事件机制等，本节采用了最直接的异步IO机制，其代码如代码清单9-18所示，关于异步IO机制可参考glibc的文档。

代码清单9-18 异步IO实现流水线模式

```
loadDataFromFileAsync(file, 0, len, data0);
for(int iter = 0; iter < numIter-1; iter++){
    S1    dataBuff = iter % 2 ? data0 : data1;
    S2    loadDataFromFileAsync(file, iter+1, len, dataBuff);
    S3    syncPreviousLoad();
    S4    data = iter % 2 ? data1 : data0;
        computeSquireSumNEON(len, data);
}
```

S2处的`loadDataFromFileAsync`函数从文件中异步加载数据，此函数不会阻塞，发出异步IO操作后，控制会立即返回。S3处的函数`syncPreviousLoad()`会等待前一次循环的异步IO操作

(`loadDataFromFileAsync`)完成。为了使用异步IO，算法使用了两个缓冲区（有人据此称此类方法为双缓冲区），S1和S4即是为当前迭代选择缓冲区的逻辑。

3.GPU实现

使用GPU运算的版本和异步的版本没有本质区别，只是将CPU上的计算转移到了GPU上，如代码清单9-19所示。在CUDA中，GPU内核是异步的，利用这一点可以设计出不用异步IO的方法，如代码清单9-20所示：

代码清单9-19 CUDA+异步IO实现流水线模式

```
loadDataFromFileAsync(file, 0, len, data0);
for(int iter = 0; iter < numIter-1; iter++){
    dataBuff = iter % 2 ? data0 : data1;
    loadDataFromFileAsync(file, iter+1, len, dataBuff);
    syncPreviousLoad();
    data = iter % 2 ? data1 : data0;
    computeSquireSumGPU(len, data);
}
```

由于CUDA的内核执行是异步的，也就是说它会在计算执行完之前返回，主线程就可以接着往下执行。

代码清单9-20 CUDA实现流水线模式

```
loadDataFromFile(file, 0, len, data0);
for(int iter = 0; iter < numIter-1; iter++){
    data = iter % 2 ? data1 : data0;
    computeSquireSumGPU(len, data);
    dataBuff = iter % 2 ? data0 : data1;
    loadDataFromFile(file, 0, len, dataBuff);
}
```

首先加载一次迭代的数据到data0中，iter=0时，computeSquireSumGPU函数使用data0运算，由于computeSquireSumGPU函数是异步的，因此CPU不会阻塞，会立刻返回接着执行加载数据到data1中，故此时GPU计算和数据加载在同时进行。

9.7 本章小结

本章介绍了如何使用SIMD向量指令、多核多线程和GPU来实现map、reduce、scan和流水线等并行实践模式。通过这些并行实践模式的介绍、解说，希望读者能够通过模式解决一系列相关的并行计算问题。

第10章 如何并行遗留代码

遗留软件是企业的财富，在并行时代，如何并行化它们非常重要。一方面几乎所有遗留软件都是以标量串行程序的方式设计的，根本没有考虑到并行性；另一方面没有多少企业愿意放弃已有的标量串行软件转而从头开发一个全新的并行版本。以遗留的标量串行软件为基础，将其移植到并行平台上会是更好的选择。这就涉及并行化现有软件的方法，通常对一个标量串行软件进行并行化有两种办法。

- **部分并行化**。适合热点比较集中，涉及代码少，代码的模块化比较好的软件。通常部分并行化容易受制于原软件的模式和代码编写水平。这种方法的工作量小，而且容易调试和验证正确性。大多数程序软件满足80% - 20%定律（即程序20%的代码消耗整体运行时间的80%），部分并行化适合这类软件。

- **整体并行化**。适合热点分散且整体的并行性很好的软件。整体并行化需要了解软件的整体流程逻辑、逻辑架构、物理架构、数据分布、数据相关性和作用在数据上的处理，等等。对于遗留软件来说，接手并行化的开发人员可能并非原始的开发人员，这更增加了困难。

部分并行化和整体并行化的区别在于需要并行的代码行数占总代码行数的比例，两者没有本质的不同。实践中常采取两者的折衷：对热点相关但又具有并行性的多个部分并行化，有时也会并行化某些并行性不好的部分，因为这样可能会改善其他部分的性能或者使其他部分更易于并行化。在一些情况下，完成局部并行化后，原来没有并行化的部分成了瓶颈，为了性能，此时可能需要转化成整体并行化。

相比局部并行化来说，整体并行化要求更大的投资，失败的风险也更大，应当作为备选。并行化遗留软件要求软件开发人员了解原来的标量串行代码的方方面面，对他们来说这是一项挑战，这考验着他们的智慧和能力。

本章以笔者的经验，给出并行化标量串行遗留软件的基本步骤、一些要注意的问题和具体的实践方法，最后以笔者并行化word2vec为例结尾。

10.1 找出软件的计算热点

通常软件运行时间的80%消耗在20%的代码上，如果能够只并行化计算热点，那么就可以以小的时间投资换取大的收益。

关于并行化热点获得的最大性能收益可依据Amdahl定律估计。如果热点时间占总运行时间的80%，那么并行化最多可加速4倍。

1. 如何选择测时数据集大小

查找软件的热点时，对于程序的运行时间也有所讲究，通常既要求时间够长，又要求时间够短。要求时间长到能够分析到正确的结果，又要尽快得到结果。如果程序运行的时间太短，那么分析工具引入的测时误差可能“淹没”测时结果；如果程序运行的时间太长，那么开发人员可能一直在等待程序运行结束。

通常可以更改程序使用的数据集的大小来更改程序运行时间。对于某些软件来说，运行不同的数据集测得的热点并不相同，通常有如下一些原因：

- 由于分析工具分析方式导致结果不准。比如gprof需要使用-gpg编译选项，并且通过“插桩”（指插入计时函数）的方式来记录函数运行时间，这和程序实际运行时不一致，导致测得的结果不准。此时可以使用多个分析工具分析，然后把各个工具的结果进行汇总分析。

- 一些函数在数据量比较小的情况下运行时间刚好比分析工具的灵敏度小，分析工具没有将其记录在册。此时增大程序运行使用的数据量即可。

- 在不同的数据集下，程序运行的路径不同。如果软件中有一些依赖于数据的分支，那么测时结果就和程序执行的分支路径有关，进而和原始的数据有关。在这种情况下，通常使用最典型的情况下的数据集。

2. 使用哪种分析工具

关于如何找出软件的计算热点可以使用许多分析工具，如gprof、valgrind和Intel Vtune等。这些分析工具使用不同的测时方法和分析方法，因此测时结果有所差别。在各种分析工具的介绍上，它们各有各的优点。

在大多数情况下，笔者优先使用gprof，不是因为其测时结果准确，而是因为GCC自带、使用简单且大多数情况下测时结果足够准确。

如果笔者怀疑测时不准时，一般会使用其他工具验证，在必要时，笔者也会手工插入测时代码分析。

10.2 判断是否并行化热点

找出热点代码后，并不是要立即着手并行化，还需要知道两个问题的答案：

- 热点是否能够并行化，即热点是否具有并行性。如果热点不能够并行化，那有没有其他办法将热点转换成能够并行化的代码。

- 并行化热点代码的投资要多少，是否能够收回投资。软件项目需要多少人力、物力，需要多长时间，这些都需要有所考虑。

当然很多情况下，这个问题在查找程序的热点代码时已经大致得出结论。

1. 热点是否具有并行性

如果热点没有并行性，那么就需要考虑是否采用其他的具有并行性的算法或者是否能够通过重构算法使其具有并行性。如果最终确定算法没有并行性，那么就应当终止项目。

大多数时候并行性来自于循环（数据并行）或函数（任务并行）。如果循环操作的数据没有依赖，那么循环就应当可以并行；如果循环操作的数据有依赖，但是可以通过代码重构转化成没有依赖，循环也有并行性；如果循环操作的数据有依赖，但是可以通过通信方式（如锁、原子函数等）并行，且通信的代价很小，也认为可以并行。如果多个函数作用在不同的数据之上，那么认为可以使用任务并行；如果多个函数作用在相同的数据之上，但是可以通过通信方式解决冲突且通信代价很小，也认为可以并行。

另外，是否具有并行性和使用的软硬件环境有关，更确切地说是和环境使用的并行模式有关。如果一个软硬件环境使用的是数据并行，那么任务并行就被认为没有并行性；如果一个软硬件环境使用的是任务并行，那么数据并行就被认为没有并行性；如果一个软硬件环境使用的是流水线并行，那么数据并行和任务并行就会被认为没有并行性。

实际上，数据并行、任务并行和流水线并行，可以相互转化，如代码清单10-1所示的数据并行代码，可转变为代码清单10-2所示的任务并行代码，也可转变成代码清单10-3所示的流水线并行代码。

代码清单10-1 数据并行示例

```
void func(const float* __restrict__ data, size_t len){
    for(int i = 0; i < len; i++){
        process(data[i]);
    }
}
```

将代码清单10-1转变成代码清单10-2，其中两个p函数调用没有依赖，因此可以使用任务并行，开启两个任务分别处理两个p函数调用。

代码清单10-2 任务并行示例

```
void func(const float* __restrict__ data, size_t len){
    int start = len/2;
    p(data, start);
    p(d+start, len-len/2);
}
void p(const float* __restrict__ data, size_t len){
    for(int i = 0; i < len; i++){
        process(data[i]);
    }
}
```

将代码清单10-1转变成代码清单10-3，其中加载数据到变量cur中和process函数没有依赖，可以使用流水线并行。

代码清单10-3 流水线并行示例

```
void func(const float* __restrict__ data, size_t len){
    float prev = data[0];
    float cur;
    for(int i = 1; i < len; i++){
        cur = data[i];
        process(prev);
        prev = cur;
    }
    process(prev);
}
```

判断代码是否具有并行性是后面的步骤的基础，所以一定要慎重和小心，避免得出错误的结论。

2.是否能够向量化热点

如果热点具有并行性，那么需要考虑是否能够向量化热点。如果能够向量化热点，就能够获得更好的性能提升。另外，目前大多数时候向量化代码性能的稳定性比多线程代码要好，应当优先采用。目前向量多核处理器支持SIMD和SIMT向量化，SIMD和SIMT是数据并行的一个子集。如果热点具有数据并行性，那么基本上认为热点可以向量化。

不同的向量多核处理器支持的向量指令类型不同且有限，因此需要考虑向量化热点代码的话，那么主要操作指令在目标平台上是否得到支持也应当考虑。

一些向量处理器要求读写数据时必须对齐到向量的长度，而另外一些向量处理器允许不对齐的读写。如果目标向量处理器不允许不对齐的读写，那么是否有办法满足对齐要求就很重要。

不同的向量处理器具有不同的限制，要避免这些限制导致热点不能向量化。

3.并行代价和收益是否匹配

需要考虑并行化要付出多少人年的投资，如果购买设备、付给开发人员的薪水超过投资方给予的资金的话，那就得不偿失了。关于如何评估项目代价，读者可参考相关的书籍。

从某种方面说，这是最难的，因为实际上软件开发的估价一直都是个难题，无数个项目都是因为资金无以为继才不得不终止。

10.3 设计算法并实现

在完成所有准备工作后，就需要考虑向量化或并行实现的问题，这是软件开发人员最重要的工作，也是软件开发人员能力和水平的展示。

在设计算法时，需要选定软硬件开发环境，比如目标代码在何种硬件平台上运行、使用什么编程环境等。

10.3.1 选择何种工具进行向量化或并行化

应当依据热点代码的并行性不同选择不同的软硬件平台，如果热点代码的向量化并行很好，那么就应当考虑向量处理器；如果热点代码不能向量化，只能多线程并行，那么处理器的向量化性能就不应当考虑，而只考虑多线程性能好的处理器。

如果应用对运行平台有所限制的话，比如必须运行在ARM上，那么需要选择支持NEON和ARM处理器多线程的软件开发环境。

如果应用对延迟要求不高，而对吞吐量要求很高的话，GPU会是一个很好的选择。如果应用对延迟和吞吐量要求都很高，此时FPGA可能会更好。

关于编程语言，笔者并无偏好，但是现实情况是：几乎所有的向量化和并行程序都是使用C或Fortran编写，一些开发人员也尝试使用C++的高编码效率和代码的可扩展性。笔者认为：[向量化和并行开发人员需要有选择地使用C++的语言特性，且要尽量保证代码对C是兼容的。](#)

对于遗留代码来说，最好的选择是编译制导语句类型的并行化方案，如OpenMP、OpenACC等。通常编译制导语句能够保证并行化后的代码依旧能够编译成串行程序，另外相对来说编译制导语句使用简单。

10.3.2 重构热点代码

遗留代码通常不是采用并行化的方式来编写的，如果直接并行热点代码，可能比较容易出错，笔者更推荐先将热点代码重构成容易并行化的模式，然后再并行化。为了更好地利用并行计算硬件强大的计算能力，在并行化时可能需要改变原软件的数据组织方式和局部的计算逻辑，此时也可以先重构。

重构热点代码的主要目的在于使热点代码更易于并行化，因此重构后的代码要更好地表达并行性，如代码清单10-4所示的代码可重构为代码清单10-5所示内容。

代码清单10-4 循环拆分前

```
for(int i = 0; i < n; i++){
    a[i+1] = b[i] + c;
    d[i] = a[i] + e;
}
```

代码清单10-4重构为代码清单10-5之后，并行性就一目了然了。

代码清单10-5 循环拆分后

```
for(int i = 0; i < n; i++){
    a[i+1] = b[i] + c;
}
for(int i = 0; i < n; i++){
    d[i] = a[i] + e;
}
```

对于代码清单10-4，由于循环内读写了数组a中不同索引的元素，故不能直接并行化，而拆分循环修改成代码清单10-5之后，两个循环就都可以并行了。

重构热点代码的次要目的在于使得并行化后的代码性能更好，因此重构后的代码要能够更好地映射到目标硬件上，如代码清单10-6所示的代码重构为代码清单10-7所示的代码就能够更好地映射到X86或GPU的向量单元上执行。

代码清单10-6 重构前

```
for( int i = 0; i < n; i++){
    short int r = rgb_buf[3*i]; // load red
    short int g = rgb_buf[3*i+1]; // load green
    short int b = rgb_buf[3*i+2]; // load blue
    short int r_ratio = 77;
    short int g_ratio = 151;
    short int b_ratio = 28;
    short int y = r * r_ratio;
    y += g*g_ratio;
    y += (b*b_ratio);
    dest[i] = (y>>8);
}
```


在代码清单10-6中，读rgb_buf在X86处理器上向量化需要使用gather指令将间隔为3的几个数加载到向量中，如果将其划分成r_buf、g_buf和b_buf后，加载到向量化中就可以避免gather指令的使用，这可提升性能。

代码清单10-7 重构后

```
for( int i = 0; i < n; i++){
    short int r = r_buf[i]; // load red
    short int g = g_buf[i]; // load green
    short int b = b_buf[i]; // load blue
    short int r_ratio = 77;
    short int g_ratio = 151;
    short int b_ratio = 28;
    short int y = r * r_ratio;
    y += g*g_ratio;
    y += (b*b_ratio);
    dest[i] = (y>>8);
}
```

为了提升性能或更好地支持向量化，ARM处理器提供了vld3系列指令来支持代码清单10-6的向量化。

10.3.3 依据硬件实现算法

高级语言隐藏了硬件的实现细节，但是为了发挥硬件的性能，软件开发人员还是需要使用硬件友好的方式来编写代码。

从硬件组织来说，目前主流的处理器都是向量多核处理器，都支持多线程并行和向量化。比如X86多核处理器既支持多核并行又支持向量并行；用于手机和平板的ARM A系列处理器支持多核与NEON向量指令；NVIDIA和AMD的GPU使用CUDA与OpenCL支持多核并行及向量并行。和X86及ARM不同的是，CUDA和OpenCL只需要一份代码即可支持多线程与向量化。

从硬件指令集实现来说，不同的硬件具有不同的特性，同一指令（比如整数加法）在不同的硬件上其吞吐量和延迟不同；即使是同一向量处理器实现同一功能的不同指令的延迟和吞吐量也不相同；要发挥同一运算的峰值性能，在不同的处理器上需要的并行度不同。以32位浮点乘加为例，Intel Haswell处理器上，其延迟为5个周期，每核吞吐量为每周期16个运算，需要的并行度为80；ARM A15处理器上，其延迟为8个周期，每核吞吐量为每周期4个运算，需要的并行度为32；而NVIDIA Maxwell GPU上，其延迟大约为6个周期，每核吞吐量为每周期128个，需要的并行度为768；在AMD GCN GPU上，其延迟大约为12个周期，每核吞吐量为每周期64个，需要的并行度为768。

由于CPU和GPU之间的数据传输是通过PCI-e总线进行的，目前这种数据传输的速率是单向16GB/s（PCI-E X16 GEN3），实际的速度可达到10~12GB/s，相比内存带宽（50GB/s左右）和显存带宽（250GB/s左右），相当慢，另外GPU仅支持有限的数据结构。因此某些在CPU上使用部分并行化工作得很好的算法在GPU体系上实现部分并行化可能性会非常差，通常使用整体并行化以尽可能回避这些因素的影响。但是部分并行在pthread和OpenMP等基于CPU的串行程序并行化过程中通常是优先选择。

不但需要考虑使用的硬件的特点，还要考虑并行化后软件和原有软件的接口。以CUDA为例，由于显存和内存之间的数据传输及CUDA的合并访问要求，这可能要求重新组织数据，而重新组织数据后，能否再转化为原来的数据接口就很重要，一旦不能的话，就意味着并行化后的代码无法嵌入到原软件中。而如果不重新组织的话，性能如何就得重新考量（如目前的GPU硬件对乱序访问的支持并不好，如果不改变数据结构的话，可能需要CPU来做很多扫尾工作，可能难以体现GPU的性能优势）。

10.4 将实现后的代码嵌入原软件

在使用部分并行时，算法实现后，需要将相应的代码嵌入原软件中。通常使用两种方法：混合编译和使用动态链接库。

10.4.1 混合编译

混合编译适合同种语言实现，因为并行化后的代码和原软件使用同一种编译系统，只是并行化后的代码使用了更多的库。此时可以将并行后的代码单独编译成编译系统的中间语言或目标代码（如可重定位二进制或汇编代码）。然后在原软件编译中链接编译后的中间代码。下例是一个混合编译使用MPI并行化的部分代码，如下所示：

```
mpicc -c xx.c
g++ yy.c xx.o -lmpi -o yy
```

在科学计算实践中经常需要混合使用Fortran、C和C++的链接库，这和编程语言使用的名字毁坏机制和数据类型机制有关。有些语言内置了调用其他语言的支持，如Java就可通过Java本地接口（Java Native Interface，JNI）调用C++库。通常库的混合使用表现为以下几种具体情况：

- 如果使用C++调用C编写的库函数，需要在声明被调用函数时增加extern"C"前缀，然后直接链接该库即可。

- 如果使用C调用C++编写的库函数B，可另外编写一个函数A以代理对B的访问，函数A的声明和实现都增加extern"C"前缀，然后使用C++编译器编译函数A，在需要调用B的地方，都调用A，最后使用C编译器链接A即可。

- 如果使用Fortran调用C，则更为简单，因为Fortran的名字毁坏机制就是在函数名后增加一个下划线。如果需要使用Fortran调用C代码中的A函数，那么只需要编写一个C函数A_以代理对A的访问即可。

- 如果需要使用C调用Fortran中A函数，使用A_调用即可。

实际上混合编译最可能出问题的是数据类型，因为不同的编译器中对数据类型的长度、对齐和大小端有不同的要求。比如，由于数据类型对齐的要求不同，在某个结构体中的成员a，可能在一种语言下使用偏移量x访问，在另一种语言下可能就要使用偏移量y访问了，如果涉及数据的复制就会导致错误，甚至程序会正常运行，但是会悄无声息地产生错误的结果。笔者曾经遇到的一个问题是在一个C和C++的混合编程项目中，由于对一个结构体的声明忘记使用extern"C"修饰，导致调试半个月。对于面向对象语言C++来说，不同的编译器会以不同的方式组织对象的虚函数表，其混合编译的难度就更大了。

10.4.2 动态链接库

Linux和Windows系统都使用动态链接库将某些代码封装以方便调用、发布、更新代码和保护知识产权。动态链接库更改时无须重新编译程序，只要编译动态链接库即可，甚至可以在运行时替换动态链接库。由于动态链接库是独立于语言的（只依赖于操作系统关于其格式的规定），因此理论上多种不同的语言可以通过动态链接库通信。

程序启动时需要加载其链接的动态链接库，程序则会在运行时再调用动态链接库中的函数，软件开发人员可通过操作系统提供的函数来惰性加载节约启动时间，如Linux提供了系统调用`dlopen`，`dlsym`，`dlclose`等在程序运行时显式地调用动态链接库中的函数。

Linux下动态链接库的后缀名是`so`，在Linux下创建和使用动态链接库非常简单，下面的命令将`dl.c`文件编译成动态链接库`libdl.so`。其中`-fPIC`表示生成位置无关代码，位置无关代码可以加载到内存的任意地址运行。

```
gcc -shared -fPIC -o libdl.so dl.c
```

Linux下的动态链接库名字必须以`lib`开始。下面的命令展示了如何使用动态链接库，其方法和操作系统自带的动态链接库一致。

```
gcc -o man man.c -ldl.so -L.
```

其中`-l`选项指定链接的动态链接库的名字，不包括前缀`lib`。`-L`指定动态链接库所在的目录。如果动态链接库所在目录在系统环境变量`LD_LIBRARY_PATH`中，则不用显式使用`-L`选项指定动态链接库所在的目录。

实际上不是所有语言或编译器都支持动态链接库，但是很多语言都提供了调用C/C++代码的机制，因此这些语言都可以调用C/C++编写的动态链接库。

在实践中，开发人员可以将并行化的代码编译成动态链接库，然后原始程序调用动态链接库。如果需要更改或移植到其他平台，则只需要处理动态链接库即可。

与混合编译类似，不同的语言在处理数据类型和函数名的时候采用了不同的规则，使用动态链接库时需要注意这一点。



注意 为了避免并行化过程中引入新的bug，需要将原来通过的测试重新跑一遍。并行程序和串行

程序在某些方面具有本质上的不同，必要时需要多设计一些相关的单元测试。

10.5 示例：如何并行化word2vec

word2vec是Google在2013年开源的一款将词表征为向量的高效工具，采用的模型有CBOW（Continuous Bag-Of-Words，连续词模型）和Skip-Gram两种，本节主要关注连续词模型。word2vec通过训练，可以得出词的K维向量表示，因此把对文本内容的处理转换成K维向量的运算，向量的乘积可以用来表示文本语义的相似度。

word2vec本身通过pthread支持多线程并行处理，但是，并没有使用向量化和GPU加速，因此本节主要评估这两种可能性。

1.查找计算热点

由于gprof不支持多线程程序，因此笔者使用手动插入计时函数的办法。分析发现单线程下，CBOW模型计算耗时占程序总运行时间的95%左右，从文件中读取等待训练的数据耗时4%左右。整体来说，这是一个热点非常集中的项目，非常适合并行化。

2.是否能够并行化热点

word2vec中对每个输入的单词计算CBOW模型相关的代码，如代码清单10-8所示，为了减少篇幅，笔者去掉了不影响评估其能否被并行化的部分代码。

由于每个单词都进行同样的运算，因此适合多核并行，考虑到每个输入都需要更新共享变量，因此需要互斥访问。但是由于模型、领域相关的原因，导致多个线程更新同一个地址的可能性非常小，故对共享变量的更新无须互斥，只需要将它们声明成volatile就完全足够。

从代码清单10-8中看，大多数运算是向量与向量乘和向量与常数乘，layer1_size即向量的维数，大多数情况下大小为几百、几千，因此适合向量化，但是代码中有许多的分支和非对齐访问，因此向量化的效果可能并不如人意。

代码清单10-8 对输入单词计算CBOW模型相关的代码

```
// in -> hidden
for (a = b; a < window * 2 + 1 - b; a++) if (a != window) {
//omit unimportant code
...
    for (c = 0; c < layer1_size; c++) neu1[c] += syn0[c + last_word * layer1_size];
}
if (hs) for (d = 0; d < vocab[word].codelen; d++) {
    f = 0;
    l2 = vocab[word].point[d] * layer1_size;
    // Propagate hidden -> output
    for (c = 0; c < layer1_size; c++) f += neu1[c] * syn1[c + l2];
//omit unimportant code
```

```

...
    // Propagate errors output -> hidden
    for (c = 0; c < layer1_size; c++) neu1e[c] += g * syn1[c + l2];
    // Learn weights hidden -> output
    for (c = 0; c < layer1_size; c++) syn1[c + l2] += g * neu1[c];
}
// NEGATIVE SAMPLING
if (negative > 0) for (d = 0; d < negative + 1; d++) {
//omit unimportant code
...
    for (c = 0; c < layer1_size; c++) f += neu1[c] * syn1neg[c + l2];
//omit unimportant code
...
    for (c = 0; c < layer1_size; c++) neu1e[c] += g * syn1neg[c + l2];
    for (c = 0; c < layer1_size; c++) syn1neg[c + l2] += g * neu1[c];
}
// hidden -> in
for (a = b; a < window * 2 + 1 - b; a++) if (a != window) {
//omit unimportant code
...
    for (c = 0; c < layer1_size; c++) syn0[c + last_word * layer1_size] += neu1e[c];
}

```

3.使用GPU或X86实现

从关键代码来看，其既可以向量化，又可以多核并行。在支持非对齐向量访问的X86和GPU这两类向量多核处理器上都可以向量化和并行。

在GPU上实现可以使用OpenCL或CUDA，在X86上实现可以使用SSE/AVX内置函数。

4.算法实现

在具体实现时，由于GPU和X86硬件的不同，实现有比较大的差别。

在X86上实现时，可以通过一个线程处理多个待训练的词，而每个词内对layer_size1的运算则可以通过AVX内置函数加速。

在GPU上实现时，可以使用一个工作组处理一个待训练的词的方式，而每个词内对layer_size1的运算则可交给工作组内的一个工作项处理。由于GPU内部有许多特殊的存储器，如本地存储器（local memory）和常量存储器（constant memory），笔者将一些数据保存在了这两个存储器中，以获得更好的性能。

下面的伪代码展示了如何向量化代码清单10-8，其中的//done by SIMD or Workgroup 注释表示循环可被向量化或在GPU上可以使用一个workgroup处理。

```

// in -> hidden
for (a = b; a < window * 2 + 1 - b; a++) if (a != window) {
//omit unimportant code
...
//done by SIMD or Workgroup
    for (c = 0; c < layer1_size; c++) neu1[c] += syn0[c + last_word * layer1_size];
}
if (hs) for (d = 0; d < vocab[word].codelen; d++) {
    f = 0;
    l2 = vocab[word].point[d] * layer1_size;
    // Propagate hidden -> output
//done by SIMD or Workgroup
    for (c = 0; c < layer1_size; c++) f += neu1[c] * syn1[c + l2];
//omit unimportant code
...
    // Propagate errors output -> hidden

```



```

//done by SIMD or Workgroup
    for (c = 0; c < layer1_size; c++) neu1e[c] += g * syn1[c + 12];
    // Learn weights hidden -> output
//done by SIMD or Workgroup
    for (c = 0; c < layer1_size; c++) syn1[c + 12] += g * neu1[c];
}
// NEGATIVE SAMPLING
if (negative > 0) for (d = 0; d < negative + 1; d++) {
//omit unimportant code
...
//done by SIMD or Workgroup
    for (c = 0; c < layer1_size; c++) f += neu1[c] * syn1neg[c + 12];
//omit unimportant code
...
//done by SIMD or Workgroup
    for (c = 0; c < layer1_size; c++) neu1e[c] += g * syn1neg[c + 12];
//done by SIMD or Workgroup
    for (c = 0; c < layer1_size; c++) syn1neg[c + 12] += g * neu1[c];
}
// hidden -> in
for (a = b; a < window * 2 + 1 - b; a++) if (a != window) {
//omit unimportant code
...
//done by SIMD or Workgroup
    for (c = 0; c < layer1_size; c++) syn0[c + last_word * layer1_size] += neu1e[c];
}

```

10.6 本章小结

本章给出了并行化遗留代码的基本步骤，并指出每个步骤的常用方法和需要注意的事项，最后以如何并行化word2vec作为示例结束。

通常并行化遗留代码的基本步骤是：①找出软件的计算热点；②判断热点是否可并行化；③设计算法并实现；④将实现后的算法嵌入原软件；⑤重新测试。

在找出软件的计算热点时，需要注意选择测试数据集的大小和使用何种分析工具。

判断热点是否可并行化需要注意：热点是否有足够的并行性，进而需要分析热点是否能够被向量化，还需要评估并行化的代价公司是否能够承担。

在设计算法并实现的时候，需要注意选择合适的实现工具，在实现前要提前重构原始代码，最后要根据硬件来实现算法。

通常将实现后的代码嵌入原软件有两种办法：混合编译和使用动态并行库。

第11章 超级并行

单纯地使用MPI、OpenMP、CUDA或者OpenCL只能利用多机、多核或者GPU中一种硬件的计算能力，可能就不能利用计算系统拥有的全部计算能力，而本章将会关注如何能够充分发挥计算机系统多层次硬件的计算能力，本章称之为超级并行。

现实中的超级并行一般分为三层。

- 多机：通常使用网络将多个计算机连接起来。基于进程的MPI天生适合此类系统。编程时，通常要求大粒度。
- 多核：目前的处理器中，四核已经非常普通，八核也已经普及，服务器支持双路八核、四路八核的技术也将投入使用。在此类系统上编程，基于线程机制的OpenMP和pthread是首选的开发环境。
- 图形处理单元或SIMD向量：X86架构支持SSE/AVX指令，在这一层次需要利用生产商提供的汇编指令接口编程，但是通常编译器已经包装了它们，以内置函数的方式提供，以方便使用。而在图形处理单元、FPGA这类加速器上编程需要使用如CUDA、OpenCL等特殊设计的语言环境。

本章以稠密矩阵向量乘为例介绍如何将计算任务划分并映射到超级并行架构上，最后以矩阵乘法为例介绍实践中如何使用超级并行。

11.1 超级并行方式编程

由于不同级别、类型、组织的硬件上有不同的适合的编程方式，创造或者设计一种能够在多种类型、平台上运行的编程环境并非难事，但是它们通常不能充分发挥不同硬件的优势，因此超级并行方式通常涉及多种编程环境的混合。这种混合通常对应着硬件的某些特点，这种混合增加了编程的难度，但是却提供了灵活性和获得更好性能的方式。

由于使用了不同的函数库和编程语言，构建超级并行程序并不像普通的程序那样简单。解决这个问题基本思想是分段编译，然后链接。

超级并行引入了更多的复杂性，封装通常可以减少程序的复杂性，但是封装也会减弱软件开发人员的控制力，潜在地降低了性能，因此实践中应根据实际情况均衡。

超级并行硬件上通常具有多种功能的硬件，如何让这些硬件同时都在全速工作非常具有挑战性。比如在单机多核多GPU平台上，如何让众多GPU、CPU和IO设备都处于忙碌状态。

本节以矩阵向量相乘来说明各种超级并行编程方式的使用。

11.1.1 进程+线程

此类编程方式适合“多机+多核”的硬件组织模式，在这类系统上有多个计算机，可以为每个计算机分配一个进程，MPI是首选。每个计算机上有多个核心，为了发挥所有核心的计算能力，需要使用多线程机制在每个进程内分配多个线程，通常OpenMP是首选。

这要求首选将应用算法划分为大粒度的任务，以将每个任务分配给一个进程处理。然后将每个进程的任务划分为更小的任务，然后使用多个线程处理这些任务。由于进程间通信代价的高昂，应尽量将通信限制在一个计算机内的线程间。

应用可以使用进程id来决定进程计算的任务。然后包装每个进程处理的任务。

在矩阵向量相乘时，可以使用一个进程计算矩阵多行与向量乘积，而每个线程计算矩阵一行和向量乘积，或者多个线程计算矩阵一行与向量乘积。核心代码如代码清单11-1所示：

代码清单11-1 进程+线程计算矩阵向量乘

```
template<typename T>
void mxv(size_t numRows, size_t numColumns, const T *matrix, const T *v, T *r ){
#pragma omp parallel for
  for(int i = 0; i < numRows; i++){
    T sum = (T)0;
    for(int j = 0; j < numColumns; j++){
      sum += matrix[i*numColumns+j]*v[j];
    }
    r[i] = sum;
  }
}

int main(int argc, char *argv[]){
  checkMPIError(MPI_Init(&argc, &argv));
  int size, rank;
  checkMPIError(MPI_Comm_size(MPI_COMM_WORLD, &size));
  checkMPIError(MPI_Comm_rank(MPI_COMM_WORLD, &rank));
  float matrix[NUM_ROWS*NUM_COLUMNS];
  float v[NUM_COLUMNS], r[NUM_ROWS];
  float *all;
  //init matrix, v and malloc space for all
  .....
  mxv(NUM_ROWS, NUM_COLUMNS, matrix, v, r);
  checkMPIError(MPI_Gather(r, NUM_ROWS, MPI_FLOAT, all, NUM_ROWS,
    MPI_FLOAT, 0, MPI_COMM_WORLD));
  //post processes
  ...
  checkMPIError(MPI_Finalize());
  return 0;
}
```

由于两者可以使用同一种编译器编译，因此只需要在编译的时候链接上对应的动态链接库或编译选项即可。如上例的编译方式如下：

```
mpic++ mpiOpenmpmxv.cpp -DNUM_ROWS=1024 -DNUM_COLUMNS=1024 -fopenmp -lgomp
```

11.1.2 进程+GPU线程

实际中适合这种模式的硬件组织有两种情况：一是多机+GPU，这并不多见，因为在每个机群计算机上只有一块GPU的现象不常见，在此类系统上编程首选的是“MPI+CUDA/OpenCL”；二是“单机多核+多GPU”，这更适合“线程+GPU线程”。

通常使用一个进程控制一个GPU。实际上由于三层模型增加了复杂性，很多时候将之降级为二层模型，在机群里的每个GPU上分配一个进程，每个进程控制一个GPU。

MPI编译多GPU程序的思想是：将CUDA程序用C封装，再编译成目标文件；然后在mpicc编译时连接，最好保证GPU数和进程数相等，因为现在有一些老的GPU还不支持在单GPU上同时运行多个CUDA kernel函数。

在矩阵向量相乘时，可以使用一个进程计算矩阵多行与向量乘积，而每个GPU计算矩阵一行和向量乘积，或者一个GPU计算矩阵多行与向量乘积。

其中CUDA矩阵向量乘内核如代码清单11-2所示：

代码清单11-2 进程+GPU线程计算矩阵向量乘CUDA部分代码

```
void __global__ mxvBlock(int rowSize, int columnSize, int pitchItem, const
float* __restrict__ d_matrix, const float* __restrict__ d_vec, float*
__restrict__ d_r){
    unsigned int tid = threadIdx.x;
    extern __shared__ volatile float s_r[];
    float temp = 0.0f;
    for(int i = tid; i < columnSize; i += blockDim.x){
        temp += d_matrix[blockIdx.x*pitchItem+i]*d_vec[i];
    }
    s_r[tid] = temp;
    __syncthreads();
    for(int i = (blockDim.x>>1); i > 32; i >= 1){
        if(tid < i){
            s_r[tid] += s_r[tid+i];
        }
        __syncthreads();
    }
    if(tid < 32){ s_r[tid] += s_r[tid+32]; }
    if(tid < 16){ s_r[tid] += s_r[tid+16]; }
    if(tid < 8){ s_r[tid] += s_r[tid+8]; }
    if(tid < 4){ s_r[tid] += s_r[tid+4]; }
    if(tid < 2){ s_r[tid] += s_r[tid+2]; }
    if(tid < 1){ s_r[tid] += s_r[tid+1]; }
    if(0 == tid){ d_r[blockIdx.x] = s_r[0]; }
}

void runmxvBlockGPU(int rowSize, int columnSize, const float *matrix, const
float *v, float *r){
    float *d_matrix; size_t pitch;
    cutiSafeCall(cudaMallocPitch((void**)&d_matrix, &pitch,
columnSize*sizeof(float), rowSize));
    cutiSafeCall(cudaMemcpy2DAsync(d_matrix, pitch, matrix,
columnSize*sizeof(float), columnSize*sizeof(float), rowSize,
cudaMemcpyHostToDevice, 0));
    float *d_v;
    cutiSafeCall(cudaMalloc((void**)&d_v, columnSize*sizeof(float)));
    cutiSafeCall(cudaMemcpyAsync(d_v, v, columnSize*sizeof(float),
cudaMemcpyHostToDevice, 0));
    float *d_r;
    cutiSafeCall(cudaMalloc((void**)&d_r, rowSize*sizeof(float)));
    int blockSize = 256;
```

```
mxvBlock<<<rowSize, blockSize, blockSize*sizeof(float)>>>(rowSize,
columnSize, pitch/sizeof(float), d_matrix, d_v, d_r);
cutilSafeCall(cudaThreadSynchronize());
cutilSafeCall(cudaMemcpyAsync(r, d_r, rowSize*sizeof(float),
cudaMemcpyDeviceToHost, 0));
cutilSafeCall(cudaFree(d_v));
cutilSafeCall(cudaFree(d_matrix));
cutilSafeCall(cudaFree(d_r));
cutilSafeCall(cudaDeviceReset());
}
```

使用下面的命令将其编译成.o文件：

```
nvcc -c cudamxv.cu
```

对应的MPI程序核心代码如代码清单11-3所示：

代码清单11-3 进程+GPU线程计算矩阵向量乘部分MPI代码

```
void runmxvBlockGPU(int rowSize, int columnSize, const float *matrix, const
float *v, float *r);
int main(int argc, char *argv[]){
    checkMPIError(MPI_Init(&argc, &argv));
    int size, rank;
    checkMPIError(MPI_Comm_size(MPI_COMM_WORLD, &size));
    checkMPIError(MPI_Comm_rank(MPI_COMM_WORLD, &rank));
    float matrix[NUM_ROWS*NUM_COLUMNS];
    float v[NUM_COLUMNS], r[NUM_ROWS];
    float *all;
    //init matrix, v and all
    int num;
    cudaGetDeviceCount(&num);
    cudaSetDevice(rank%num);
    runmxvBlockGPU(NUM_ROWS, NUM_COLUMNS, matrix, v, r);
    checkMPIError(MPI_Gather(r, NUM_ROWS, MPI_FLOAT, all, NUM_ROWS,
MPI_FLOAT, 0, MPI_COMM_WORLD));
    //post process
    checkMPIError(MPI_Finalize());
    return 0;
}
```

使用下面的命令将其编译并链接CUDA内核函数：

```
mpic++ mpiCUDAmxv.cpp -DNUM_ROWS=1024 -DNUM_COLUMNS=1024 cudamxv.o -lcudart
-L/usr/local/cuda-5.0/lib64/
```

有时在运行时，程序会抱怨找不到相关的动态链接库，此时只需要将该动态链接所在路径写入/etc/ld.so.conf文件中，然后运行ldconfig命令即可。

11.1.3 线程+GPU线程

这种模式比较常见，硬件组织通常是多核+多GPU，使用的编程环境首选OpenMP+CUDA/OpenCL。此时通常使用的模型是一个线程控制一个GPU。

编程时需要将任务划分为更小的任务，每个线程处理一个或多个。然后线程将任务交给GPU处理。由于硬件体系结构的不同，导致编程时考虑的因素更多。

编译OpenMP+CUDA/OpenCL程序的思想是：首先将CUDA/OpenCL包装成C函数，再编译成目标文件或动态链接库，然后在OpenMP程序中调用它即可。

在矩阵向量相乘时，可以使用一个线程计算矩阵多行与向量乘积，而每个GPU计算矩阵一行和向量乘积，或者一个GPU计算矩阵多行与向量乘积（更优）。其核心OpenMP代码如代码清单11-4所示：

代码清单11-4 线程+GPU线程示例

```
void runmxvBlockGPU(int rowSize, int columnSize, const float *matrix, const
float *v, float *r);
int main(int argc, char *argv[]){
    float matrix[NUM_COLUMNS*NUM_ROWS];
    float v[NUM_COLUMNS], r[NUM_ROWS];
    int deviceNum;
    cudaGetDeviceCount(&deviceNum);
    #pragma omp parallel
    {
        cudaSetDevice(omp_get_thread_num()%deviceNum);
        #pragma omp for schedule(dynamic, 1)
        for(int i = 0; i < NUM_ROWS; i += PATCH){
            runmxvBlockGPU(PATCH, NUM_COLUMNS, matrix+NUM_COLUMNS*i, v, r+i);
        }
    }
    return 0;
}
```

可以使用如下的命令编译并链接CUDA内核函数：

```
g++ openmpCUDAmxv.cpp -DNUM_ROWS=1024 -DNUM_COLUMNS=1024 -DPATCH=256
cudamxv.o -lcudart -L/usr/local/cuda-5.0/lib64 -fopenmp
```


11.1.4 线程+向量指令

这种模式适合在多核X86和ARM处理器上使用，线程技术可以发挥多核的计算能力，而向量指令又可以发挥核心内向量指令的计算能力，编程方式首选OpenMP+SSE/AVX/NEON。

在矩阵向量相乘时，可以使用一个线程计算矩阵多行与向量乘积，而矩阵一行和向量乘积则由向量指令来计算。核心代码如代码清单11-5所示：

代码清单11-5 线程+SSE向量指令示例

```
void mxvSSEOpenmp(const int rowSize, const int columnSize, float *matrix,
float *v, float *r){
    __m128 *mv = (__m128*)v;
    __m128 *mm = (__m128*)matrix;
    #pragma omp parallel for
    for(int i = 0; i < rowSize; i++){
        __m128 re = _mm_set_ps(0.0f, 0.0f, 0.0f, 0.0f);
        for(int j = 0; j < columnSize/4; j++){
            re = _mm_add_ps(re, _mm_mul_ps(mm[i*columnSize/4+j], mv[j]));
        }
        float __attribute__((aligned(16))) a[4];
        _mm_store_ps(a, re);
        r[i] = a[0] + a[1] + a[2] + a[3];
    }
}
```

编译命令和编译普通的OpenMP程序一致，如下所示：

```
g++ openmpSSEmxv.cpp -lgomp -fopenmp -O3
```

11.1.5 进程+线程+向量指令

这种模式适合在X86机群上使用，为每个计算机分配一个进程，进程中再为每个核心分配一个线程，每个线程中使用向量指令以充分发挥机群的计算能力。通常使用MPI+OpenMP+SSE/AVX环境编程，相比双层结构，三层结构更为复杂，更难驾驭。

在矩阵向量相乘时，可以使用一个进程计算矩阵多行与向量乘积，而每个线程计算矩阵一行和向量乘积，然后向量指令实际负责计算矩阵一行与向量乘积。核心代码如代码清单11-6所示：

代码清单11-6 进程+线程+向量指令示例

```
template<typename T>
void mxv(size_t rowSize, size_t columnSize, const T *matrix, const T *v, T *r){
    __m128 *mv = (__m128*)v;
    __m128 *mm = (__m128*)matrix;
#pragma omp parallel for
    for(int i = 0; i < rowSize; i++){
        __m128 re = _mm_set_ps(0.0f, 0.0f, 0.0f, 0.0f);
        for(int j = 0; j < columnSize/4; j++){
            re = _mm_add_ps(re, _mm_mul_ps(mm[i*columnSize/4+j], mv[j]));
        }
        float __attribute__((aligned(16))) a[4];
        _mm_store_ps(a, re);
        r[i] = a[0] + a[1] + a[2] + a[3];
    }
}

int main(int argc, char *argv[]){
    float __attribute__((aligned(16))) matrix[NUM_ROWS*NUM_COLUMNS];
    float __attribute__((aligned(16))) v[NUM_COLUMNS], r[NUM_ROWS];
    checkMPIError(MPI_Init(&argc, &argv));
    int size, rank;
    checkMPIError(MPI_Comm_size(MPI_COMM_WORLD, &size));
    checkMPIError(MPI_Comm_rank(MPI_COMM_WORLD, &rank));
    float *all;
    //init matrix, v and all
    mxv(NUM_ROWS, NUM_COLUMNS, matrix, v, r);
    checkMPIError(MPI_Gather(r, NUM_ROWS, MPI_FLOAT, all, NUM_ROWS, MPI_
        FLOAT, 0, MPI_COMM_WORLD));
    checkMPIError(MPI_Finalize());
    return 0;
}
```

其编译命令为：

```
mpic++ mpiOpenmpSSEmxv.cpp -DNUM_ROWS=1024 -DNUM_COLUMNS=1024 -fopenmp
```

11.1.6 进程+线程+GPU线程

还有一种三层模式是使用GPU取代向量指令，相比于向量指令，GPU要求更大的并行粒度。编程环境是MPI+OpenMP+CUDA/OpenCL。由于硬件结构的差异，使用这种模式并行化相比X86有许多不同。

在此类系统上编程时，可采用每个计算节点机上分配一个进程，每个进程中有多个线程，每个线程控制一个GPU的模式。

在矩阵向量相乘时，可以使用一个进程计算矩阵多行与向量乘积，而进程中的每个线程计算矩阵一行和向量乘积，而一个GPU具体计算矩阵一行与向量乘积。核心代码如代码清单11-7所示：

代码清单11-7 进程+线程+GPU线程示例

```
void runmxvBlockGPU(int rowSize, int columnSize, const float *matrix, const
float *v, float *r);
int main(int argc, char *argv[]){
    checkMPIError(MPI_Init(&argc, &argv));
    int size, rank;
    checkMPIError(MPI_Comm_size(MPI_COMM_WORLD, &size));
    checkMPIError(MPI_Comm_rank(MPI_COMM_WORLD, &rank));
    float matrix[NUM_ROWS*NUM_COLUMNS];
    float v[NUM_COLUMNS], r[NUM_ROWS];
    float *all;
    //init matrix, v and all
    int num;
    cudaGetDeviceCount(&num);
    #pragma omp parallel
    {
        cudaSetDevice(omp_get_thread_num()%num);
        #pragma omp for schedule(dynamic, 1)
        for(int i = 0; i < NUM_ROWS; i += PATCH)
            runmxvBlockGPU(PATCH, NUM_COLUMNS, matrix+i*NUM_COLUMNS, v, r+i);
    }
    checkMPIError(MPI_Gather(r, NUM_ROWS, MPI_FLOAT, all, NUM_ROWS, MPI_
        FLOAT, 0, MPI_COMM_WORLD));
    checkMPIError(MPI_Finalize());
    return 0;
}
```

其编译命令为：

```
mpic++ mpiOpenmpCUDAmxv.cpp -DNUM_ROWS=1024 -DNUM_COLUMNS=1024 -DPATCH=256
-fopenmp cudamxv.o -lcudart -L/usr/local/cuda-5.0/lib64
```

11.2 矩阵乘法

矩阵乘法在科学计算中广泛使用，成为在各种硬件架构上研究得最多的算法之一。本节以矩阵乘法为例，展示在多机、GPU集群上如何优化矩阵乘法。

11.2.1 多机CPU矩阵乘法

在基于MPI的多机环境中，卡诺算法是应用得最广泛的矩阵乘法实现。卡诺算法大致原理如下：将进程组织成二维结构，假设当前进程索引为 (i, j) ；将结果矩阵二维划分成和进程数相同的块；当前进程 (i, j) 计算结果矩阵块 (i, j) ；按照相同的方式划分矩阵A和B。从算法可以知道：结果矩阵中块 (i, j) ，为A的分块中行索引为 i 的块与B的分块中列索引为 j 的块的对应乘积（遍历A的列和B的行）。基于这一认识，笔者基于MPI编写第1版代码，核心代码如代码清单11-8所示，其中函数getXYUpDownProcess返回二维进程拓扑中与当前进程相距disp的4个进程编号，关于进程拓扑的含义可参考MPI手册。

代码清单11-8 多机CPU矩阵乘法版本1

```
#define NUM_DIMS 2
int main( int argc, char **argv ){
    checkMPIError(MPI_Init(&argc, &argv));
    int M = 1024;
    int K = 1024;
    int N = 1024;
    int rank;
    checkMPIError(MPI_Comm_rank(MPI_COMM_WORLD, &rank));
    int numProcesses;
    checkMPIError(MPI_Comm_size(MPI_COMM_WORLD, &numProcesses));
    int dims[NUM_DIMS] = {0,0};
    checkMPIError(MPI_Dims_create(numProcesses, NUM_DIMS, dims));
    MPI_Comm my_comm;
    int periods[NUM_DIMS] = {1, 1};
    checkMPIError(MPI_Cart_create(MPI_COMM_WORLD, NUM_DIMS, dims, periods, 0,
        &my_comm));
    //malloc space for A, B, C, bufA, bufB, bufC, and init them
    //0 y, 1 x
    int numPartitions = dims[0];
    int myid[NUM_DIMS];
    int xidup, xiddown, yidup, yiddown;
    matrixMul<float, false>(A, B, C, M, K, N);
    for(int disp = 1; disp < numPartitions; disp++){
        getXYUpDownProcess(2, rank, my_comm, disp, myid, &xidup, &xiddown,
            &yidup, &yiddown);
        checkMPIError(MPI_Sendrecv(A, M*K, MPI_FLOAT, xiddown, rank, bufA,
            M*K, MPI_FLOAT, xidup, xidup, MPI_COMM_WORLD, MPI_STATUS_IGNORE));
        checkMPIError(MPI_Sendrecv(B, K*N, MPI_FLOAT, yiddown, rank, bufB,
            K*N, MPI_FLOAT, yidup, yidup, MPI_COMM_WORLD, MPI_STATUS_IGNORE));
        matrixMul<float, true>(bufA, bufB, C, M, K, N);
    }
    checkMPIError(MPI_Finalize());
    return 0;
}
```

考虑到对于结果分块而言，行索引相同的块共享A中相同行索引的块，列索引相同的块共享B中相同列索引的块，因此只要将A分块每次移动一个位置（把数据交给列相邻进程），同时将B分块向相应的方向每次移动一个位置。相比前面的算法，这个算法保证了所有的通信进程都是二维相邻的，因此在基于网格互联的集群中，其性能会更好。核心代码如代码清单11-9所示：

代码清单11-9 多机CPU矩阵乘法版本2

```
float *sendBufB, *sendBufA, *recvBufA, *recvBufB;
for(int disp = 1; disp < numPartitions; disp++){
    sendBufA = (disp % 2) ? A: bufA;
    recvBufA = (disp % 2) ? bufA: A;
    sendBufB = (disp % 2) ? B: bufB;
```

```

recvBufB = (disp % 2) ? bufB: B;
if(1 == disp){
    matrixMul<float, false>(A, B, C, M, K, N);
}else{
    matrixMul<float, true>(sendBufA, sendBufB, C, M, K, N);
}
MPI_Sendrecv(sendBufA, M*K, MPI_FLOAT, xiddown, rank, recvBufA,
    M*K, MPI_FLOAT, xidup, xidup, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
MPI_Sendrecv(sendBufB, K*N, MPI_FLOAT, yiddown, rank, recvBufB,
    K*N, MPI_FLOAT, yidup, yidup, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
checkMPIError(MPI_Barrier(MPI_COMM_WORLD));
}
matrixMul<float, true>(recvBufA, recvBufB, C, M, K, N);

```

笔者使用了MPI_Sendrecv在进程中交换数据，但是这个函数是阻塞的，因此计算和通信是串行的。要解决这个问题有两个办法：①使用异步通信MPI_Isend、MPI_Irecv，此时需要特别安排以避免死锁，许多开发人员（包括几大互联网公司的一些资深人员）认为MPI的异步通信不会出现死锁，这是严重错误的；②使用线程，将计算交给线程，通信和计算自然就并行了。

使用第二个方法的核心代码如代码清单11-10所示：

代码清单11-10 多机CPU矩阵乘法版本3

```

pthread_t thread;
for(int disp = 1; disp < numPartitions; disp++){
    sendBufA = (disp % 2) ? A: bufA;
    recvBufA = (disp % 2) ? bufA: A;
    sendBufB = (disp % 2) ? B: bufB;
    recvBufB = (disp % 2) ? bufB: B;
    DataArg arg;
    arg.A = sendBufA;
    arg.B = sendBufB;
    arg.C = C;
    arg.M = M;
    arg.K = K;
    arg.N = N;
    assert(0 == pthread_create(&thread, NULL, thread_matrixMul, &arg));
    checkMPIError(MPI_Sendrecv(sendBufA, M*K, MPI_FLOAT, xiddown,
        rank, recvBufA, M*K, MPI_FLOAT, xidup, xidup, MPI_COMM_WORLD,
        MPI_STATUS_IGNORE));
    checkMPIError(MPI_Sendrecv(sendBufB, K*N, MPI_FLOAT, yiddown,
        rank, recvBufB, K*N, MPI_FLOAT, yidup, yidup, MPI_COMM_WORLD,
        MPI_STATUS_IGNORE));
    pthread_join(thread, NULL);
    checkMPIError(MPI_Barrier(MPI_COMM_WORLD));
}
matrixMul<float, true>(recvBufA, recvBufB, C, M, K, N);

```

笔者使用pthread新建了一个线程，由线程来执行矩阵乘法，然后在数据传输完成后等待矩阵乘法计算完成，同时同步所有进程。

需要提醒的是：有些MPI实现不支持直接在MPI进程中使用线程，目前OpenMPI实现已经提供了支持。

11.2.2 单机多GPU矩阵乘法

可以将卡诺算法简单地映射到单节点多GPU上，但是相对来说，有更好、更简单的划分方式。更简单的方式是将A按行划分，B按列划分，通过使用流来安排依赖关系。相比使用MPI来说，使用CUDA流依赖能够很好地发挥PCI-E的双向带宽。

笔者使用CUDA实现的单机多GPU矩阵乘法核心代码如代码清单11-11所示。此算法在单机4个GPU上，使用4096×4096的矩阵和4096×4096的矩阵相乘获得了线性扩展。

代码清单11-11 单机多GPU矩阵乘法

```
float alpha = 1.0f;
float beta = 0.0f;
int lda = k;
int ldb = n/numGPU;
int ldc = n;
for(int id = 0; id < numGPU; id++){
    cudaSetDevice(id);
    cublasSetStream(handle[id], streams[id][0]);
    cublasSgemm(handle[id], CUBLAS_OP_N, CUBLAS_OP_N, n/numGPU, m/
        numGPU, k, &alpha, d_b[id][0], ldb, d_a[id], lda, &beta,
        c[id]+id*(n/numGPU), ldc);
    for(int offset = 1; offset < numGPU; offset++){
        int srcDevice = (id+offset) % numGPU;
        checkCUDAError(cudaMemcpyPeerAsync(d_b[id][offset],
            id, d_b[srcDevice][0], srcDevice, n/numGPU*k*sizeof(float),
            streams[id][offset]));
    }
    for(int offset = 1; offset < numGPU; offset++){
        int srcDevice = (id+offset) % numGPU;
        cublasSetStream(handle[id], streams[id][offset]);
        cublasSgemm(handle[id], CUBLAS_OP_N, CUBLAS_OP_N, n/
            numGPU, m/numGPU, k, &alpha, d_b[id][offset], ldb,
            d_a[id], lda, &beta, d_c[id]+srcDevice*(n/numGPU), ldc);
    }
}
for(int id = 0; id < numGPU; id++){
    cudaSetDevice(id);
    checkCUDAError(cudaDeviceSynchronize());
}
```

此算法的主要核心是双层循环，外层循环遍历每个GPU，内层循环从其他GPU获得B数据。由于每个GPU只保存了一部分的A和一部分的B，因此需要在计算时从其他GPU获得B数据。为了更好地利用GPU之间的带宽，笔者使用了cudaMemcpyPeerAsync函数。

为了更好地利用GPU计算和数据传输同时进行，每个GPU先在其拥有的数据上执行矩阵乘法运算，由于这个执行是异步的，CPU会接着执行下面的、从其他GPU获取数据的传输。为了保证GPU开始计算时，其需要的数据已经传输到了，即计算和数据传输存在依赖关系，笔者使用了CUDA流来保证这种依赖关系。在代码的最后调用cudaDeviceSynchronize来等待所有GPU运算完成，此时GPU上的计算结果才可用。

11.2.3 多机多GPU矩阵乘法

由于CUDA内核运行是异步的，因此其天然可以和MPI通信并行，为了正确性，需要在MPI通信完成后等待GPU计算完成。本节使用的算法本质上和多机CPU矩阵乘法一样，其核心代码如代码清单11-12所示：

代码清单11-12 多机多GPU矩阵乘法

```
float *sendBufB, *sendBufA, *recvBufA, *recvBufB;
for(int disp = 1; disp < numPartitions; disp++){
    sendBufA = (disp % 2) ? d_A: d_bufA;
    recvBufA = (disp % 2) ? d_bufA: d_A;
    sendBufB = (disp % 2) ? d_B: d_bufB;
    recvBufB = (disp % 2) ? d_bufB: d_B;
    cublasSgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N, N, M, K,&alpha,
        sendBufB, ldb, sendBufA, lda, &beta, d_C, ldc);
    checkMPIError(MPI_Sendrecv(sendBufA, M*K, MPI_FLOAT, xiddown,
        rank, recvBufA, M*K, MPI_FLOAT, xidup, xidup, MPI_COMM_WORLD, MPI_
        STATUS_IGNORE));
    checkMPIError(MPI_Sendrecv(sendBufB, K*N, MPI_FLOAT, yiddown,
        rank, recvBufB, K*N, MPI_FLOAT, yidup, yidup, MPI_COMM_WORLD, MPI_
        STATUS_IGNORE));
    checkCUDAError(cudaDeviceSynchronize());
    checkMPIError(MPI_Barrier(MPI_COMM_WORLD));
}
cublasSgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N, N, M, K,&alpha, recvBufB,
    ldb, recvBufA, lda, &beta, d_C, ldc);
```

和单机多GPU矩阵乘法不相同的是：为了减少临时空间的大小（只使用两倍分块大小的空间），采用在一块数据上计算的同时从其他进程上获得下一次计算需要的数据，同时也把自己的数据传给需要它的进程，故需要在循环内部调用cudaDeviceSynchronize和MPI_Barrier来保证进程在执行下一次计算前，当前的计算和传输都已经完成。

11.3 本章小结

本章介绍了常见的几种超级并行方式，并且以矩阵向量乘为例展示在各种超级并行模式下如何划分数据和计算。常见的超级并行方式有：进程+线程、进程+GPU、线程+GPU、线程+SIMD向量指令、进程+线程+向量指令、进程+线程+GPU这几种模式。最后以矩阵乘法为例，讲解如何在多机多CPU、单机多GPU和多机多GPU的环境下设计矩阵乘法。

第12章 并行算法设计的一般准则

并行编程更像是一门艺术，所关注的往往不是能否实现它们，而是如何优雅、高效地实现。本章将会给出一些设计并行程序的简单规则。笔者希望通过这个短短列表，读者能够了解写出高质量、高效率的并行程序要记住的一些法则。笔者尽量按照设计并行程序要考虑的问题的重要程度来组织它们。

12.1 并行算法设计14准则

1.查找算法的热点并优先考虑热点的并行化

依据Amdahl定律，对热点的优化通常会产生更好的结果。一个简单的说明就是并行化占总时间80%的程序加速1倍得到的效果比对余下的20%并行化后加速10倍的效果要好。

有时会遇上热点比较均匀的算法，此时并行的工作量会比较大，选择合适的工具就非常重要。

通常经过对简单算法的步骤的分析可知道哪些部分耗时比较多，但是大的程序可能就不行了，关于如何查找程序的热点可参考剖分软件的手册。

2.找出算法中的并行性

如果对一个没有并行性的算法进行并行化，你的努力将会白费。一般而言，并行性意味着不相关/独立，更详细点说，一些计算任务与另一些计算任务无关，这样它们之间就可以并行。或者作用在某些数据上的操作与作用在另一些数据上的操作无关，也可并行。或者同样操作作用在不同的数据上。

也可能会碰到某些只能串行执行的计算任务，大多数是因为计算任务或操作的数据之间存在依赖关系，从而导致它们只能按照某些特定的顺序串行执行。女人生孩子便是一个很好的例子。你没有任何办法可以让生产周期缩为5个月（早产和流产不算），当然你可以娶10个女人，让她们10个月给你生10个孩子。

3.自外而内与自内而外

寻找算法的并行性时有两种方法可以选择，一种是自外而内，一种是自内向外。相比自内而外，自外而内通常能够更早找出可并行化的部分，并且粒度通常也比较大，因此更适合多核并行，在多核处理器上并行也会得到更好的效果。

在自外而内的方法中，首先对高层进行并行化，如果这一层没有并行性的话，看下一层能否并行化。假设程序可并行化的部分在一个嵌套循环的最里层，我们可以由外而内逐一检查每一层循环，直到找到可以并行化的部分。

通常外层比内层更易找到，且自外而内的方式比自内而外的算法更易于找到大粒度的并行化方法，因此对于多进程和多线程程序，笔者推荐使用自外而内的方法。

自内而外的方法要首先找到算法最内层的部分，然后再查外层部分。即使内层能够并行化，也应该检

查一下是否可能在更高的某一层上实现并行化。这能提高每个线程所执行的任务的粒度。

由于自内而外的方法容易找到细粒度的并行，而向量化需要细粒度并行，因此对于向量化的任务，笔者推荐自内而外的方法。

4.注意算法的通信计算比

并行算法的计算通信比是指算法中计算与通信的比例。计算量越大，计算通信比越大；通信量越大，计算通信比越小。计算通信比如果小到不能掩盖使用多个控制流所带来的控制流创建、调度和销毁的开销，就不应当并行化。在计算量保持不变的前提下使用越多的控制流，计算通信比会降低，在计算通信比降至1之后，再增加控制流的数目，程序的性能会越差。创建、调度和销毁控制流的代价越大，算法的计算通信比就应该越大。并行计算通信比大的算法会有更好的可扩展性。

5.选择并行算法时不要只看评价串行算法的标准

当比较串行或者并程序的性能的时候，执行时间就是衡量的首要标准。在实现算法前，软件开发人员通常会依据时间复杂度评价算法。

在并行程序中，时间复杂度更好的算法通常会更快一些。然而，有时时间复杂度更好的算法却不易并行化。此时可换一个时间复杂度稍微差一些但是却更容易并行化的算法，在硬件的能力超越时间复杂度的差别的时候，也可能得到加速。另外在某些情况下，对算法做一些转换或者从另外一个角度看，不适合并行的算法也就适合并行了。

更适合评价并程序的标准是控制流的最大计算复杂度和最大访存复杂度，而控制流的最大计算复杂度/最小计算复杂度和最大访存复杂度/最小访存复杂度反映了负载不均衡的程度。

6.采用合适的并行模型使算法更好地映射到硬件上

相比显式的多线程机制，隐式的多线程模型更易于编程和调试。因此在能满足功能需求的前提下，应尽量使用隐式模型（如OpenMP）而将具体的线程建立、调度和销毁的实现交给编译器。显式的多线程模型（例如Pthread）能提供对线程更精确的控制，使用也更加自由，另外很多操作系统环境也有显式的多线程实现（Linux/Pthread、Windows/winthread），但是其实现更复杂，更容易出错，且代码的维护成本也会更大。

为了充分发挥硬件的计算能力，应当将程序适当地映射到硬件上，这一点经常和程序的可兼容性相冲突，要特别注意。

7.尽可能利用已有的并行库

如果程序的热点并行化能通过调用库函数来实现，那么就不要考虑调用自己手写的代码。“重新造轮子”不是一个好主意。一方面软件开发人员可能并没有库代码作者那么丰富的经验和知识，另外库代码通常经过了非常丰富完备的测试，因此也更可靠。

8.注意并程序的可扩展性

目前四核处理器已经成为了主流，八核、十六核的处理器也已经量产，而GPU有成百上千核心。从目前的发展趋势来看，未来处理器的核心数量只会越来越多。为了保证你的软件在未来五年内依旧可用，应该尽量使软件具有可扩展性。

可扩展性（scalability）用来衡量一个程序应对条件变化的能力，典型的变化有核心数量、内存大小、总线速度或数据量的增加等。在面对越来越多可用的核心时，应该保证软件能够利用更多数量的核心。

数据并行相比任务并行具有更高的可扩展性。任务并行可能会面临程序中可独立运行的函数数量有限或运行时间相差很大的问题。如果每一个独立的任务已经在单独的线程和核心上运行的时候，就不能通过增加线程数量来利用更多核心的计算能力。而数据并行只需要增加待处理的数据量就可以利用更多的核心资源。

9.永远不要假设具体的执行顺序

串行程序的状态是确定的，可以非常容易地预测某个程序的某个状态之后会是什么状态。然而，由于多线程执行顺序的不确定性（由操作系统的调度器决定），不能准确预测多个线程的执行顺序，甚至在某个时刻哪些线程会被调度到核心上执行也不能预测。这主要是为了隐藏程序运行时的延迟，特别是当线程的数量多于核心的数量时。例如，如果一个线程因为缺页中断而等待时，操作系统就会把另一个等待执行的线程调度进来并执行它。

顺序一致性问题就是由这种执行顺序的不确定性引起的。如果你假设一个线程对共享变量的写操作会在另一个线程对该共享变量的读操作之前完成，你的预测可能会一直正确，有可能有些时候会正确，也有可能从来都不会正确。然而由于处理器和操作系统调度算法不同，这类程序可能在一个平台上产生一个结果而在另一个平台上产生另一个结果。关于顺序一致性的详细说明请参考7.2.3节。

10.注意通信开销

通信本身并不属于计算任务，它只是为了确保程序的并行执行能得到正确的结果所产生的额外开销。

虽然通信产生了额外的开销但是又必不可少。因此要尽可能把同步所产生的开销降低到最低，可以采用不同的通信方式达到。另外一个常用的方法是使用线程私有的存储空间或者局部变量来达到目的。

11.使用线程私有变量

无须在线程间共享的临时变量和结果应由每个线程单独声明或分配。但是有些操作之间通信不可避免，此时尽量使通信比较少且在线程间分布均匀。通常使用锁、临界区和原子函数等协调对共享资源的访问，以确保程序对数据进行了适当的操作。

12.注意锁的粒度

在大量的数据需要加锁的情况下，如果只用一个锁来保护数据的话，可能会造成严重的锁竞争从而导致性能瓶颈。此时可以使用多个锁来保护数据，每个锁保护部分数据，这样多个锁可并行保护数据，提高了并行性，此时要尽量保证锁竞争数量相近。通常采用模余算法来确定锁保护的数据。假设有两个锁，可以采取一个锁保护所有的奇数索引的元素，另一个锁保护所有的偶数索引的元素。

13.全局设计

并行算法实现前最好考虑清楚大的层次上的每一个细节，把相关的数据结构设计好。一旦数据结构设计好后，算法如何实现就基本清楚了。

通常并行算法的设计是一个迭代的过程，很可能实现一个之后，对某些以前没考虑清楚的细节的了解会帮助设计更好的算法。

14.步步验证

目前用于并行程序调试、查错的工具非常少，故最好边编码边验证，尽量将bug消灭在出现之前。一个非常好的实践是每次只执行一个小的更改，然后验证。

12.2 本章小结

本章总结了并行算法设计的一般准则，希望读者可以据此编写出高质量、高效率的并行程序。

附录A 整型数据与浮点数据

相邻整数之间的距离为1，这是精确的，因此可以精确表示，但是整数数目却是无穷的，要表示这无穷个数据，需要无数个二进制位。为了在存储量和大小之间均衡，编程语言都支持不同大小的整数数据类型。由于采用了有限的位数存储整数，其计算和现实中的整数计算不一样，本章会详细说明C语言中整数的表示方式以及其运算和现实运算的不同之处。

现代X86处理器使用SSE或X87指令做浮点计算，为了可移植性，在32位机器上编译器通常将浮点运算编译成X87指令，而在64位机器上则编译成SSE指令。浮点表示具有不同于整形表示的特点，因为整形相邻数据之间的差异为1，而浮点数据的差异为无穷小，这就意味着要准确表示浮点数值，需要无穷二进制位，这并不现实。目前处理器普遍采用的浮点表示方法是IEEE-754标准，本章会详细说明该标准的主要内容。

A.1 C语言数据类型

C语言中的整数分为有符号和无符号两种类别，有符号表示其值可以表示正数和负数，而无符号则只表示正数。C中的浮点数有单精度和双精度两种，除了精度之外，两者没有区别。表A-1列出了C语言支持的数据类型及其在IA32和X86-64机器上所占据的字节数。

表A-1 C语言的数据类型及在IA32和X86-64上位数

类型	IA32	X86-64
char	8	8
short	16	16
int	32	32
long	32	64
float	32	32
double	64	64
void*	32	64

A.2 整型运算

由于使用有限的位表示整型，因此其运算可能出现溢出，溢出的主要原因是运算结果改变了最高位或者超过了表示能力。和浮点表示不同的是：整型的表示是精确的，其满足分配律和结合律。

对于整数加减法来说，需要注意运算结果超出表示范围的情况，即两个正数的和有可能是负数，而两个负数的和有可能是正数。有符号数取负需要注意其表示并非对称。两个整数乘法运算的结果有可能超出

其表示范围。在进行数据类型转换时，需要特别注意涉及数据大小转换的时候。

1.整型的补码表示

C语言中整数的表示有原码、反码和补码三种，三者在表示正数时没有区别，区别的只是如何表示负数。虽然C语言标准没有规定整数的编码标准，但是现在的编译器都使用补码表示，因此本文也只介绍补码表示。

对于一个二进制表示为 $(x_{n-1}, x_{n-2}, \dots, x_0)$ 、二进制长度为 n 的整型数据，其补码表示的数据为： $-x_{n-1}2^{n-1} + \sum_{i=0}^{i=n-2} x_i 2^i$ ，其表示的无符号数据是： $\sum_{i=0}^{i=n-2} x_i 2^i$ 。对于有符号整数，通常称 x_{n-1} 为符号位，其值为1表示负数，为0表示正数。

本节使用函数 r 表示C语言中某种运算的结果，而表达式表示无限精度运算结果。

2.加减

对于二进制长度为 n 位的两个数的加减运算，产生的结果可能是 $n+1$ 位数据，而实际上C还是使用 n 位表示（原因是提供无限精度的表示代价太大），这就可能导致某些奇怪的行为。

无符号数的加减结果还是无符号数，但是两个无符号数的和可能小于两者中任何一个，两个无符号数减的结果可能大于被减数。无符号数加减的结果如式A-1所示：

$$r(x \pm y) = \begin{cases} x \pm y - 2^n & x \pm y \geq 2^n \\ x \pm y + 2^n & x \pm y < 0 \\ x \pm y & 0 \leq x \pm y < 2^n \end{cases} \quad (\text{A-1})$$

两个正的有符号数的和可能为负数，两个负的有符号数的和可能是正数。有符号数加减的结果如式A-2所示：

$$r(x \pm y) = \begin{cases} x \pm y - 2^n & x \pm y \geq 2^{n-1} \\ x \pm y + 2^n & x \pm y < -2^{n-1} \\ x \pm y & -2^{n-1} \leq x \pm y < 2^{n-1} \end{cases} \quad (\text{A-2})$$

3.取负

由于补码表示的数不是对称的（最小的负数没有对应的正数表示），因此取负存在特例。无符号数取负却是正数，而且0是其特例。

无符号数的负数如式A-3所示：

$$r(-x) = \begin{cases} 0 & x = 0 \\ 2^n - x & \end{cases} \quad (\text{A-3})$$

有符号数的负数如式A-4所示：

$$r(-x) = \begin{cases} x & x = -2^{n-1} \\ -x & \end{cases} \quad (\text{A-4})$$

4.乘法

两个n位二进制的乘法结果最多可能为2n位，但是C只使用了其低n位而丢弃高n位，因此其结果可能和直观不一致。整数乘法的结果如式A-5所示：

$$r(x \times y) = (x \times y) \% 2^n \quad (\text{A-5})$$

对于有符号数的乘法来说，其运算相当于先将两个乘数的位表示解释成无符号数，再执行乘法，然后将结果的位表示解释成有符号数，因此有符号数和无符号数的乘法可使用同一个指令。

整数乘以常整数时，可以将其转化为左移和加减法，因为通常乘法需要几个时钟周期而移位和加法一个周期能够处理多个。如 $x \times 8$ 可转化为 $x \ll 3$ ，而 $x \times 9$ 可转化为 $(x \ll 3) + x$ 。但是 $x \times 7$ 有两种表示方式： $(x \ll 3) - x$ 和 $(x \ll 2) + (x \ll 1) + x$ ，很明显前一个更高效。

5.除法

C语言中，整数除法的结果还是整数，因此存在一个舍入的问题，C语言默认采用向零舍入，即 $(\text{int}) 3.5 = 3$ ， $(\text{int}) -3.5 = -3$ 。

除法运算需要的周期数远超乘法，通常要几百个。除以2的n次方时，可以将其转化为右移，但是有一点需要注意：除法是向零舍入，而右移是向下舍入，因此如果被除数是负数时，需要特殊处理。

```
int r = ((x > 0 ? x : (x+(1<<n)-1)) >> n);
```

6.移位

C语言中，有两种移位操作，一是左移，用符号<<表示，左移一位相当于乘以2，左移产生的右侧空位填充0，左移产生的进位会被忽略，其行为和乘法一致；二是右移，用符号>>表示，右移一位相当于除以2，对于右移产生的高位空位有两种处理方式，一种是填充0，这称为逻辑右移；一种是填充最高位，这称为算术右移。

关于采用何种填充方式，现行的C标准没有对此做出规定，但是所有的编译器对此的处理表现出惊人的一致：对于无符号数使用逻辑右移，对于有符号数使用算术右移。这种处理能够保证移位和乘除法方便地进行转换。另外，右移是向下舍入，而除法是向零舍入的，关于向下舍入和向零舍入的概念，请参考相关书目。

7.数据类型转换

C支持强制数据类型转换，由于强制类型转换可能会改变数据的表示，因此结果可能会发生变化。另外由于C支持隐式数据类型转换，就导致了許多潜在的错误，而且非常隐蔽，难以排除故障。下面列出了C中支持的数据类型转换类别。

- 大转小：此时会去掉高位，留下低位，如int转化为char，便只会留下最低8位。
- 小转大：存在两种高位填充模式，一种是填充0，无符号数大小转换采用这种。另一种是以符号位填充高位，有符号数转换采用这种方式，它能够保证其数字结果和原来一致。这一点和编译器对右移的处理保持了一致。
- 有符号转无符号：此时会保持数据的位表示，只是对位做重新解释。
- 混合：同时有数据大小转换和符号转换的情况下，C会先进行数据类型大小转换，然后再进行符号转换。如将int强制转换成unsigned short，会先将int转换成short，再将short转换成unsigned short。对于混合转换时，特别需要注意小数据转大数据的情况，尤其是同时执行有符号数转无符号数，如（unsigned int）（short-2），这将会产生一个非常大的无符号数。

A.3 IEEE-754浮点格式

通常任一浮点数据x都可以表示成式A-6所示：

$$x=(-1)^S \times M \times 2^E \quad (\text{A-6})$$

其中S表示x的符号，等于1或者0，因此只要一位便可；M是一个二进制数，称为尾数，它的范围是[0, 2)；E为阶码，可以是负数也可以是正数。

IEEE-754规定32位的单精度数据中E由8位表示，M由23位表示；而64位的双精度数据中E由11位表示，M由52位表示；半精度数据类型，其M由10位表示，E由5位表示。本文使用k表示阶码的位数。

1.规格化数

如果E的二进制表示非全0或全1，表示规格化数。此时M的范围是[1, 2]，可以省略1的表示，称为隐含的1，因此实际的10/23/52位只表示小数部分，其第一位表示是否有0.5，第二位表示是否有0.25，其余类推。而此时E表示 $e-bia$ ，e是E的二进制表示的无符号值，bia表示 $2^{k-1}-1$ ，其中k表示E的位数。

可以知道最小的正规格化数为：M全0，表示1.0；E的最低有效位为1，表示 $2-2^{k-1}$ ，故最小的正规格化数为 $2^{2-2^{k-1}}$ 。

2.非规格化数

如果E的二进制表示全0，这称为非规格化数。非规格化数的尾数M由其二进制小数表示（没有隐含的1），故其大小范围为[0, 1)，而此时E大小为 $2-2^{k-1}$ ，故最大的正非规格化数为：M全1，表示 $1-2^{-n}$ ，n表示M的位数，值为 $(1-2^{-n}) \times 2^{2-2^{k-1}}$ 。可以看出最大的正非规格化数和最小的正规格化数非常接近。

非规格化数提供了一种表示非常小的数的方式，且使得这些小数近似对称的分布在0左右。IEEE的表示有一个非常好的优点：二进制相邻（即二进制x和x+1）表示的浮点数并不均匀，越靠近0，其二进制相邻表示的浮点数之间的距离越小，这就能够保证更好的精度。

现在处理器处理非规格化数时，有一个称为下溢的术语，它表示一个存在但是无法表示的数被当成了0处理，下溢会造成精度损失。由于最小的非规格化数为：M最后一位为1，即 2^{-n} ，值为 $2^{2-2^{k-1}-n}$ ，其绝对值非常小，虽然其绝对值非常小，但是下溢会损失精度，尤其是在参与运算的数绝对值大小差距明显时。

3.特殊值

当E的二进制表示全1，M全0时，表示正负无穷大，无穷大表示数据类型不能表示但确实存在的数据；当E全1，而M非全0时，结果表示nan，意味着不能表示的数据，通常意味着除以0或无穷相关运算得到的结果。

怪异的是：有两种0表示法，一种是正0，一种是负0，对应着S为0或1。在标准看来这两种0是不同的。

4.舍入模式

由于IEEE-754采用有限位来表示浮点数据，因此有许多数据不能精确表示，此时需要采用舍入规则使用一个离它很近的值来表示。常用的舍入模式有：向上舍入、向下舍入、舍入到偶、向零舍入。

- **向上舍入**是指将结果舍入到第一个比它大的可表示的数，类似于标准数学函数ceil，只是舍入的是最低有效位；
- **向下舍入**指将结果舍入到第一个比它小的可表示的数，类似于floor函数；
- **舍入到偶**是指将结果向上或者向下舍入，使得结果的最低有效位为偶数，这是标准支持的默认舍入模式；
- **向零舍入**是指最后的结果向0靠近，如果是负数就是向上舍入，如果是正数就是向下舍入。

将浮点数转型为整数采用的舍入模式是向零舍入。一些标准库为其某些函数提供了支持这些舍入模式的版本。

5.浮点运算特性

由于IEEE-754标准是近似的表示浮点数，因此其计算和符号计算有许多不同的地方。

· 整形和单精度浮点型之间转换会有精度损失，即使将单精度浮点转成8字节的整形，因为有些浮点值是整形无法表示的，同样有些整形也无法使用浮点值表示。大致来说，在C语言中，将整型转化为浮点会选择一个最接近整型的浮点可表示的浮点值。将浮点型强制转换为整型的法则是直接去掉小数即向零舍入，而不管浮点数本身是正是负，如果超出了整数的表示范围，则使用整数的最大或最小值。如果需要对十分位进行四舍五入，则其公式如式A-7所示：

$$x > 0 ? (\text{int})(x + 0.5) : (\text{int})(x - 0.5) \quad (\text{A-7})$$

下面的代码展示了这一公式。

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char* argv[]) {
    printf("(int)8.3=%d\n", (int)8.3);
    printf("(int)8.7=%d\n", (int)8.7);
    printf("(int)(8.3+0.5)=%d\n", (int)(8.3+0.5));
    printf("(int)(8.7+0.5)=%d\n", (int)(8.7+0.5));
    printf("(int)-8.3=%d\n", (int)-8.3);
    printf("(int)-8.7=%d\n", (int)-8.7);
    printf("(int)(-8.3+0.5)=%d\n", (int)(-8.3+0.5));
    printf("(int)(-8.7+0.5)=%d\n", (int)(-8.7+0.5));
    printf("(int)(-8.3-0.5)=%d\n", (int)(-8.3-0.5));
    printf("(int)(-8.7-0.5)=%d\n", (int)(-8.7-0.5));
    return 0;
}
```

```
/*
(int)8.3=8
(int)8.7=8
(int)(8.3+0.5)=8
(int)(8.7+0.5)=9
(int)-8.3=-8
(int)-8.7=-8
(int)(-8.3+0.5)=-7
(int)(-8.7+0.5)=-8
(int)(-8.3-0.5)=-8
(int)(-8.7-0.5)=-9
*/
```

- 将int转换为double没有精度损失，因为double能够表示每一个int。
- 将单精度转成双精度不会有损失，但是将双精度转化成单精度有损失，甚至可能产生无穷大。浮点（双精度和单精度）之间的转化与整型之间的转化的不同在于：浮点之间的转化尽量保证转化前后结果一致或相近，而没有符号位扩展的问题。
- 浮点加法和乘满足交换律，但是不满足结合律与分配律，在某些情况下，这阻止了编译器的优化。在必要的时候可以通过加括号来告诉编译器某些操作可乱序或同时执行。

如下面的代码中下一条就比上一条高效。

```
float x = a + b + c + d;
float x = (a+b) + (c+d)
```

6.不同条件下浮点运算结果

对于涉及浮点运算的算法实现，直接拿不同硬件的结果一步一步地比较是不恰当的，因为即使在同一台机器上同一程序源码在不同的编译器上、不同的操作系统上、不同的编译选项上、串行和并行、结果都会有差别，这个问题的根本原因在于现行的IEEE-754浮点标准，浮点加减乘不满足结合律和分配律，这种问题在操作数为非规格化数时更为明显。由于浮点运算不满足分配律和结合律，由不同的人实现的算法结果也可能有差别。通常这并不意味着程序是错误的，而是说结果的确存在着差异。

由于浮点运算存在误差，在某种对精确度要求非常严格的应用中，对误差的控制就非常有必要。