摘 要

安全算法是并行计算的其中一个应用分支，包括对密码算法，随机序列等的各种安全检测算法。这些安全算法的性能直接影响着整个应用系统的性能。随着软硬件技术的提高，人们对安全算法提出了新的要求。本文主要研究了安全算法的一般并行实现方式，并使用 OpenMP 进行仿真实现，指出并行实现过程中遇到的一般问题与其相对应的解决思路与方案以及并行程序的性能优化方法。

重点介绍了并行随机数生成器的实现方式、优缺点和检测方法。设计了具有良好可扩展性的两种高性能并行生成器的并行实现步骤。总结了在并行环境下，对并行随机数生成器的检测方法的新的要求和发展趋势。

最后给出MD6的两种并行实现思路，通过对两个程序实现结果的分析和对比，明确指出并行实现需考虑和解决的问题以及 OpenMP 实现并行的局限性。

关键词：并行　安全算法　OpenMP　并行随机数生成器　MD6

Abstract

Security algorithm, which includes test algorithms in Cipher algorithm and random numbers, is one of the applications in parallel computing. The performance of the cipher algorithm and random numbers used in the application system directly affects the safety performance of the entire system. With the improvement of hardware and software technologies, people make a new requirement for the research on Security algorithm. Based on the deep understanding of the security algorithms and experience in parallel implementing, this paper first summarizes the general implementing ideas of the Security algorithm, and then puts forward the general technique of parallel computing in implementing, also the solutions to the problems encountered in the process of programming. Examples of parallel computing in OpenMP and parallel performance optimization used by Intel software are presented.

And then, methods of the parallel implementing, relative merits and test methods in parallel random number generators are described. Focusing on the scalability of the application, steps of implementing two high-performance random number generators applied to a variety of hardware are proposed. Then the development trend and new requirements for the test methods in parallel environment are concluded.

At last, two methods of parallel implementing on MD6 hash funcion are designed and implemented. By analyzing and comparing the results of these two methods, the problems needed to be considered and solved in parallel computing are straightly pointed out, also the inherent restrictions of implementing using OpenMP.

Keywords: Parallel   Security algorithm   OpenMP
Parallel Random Number Generator   MD6

西安电子科技大学

学位论文独创性（或创新性）声明

秉承学校严谨的学风和优良的科学道德，本人声明所呈交的论文是我个人在导师指导下进行的研究工作及取得的研究成果。尽我所知，除了文中特别加以标注和致谢中所罗列的内容以外，论文中不包含其他人已经发表或撰写过的研究成果；也不包含为获得西安电子科技大学或其它教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中做了明确的说明并表示了谢意。

申请学位论文与资料若有不实之处，本人承担一切的法律责任。

本人签名：______ 日期：2010.3.16

西安电子科技大学

关于论文使用授权的说明

本人完全了解西安电子科技大学有关保留和使用学位论文的规定，即：研究生在校攻读学位期间论文工作的知识产权单位属西安电子科技大学。学校有权保留送交论文的复印件，允许查阅和借阅论文；学校可以公布论文的全部或部分内容，可以允许采用影印、缩印或其它复制手段保存论文。同时本人保证，毕业后结合学位论文研究课题再撰写的文章一律署名单位为西安电子科技大学。

（保密的论文在解密后遵守此规定）

本人签名：______ 日期：2010.3.16

导师签名：______ 日期：2010.3.16

第一章 绪 论

本章从计算机的发展和应用需求出发说明对并行计算研究的必要性以及紧迫性。以及并行计算在安全算法中的地位。最后简单介绍后续章节的内容安排。

1.1 并行计算的发展背景

1.1.1 计算机科学的发展

1965年，戈登·摩尔(Gordon Moore)发现了这样一条规律：半导体厂商能够集成在芯片中的晶体管数量大约每18~24个月翻一番---摩尔定律[1]。

在过去的四十年中，摩尔定律一直引导着计算机设计人员的思维和计算机产业的发展。而计算机也从晶体管数字计算机发展到集成电路数字计算机、大规模集成电路数字计算机。晶体管尺寸缩小是集成电路集成度增加、性能提高的主要方法，但是晶体管的尺寸缩小必将有一个极限。若不能在设计原理、工艺技术和原材料方面有所突破，再往后将无法在更大程度上缩小晶体管尺寸，届时摩尔定律将走到终点。

随着应用和科学的发展，人们对计算机性能的要求越来越高，而受到来自硬件技术的限制，计算机体系结构设计师们试图采用其他的技术来提高处理器性能，如指令集并行技术，使不相关的指令得以并行执行，进而保持处理器的执行单元尽量处于工作状态。

随着软件技术的不断发展以及需求的提高，逐渐出现了多线程技术以及支持超线程的CPU。2005年4月18日，Intel全球同步首发基于双核技术产品Intel Pentium D处理器，正式揭开x86处理器多核新时代，从而实现了真正意义上的并行。

1.1.2 对并行计算的迫切需求

并行计算，就是在并行计算机上所做的计算，它和常说的高性能计算是同义词，因为任何的高性能计算总离不开使用并行技术。随着计算机和计算方法的飞速发展，几乎所有的学科都走向定量化和精确化，在理论模型复杂甚至理论尚未建立，或者实验费昂贵甚至试验无法进行的情况下，此时计算就成为求解问题的唯一或主要手段。

在当代科学与工程问题中，如1993年美国科学、工程、技术联邦协调理事会向国会提交了题为“重大挑战项目[2][3]：高性能计算和通信”的报告，其中

汇总了磁记录技术、新药设计、高速民航、催化作用、燃料燃烧、海洋建模、臭氧耗损、数字解析、大气污染、蛋白质结构设计、图像理解、密码破译等，都对计算性能有着极高的要求。在其他的应用领域中，计算性能也起着关键性的作用，它的效率甚至影响着结果的准确性、精确度，比如数值气象预报、商务应用、网络计算等，这些应用都迫切的需要高性能的计算来支撑。随着计算机的高速发展，研究并行计算机上的计算已经成为历史的潮流，必然的趋势。

1.2 安全算法的分析研究

安全算法包括密码安全检测算法，随机序列检测算法等等。毋庸置疑，对安全算法的研究直接影响甚至决定着整个应用系统的安全。

密码安全检测算法。密码算法的安全性取决于哪些因素呢？对称密码的安全性取决于密钥的安全，如果一个对称密码系统的密钥被攻破，则意味着整个系统被攻破。非对称密码的安全性取决于数学的难解问题。这两种密码体制，都可以用穷搜索或者其他的方法进行攻击，串行计算中相对安全的密码算法，在并行计算中不一定安全，甚至很容易被攻击。串行中需要一个月才能找到密码算法的密钥，在高性能计算机中，可能只需要 1~2 天。

随机序列检测算法，包括经验性检测算法和统计性检测算法。一般用以检测随机数的随机性。被广泛使用的随机数，其随机特性直接影响着整个系统的性能。而现有的检测算法均有其局限性，没有一个检测算法足以说明一个随机序列具有随机性。随着并行伪随机数生成器的提出，对其实现方式的研究和检测也面临着新的问题。

高性能计算机的出现，使得密码算法的并行计算研究迫在眉睫。而对密码算法的分析也不只局限于串行，并行分析也已成为一种及其有效的方法。而对现有的安全算法的并行分析，并不只是将其简单的并行化，在串行分析中安全的算法，并不足以证明其在并行环境下也是安全的，因此并行计算的出现使得对安全算法的分析与研究变得更加复杂。

1.3 论文的研究内容及论文安排

安全算法是一个特殊的领域，它有着不同于其他应用的特点，本文针对其算法特性，对安全算法进行并行计算的研究与分析。

安全算法包括密码安全检测算法，随机序列检测算法，本文主要总结了这些算法的一般思路，提出在并行算法设计上的实现技巧和算法实现中可能遇到的问题及其解决方案，对现有的并行伪随机数生成器进行并行实现和检测研究。最后设计并实现了两种 MD6 的并行实现方法。

论文分为六章，各章内容如下：

第一章：简单介绍了对安全算法进行并行计算研究的必要性和紧迫性。

第二章：首先介绍了有关于并行计算的一些发展与分析，而后概述并行设计的一般技术以及可能遇到的问题及其解决方案。

第三章：总结安全算法的一般思想步骤，设计一般可行的并行实现方法。简单介绍现有的可实现多线程功能的函数库。并重点使用OpenMP对算法思路进行并行实现。

第四章：介绍现有的并行伪随机数生成器的优缺点及其并行检测技术，并设计生成器的并行实现算法，给出检测技术的并行实现思路，总结检测技术的发展趋势。

第五章：提出两种MD6的并行实现方法，对其程序实现结果进行分析对比，直接给出了并行程序设计过程中必须要考虑和解决的问题，同时揭示了OpenMP并行设计固有的局限性。

第六章：总结了全文所做的工作，提出了并行研究的发展方向。

第二章　并行计算的实现技术

计算机的发展从主频的提高到改变硬件构造，最后发展到拥有高主频的并行机。而对应的软件技术也有了很大的发展，包括多线程技术，并行实现函数库。实现并行计算要求程序设计者需将软件技术结合硬件结构，才能编写出高效的并行代码。由于并行计算打破了传统的串行实现方式，对程序设计者也提出了新的挑战。

2.1　硬件背景

2.1.1 计算机的发展

1946 年，世界上第一台电子数字计算机（ENIAC）在美国诞生。这台计算机共用了 18000 多个电于管组成，占地 170m^2，总重量为 30t，耗电 140kw，运算速度达到每秒能进行 5000 次加法、300 次乘法。

之后，电子计算机在短短的 50 多年里经过了电子管、晶体管、集成电路（IC）和超大规模集成电路（VLSI）四个阶段的发展，使计算机的体积越来越小，功能越来越强，价格越来越低，应用越来越广泛，目前正朝智能化（第五代）[8]计算机方向发展。

(1) 第一代电子计算机

第一代电子计算机是从 1946 年至 1958 年。它们体积较大，运算速度较低，存储容量不大，而且价格昂贵。使用也不方便，为了解决一个问题，所编制的程序的复杂程度难以表述。这一代计算机主要用于科学计算，只在重要部门或科学研究部门使用。

(2) 第二代电子计算机

第二代计算机是从 1958 年到 1965 年，它们全部采用晶体管作为电子器件，其运算速度比第一代计算机的速度提高了近百倍，体积为原来的几十分之一。在软件方面开始使用计算机算法语言。这一代计算机不仅用于科学计算，还用于数据处理和事务处理及工业控制。

(3) 第三代电子计算机

第三代计算机是从 1965 年到 1970 年。这一时期的主要特征是以中、小规模集成电路为电子器件，并且出现操作系统，使计算机的功能越来越强，应用范围越来越广。它们不仅用于科学计算，还用于文字处理、企业管理、自动控制等领域，出现了计算机技术与通信技术相结合的信息管理系统，可用于生产管理、交通管理、情报检索等领域。

(4) 第四代电子计算机

第四代计算机是指从 1970 年以后采用大规模集成电路（LSI）和超大规模集成电路（VLSI）为主要电子器件制成的计算机。它的另一个重要分支是以大规模、超大规模集成电路为基础发展起来的微处理器和微型计算机。

微型计算机的性能主要取决于它的核心器件——微处理器（CPU）的性能。目前的电子计算机虽然能以惊人的信息处理来完成人类无法完成的工作（例如遥控已发射的火箭），但是它仍不能满足某些科技领域的高速、大量的计算任务的要求。例如，原子反应堆事故和核聚变反应的模拟实验、资源探测卫星发回的图象数据的实时解析、飞行器的风洞实验、天气预报、地震预测等要求极高的计算速度和精度，都远远超出目前电子计算机的能力极限。由此可见，当今的电子计算机已不能适应信息社会的需要，必须在崭新的理论和技术基础上创制新一代计算机。

(5) 第五代计算机

第五代计算机是把信息采集、存储、处理、通信同人工智能结合在一起的智能计算机系统。它能进行数值计算或处理一般的信息，主要能面向知识处理，具有形式化推理、联想、学习和解释的能力，能够帮助人们进行判断、决策、开拓未知领域和获得新的知识。人-机之间可以直接通过自然语言（声音、文字）或图形图象交换信息。第五代计算机又称新一代计算机。它突破了传统的冯·诺伊曼式机器的概念，舍弃了二进制结构，把许多处理机并联起来，并行处理信息，速度大大提高。1991 年，美国加州理工学院推出了一种大容量并行处理系统，用 528 台处理器并行进行工作，其运算速度可达到每秒 320 亿次浮点运算。

2.1.2 并行机的发展

开发并行度和提高性能是研究并行计算机体系结构的根本出发点。以下介绍对微处理器级开发并行度的过程，而并行机体系结构发展提高了计算机的使用性能。

(1) 微处理器级并行度的开发[3]

位级并行(1970 年-1986 年) 处理器芯片上位并行的演变。由 4 位微处理器芯片到 8 位、16 位。20 世纪 80 年代中期 32 位的微处理器出现，而 10 年之后才出现部分采用 64 位操作的芯片。

指令集并行(20 世纪 80 年代中期-90 年代中期) 并发地执行多条机器指令的部分叫做指令集并行。全字长操作意味着指令执行的基本步(指令译码、整数运算、地址计算)可在单周期内完成。精简指令集计算机 RISC 方法展示了平

均几乎每个周期可执行一条指令。RISC 微处理器性能的进展，开拓了指令级并行度。

线程级并行(2000 年以后) 单控制线程内的指令级并行度是有限的，为了获得可观的并行度，必须同时施行多控制线程，即一台处理机有多个控制线程，能同时执行多条指令序列。多线程控制为大型多处理机隐藏掉较长的时延提供了一种有效机制。

(2) 并行结构的发展变化

SIMD 阵列处理机(1964 年-1975 年) 按照 Flynn 的分类法，阵列处理机属于单指令多数据流 SIMD 并行机，其名称反映出多台处理器排成阵列的拓补结构，利用资源重复的方法开拓并行性。单一的指令运行在大型规则的数据结构上，使阵列处理机也常叫做数据并行结构。

并行向量处理机 PVP (1976 年-1990 年) 在向量机中，标量处理器被集成为一组功能单元，它们以流水线方式执行存储器中的向量数据。能够操作于存储器中任何地方的向量就没必要将应用数据结构映射到不变的互连结构上，从而大大地简化了数据对准的问题。

大规模并行处理机 MPP(1990 年-1995 年) 分布式存储的多指令多数据流 MIMD 计算机。其主要特点为采用高通信带宽和低延迟的互联网络，能扩放至成百上千个处理器。主要应用于科学计算、工程模拟和信号处理等以计算为主的领域。

各种体系结构并存(1995 年后) PVP、MPP、SMP(对称多处理机)、DSM(分布共享存储)、COW(工作站机群)。SMP 的系统是对称的，每个处理器可等同地访问共享存储器、I/O 设备和操作系统服务。正是对称，才能开拓较高的并行度；也正是共享存储，限制了系统中的处理器不能太多，同时总线和交叉开关互连一旦做成也难于扩展。DSM 结合了 MPP 和 SMP 的优点。COW 的每个节点都是一个完整的工作站，一个节点可以是一个 PC 或 SMP，节点间通过一种低成本的商品网络互连。

2.2 并行程序设计

并行程序设计的整个概念都集中在设计、开发、设置应用程序的多线程和协调各线程及其操作之间的关系这四个方面。

2.2.1 设计方式

将串行转换成并行程序，设计人员必须将程序中能够并行执行的部分识别出来。将应用程序划分成多个独立的任务，并确定这些任务之间的相互依赖关系的

过程称为分解。分解问题的方式主要有三种：任务分解、数据分解和数据流分解[1]。

(1) 任务分解（功能并行性）。对应用程序根据其执行的功能进行分解的过程称为任务分解。任务分解是一种能够简单实现并行执行的方法。开发人员可以对众多的独立任务进行调度，实现并行。当然开发人员应该尽量避免这些任务之间的冲突。

(2) 数据分解（数据并行性），是将应用程序根据各任务所处理的数据来进行分解的方法。一般来讲，能够按照数据分解方式进行分解的应用程序都包含多个线程，这些线程分别对不同的数据对象执行相同的操作。数据分解方式所能处理的问题规模随着处理器核数量的增加而增长。这就意味着采用数据分解之后，在相同的时间内能够完成更多的工作量。

(3) 数据流分解（流水线技术）。很多情况下，当对一个问题进行分解的时候，关键问题不在于采用一些什么任务来完成这个工作，而在于数据在这些任务之间是如何流动的。这种时候就需要采用数据流分解方式，根据任务之间的数据流关系对问题进行分解。当一个线程的输出为另一个线程的输入时，应如何减少由于第一个线程的延迟而引起的第二个线程的暂停时间。其基本思想是将一个计算任务t分成一系列子任务$t_1,t_2,\cdots,t_m$，使得一旦t_1完成，后继的子任务就可立即开始，并以同样的速率进行计算。

各种分解方式都有自身的优势。根据不同的问题域和条件，开发人员必须经过认真的计划和测试，才能决定采用何种并行程序设计方法来实现应用程序的正确方案。相对于传统单线程程序设计而言，并行程序设计更需要以丰富的程序设计经验作为背景，从而对时序和评估作出正确的判断。

2.2.2 性能测评

一般来讲，采用多线程来实现应用程序的一个源动力就是性能，那么如何去衡量并行程序设计所带来的性能收益？直觉告诉我们，如果我们将任务划分为许多不同的子任务，然后同时处理这些子任务，就能够大幅度提高性能。在子任务完全不相关的情况下，可以获得性能提升是显然的，但是大多数实际情况并没有如此简单。一种衡量指标就是用最优串行算法的执行时间除以并行程序的执行时间所得到的比值，这就是加速比[1][2][3]的概念，加速比能够准确描述对程序并行化之后所获得的性能收益。

$$\text{加速比}(n_t)=\frac{\text{最优串行算法的执行时间}}{\text{并行程序的执行时间}(n_t)} \quad \text{式(2-1)}$$

(1) Amdahl 定律。用以计算并行程序相对于最优串行算法在性能提升上的理论最大值。

$$加速比=\frac{1}{S+(1-S)/n} \qquad 式(2\text{-}2)$$

式(2-2)中，S 表示执行程序中串行部分的比例，n 表示处理器核的数量。当 $n\to\infty$ 时，公式(2-2)就变成了公式(2-3)：

$$加速比=\frac{1}{S} \qquad 式(2\text{-}3)$$

公式(2-3)表明，串行部分执行时间为 S 的应用程序并行化之后能够达到的加速比上限为 $1/S$，无限的处理器核并不能带来性能上的无限增长，无论如何，程序性能总是存在一个上限，即应用程序从可并行部分所获得的性能提升最大值受限于串行部分所占的比例。

Amdahl定律本身做了几个假设：1) 最优串行算法的性能严格受限于CPU资源的可用性。但是实际情况并非如此，多核处理器可能会为每个核实现一个单独的cache，这样，cache中就能够存放更多的数据，从而降低了存储延迟。2) 串行算法是给定问题的最优解决方案。但是，一些问题本质上就是并行的，因此采用并行实现时所需的计算步骤就会比串行算法减少很多。3) 关于问题大小的假设。Amdahl定律假设在处理器核数量增长的时候，问题的规模却保持不变。这在大多数情况下都是不成立的。一般来讲，当给予更多的计算资源的时候，问题规模都会随之增大以适应资源规模的扩大。事实上，应用程序的执行时间保持不变才是多数情况。

(2) Gustafson 定律：

$$扩展加速比=N+(1-N)S \qquad 式(2\text{-}4)$$

式(2-4)中，N 表示处理器核的数量，S 表示执行程序中串行部分的比例。从公式看，加速比是线性增长的，属增长式回报。Gustafson定律对多核处理器上并行计算的潜力有着更加现实和乐观的展望。

由以上加速定律可知，增加处理器数和求解问题的规模都可提高加速比，而影响加速的因素有：①求解问题中的串行分量；②并行处理所引起的额外开销(通信、等待、竞争、冗余操作和同步等)；③加大的处理器数超过了算法中的并发程度。增加问题的规模有利于提高加速的因素是：①较大的问题规模可提供较高的并发度；②额外开销的增加可能慢于有效计算的增加；③算法中的串行分量比例不是固定不变的。一般情况下，增加处理器数，是会增大额外开销和降低处理器的利用率的，所以对于一个特定的并行系统、并行算法或并行程序，它们能否有效利用不断增加的处理器的能力是受限的，而度量这种能力就是可扩放性这一指标。

可扩放性最简朴的含义是在确定的应用背景下，计算机系统性能随处理器数的增加而按比例提高的能力。其研究的主要目的是：①确定解决某类问题用何种

并行算法与何种并行体系结构的组合，可以有效地利用大量的处理器；②对于运行于某种体系结构的并行机上的某种算法，根据算法在小规模处理机上的运行性能，预测该并行算法当移植到大规模处理机上后运行的性能；③对固定的问题规模，确定在某类并行机上最优的处理器数与可获得的最大的加速比；④用于指导改进并行算法和并行机体系结构，以使并行算法尽可能地充分利用可扩充的大量处理器。

2.3 具体实现技术

针对不同的操作系统，不同的计算机体系结构，有着不同的实现技术。在诸如SMP的共享存储系统中，可以使用OpenMP这种API，分布式存储系统中，可以使用MPI这种被广泛采用的消息传递标准。在Windows平台下可以通过Windows的线程库来实现多线程编程，可以利用Win32 API或MFC以及.Net Framework提供的接口来实现。Pthreads现在已成为Linux操作系统中多线程接口的标准，并且也已广泛使用在大多数的UNIX平台上。针对Windows操作系统，Pthreads也存在一个开放源代码的版本，称为pthreads-win32。

一般来讲，应用程序线程可以采用内建的API调用来实现，这就是线程在应用程序中的实现方式。最常用的API是OpenMP库和显式低级线程库(例如Pthreads库和Windows线程库)。具体选择哪种API是由具体的需求和系统平台来决定的。一般来讲，采用低级线程库的好处就是可以对线程进行细粒度的控制。相对来说，OpenMP库的使用更加方便，并且实现多线程的方式更加友好。采用OpenMP进行实现需要支持OpenMP API的编译器。采用低级线程实现则只需要访问操作系统的多线程库即可[1][4]。

2.3.1 共享存储系统

(1) Win32 API 是 Windows 操作系统为内核以及应用程序之间提供的接口，将内核提供的功能进行函数封装，应用程序通过调用相关的函数获得相应的系统功能。包括创建线程、管理线程、终止线程、线程同步等。用 Win32 API 直接编写应用程序要求程序员对 Windows 操作系统具有一定的了解，否则会占用程序员很多的时间来对系统的资源进行管理，降低程序员的工作效率。但直接用 Win32 API 编写的应用程序，程序的执行代码小，运行效率高。

(2) MFC 是由微软公司提供的，称为“微软基础函数类库”(Microsoft Foundation Classes)，即用类库的方式将 Win32 API 进行封装，以类的方式提供给开发者。在 MFC 类库中，提供了对多线程的支持。由于 MFC 是在 Win32 API 基础之上进行封装的，其基本原理与 Win32 API 的基本实现原理很类似，且 MFC 对同步对象作

了封装，因为对用户编程实现来说更加方便。由于 MFC 具有快速、简捷、功能强大等特点，因此深受广大用户的喜爱。

(3) .Net Framework 由两部分构成：公共语言运行库 CLR(Common Language Runtime)和 Framework 类库 FCL(Framework Class Library)。CLR 包括自己的文件加载器、垃圾收集器、安全系统等，提供了一个可靠而完善的多语言运行环境。CLR 是一个软件引擎；用于加载应用程序、检查错误、进行安全许可认证、执行和清空内存。Framework 类库提供了所有应用程序模型都要使用的一个面向对象的 API 集合。.Net 基础类库的 System.Threading 命名空间提供了大量的类和接口来支持多线程。所有与多线程机制相关的类都存放在 System.Threading 命名空间中。其中 Thread 类用于创建及管理线程，ThreadPool 类用于管理线程池等，此外还提供线程间通信等实际问题的机制。

(4) POSIX(Portable Operating System Interface)Threads，即 Pthreads，代表官方 IEEE POSIX1003.IC-1995 线程标准，系由 IEEE 标准委员会所建立，其功能和界面类似于 Solaris 线程的功能与界面。Pthreads 的主要功能集中在线程创建与终止、线程同步以及一些辅助功能上。而类似于线程优先级之类的功能并不包括在核心 Pthreads 库中，而是作为可选功能的一部分由制造商实现。

(5) OpenMP 标准诞生于 1997 年，目前由其结构审议委员会正在制定并即将推出 OpenMP3.0 版本。OpenMP 是一种针对共享内存的多线程编程技术，由一些具有国际影响力的大规模软件和硬件厂商共同定义的标准。它是一种编译指导语句，指导多线程、共享内存并行的应用程序编程接口(API)。它是一种面向共享内存以及分布式共享内存的多处理器多线程并行编程语句。OpenMP 具有良好的可移植性，支持多种编程语言，包括 Fortran77、Fortran90、Fortran95 以及 C/C++；同时在平台支持上，OpenMP 能够支持多种平台，包括大多数的类 UNIX 系统以及 Windows NT 系统。OpenMP 的提出，是希望遵循该并行程序模型的并行程序，可以在不同的厂商提供的共享存储体系结构间比较容易地移植。

OpenMP 可以根据目标系统自动使用适当数量的线程，因此只要有可能的话，开发人员都应该考虑使用 OpenMP 来简化串行代码到并行代码的转换，并使代码更具有可移植性，更加易于维护。只有在无法使用 OpenMP 的情况下，才应该考虑系统内嵌的或半内嵌的多线程机制，如 Windows 多线程 API 和 Pthreads。如果打算使用 C 语言开发多线程程序，并且需要一个能比 OpenMP 提供更多直接控制的可移植的多线程 API，那么 Pthreads 会是一个不错的选择[1][4]。

2.3.2 分布存储系统

从并行程序设计的角度来看，分布存储系统的主要特点是：系统通过互连网

络将多个处理器连接起来，每个处理器均有自己的局部存储器，所有的局部存储器就构成了整个地址空间。

分布存储系统有两种并行编程模型：数据并行模型和消息传递模型。所谓基于消息传递的并行编程，是指用户必须显式地通过发送和接受消息来实现处理器之间的数据交换。每个进程均有自己独立的地址空间，一个进程不能直接访问其他进程中的数据，这种远程访问必须通过消息传递来实现。

MPI 是 1994 年发布的一种消息传递接口。伴随着高性能计算技术的普及，尤其是集群系统的普及，MPI 标准如今已经成为事实意义上的消息传递并行编程标准，也是最为流行的并行编程接口。

MPI 具有许多优点：具有可移植性和易用性；有完备的异步通信功能；有正式和详细的精确定义。在基于 MPI 编程模型中，计算是由一个或多个彼此通过调用库函数进行消息收、发通信的进程所组成。在绝大部分 MPI 实现中，一组固定的进程在程序初始化时生成，一个处理器生成一个进程。这些进程可以执行相同或不同的程序（相应地称为 SPMD 或 MPMD 模式)。

目前，MPI 支持多种编程语言，包括 Fortran77、Fortran90 以及 C/C++；同时，MPI 支持多种操作系统，包括大多数的类 UNIX 系统以及 Windows 系统；MPI 还支持多核、对称多处理机、集群等各种硬件平台[1][2][4]。

2.4 程序设计中遇到的问题与解决方案

2.4.1 面临的问题

因为多线程技术支持多个操作同时执行，所以能够显著提高程序性能。但是并行程序设计人员必须认识到：多线程同时也使得应用程序行为变得更加复杂，因此需要更加深入的思考才能正确驾驭。多线程技术使程序行为变得更加复杂的根本原因在于：程序会同时发生多个动作。对这些同时发生的动作以及它们之间的交互进行管理将面临四个方面的挑战[1]：

(1) 同步。指两个或者多个线程协调其行为的过程。例如，一个线程停下来等待另一个线程完成某项任务。在此，要避免死锁问题。

(2) 通信。指与线程之间交换数据相关的带宽和延迟问题。

(3) 负载平衡。指多个线程之间工作量分布的情况。负载平衡能够使个线程的工作量平均分配。

(4) 可扩展性。衡量在性能更加强劲的系统上运行软件时能否有效利用更多线程的指标。例如，如果一个应用程序是面向四核系统编写的，那么当该程序在八核系统上运行时，其性能是否能够线性增长。

多核属于共享存储的 CPU，在多核环境下编程与单核编程有着很大的不同，

将不可避免的遇到很多问题，如共享数据的访问，负载平衡和可扩展问题[1][5]。

(1) 并发性问题。当程序并发地在多个 CPU 核上执行后，由于所有程序都需要并发执行，因此创建线程成了首要解决的问题。多核编程时，由于所有代码都需要并发执行，不再限于将某个函数线程化；在函数的内部也需要创建线程使关键计算并行执行，因为创建线程是一个很频繁的工作。如果仍然仅仅使用 API 创建线程，那么将给程序员设计程序带来很多额外的工作开销，因此需要更多有效的方法来帮助程序员更快地写出并行化的程序。而 OpenMP 就是一个可以帮助程序员快速创建并行程序的标准库。

(2) CPU 饥饿问题。程序并行化后，会遇到共享数据访问的问题。如果多个线程对共享数据都是只读操作，那么对共享数据的访问不需要加锁保护。如果多个线程对共享数据的访问存在写操作，那么对共享数据的访问必须加锁保护。在有锁保护的共享数据访问模型中，一旦一个线程取得了锁，那么其他线程在进行锁操作时都必须等待。这样，就只有一个线程在运行，只有一个 CPU 核在运行，其他 CPU 核都处于饥饿状态。

(3) 任务的分解和调度问题。程序并发运行后，除了锁竞争外，还有一个问题也会导致 CPU 饥饿，那就是任务的分解和调度问题。所谓任务指的是执行的某个程序功能；任务和线程不同，一个线程内可以执行一个或多个任务。如果任务分解的不好，就很难均匀地分配到各个 CPU 核上；对于分解好的多个任务，如何将其均匀地分配到各个 CPU 核上进行计算是很重要的问题—任务调度。任务分解和调度的好坏会影响各 CPU 核上计算的负载均衡。在实际运用中，任务数量一般会比较多，任务间耗时差距也非常大，要均匀地将任务分配到各 CPU 核上执行并非易事；当任务间的执行存在依赖关系时，任务调度又复杂了一些；还有许多任务是动态产生的，事先并不知道任务耗时，如何调度这些动态任务、使计算工作均匀地分配到各个 CPU 核上则是更大的挑战。事实上，任务的调度算法问题是一个 NP 难题。

2.4.2 解决方案

同步是对线程执行的顺序进行强行限制的一种机制，用来控制线程执行的相对顺序，可以有效解决任何线程间的冲突，而这些冲突有可能会导致线程的执行出现异常行为。简而言之，同步主要用于协调线程执行和管理共享数据。在共享存储器系统中，广泛使用的同步操作主要有两种类型：互斥和条件同步。

有两个概念：

(1) 临界段[1]，指包含有共享变量的一段代码块，这些共享变量和多个线程之间存在相关关系。采用合适的同步技术，可以保证任何时刻临界段仅被一个线程

访问。多线程程序设计的主要挑战在于需要以多个线程执行互斥操作的方式实现临界段，并保证多个线程不会同时访问临界段。

临界段也可以叫做同步块。临界段的大小十分重要，它主要依赖于临界段的使用方式。在实际使用时，应尽可能地减少临界段的大小。较大的临界段代码块应该分割成多个较小的代码块，这对于有可能出现明显线程竞争的代码块尤为重要。

(2) 死锁[1]。当一个线程因等待另一个线程的资源而阻塞，而同时该资源永远不会被释放时，就发生死锁。在不同的情况下，可能出现不同类型的死锁：自死锁，递归死锁和错序死锁。

同步一般通过三种原语实现：信号量、锁和条件变量[1][4]。这三种原语的使用依赖于应用程序的需求。这些同步原语可以通过原子操作和使用适当的存储栅栏指令实现。存储栅栏，也称为存储屏障，是一种依赖于处理器的操作，它可以将存储器操作维持合理的顺序以确保所有的线程都可以看到其他线程的存储器操作。为了隐藏这些同步原语的粒度，可采用更高级别的同步操作，这样应用程序开发人员就可以较少地考虑同步的内部细节。

(1) 信号量。信号量是第一套用于实现并行进程同步互斥操作的软件原语，它于 1968 年由著名的数学家 Edsger Dijkstra 在其论文“多道程序设计系统”中提出并做了介绍。Dijkstra 阐明了同步操作可以由传统机器指令或层次结构来实现。他提出信号量可以用一个整数 sem 表示，对信号量有两个基本的原子操作：P 操作（Proberen，检测操作）和 V 操作（verhogen，增量操作），这些原子操作也被称为同步原语。其中 P 操作表示使线程“延迟”或“等待”，V 操作表示“移除栅栏”或“释放线程”。用这两个同步原子操作可以实现对信号量 S 的如下操作[1]：

线程T执行的P操作：

```
P(s):原子操作{sem=sem-1;temp=sem}
     if(temp<0)
     {线程T被阻塞并插入到信号量s的等待队列中}
```

线程T执行的V操作：

```
V(s):原子操作{sem=sem+1;temp=sem}
     if(temp≤0)
     {从信号量s的等待队列中移出一个线程}
```

如果信号量为 0，则 P 操作将阻塞一个请求线程，而 V 操作独立于 P 操作，它通知一个被阻塞的线程可恢复执行。sem 为正表示可无阻塞执行的线程数目，为负则表示已被阻塞的线程数。当 sem 值为 0，表示没有等待的线程。如果一个线程对 sem 做减量操作，则该线程被阻塞在一个等待队列中。

(2) 锁[1]。锁类似于信号量，不同之处在于一个线程在同一个时刻只能使用一

个锁。锁对应的两种原子操作为：

Acquire()：获取原子操作，等待锁状态变为未加锁状态，然后再将锁状态置为已加锁。

Release()：释放原子操作，将锁状态由已加锁变为未加锁。

一个锁至多由一个线程获得。线程对共享资源进行访问之前必须先获取锁；否则，线程保持等待，直到该锁可用。一个线程访问共享数据的流程是，获取锁，之后独占共享数据进行排它性操作，然后释放锁供其他线程使用。

锁粒度。设t_s表示锁内计算时间，大小由共享资源的操作时间决定，与共享资源类型有关，并且与程序员的程序设计有关；t_1表示加锁操作和解锁操作耗费的时间，如果 CPU 核的速度固定，那么它为一常量。$f_k = t_s/t_1$ 表示锁粒度因子，反映了一个线程内锁操作的粒度关系。将粒度非常小的锁称为细粒度锁，采用细粒度锁不仅可以取得分时效果，还可以使得加速比性能得到更大的提升。锁粒度因子的大小取决于锁内计算的大小，因此要想使锁粒度变小就必须减少锁内计算量。锁内计算的大小是受共享资源的操作时间和锁内非共享资源操作时间决定的。因此可以从两个方面来减小锁粒度因子。①将非共享资源操作从锁内移到锁外；②将大的共享资源操作设计成小的共享资源操作，减少共享资源操作时间。

(3) 条件变量。除了没有存储值与操作绑定外，条件变量和信号量机制具有相同的语义。也就是说条件变量实质上并没有需要检验的条件值。当特定条件满足时，线程等待或者唤醒其他合作线程。当有很多线程试图获取锁并需要在这些线程之间进行调度时，条件变量要优于锁机制。条件标量 C 使用锁 L 来完成对共享数据的访问。可以对条件变量 C 执行 3 种原子操作[1]：

wait(L)：原子操作，该操作释放自身持有的锁并等待，其执行完毕即表示锁已被其他线程获得。

signal(L)：允许其中一个等待线程往下执行，该操作执行完毕即表示锁仍然被持有。

broadcast(L)：广播，允许所有等待线程往下执行，该操作执行完毕即表示锁仍然被持有。

当需要控制很多线程时，推荐使用 signal 函数。基于广播机制的通知函数的开销很大，因此当要唤醒所有的等待线程时，需格外小心。但在一些场合，广播机制会很高效，例如，一个“写”锁可以使用广播机制来允许所有的“读者”线程同时继续执行。

任务的分解和动态调度问题的解决思路将在第三章中提出。

2.5 本章小结

本章介绍了计算机硬件的发展历史、多种计算机硬件架构以及在各种架构下的多种并行实现方式与技术。总结并行算法的一般设计技术和设计过程中面临的新的四大问题与其相对应的解决方案。

第三章 安全算法的并行实现与优化

安全算法包括密码安全检测算法，随机序列检测算法。这些算法都可以总结出一般的规律。针对这些规律，利用现有的并行实现技术对其进行并行设计。

3.1 安全算法的特性

在各种安全算法中，包括对密码算法和随机序列的检测与分析算法，基本都是以对数据集的搜索遍历为出发点，穷搜索对于每个算法都是有效的，而对安全算法研究的目的在于如何降低搜索的次数，提高算法的效率。比如分组密码中的差分线性分析，其数据集为明密文对，通过算法优化，可以使对数据集从复杂度为2^{56}（如 DES）降低到2^{30}；新统计测试中，数据集为所有满足条件的明文密文对；代数免疫中，数据集为满足条件的多项式理想集；平衡性检验中，数据集为至少2^{18}个不同的随机明文输入等等。这些算法中的数据集通常都是非常庞大的，而对数据集进行遍历检测时通常使用的是同一个检测算法，一般可采用数据划分方式进行并行计算。

当数据集确定时，相当于整个并行执行的规模已经确定，在此可以很简单的确定线程的数目以及给各个线程分配任务，达到负载平衡的效果；在对共享资源的访问方面，可以给出确定性的解决方案，明确的防止死锁和线程饥饿现象。

但是得到数据集的过程，通常是一个动态的过程，比如一个数据元素可以产生零个或多个不同的数据元素加入到数据集中。此时将使用特殊的数据结构对数据集进行存储，尤其是当数据量非常大时，还涉及到动态添加或删除数据结构的操作，此时无法简单的确定线程的数目和各个线程执行的工作负载。因此，可采用的一般设计思路是，在对存储数据集的数据结构进行添加删除时使用锁操作，具体实现思路将在 3.2.2 中给出。

数据集的大小决定程序的运行规模，当程序运行规模无法确定时，需要使用到任务的动态调度思路，即如何动态的给线程分配任务，如何最大化的利用计算机硬件资源，使所有的线程负载均衡？解决思路将在后续章节中提出。

3.2 基本并行算法设计

3.2.1 数据划分

根据不同的算法思路，数据划分可分为均匀划分、方根划分、对数划分等[2]。以下以并行快速归并排序算法[5]的设计思路为例：

如果要取得负载均衡，一种较好的方案是先将待排序区间划分成若干相等的

小区间，然后并行地对这些小区间进行排序，最后通过归并的方法将所有排好序的小区间归并成一个有序系列。由于归并时是一层层向上进行的，因此需要将区间划分成2^k个小区间，这样第 1 轮归并时，可以将2^k个小区间归并成2^{k-1}个区间，……，经过k轮归并操作就归并成一个有序的大区间。

1) 对待排序区间分解成2^k个小区间；
2) 使用多个线程对2^k个小区间进行排序（区间相互独立的，可并行执行）；
3) 设$i=k$；
4) 使用串行归并算法对2^i个小区间进行归并；
5) $i \leftarrow i-1$，若$i>0$，执行步骤 4)；
6) 得到有序区间。

步骤3~5:
```
    for(i=k;i>0;i--)
    {
#pragma omp parallel for num_threads(nCore)
        for(j=0;j<2^i;j+2)
        {   并行执行两两归并; }
    }
```

3.2.2 锁操作

锁属于操作系统中的一种同步操作原语，有只支持线程间同步的锁，也有用于进程间同步的锁。一般来说，用于进程间同步的锁也可以用于线程间的同步，但比只支持线程间同步的锁的效率要低得多。

一把锁在同一时刻只能由一个线程持有，不能被多个线程同时持有。在多个线程对共享数据进行操作时，还需要用到许多其他同步原语，如信号量、事件等，其实这些信号量操作都可以由锁操作通过一定的算法来实现。

锁的最主要特点就是互斥特性，当然还有些锁具有旋转特性、递归特性、多读单写特性[1]等。

(1) 互斥特性

1）基本互斥特性。互斥特性是锁的最基本属性，具有互斥特性的锁称为互斥锁，是最简单的一种锁。互斥锁有两个最主要的操作：获取锁和释放锁。当一个线程获取锁时，其他访问该锁的过程都会被阻塞。如果一个线程获取锁但还没有释放该锁，那么称该线程持有该锁。同一时刻只能有一个线程持有锁。

使用互斥锁时需要注意的是，在获取锁后必须释放锁，否则其他线程都会被阻塞不能运行，出现死锁现象。

互斥锁属于基本的同步原语，其他的各种同步操作（如 Windows 系统中的 Semaphore、Event 等）都可以由互斥锁通过一定的算法来实现。

2）条件互斥特性。条件互斥特性是满足一定条件下的互斥特性，比如奇数信号量就是一种条件互斥锁，在释放锁的操作中将计数加 1，在获取锁的操作中将计数减 1，如果计数为 0，则获取锁的操作会被阻塞，直到其他线程进行释放锁操作使计数不为 0 时才能获取锁。

事件也是一种条件互斥锁，事件具有唤醒和非唤醒两种状态，当等待一个非唤醒的事件时，线程会被阻塞，知道其他线程将这个事件唤醒为止。如果一个线程成功等到一个事件，那么它会重新将这个事件设置为非唤醒状态。在同一时刻，只能有一个线程等到事件。

具有条件互斥特性的锁也都可以由最基本的互斥锁通过一定的算法来实现。

(2) 旋转特性

旋转特性是指在等待锁时并不立即使线程进入睡眠状态，而是在等待锁时不断循环尝试，直到最终获取锁。

具有旋转特性的锁称为旋转锁，旋转锁是一种用于多处理器系统的锁。旋转锁也具有互斥属性，即同一时刻只能有一个线程持有同一把锁。

旋转锁和互斥锁的区别是在等待锁的过程中不需要进行线程的挂起和唤醒，因此，如果等待锁的时间少于线程挂起和唤醒的时间，那么旋转锁将比非旋转锁有更好的效率。

实际情况中，旋转锁并不是无限旋转，有一个旋转次数设置，当旋转了规定次数还没有获取锁后，线程进入睡眠状态。旋转次数设置过少，将可能导致在旋转了规定次数后仍然获取不到锁；旋转次数设置过多，则旋转花费的时间可能超过线程上下文切换的时间。

(3) 递归特性

递归特性是指在同一线程内，没有释放锁的情况下就可以多次重复获取锁的特性。满足递归特性的锁称为递归锁。递归锁在同一线程内释放的次数必须与获取的次数一样多，否则其他线程将无法获取该锁。递归锁通常用于递归函数中，并且速度比普通互斥锁的速度慢。

(4) 多读单写特性

多读单写特性是指同一时刻只能有一个线程进行写操作，但是在没有写操作的情况下可以有多个线程同时进行读操作。具有多读单写特性的锁称为读写锁。

读写锁在读操作时，每个线程持有锁的时间一般为一次加锁操作的时间，最坏情况下读锁为执行两次加锁、解锁操作的时间，锁内计算时间较少，因此读写锁中读锁的粒度较小；读写锁是一种粒度较小的锁。

在上一节中提到对共享数据结构进行读写时应使用锁操作。常用的数据结构为数组、队列、链表、哈希表和树。一般而言，对数组的读写可能会存在伪共享的问题，具体参照 3.3.3，因此根据需要在写数组之前加锁或者临界区；队列的存

取结构也是数组，不同的是在对队列的头尾指标进行修改时，也需要加锁；链表中结点之间是依据指针指向的，一个结点的添加或者删除都影响着另一个结点，因此一个链表只能有一把锁；而哈希表一般用作索引，当索引指向链表或者树结构时，索引结构需要加一个锁类型变量；同样，当树的结点需要添加或删除时，当前结点与当前结点的子结点共享一把锁。

3.3 OpenMP 实现并行

简单介绍 OpenMP 的语法规则，简单函数操作。最后用 OpenMP 实现并行程序，并总结出并行实现的一般技巧。

3.3.1 OpenMP 简介

(1) OpenMP[1][5]概述。OpenMP 程序设计模块提供了一组与平台无关的编译指导(pragmas)、指导命令(directive)、函数调用和环境变量，可以显式的指导编译器如何以及合适利用应用程序中的并行性。对于很多循环来说，都可以在其开始之前插入一条编译指导，使其以多线程执行。开发人员不需要关心那些实质性的实现细节，如复杂的线程创建、同步、负载平衡和销毁工作等，这是编译器和 OpenMP 线程库的工作，开发人员只需要认真考虑哪些循环应该以多线程方式执行，以及如何重构算法以便在多核处理器上获得更好的性能等问题。当使用 OpenMP 将那些最耗时的循环，也就是所谓的“热点”以多线程执行的时候，OpenMP 的能力就体现出来了。

OpenMP 应用编程接口 API 是在共享存储体系结构上的一个编程模型，包含编译制导(Compiler Directive)、运行库例程(Runtime Library)和环境变量(Environment Variables)，还支持增量并行化(Incremental Parallelization)。

OpenMP 不包含的性质：

① 不是建立在分布式存储系统上的；

② 不是在所有的环境下都是一样的；

③ 不是能保证让多数共享存储器均能有效的利用；

OpenMP 的编程模型以线程为基础，通过编译指导语句来显示地指导并行化，为编程人员提供了对并行化的完整的控制。

OpenMP 的执行模型采用 Fork-Join[1][5]的形式。在开始时，只有一个叫做主线程的运行线程存在，在运行过程中，当遇到需要进行并行计算的时候，派生出(Fork)线程来执行并行任务，在并行代码结束执行后，派生线程退出或挂起，控制流程回到单独的主线程中（Join）：

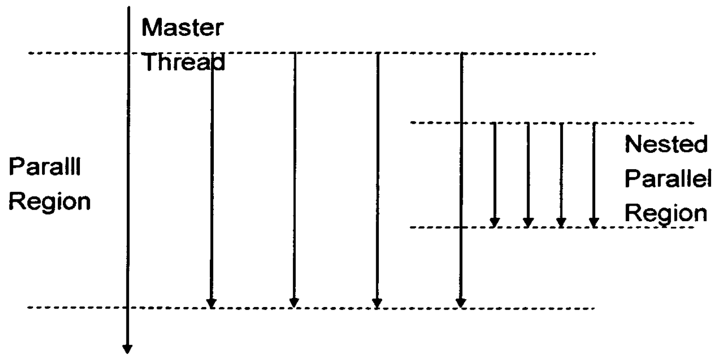


图 3.1 OpenMP(Fork-Join)执行模式

(2) 函数介绍。OpenMP 中用于创建线程的命令主要有 parallel，for，sections[1][5] 等命令。以下将简要介绍。

OpenMP 的#pragma 语句的格式为：

#pragma omp directive_name …

具体格式：#pragma omp parallel [clause[[,]clause]…]newline

表 3.1 OpenMP 语句格式

#pragma omp	directive-name	[clause, ...]	newline
制导指令前缀。对所有的 OpenMP 语句都需要这样的前缀。	制导指令。在制导指令前缀和子句之间必须有一个正确的 OpenMP 制导指令。	子句。在没有其它约束条件下，子句可以无序，也可以任意的选择。这一部分也可以没有。	换行符。表明这条制导语句的终止。

① for 编译制导语句

for 语句指定紧随它的循环语句必须由线程组并行执行；

语句格式：#pragma omp for [clause[[,]clause]…] newline

[clause]=schedule，ordered，private，firstprivate，lastprivate，shared，reduction，nowait。

② sections 编译制导语句

sections 编译制导语句指定内部的代码被划分给线程组中的各线程，不同的 section 由不同的线程执行。section 语句格式：

```
#pragma omp sections [ clause[[,]clause]…] newline
    {
        [#pragma omp section newline]
        …
        [#pragma omp section newline]
        …
    }
```

在 sections 语句结束处有一个隐含的路障，使用了 nowait 子句除外。

③ single 编译制导语句

single 编译制导语句指定内部代码只有线程组中的一个线程执行。线程组中没有执行 single 语句的线程会一直等待代码块的结束，使用 nowait 子句除外。

语句格式：#pragma omp single [clause[[,]clause]…] newline

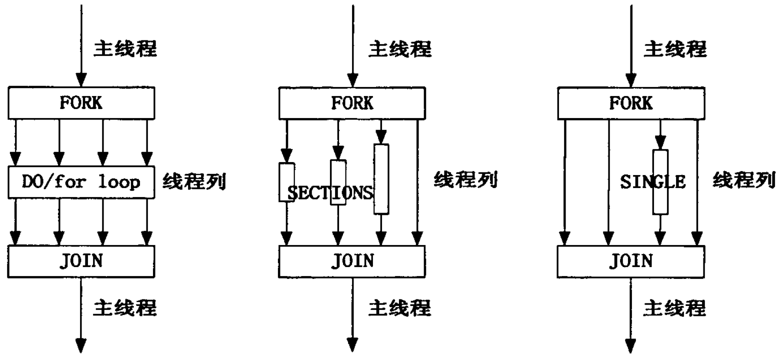


图 3.2 for，sections 和 single 执行模式

(3) 环境变量[5]

OMP_SCHEDULE：只能用到 for，parallel for 中。它的值就是处理器中循环的次数。

OMP_NUM_THREADS：定义执行中最大的线程数。

OMP_DYNAMIC：通过设定变量值 TRUE 或 FALSE，来确定是否动态设定并行域执行的线程数。

OMP_NESTED：确定是否可以并行嵌套。

(4) 循环并行化语句的限制

并不是所有的循环语句都能在其前面加上#pragma omp parallel 来实现并行化：

① 并行化的语句必须是 for 循环语句并具有规范格式；

② 能够推测出循环的次数；

③ 在循环过程中不能使用 break 语句，不能使用 goto 和 return 语句从循环中跳出，可以使用 continue 语句。

(5) parallel 编译指导语句的执行过程

当程序遇到 parallel 编译指导语句的时候，就会生成相应数目（根据环境变量）的线程组成一个线程组，并将代码重复地在各个线程内部执行；

parallel 的末尾有一个隐含的同步屏障（barrier），所有线程完成所需的重复任务后，在这个同步屏障处会和（join）。

(6) 根据线程号分配任务

每一个线程在执行的过程中的线程标识号[5]是不同的，可以根据这个线程标识

号给线程分配不同的任务。

例 1：

```
#pragma omp parallel private（myid）
    {
        nthreads=omp_get_num_threads（）;
        myid=omp_get_thread_num（）;
        get_my_work_done（myid,nthreads）;
    }
```

(7) OpenMP 支持两种不同类型的线程同步机制：互斥锁和事件通知机制[1][5]。如存在这样一种数据竞争：

例 2：

```
    int i; int max_num=-1;
#pragma omp parallel for
    for（i=0;i<n;i++）
        if（ar[i]>max_num）
            max_num=ar[i];
```

1）互斥锁机制。在 OpenMP 中，提供了三种不同的互斥锁机制用来对一块内存进行保护，它们分别是临界区（critical），原子操作（atomic）以及由库函数来提供同步操作。

① 在程序需要访问可能产生竞争的内存数据的时候，都需要插入相应的临界区代码。临界区编译制导语句的格式如下所示：

```
#pragma omp critical [（name）]
    Block
```

name 是一个临界区的属性，为临界区的一个命名操作。在对不同的内存区域进行保护的时候，如果两个线程实际上访问的并不是同一块内存区域，则不会产生冲突，也没必要对它们之间进行互斥锁的操作。

例 3：

```
    int i; int max_num_x=-1;
#pragma omp parallel for
    for（i=0;i<n;i++）
    {
#pragma omp critical （max_num_x）
        if（arx[i]>max_num_x）
            max_num_x=arx[i];
    }
```

② 原子操作是 OpenMP 编程方式给同步编程带来的特殊的编程功能，通过编译指导语句的方式直接获取了现在多处理器计算机体系结构的功能。通过#pragma omp atomic 编译指导语句提供；只能作用在语言内建的基本数据结构。

```
#pragma omp atomic
    x <binop>=expr，x++//or x--, --x, ++x
```

例 4：

```
    int counter=0;
#pragma omp parallel
    {
        for (int i=0;i<10000;i++)
#pragma omp atomic            //atomic operation
            counter++;
    }
```

③ 库函数[4]

OpenMP 通过一系列的库函数支持更加细致的互斥锁操作。编译制导语句进行的互斥锁支持只能放置在一段代码之前，作用在这段代码之上。程序员必须自己保证在调用相应锁操作之后释放相应的锁，否则就会造成多线程程序的死锁。

表 3.2 运行时库函数的互斥锁支持

函数名称	描　述
void omp_init_lock（omp_lock_t *）	初始化一个互斥锁
void omp_destroy_lock（omp_lock_t*）	结束一个互斥锁的使用并释放内存
void omp_set_lock（omp_lock_t *）	获得一个互斥锁
void omp_unset_lock（omp_lock_t *）	释放一个互斥锁
int omp_test_lock（omp_lock_t *）	试图获得一个互斥锁，并在成功是返回真（true），失败是返回假（false）

表 3.3 常用的库函数

函数名称	描　述
int omp_get_num_procs(void)	返回当前多处理机的处理器个数
int omp_get_num_threads(void)	返回当前并行区域中的活动线程个数
int omp_get_thread_num(void)	返回线程号
int omp_set_thread_num(int Num_Threadas)	设置并行执行代码时的线程个数。它可以覆盖 OMP_NUM_THREADS 环境变量的值

2）事件同步机制

用来控制代码的执行顺序，使得某一部分代码必须在其它的代码执行完毕之后才能执行。

OpenMP 中的事件同步主要包括[5]：同步屏障（barrier）、定序区段（ordered sections）、主线程执行（master）和隐含的同步屏障（barrier）。

在每一个并行区域都会有一个隐含的同步屏障；一个同步屏障要求所有的线程执行到此屏障，然后才能够继续执行下面的代码。

#pragma omp for，#pragma omp single，#pragma omp sections 程序块都包含自

己的隐含的同步屏障。

为了避免在循环过程中不必要的同步屏障，可以增加 nowait 子句到相应的编译制导语句中。

明确的同步屏障语句。在有些情况下，隐含的同步屏障并不能提供有效的同步措施，程序员可以在需要的地方插入明确的同步屏障语句#pragma omp barrier，在并行区域的执行过程中，所有的执行线程都会在同步屏障语句上进行同步：

```
#pragma omp parallel
    {
        initialization ( ) ;
#pragma omp barrier
        process ( ) ;
    }
```

(8) 内存模型[5]

1）私有变量和共享变量的区别

OpenMP 提供了不严格一致的共享内存模型，所有的线程在某个地方存储或检索变量，则这个地方被称为内存。

OpenMP 里访问的变量有共享和私有两种方式。并行区域内的共享变量是对并行区域外同名变量的引用；并行区域内的私有变量，即每个线程（主线程除外）会各自创建并行区域外同名原始变量的一个拷贝。并行区域内各个线程内实际操作的变量都是自己的私有副本，不会出现数据竞争现象。

如果访问共享变量，多个线程访问时存在写操作，则会发生数据竞争现象，其结果是不可预测的。如果存在同时写或者同时读写情况，必须进行锁保护或者使用原子操作等以避免出现数据竞争现象。

2）Flush 操作

OpenMP 的内存模型之所以被称为不严格一致的共享内存模型，是因为每个线程都有它自己的内存临时视图（CPU Cache），内存临时视图和实际的内存并不是在所有时间内都一致。

当将某个值写入一个变量时，实际上是在线程的内存临时视图自己中写入，变量在内存中的值并没有改变，除非强迫将其写入内存之中。

同理，当读入一个变量时，读出的是线程的内存临时视图中的值，并非内存中变量的值，除非强迫从内存中读取。

OpenMP 中提供 flush 操作来保证内存临时视图和内存的一致性。Flush 操作可以保证一个线程写入变量的值被另外一个线程读取；要完成这个操作，程序员必须保证自从最后一次对指定变量的 flush 操作之后，第二个线程没有对该变量进行写操作，即按照以下的操作顺序进行。

① 第一个线程写共享变量；

② flush 该共享变量；

③ 第二个线程 flush 该共享变量；

④ 第二个线程读取该共享变量。

3.3.2 编译器对 OpenMP 的支持

目前，支持 OpenMP 的 C/C++编译器主要有微软的 VC 和 Intel 的 C/C++编译器。

在微软的 VC 8.0 中，配置属性->C/C++->语言，将"OpenMP 支持"由"否"改成"是(/openmp)"即可。

在 VC 8.0 环境中，编译连接完 OpenMP 程序时，在有些 Windows 系统上执行时会遇到无法找到 vcomp.dll 的问题，碰到这个问题时，需要对 VC 进行一些设置：配置属性->链接器->清单文件，在"附加清单依赖项"中需要增加类似如下所示的内容：

"type='win32'name='Microsoft.VC80.OpenMP'version='8.0.50727.42'processorArchitecture='x86'publicKeyToken='1fc8b9a1e18e3b'"。

其中，version 是 VC 的版本号，比如 Windows 安装在 C:\Windows 目录下，那么安装完后可以在目录 C:\Windows\winsxs\x86_microsoft.vc80.openmp_1fc8b9a1e18e3b_8.0.50727.42_none_45e008191e507087 中发现 vcomp.dll 文件，目录名称中 VC 80.OpenMP 后接着的就是 publicKeyToken 字符串，再接着就是 version 号。在"附加清单依赖项"里输入的内容必须与目录名称中的内容一致[5]。

3.3.3 算法并行化基本技巧

(1) 动态设置并行循环的线程数量[5]

在实际情况中，程序可能运行在不同的机器环境里，有些机器是双核，有些机器是 4 核甚至更多核。并且未来硬件存在升级的可能，CPU 核数会变得越来越多。如何根据机器硬件的不同来自动设置合适的线程数量就显得很重要了，否则硬件升级后程序就得进行修改，那将是一件很麻烦的事情。

线程数量的设置除了要满足机器硬件升级的可扩展性外，还需要考虑程序的可扩展性，当程序运算量增加或减少后，设置的线程数量仍然能够满足要求。显然这也不能通过设置静态的线程数量来解决。

在具体计算需要使用多少线程时，主要需要考虑以下两点：

1）当循环次数比较少时，如果分成过多数量的线程来执行，可能会使得总运行时间高于较少线程或一个线程执行的情况。并且会增加能耗。

2）如果设置的线程数量远大于 CPU 核数的话，那么存在着大量的任务切换

和调度等开销，也会降低整体效率。

那么如何根据循环的次数和 CPU 核数来动态地设置线程的数量呢？下面以一个例子来说明动态设置线程数量的算法，假设一个需要动态设置线程数的需求为：

1） 以多个线程运行时的每个线程运行的循环次数不低于 4 次；

2） 总的运行线程数最大不超过 2 倍 CPU 核数。

下面代码便是一个实现上述需求的动态设置线程数量的例子：

```
    const int MIN_ITERATOR_NUM = 4;
    int ncore = omp_get_num_procs(); //获取执行核的数量
    int max_tn = n / MIN_ITERATOR_NUM;
    int tn = max_tn > 2*ncore ? 2*ncore : max_tn; //tn表示要设置的线程数量
#pragma omp parallel for if( tn > 1) num_threads(tn)
    for ( i = 0; i < n; i++ )
    {
        printf("Thread Id = %ld\n", omp_get_thread_num());
        //Do some work here
    }
```

当然具体设置多少线程要视情况而定的，一般情况下线程数量刚好等于 CPU 核数可以取得比较好的性能，因为线程数等于 CPU 核数时，每个核执行一个任务，没有任务切换开销。

(2) 任务调度[5]

OpenMP 中，任务调度主要用于并行的 for 循环。当循环中每次迭代的计算量不等时，如果简单地给各个线程分配相同次数的迭代，则会造成各个线程计算负载不均衡，使得有些线程先执行完、有些后执行完，造成某些 CPU 核空闲，影响程序性能。例如：

```
int i,j;
int a[100][100]={0};
for(i=0;i<100;i++)
{
    for(j=i;j<100;j++)
        a[i][j]=i*j;
}
```

如果将最外层循环并行执行，使用 4 个线程，给每个线程平均分配 25 次循环迭代计算，显然 $i=0$ 和 $i=99$ 的计算量相差 100 倍，那么各个线程间可能出现较大的负载不平衡情况。为了解决这个问题，OpenMP 提供了几种对 for 循环并行化的任务调度方案。

在 OpenMP 中，对 for 循环并行化的任务调度是使用 schedule 子句来实现的。

Schedule 子句的使用格式为：Schedule(type [, size])。

① type 参数：dynamic、guided、runtime、static。这四种调度类型实际上只有 dynamic、guided、static 三种，runtime 实际上是根据环境变量 OMP_SCHEDULE

来选择前三种类型中的某一种；

② size 参数（可选）：表示循环次数，size 参数必须是整数。dynamic、guided、static 三种调度方式都可以使用 size 参数，也可以不使用。Runtime 时，size 参数为非法的。

1）static 静态调度。当 parallel for 编译指导语句没有带 schedule 子句时，大部分系统默认采用静态调度方式。假设有 n 次循环迭代，t 个线程，则给每个线程分配大约 n/t 次迭代计算。如果指定了 size，则每个线程一次运行 size 次迭代计算；

2）danamic 动态调度。动态的将迭代分配到各个线程。当使用 size 参数时表示每次动态分配任务时分配给线程的迭代次数为指定的 size 次；不使用 size 参数时，动态的将迭代逐个地分配给各个线程；

3）guided 调度。采用指导性的启发式自调度方法进行调度。开始时每个线程会分配到较大的迭代块，之后分配到的迭代块会逐渐递减。迭代块的大小会按指数级下降到指定的 size 大小；如果没有指定 size 参数，那么迭代块最小会降到 1。

(3) 终止检测算法

然而，很多时候，我们不需要运行完整个 for 循环。看以下两个例子：

1）使用 for 循环，寻找一个指定的数值，当找到后，即可跳出循环。一个限制是，在 parallel for 中不能使用诸如 break、return 语句跳出循环，此时的 parallel for 是一种方法，不是一个循环回路；

2）多线程按数据并行运行时，如何在其中一个线程完成指定任务之后，通过某种机制通知其他线程终止运行？

因此我们需要研究并行环境下的终止检测算法。

一个可行的通用方法是，设定一个全局变量标记是否结束运行。通过在并行区域中对全局变量进行访问和判断，执行一定的动作，如下简单例子：

```
static int break_flag=1;    //全局变量
int Parallel_Search(int *Data,int fdata) //并行查找数据fdata
{
#pragma omp parallel for num_threads(nCore)
     for(k=0;k<nCore;k++)      //并行区域中不能使用break,return
     {
          ...
          SerialSearch(Data,begin,end,fdata); //查找操作
     }
}
```

```
void SerialSearch(int *Data, int begin, int end, int fdata)
{
    if(break_flag) //如果已需要退出，则不必做任何的操作，以此节省时间
    {
        for(i=begin;i<end;i++)
                if_find_data(fdata) //找到数据
                {
                        break_flag=0;
                        break;      //在此可使用break;
                }
    }
}
```

(4) 伪共享[5]问题

伪共享问题是由 CPU Cache 机制造成的。CPU 读取 Cache 时是以行为单位读取的，如果两个硬件线程的两块不同内存位于同一 Cache 行里，当两个硬件线程同时在对各自的内存进行写操作时，将会造成两个硬件线程写同一 Cache 行的问题；它会引起竞争，将使效率下降到原来的百分之一。

对于分配的内存，可以采取一定的内存分配算法使各块内存不在同一 Cache 行里，但对于数组或变量的访问，就必须由程序员在设计时考虑如何避免伪共享的问题。

在一个 Intel 处理器系统中，每个 Cache 行首地址都是 Cache 行大小的整数倍。如 Cache 行大小为 64B，当一块内存首地址为 0x0012ff52，因其除以 64 后余数为 0x12，因此这个地址不是 Cache 行的首地址。根据这个特点，可以利用以下算法得到给定地址的下一个 Cache 行首地址：

```
int *GetCacheAlignedAddr(int *pAddr)
{
    int m=CACHE_LINE_SIZE;
    int *pRet=(int *)((pAddr+m-1)&(-m));
    return pRet;
}
```

(5) 嵌套并行

当运行数据无法预先得知时，要编写一个自适应于硬件环境的并行程序，需要自动根据当前环境，给线程动态分配任务。

1）一个可行的方法是，设定一个线程的最小负载规模，当当前处理器核数超过一定的数量时，可执行嵌套并行思路。以下例子设定线程最小负载规模为 2，当运行数据 $n>2$ 时，继续嵌套。

例 5：

```
void d01(int a,int n)
{
    int i;
    if(n>2)
    {
#pragma omp parallel for
        for(i=a;i<n;i++)
        {
            d01(i,n-2);
        }
    }
    else
        Serial_do_something(a,n);
}
void main()
{
    omp_set_nested(1); //开启嵌套
    omp_set_dynamic(0); //OMP_DYNAMIC无效
    d01(0,6);
}
```

2）由 3.1 节中提到的，在使用特殊的数据结构对数据集进行动态存储的情况下，如何实现良好的并行运算。以下代码可用于解决链表结构问题：

```
    data_struct *headlist, *phead,*pcur;
    phead=headlist;
    while(phead!=NULL)
    {
#pragma omp parallel for num_threads(nCore) private(pcur)
        for(i=0;i<num_threads;i++)        //并行执行设置的线程数
        {
#pragma omp critical          //临界区，防止读写冲突
            {
                flush(phead);    //读phead之前
                pcur=phead->next;
                phead=pcur;
                flush(phead);     //写phead之后
            }
            do_for_datanode(pcur);  //每个线程同时单独处理一个数据结点
        }
    }
```

3.4 并行程序优化

3.4.1 Intel 软件介绍

大多数程序设计工作都是基于已有的应用程序来展开，如要开发一个全新的程序，通常首先基于一个框架原型或者应用程序的关键部分来进行。但是，不管是基于框架原型还是基于已有应用程序来进行开发，开发人员都要进行一些前期的调查工作以指导后续的开发，这些调查工作对整个开发工作具有非常重要的意义。

这里简要介绍 Intel 的三个软件，它们使得开发人员在观察和探测系统中所有线程的行为的工作变得简单。

1）Intel(R) VTune(TM) Performance Analyzer[1]。又称 Intel VTune 性能分析器，它是一种系统分析器，不仅可以提供进程/任务级的事件取样和调用图等所有可用信息，而且可以提供进程内线程级别的信息。

Intel VTune 性能分析器是一种测量工具，能够辅助用户更好地解析测量结果，甚至能够解读所出现的异常值的意义和解决方法。开发人员可以根据应用程序的运行特点来决定实现多线程的方案。开发人员应该首先对应用程序中的热点——主要的性能瓶颈进行分析，从而决定是否采用多线程来实现这些热点代码。通过 Intel VTune 性能分析器的事件取样功能可以发现热点。如果热点位于不能并行的地方，那么开放人员可以沿着调用序列重新定位热点，这样可能会找到更好的解决方法。该调用序列可以采用 Intel VTune 性能分析器的调用图来回溯。

Intel VTune 性能分析器能够发现程序中的耗时部分，例如模块、函数、线程，甚至源码中的某一行。

Intel VTune 性能分析器能够画出应用程序的调用图。通过观察调用图，开发人员就能够发现调用树中除了函数中的热点以外的其他可改进代码。单独创建一个线程来实现这些代码可能会更好，而后对包含这些代码的更高层代码进行修改，采用多线程实现，从而将任务分配给几个线程并行处理，就能够提高应用程序的性能。

2）Intel(R) Thread Profiler[1]。又称 Intel 线程档案器，它是 Intel VTune 性能分析器的一个部分，与 Intel VTune 性能分析器中其他组成部分不同，Intel 线程档案器知道用于协调各线程行为的同步对象的相关信息。因为协调线程行为可能要求某个线程等待，所以，知道同步对象的信息这一特点，对于 Intel 线程档案器显示等待时间或者浪费时间的信息非常重要。Intel 线程档案器能够向开发人员提高关于同步对象和线程工作负载不平衡（导致程序中最长的执行路径与其他路径之间出现较大的延迟）方面的信息，据此，开发人员可以对程序进行一些性能优化。

Intel 线程档案器能够检测到 Win32、POSIX 和 OpenMP 多线程程序中存在的同步问题和导致延迟的过长阻塞时间问题。Intel 线程档案器还能够发现线程工作负载不平衡问题，这样，开发人员就能对程序进行修改，从而使得程序在并行区中的大部分时间都在做实际的工作（而非等待或者闲置），最终达到多线程应用程序性能最大化的目标。

Intel 线程档案器的时间轴视图显示了每个线程对整个程序的贡献信息，而不管其是否位于关键路径上。Intel 线程档案器具有聚焦关键路径的功能：关键路径视图（Critical Path view）能够显示程序中关键路径上的时间耗费情况，直方视图（Profile view）会给出关于关键路径上的时间耗费情况的一个高层次总结。

联合使用 Intel VTune 性能分析器和 Intel 线程档案器能够使开发人员更深入地了解应用程序和系统内的多线程。综合采用这些分析器有助于开发人员避免盲目地优化程序，同时这些分析器所提供的直接反馈信息也有助于避免一些错误。

3）Intel(R) Thread Checker[1]。又称 Intel 线程检测器，它能够检测多线程应用程序中存在的关于线程互操作的编码错误，这些错误可能导致程序执行失败。Intel 线程检测器能够发现那些看似功能正确的程序中所隐藏的问题。要发现多线程编程错误有时非常困难，也是非常令人沮丧的事情，因为在程序每次运行的过程中，这些错误都是不确定出现的，有时会导致程序失败，有时则不会，当采用调试工具对这些错误进行检查的时候，这些错误的行为经常发生变化。

开发人员可以采用 Intel 线程检测器来定位多线程程序中存在的一类特殊线程编码错误，这些错误可能导致程序执行失败，也可能不会。Intel 线程检测器会对程序中在多线程运行环境下可能出现不确定行为的地方给出一些诊断信息，并将会发生问题的函数、上下文、发生位置（源码行）、变量以及调用栈等信息报告出来。它能够识别的问题包括数据竞争、死锁、停止线程、丢失信号以及废弃锁。

3.4.2 程序优化具体实例

实现这样的一个功能函数：函数中有三个 for 循环，每个 for 循环迭代次数不一样，分别为 4000、3500、3000。三个循环并行运行时，存在对数组 num_of_degree[] 的写冲突可能。以下使用 2 种不同的实现方式，利用 Intel 软件，对并行程序进行分析和优化。程序均运行在双核机器上。以下为实现步骤：

(1) 在对程序进行优化之前，应先确保程序的正确性，使用 Intel 线程检测器对其进行冲突检测。

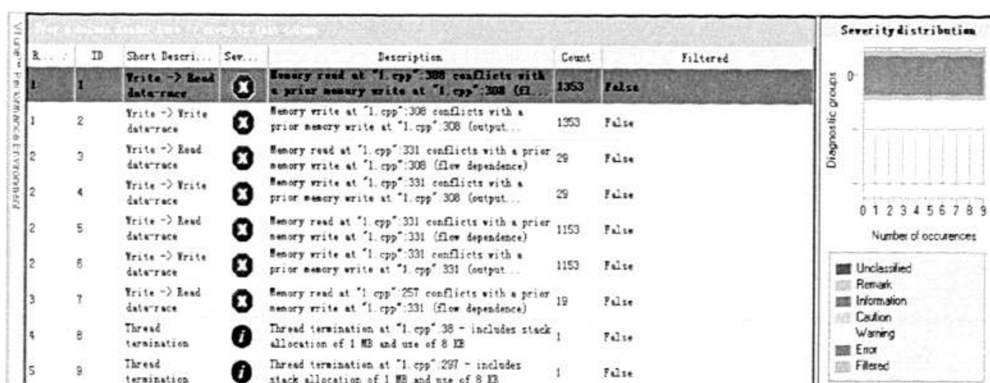


图 3.3 Intel 线程检测器的诊断视图

可看到，程序中存在读写冲突，双击图中显示，即察看对应的源代码行，发现问题发生在对 num_of_degree[]++的写操作上，引起此问题的原因是伪共享，在写操作前加语句：#pragma omp atomic 即可。

(2) 使用 Intel VTune 性能分析器分析程序中最耗时间的子函数：

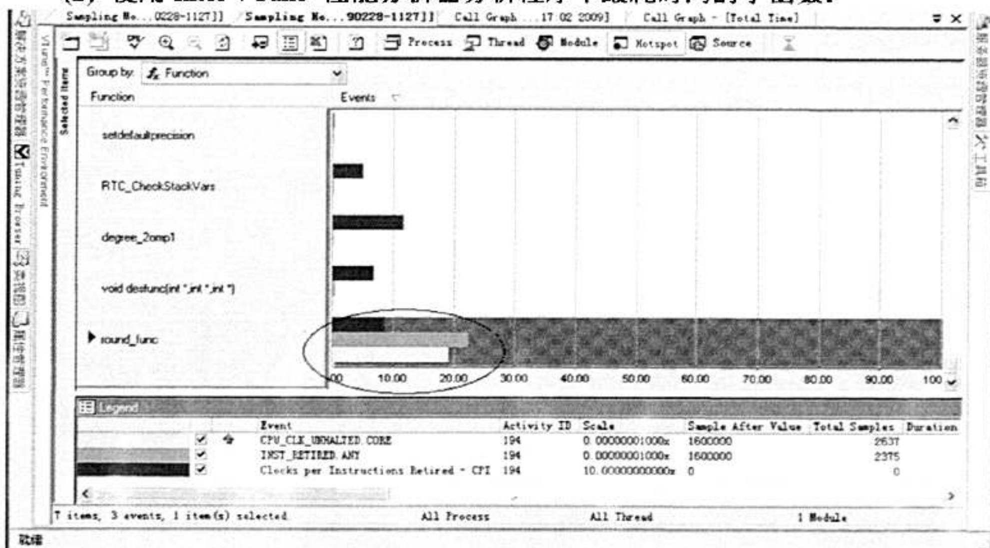


图 3.4 Intel VTune 性能分析器的取样结果

从图 3.4 可看出，子函数 round_func 被调用的次数最多，应从此函数着手，对程序进行优化。经分析发现，每一次子函数 round_fun 的调用都是独立的个体，如果对子函数内部进行并行优化，必将导致多次重复地创建销毁线程，引起不必要的开销，应从调用此子函数的函数 degree(3)着手。

(3) 对函数 degree(3)进行优化：

首先使用的实现方法是：使用 sections，即任务分解方式实现功能并行性。具体实现代码为：

```
void degree( int n )
{
#pragma omp parallel sections
    {
        #pragma omp section
        for(i=0;i<4000;i++) //线程2运行
        {
            (round_func, 1);
        }
        #pragma omp section
        for(i=0;i<3500;i++) //线程1运行
        {
            (round_func, 2);
        }
        #pragma omp section
        for(i=0;i<3000;i++)  //线程1运行
        {
            (round_func, 3);
        }
    }
}
```

使用 Intel 线程档案器的时间轴视图可看到以下线程负载情况：

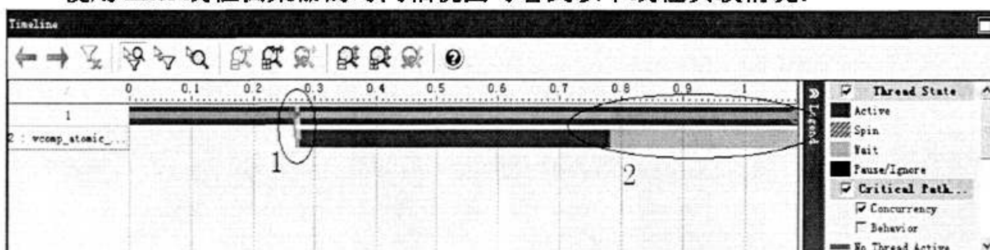


图 3.5 Intel 线程档案器的时间轴视图

标记“1”处表示创建线程消耗的时间；标记“2”处表示两个线程负载不均衡，一个线程工作结束后停止，而另一个线程仍需继续运行一段时间。分析以上程序思路，线程 1 运行完循环 for(i=0;i<3500;i++)之后，继续运行 for(i=0;i<3000;i++)，而此时的线程 2 只剩下 500 次循环。标记“2”处就是标记着线程 1 比线程 2 多运行的时间。

根据此分析结果，将原线程负载的分配方式，即单独使用 sections 更改成另一种实现方式：将第三个 section 换成 for 循环，目的在于将第三个 section 的任务分配给所有的线程一起完成，相关实现代码为：

```
#pragma omp parallel
{
#pragma omp sections nowait
    {
    #pragma omp section
        for(i=0;i<4000;i++) //线程1
        {...}
    #pragma omp section
        for(i=0;i<3500;i++) //线程2
        {...}
    }
    #pragma omp for schedule(static,8)
        for(i=0;i<3000;i++) //线程1、2
        {...}
}
```

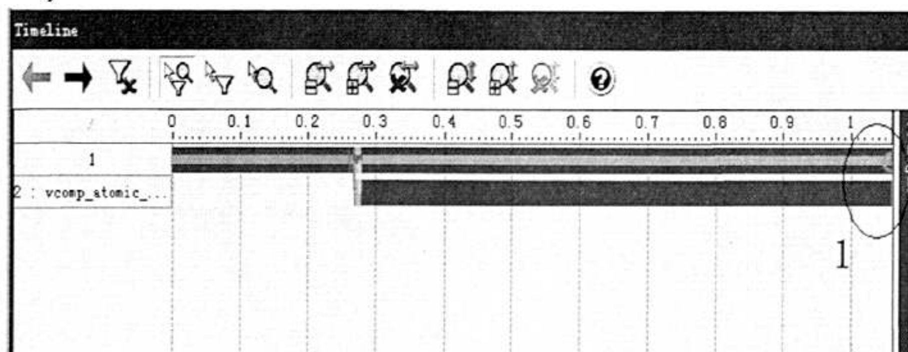


图 3.6 改进后的 Intel 线程档案器的时间轴视图

由图 3.6 中标记"1"处可知，当线程 1 和线程 2 在执行完各自的 section 的工作后，接着继续运行第三个 for 循环。程序中使用 schedule(static，8)进行任务调度，一次给一个线程静态分配 8 次循环的任务，通过这种细粒度的分配，达到两个线程负载均衡的效果。由以上分析可知，sections 有个天然的屏障，因此需要加入 nowait 语句，线程 2 运行完 section 的任务后，由于 nowait 的关系，不能空闲，接着运行 for 循环语句，而此时线程 1 还在运行 section 的任务。由图 3.6 可以看出，线程 1 和 2 一直都是活跃着的，直到任务结束，线程终止，最终达到充分利用线程资源的效果。

3.5 本章小结

本章总结了安全算法的一般规律，并针对这些规律提出一般可用的并行设计方案。重点介绍 OpenMP 的实现技巧。提出在并行实现过程中遇到的问题与具体解决方法。最后利用 Intel VTune 性能分析器、Intel 线程档案器和 Intel 线程检测器对程序进行分析和优化。

第四章 并行伪随机数生成器检测研究

随机数生成器的发展已经经过了很长的过程，而随着计算机硬件的发展，对随机数生成器的要求也随着改变，并行实现随机数生成器的设计和检测也面临着新的问题和挑战。

4.1 伪随机数的基本概况

随机数生成器被广泛应用于多种应用领域，从投币机到核反应堆的模拟，从计算科学到工程的仿真。对于计算科学应用而言，诸如蒙特卡洛模拟，要求随机数生成器拥有很好的随机性。尤其是对于运行在高性能计算机上的大规模仿真试验，随机数生成器的随机性质更显重要。然而，良好的随机数生成器是很难得到的，现有的被广泛运用的多种技术仍不成熟。而在并行机上寻找高质量高效率的随机数生成器更是一个难题。

由于在现实应用中，随机数通常由确定性算法得到，因此随机数生成器通常更为严格的称为伪随机数生成器。

随机数生成器使用迭代的确定性算法生成具有良好的随机性能的伪随机序列。理论上这些伪随机生成序列应具有以下特点[10][11]：

1）统一分布；

2）不相关性；

3）可重复生成；

4）使用简单；

5）同一算法输出序列应由初始向量决定；

6）可以分成相互独立的子序列；

7）大的周期；

8）通过所有的随机性经验检验；

9）在有限的计算机存储上能快速产生等特点。

并行伪随机数生成器则多加三条要求：

1）由不同的计算机核产生的随机序列间具有不相关性；

2）不同的核数能生成相同的序列；

3）计算机核之间不需要任何的数据通信，每个处理器都能独立的完成任务。

事实上不可能存在一个随机数生成器满足以上所有的性质，因为计算机使用有限的精确算法去存储生成器的状态，这必然导致在一定的周期内，存储出现重复性。而且，我们需要序列可重复生成，即不是真正的随机，输出序列只是一个确定性的遍历过程，因此不能实现完全不相关。

人们通常把太多的精力放在随机数生成器的实现速度上，然而，一个快速的生成器要求最少数量的简单操作，这将直接影响到生成器的随机特性。因此，在多数的应用中，人们更愿意以速度为代价换取生成器更好的随机特性。

人们更自然的想到，如果每次生成器的输出是一组数值而不是单个数值，这将大大的提升了生成器的速度。很明显，这需要在并行机上实现。因此出现了对并行伪随机数生成器的研究。

从这个角度看，对并行伪随机数生成器的研究似乎只是为了追求速度的提升，而生成器的安全性则完全取决于单个生成器的实现方式上。

4.2 并行伪随机数生成器的基本概况

4.2.1 各种生成器介绍

(1) 随机数生成器的主要算法有[10] [11]：

1）线性同余生成器（Linear Congruential Generators）。标准 C 语言和 Unix 中函数 RAND（32-bit）、RANF（48-bit）都使用这种算法。算法根据以下公式得到随机数 X_i：

$$X_i = (a * X_{i-1} + c) \bmod M \qquad 式(4\text{-}1)$$

其中，a 为非零系数，M 为素数，c 为常数。每个参数的选取都影响着输出序列的周期大小，直接影响到其随机性。

48-bit 的线性同余生成器在频谱测试中表现出了良好的性能，64-bit 的随机性更好，不建议使用 32-bit。

2）延迟斐波纳契数列发生器（Lagged Fibonacci Generators）。这种算法越来越流行，因为它能提供一个很简单的方法很快的生成大的周期序列。标准 C 语言和 Unix 中函数 RANDOM 就属于这种算法。算法定义：

$$X_i = X_{i-p} \odot X_{i-q} \bmod 2^m \qquad 式(4\text{-}2)$$

p,q 分别为延迟系数，$p > q$，$\odot$ 为任意的二进制运算，如模 M 的加减运算、逐位异或等。对于延迟系数的要求是，在二进制运算为乘法的情况下，p 至少为 127，加法时至少为 1279。一个延迟系数为 17 的乘法延迟斐波纳契数列发生器就可以通过现有的很多经验性测试。

延迟斐波纳契数列发生器需要一个存储延迟序列的额外空间，存储当前序列的前 p 个序列。

3）移位寄存器（Shift Register Generators）。移位寄存器可以看作是延迟斐波纳契数列发生器的特殊形式，即二进制运算为异或。异或形式的延迟斐波纳契数列发生器拥有最差的随机特性，所以我们不提倡使用移位寄存器来生成随机序列。

4）生成器组合。将两种不同的生成器组合使用。如将一个延迟斐波纳契数列

发生器和线性同余生成器组合，或者将两个线性同余生成器组合使用。

例如：设有两个线性同余生成器生成两个序列：$\{y_i\} \bmod m_y$和$\{z_i\} \bmod m_z$，设$m_y > m_z$，则组合生成器可以使用加法或减法的形式，由于减法可以相对容易地避免数据溢出，以下使用减法法则，设组合生成的序列为$\{x_i\}$，则组合生成器可以表示为：$x_i = (y_i - z_i) \bmod m_y$。若$m_y$为$2^{31}-1$，$m_z$为$2^{31}-19$，两个生成器周期大概分别为$2.15 \cdot 10^9$，则组合生成器的周期为$4.61 \cdot 10^{18}$。

(2) 现今主要的将随机数生成器并行化的技术有[17] [18]：

1）交互跃进（Leapfrog）。生成序列被各个处理器按一定地顺序分割，犹如牌局上向所有的选手发牌。假设一共有N个处理器，则处理器P生成的序列为：$X_P, X_{P+N}, X_{P+2N}, \cdots$；

2）序列分割（Sequence splitting）。序列被分割成相邻互不重叠的子序列。将一个周期的序列平分成N个子序列，设子序列长度为L，则处理器P生成的序列为：$X_{PL}, X_{PL+l}, \cdots, X_{PL+L-1}$；

3）独立序列群（Independent sequences）。各个单独的处理器随机选择初始种子，生成独立的长周期子序列。与序列分割类似，不同的是各个子序列长度不等，这样可以避免长程相关（long-range correlations），只要各个处理器中的种子选取是随机并相互独立的；

4）细胞自动机生成器（The cellular automata generator）。它是基于细胞自动机规则的移位寄存器的一般形式。一种称为 CMF_RANDOM 的生成器并行版本可以通过很多标准的统计测试，然而却不能通过蒙特卡洛伊新模型检测，因此不推荐使用这种生成器。

(3) 推荐使用的并行随机数生成器的生成方法有[17] [18]：

1）多个线性同余生成器使用序列分割方式组合，同样的参数，不同的初始值；

2）多个乘法延迟斐波纳契数列发生器使用独立序列群方式组合，初始种子的选取是随机并且互不相关的；

3）多个加法延迟斐波纳契数列发生器使用独立序列群方式组合，初始种子的选取是随机的并且互不相关，生成器的延迟系数必须足够大。

例子：有l个生成器，$j=1,\cdots,l$，每个生成器周期为p_j，表达式为：

$$s_{j,i} = f_j(s_{j,i-1}) \quad \text{式(4-3)}$$

组合序列为$\{s_i = (s_{1,i}, \cdots, s_{l,i}), i = 0,1,2,\cdots\}$，其中$s_0 = (s_{1,0}, \cdots, s_{l,0})$是初始种子。即组合生成器的表达式为：

$$s_i = f(s_{i-1}) \quad \text{式(4-4)}$$

对于组合线性同余生成器而已，表达式为：

$$s_i = a s_{i-1} \bmod m \quad \text{式(4-5)}$$

4.2.2 生成器的优缺点

要找到一个性能很好的并行随机数生成器是相当困难的。原因之一是，如果单个生成器存在很小的相关性，组合成并行随机数生成器后，各个生成器间的相关性会被放大。

初始化并行随机数生成器的方法也很重要，即选取每个生成器的初始种子。它的重要性不亚于单个生成器算法的选择，因为初始种子间任何小的相关性都会导致生成器间很大的相关性[12] [13] [14] [16]。

(1) 交互跃进（Leapfrog）

因为组合成并行随机数生成器的所有单个生成器，其参数一样，只是初始种子的选取的不同。如果使用的并行随机数生成器的生成方法是多个线性同余生成器使用交互跃进方式组合，设单个生成器的表达式如式(4-1)所示。交互跃进方式中，有 N 个处理器，则处理器 P 生成的序列为：$X_P, X_{P+N}, X_{P+2N}, \cdots$，组合成的并行随机数生成器单个生成器的表达式为：

$$X_{p+(i+1)N} = a^N \cdot X_{p+iN} + c \cdot (a^N - 1)/(a-1) \bmod M \qquad \text{式(4-6)}$$

存在的问题是：在选取参数 a 的时候，判断标准是，由式(4-1)生成的序列可以通过频谱测试，但是并不能保证由式(4-6)生在的序列也可以通过频谱测试，尤其是当 N 为任意数时。即参数 a 和 a^N 的区别。这种方法只适用于 N 为确定数时，即运行在一定数量的处理器上，不推荐作为一般生成器使用。

存在的另一个问题是：众所周知，在线性同余生成器中，当 M 为 2 的幂次时，生成器生成的序列中相隔为 M 的元素间存在相关性。而现有的多处理器体系中处理器个数通常为 2 的幂次，这将导致序列 $X_P, X_{P+N}, X_{P+2N}, \cdots$ 有更强的相关性。除非 M 为素数，方可避免这个问题。

即当选取的并行随机数生成器的生成方法是多个线性同余生成器使用交互跃进方式组合时，线性同余生成器至少为 48-bit，且 M 为素数。

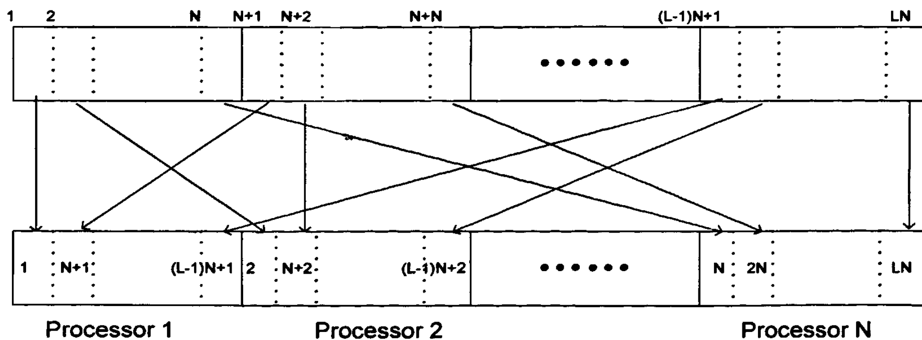


图 4.1 交互跃进方式

(2) 序列分割（Sequence splitting）

设组合并行随机数生成器中每个处理器生成的子序列长度为L，则处理器P生成的序列为：$X_{PL},X_{PL+l},\cdots,X_{PL+L-1}$。一个处理器对应一个生成器，即一个初始种子。

此方法存在的问题是：在并行随机数生成器生成的序列中，虽然每个处理器生成的子序列互不重叠，但并不能保证子序列间没有相关性。

而序列分割所带来的缺点是：在不同的处理器数下生成的序列不相同。当组合生成的序列长度为T时，在处理器个数为N的情况下，单个处理器生成的子序列长度为T/N；处理器个数为N'时，单个处理器生成的子序列长度为T/N'。

所幸的是，这种序列分割方式可以用第二章提到的数据并行性的并行设计方法来实现。即可以先确定子序列长度，即确定使用的不同的生成器个数，再根据不同的处理器数硬件环境调用不同数量的线程去实现。此时，生成器数对应的不再是处理器数。

此方法符合现有的数据并行编程模型，满足高性能程序语言的设计规则，易于实现，需要的存储空间少，可以提供一个很好的并行随机数生成器。

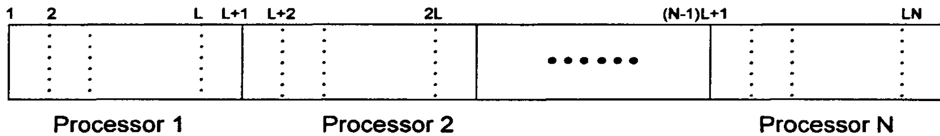


图 4.2　序列分割方式

(3) 独立序列群（Independent sequences）

事实上，当一个仿真需要很多个不同的随机序列时，这种方法得到的序列效果跟序列分割方法一样。

在序列分割中，每个处理器生成的子序列起始点均为固定的预先设定好的，如$X_{PL},X_{(P+1)L},\cdots,X_{(P+N-1)L}$。而在独立序列群这种方式中，子序列起始点为随机选取的而不是以一定的规律递增。这种选择方式带来的好处是可以避免序列间长程相关或者降低其可能性，当然前提是初始种子的选取必须是随机的和相互独立的。而以上提到的交互跃进和序列分割两种方法不适合用于延迟斐波纳契数列发生器，因为这两种方法只适用于子序列生成方法简便的发生器。

对于每个处理器的初始种子的选取在独立序列群这种方式中是至关重要的。任何初始种子间的相关性将引起子序列间严重的相关性。

一个潜在的缺点是：由于各个生成器的初始种子是随机选取的，这就不能确保生成器间生成的子序列没有重复性。当然，在使用延迟斐波纳契数列发生器时，选取大的延迟系数可以避免这种可能性。

参数相同的多个线性同余生成器，其生成的是同一个周期序列，只是选取的初始种子不同时，序列的起始点不同。如其中一个线性同余生成器生成的序列为010011000111010011，另一个生成器生成的序列为000111010011000111。

而对于延迟斐波纳契数列发生器而言，不同的初始种子所生成的周期序列是不一样的，为不相交的周期序列，可称之为等价类。如式(4-2)所示的生成器的最大周期为$(2^p-1)2^{m-1}$，即存在$(2^p-1)2^{m-1}$个初始种子能得到最大周期的周期序列。即不同的生成器要生成相同的周期序列其概率是非常小的，几乎可以忽略。

当延迟斐波纳契数列发生器的延迟系数非常大时，其初始化所花费的时间也随着增多，对于一般的应用而已，这个花费的时间是难以接受的。因此需要更好的算法提高初始化的速度。现有的实现方式要求延迟系数的范围为 17~127。因此在对初始速度有严格要求的应用中，不推荐使用加法延迟斐波纳契数列发生器。

独立序列群这种方法存在的问题跟序列分割是一样的，不同的处理器数环境下生成的序列是不一样的。也可以使用与序列分割类似的思路去解决这个问题，此时一个生成器不一定要对应一个处理器，可以对应一个线程。只是，每个生成器都需要开辟一个自己的存储延迟序列的空间，当延迟系数大到足以避免子序列间的重复性时，此时每个生成器需要的存储空间也变的相当大。

4.3 并行伪随机数生成器实现

综上所述，对随机数生成器的研究分两个方向：

1）生成器生成的随机序列的随机性能；

2）生成随机序列的实现效率和性能，如速度、占用的内存等。

应确保生成器生成的随机序列满足一定的随机特性的情况下，提高实现效率和性能。

本节针对 4.2.1 提出的三种并行伪随机数生成器生成方法，从并行实现方法的角度对其进行分析和实现。由于加法延迟斐波纳契数列发生器使用独立序列群方式组合的实现方式与乘法延迟斐波纳契数列发生器使用独立序列群方式组合的实现方式类似，以下只分析前两种。

4.3.1 线性同余生成器使用序列分割方式组合

根据式(4-1)：$X_i=(a*X_{i-1}+c)\bmod M$，当多个线性同余生成器的参数相同初始值不同时，它们的生成序列为同一周期序列，只是序列的起始点不同。因此不能使用第三种组合方式，即独立序列群方式，这样会导致各生成器间序列覆盖。

从并行实现方式上看，交互跃进方式和序列分割方式是一样的，都需要预先求出N个处理器的初始值分别为$X_1,X_2,\cdots,X_N$，而后各个处理器再根据各自相对

应的初始值使用相同的公式生成序列。

交互跃进方式中各个生成器使用上述(4-6)公式，而求初始值使用的公式为式(4-1)。

序列分割方式中，根据 4.2.2 中提到的数据并行性的并行设计方法，先确定使用的组合成并行伪随机数生成器中的生成器个数，设为N，处理器数为P，生成的序列周期为LN，则一个生成器生成的子序列长度为L，一般而言，处理器数P为N的整数倍，设$m=P/N$，则一个处理器需要生成的序列长度为$k=L/m$。

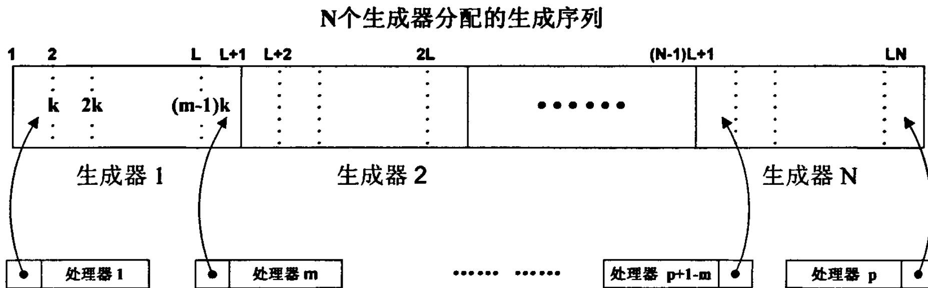


图 4.3 序列分割方式自适应性算法

由图 4.3 可知，对于每个生成器，分配m个处理器，设N个生成器的初始值分别为$X_{11},X_{21},\cdots,X_{N1}$，而各个处理器的初始值可根据对应的生成器的初始值使用以下公式求得：

$$X_{p+j}=a^j\cdot X_p+c\cdot(a^j-1)/(a-1)\bmod M \qquad \text{式(4-7)}$$

X_p表示对应的生成器的初始值。对应于生成器 1，m个处理器的初始值分别为$X_{11},X_{1k},\cdots,X_{1(m-1)k}$。

因此，实现此并行伪随机数生成器的基本步骤为：

(1) 选定生成器个数N；

(2) 选定各个生成器对应的初始种子$X_{11},X_{21},\cdots,X_{N1}$；

(3) 计算当前的处理器数P，由此求出每个生成器对应的$m=P/N$个处理器；

(4) 根据已知的初始种子，带入公式(4-7)，求出各个处理器对应的初始种子；

(5) 所有的处理器利用公式(4-1)并行运行；

(6) 各个处理器生成的子序列按照相对应的规则存入此并行伪随机数生成器的生成序列中。

在此必须要注意的问题是，不是使用越多的处理器，就能使得此并行伪随机数生成器的实现效率越高。由第三章分析可知，运行线程需要一定的额外开销，如创建、调用、销毁等。当分配给每个线程的任务细化到一定的程度后，再细化将会造成线程间频繁的调用，引起过多的开销，使生成器的性能变差。在此加大生成器个数N是个不错的选择，因此在实现的程序中，N应该为不定数。

4.3.2 乘法延迟斐波纳契数列发生器使用独立序列群方式组合

设乘法延迟斐波纳契数列发生器（LFG）的生成公式为：

$$X_i = X_{i-p} \bullet X_{i-q} \bmod 2^m, \quad p > q \qquad 式(4-8)$$

则该生成器需要的存储延迟序列的额外空间大小为 p 个单位空间。

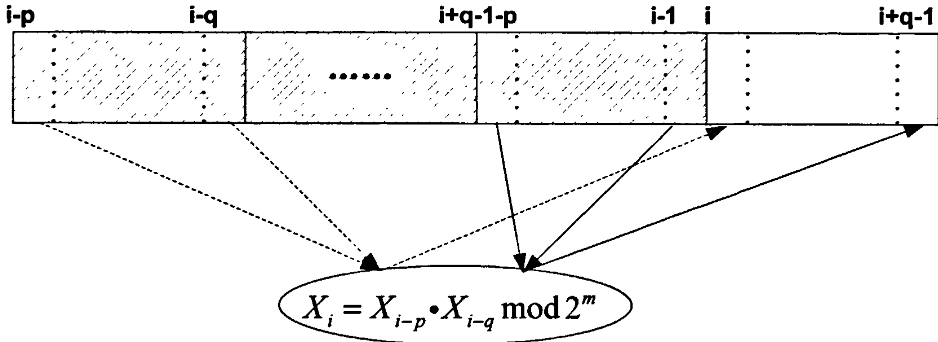


图 4.4 LFG 产生第 i 个数值时的存储空间状态

$i-p \sim i-1$ 为存储空间存储的数据，由此数据即可根据公式(4-8)得到数值 $i \sim i+q-1$，$i > p$，这 q 个数值的产生是没有相关性的，即相互独立的，可以并行实现，下一轮实现时的存储空间状态为：

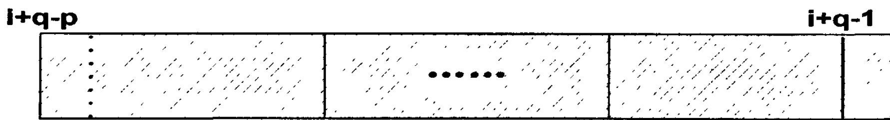


图 4.5 下一轮实现时的存储空间状态

使用的延迟斐波纳契数列发生器，当其生成器使用的公式系数一样，初始种子序列不一样时，生成的是不同的周期序列。因此在这个并行伪随机数生成器中使用的算法与 4.3.1 中提到的不同，在此，应使用尽可能多的延迟斐波纳契数列发生器组合成并行生成器。即此时生成器的个数 N 大于处理器数 P，处理器多次运行对应的生成器。只有这样，才能保证，在不同的处理器核数的环境下，可以生成相同的并行伪随机序列。此时的生成器 N 对应全部运行的线程数。

因此，实现此并行伪随机数生成器的基本步骤为：

(1) 选定生成器个数 N；

(2) 选定各个生成器对应的初始种子 $\{X_{11}\},\{X_{21}\},\cdots,\{X_{N1}\}$；

(3) 计算出现有的处理器数 P，$n = N / P$，设 $i \leftarrow 1$；

(4) 并行运行各个处理器，调用的线程将相对应的生成器的初始种子带入公式(4-8)，求出子序列；

(5) 各个处理器生成的子序列按照相对应的规则存入此并行伪随机数生成器的生成序列中；

(6) $i \leftarrow i+1$，当$i<n$时，执行 4，否则结束。

当然，此实现方法也可应用于产生线性同余生成器使用序列分割方式组合这种并行伪随机数生成器。

4.4 检测方法

4.4.1 随机性检测技术

根据 4.2 节提到的随机数生成器的输出序列应满足的多种特性，现今已提出了多种标准测试。随机性检测可以分成两类：一、基于物理模型的检测技术。如应用于蒙特卡洛仿真(Monto Carlo simulation)的二维伊辛模型(Ising model)，在应用模型中使用待测随机序列，将结果与已知应用结果比较；二、统计性测试。如频率测试、LZ 压缩测试、随机散漫测试、线性复杂度测试等等。统计性测试通常将测试结果与一标准值进行比较，由假设检验 H_0 判断生成器的随机性能，其中标准值由真正随机序列，或者已知的随机性概率分布得到。

所有的生成器在使用之前都应该先进行所有性质的统计性测试，至少要通过两个测试才能使用。一般而言，当一个生成器通过了频谱测试的检测后，很容易通过其他的测试，如标准正态分布的相关性测试、游程测试等等。

然而这些检测方法均有其局限性。如果一个生成器没有通过一个检测方法的测试，并不能说明这个生成器在某些特殊应用中性能很差，并不能说明所有使用了这个生成器的研究结果都不可靠。

一些检测技术的检测规模是由现有的计算能力决定的。比如新统计测试，在串行环境下，只建议检测到度数为 3 的统计量。当度数为 2 时，程序的运行时间为因为 1234ms，而度数为 3 时，为 98235ms，接近 100 倍。随着度数的增加，所需要的运行时间成超幂指数倍递增。而随着现今计算机计算能力的增强，以及并行计算的提出，新统计测试可以检测到度数更高的统计量，从而使随机性检测结果更加精确。

因此随着并行计算的提出，现有的一些随机性检测技术均可以扩大自己的检测规模，因而要研究这些检测技术的并行实现方式，才能适应时代的需求。

现有常用的统计性测试有 26 种，而对基于物理模型的检测技术的研究尚需加强，常用的有二维伊辛模型、渗透模型（percolation models）和随机游走（random walks）。统计性测试的优点在于运行速度比基于物理模型的检测技术快；然而后者的检测是基于实际应用环境之中，其检测结果更接近实际应用，更加可靠，而且它可以有效的检测序列间的相关性。一些可以通过统计性测试的随机数生成器在一些特殊应用中却并不一定可靠。由此，迫切需要对基于物理模型的检测技术进行更多的研究。

4.4.2 并行随机性检测技术

一个具有良好随机性质的并行伪随机数生成器应该同时也是一个良好的串行伪随机数生成器，即所有这些对随机数生成器的检测方法均可以适用于并行随机数生成器的检测，用以检测各个处理器生成的序列或者所有处理器的组合序列的随机性。不同的是，对并行伪随机数生成器的检测除了要进行以上方面的串行检测之外，还需要检测生成器组合后产生的新的问题。比如除了要检测各个生成器生成的子序列的相关性之外，还要检测子序列间的相关性。各个子序列的任何小小的相关性，在用 4.2.1 中提到的独立序列群等组合方法进行组合生成并行伪随机序列后，序列的相关性将被不可预知的放大，严重的影响了并行伪随机数生成器的随机性能。而在一般的应用中，通常需要检测序列间的高维数相关性，甚至是几千维以上，这样的一个计算量将是相当庞大的。而且子序列间的相关性的误差限难以从数学的角度去证明，因此迫切需要对此进行大量的经验性检测。这就是检测并行伪随机序列的难点所在[15][18][19][20][23]。

以下分别分析现有的两类检测方法：

(一) 统计性检测

设有 N 个串行生成器组合成并行随机性生成器，独立生成 N 个子序列。目前的并行伪随机序列的检测方法有：

(1) 交错测试（Interleaved tests）。

1）将 N 个子序列分成 M 组，将每组序列进行交错，得到 M 个交错序列；

2）用统计检测方法分别对 M 个交错序列进行统计，得到一组相对应的 M 个卡方值；

3）对 M 个卡方值用 Kolmogorov-Smirnov（KS）测试方法进行检测，看其是否服从卡方分布。

例如：$N=4$，序列长为 4，分成 2 组：（1，2）、（3，4）。将组合成的新序列分成 2 组：

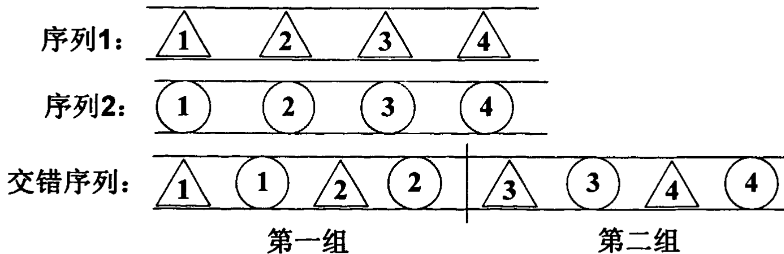


图 4.6 将两组序列进行交错组成新序列，而后进行串行检测

其中，以上 1）和 2）可以并行执行。2）中提到的统计检测方法可以使用：碰撞测试、集券测试、间隔测试、最大值测试、排列测试、扑克测试、游程测

试和连续性测试等，具体参考参考文献Knuth[21]。

交错测试的思路在于将现有的串行检测技术重复并行执行，可以有效的检测并行随机数生成器中子序列间的相关性。

还存在另外一些统计性检测技术，它们的实现方式隐含着对序列的并行检测，此时，不需要将并行随机数生成器生成的各个子序列进行交错后测试，如：

(2) 傅立叶变换测试（Fourier transform test）。将 N 个子序列输入一个二维矩阵，矩阵中每一行数据为一个子序列。计算这个二维矩阵的傅立叶变换，并将之与期望值进行对比。

(3) 封闭性测试（Blocking test）。已知几个相互独立的服从正态分布的变量之和也服从正态分布。将 N 个子序列逐位相加，最后生成和序列，对和序列进行统计测试。

(二) 基于物理模型的检测技术

(1) 二维伊辛模型[22][23][26]

二维伊辛模型是一种简单的点阵旋转模型，模型中相应的结果数据是已知的，比如能量和比热容。由此可以判断应用于此模型的随机数生成器的性能。由于二维伊辛模型有一个相移变换，因此可以判断随机数生成器的长程相关性。如果二维伊辛模型中使用的随机序列间存在相关性，也许可以得到正确的能量值，但是一定得不到正确的比热容。以此检验序列间的相关性。

常用的实现二维伊辛模型的算法有Metropolis算法和Wolff算法。每一种算法中的随机数的使用方式都不同。将并行随机数生成器生成的各个子序列放进模型中不同的网格子集中，可以有效的检测子序列间的相关性。对不同的网格进行运算操作是相互独立的，可以并行执行。

(2) 随机游走测试[22][23][26]

随机游走的实现方法比较简单，设有一个二维网格，随机游走始于网格中的某一点。将二维网格平分成四等分，其中下一步的走向由随机数生成器中的下一个数值确定。记下随机游走的终止位置。重复 N 次随机游走，对终止位置进行度数为3的卡方检验。其中 N 次游走可以并行执行。

由于基于物理模型的检测技术的规模比较大，使用分布式存储模式实现将更加高效，即应使用2.3.2中提到的MPI技术实现。

所有的统计性检测都不足以证明一个序列具有随机性，只能表明这个序列满足某一个随机特性，或者适用于某一应用。为了避免在某一应用中使用了性能不合格的随机数生成器，强烈建议在任何使用到随机数生成器的应用中，至少使用不同的两种甚至多种生成器运行于该应用中，并将其结果进行对比，以确保应用结果的准确性。相反，当一个应用模型的某些结果数值已知时，可用此应用模型检测应用于其中的随机数生成器的随机特性。

4.5 本章小结

本章先介绍了随机数生成器应有的随机特性，以及对并行随机数生成器提出的新的随机特性需求。

随后介绍现有的并行伪随机数生成器的结构与优缺点，并给出了线性同余生成器使用序列分割方式组合和乘法延迟斐波纳契数列发生器使用独立序列群方式组合这两种并行伪随机数生成器的并行实现算法步骤。

最后介绍了现有的检测并行伪随机数生成器的检测技术和并行实现思路。指出对随机数生成器和随机性检测技术进行研究的发展方向。

第五章 MD6 并行实现

密码算法中的哈希函数应用于消息认证和数字签字等多个领域。长期以来，人们使用哈希函数必须要权衡好其安全性与处理速度之间的关系。如何使哈希函数在保证其安全性的基础上提高自身的处理速度已经成为研究的一个方向。本章给出了 MD6 两种并行实现方式，并比较其运行效率。

5.1 MD6 原理

5.1.1 MD6 参数介绍

MD6 哈希函数为树形架构[37]，它的特点是输入不定长消息，得到固定长度的输出。一共有 5 个输入参数，分别为：

(1) M：输入的消息，设 m 表示 M 的比特长，则 $0 \le m < 2^{64}$，计算机中以字节为基本存储单位，因此 m 为 8 的整数倍；

(2) d：指定的输出比特长，$0 < d \le 512$；

(3) K（可选）：输入的密钥，设 $keylen$ 为 K 的字节长，则 $0 \le keylen \le 64$，用户可以指定 $keylen$ 长的密钥，当 $keylen$ 超过指定范围时，可以将此密钥先经过 MD6 压缩函数处理，即使得 $d = 512$，得到 $keylen = 64$ 的用户密钥；

(4) L（可选）：执行模式。MD6 有两种执行模式，$L = 0$ 表示如图 5.1 所示的串行模式，只需要很少的存储空间；$L = 64$ 为可并行化的执行模式，如图 5.2 所示。本文讨论 $L = 64$ 的并行执行模式；

(5) r（可选）：轮数，控制 MD6 中压缩函数的执行轮数。一般 r 的缺省值为：

$$r = 40 + \lfloor d/4 \rfloor \qquad 式（5-1）$$

当有输入参数 K 时，为了保护密钥的安全性，r 的取值为：

$$r = \max(80, 40 + \lfloor d/4 \rfloor) \qquad 式（5-2）$$

对于 MD6 而已，r 为输入参数，可由用户自己设定，根据当前应用对时间和安全性的需求而定。其中，轮数越大，函数的安全性越高，消耗的时间也越多。

5.1.2 MD6 实现模式

哈希函数中起关键作用的是其中的压缩函数[37]。MD6 的压缩函数为 89 字长的输入，16 字长的输出。本文讨论的字长均为 64 比特。

$$f: W^{89} \to W^{16} \qquad 式（5-3）$$

89 字长的输入组成为：15 字长的常量Q、8 字长的密钥K、1 字长的对应于每个压缩函数的唯一标识符ID、1 字长的控制向量V和 64 字长的输入数据块B，如图 5.3 所示；

MD6 压缩函数的执行模式。图 5.1 和图 5.2 中的斜杠圈表示 16 字长的输入消息块，白圈表示全 0 填充，点圈表示输入的最后消息块需填充 0 才满足 16 字长，一条线表示一个压缩函数操作。图 5.1 所示的串行模式，该模式从左到右串行执行，对于 Level 1 中的斜杠圈而已，其输入包括左边 1 个输入和底下 Level 0 中的 3 个输入，该斜杠圈表示压缩函数的输出。整个执行模式的最后输出为 Level 1 中最后一个斜杠圈；图 5.2 所示的并行执行模式，该模式从下往上，逐层操作，Level 0 的输出为 Level 1 的输入。

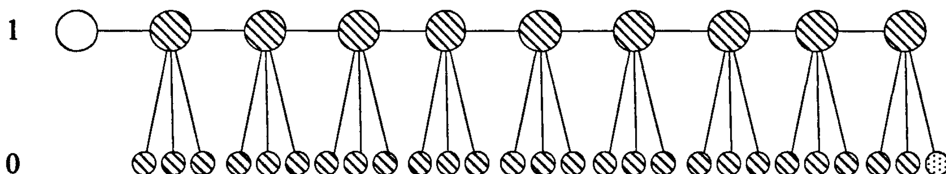


图 5.1 MD6 串行执行模式，L=0

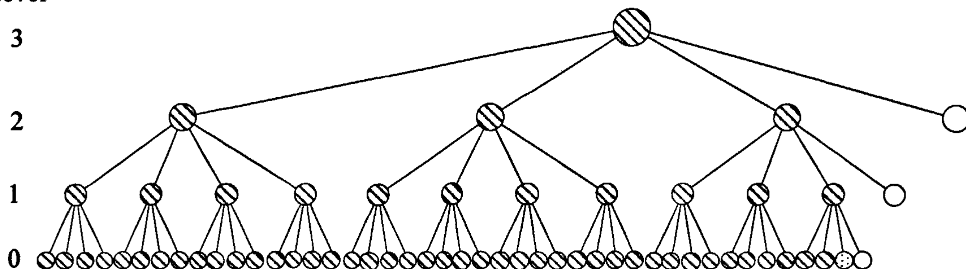


图 5.2 MD6 并行执行模式，L=64

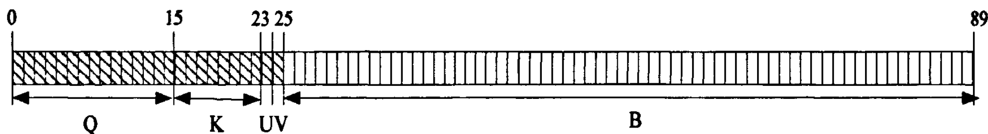


图 5.3 压缩函数输入组成，以字长为单位

5.2 MD6 具体并行实现

使用 OpenMP 对 MD6 进行并行实现，本文使用了两种方法，并将实现结果进行对比分析。以下实现结果均在工作站 HP Z800 Intel(R) Xeon(R) CPU E5530 x64，主频 2.40GHz，内存 15.9GB，MS Visual Studio2005 上运行得到。

由程序实现结果可知，MD6 哈希函数中消耗时间最长的是其压缩函数的处理过程，高达 97.5%。并行实现的一个思路是，在压缩函数中实现并行，提高处理速度。然而，压缩函数的处理规模随着轮数r的确定而确定，在压缩函

数中设计并行算法并不能达到可扩展性的要求。因此本文的两种并行实现算法思路均从 MD6 的树形结构出发。

5.2.1 逐级单独实现

由图 5.2 可知，MD6 并行模式为树形结构，每一层的结果输出为其父级的输入，根据这个特点，一种最直接的并行实现思路是，逐级实现，即先并行实现 Level 0，而后 Level 1，依次类推。设有 N 个处理器，实现的最后一级为 Level L，则个处理器工作方式如图 5.4 所示：

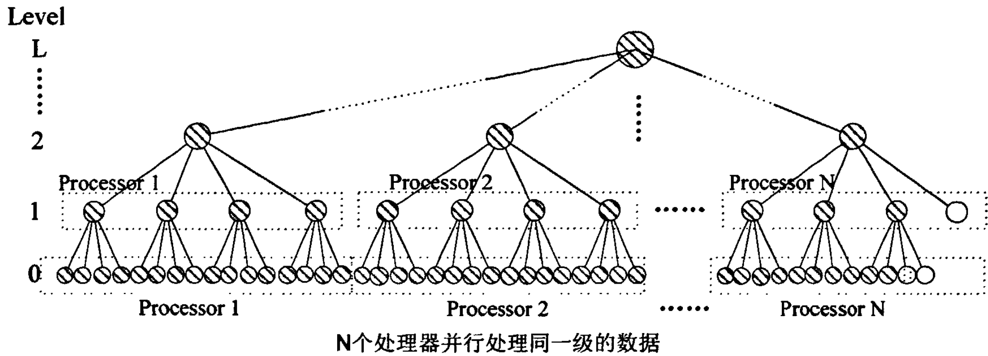


图 5.4 MD6 逐级单独实现

对于每一级而言，都使用了同一个函数，不同的是，根据每一级处理的总的数据块数不同，给每个线程分配的任务每一级都不一样，假设 Level 0 需要处理的数据块数为 B，则每个线程需处理的数据为 B/N，可以使用 OpenMP 中指令 $schedule(static, B/N)$ 实现线程的任务分配。

该实现方式存在以下问题：

(1) 每一级的实现都需要保存输入和输出，一般使用线性存储方式如数组，由此引起的一个问题是，函数中的空间存储于栈中，而计算机分配给每个程序的栈是有大小限制的，相当于 MD6 处理的数据块数局限于一定的规模。由第二章分析可知，当数据规模一定时，随着处理器数的增加，处理速度在达到一个高度后，必将不再提高，甚至由于线程的额外固定开销，处理速度还会降低；

(2) 由于每一级的处理都调用了同一个函数，在函数中实现了线程的创建和销毁，如果每次函数的调用都创建 N 个线程，在一共有 L 级的情况下，整个数据处理完将实现 $N*L$ 次线程的创建和销毁，由此占用了很多的系统资源，引起不必要的开销；

(3) 需要注意的是，随着级数的增加，需要处理的数据块数将减少，此时将不能简单的使用 $schedule(static, B/N)$ 给线程分配任务，这时应该比较并行处理 B/N 个数据块的性能优化程度与线程的额外开销，即此时要达到最好的并行性能可能只需要小于 N 个线程，此时可根据需要使用指令 $Num_threads()$ 设

定线程数或者使用 *if()* 指令。

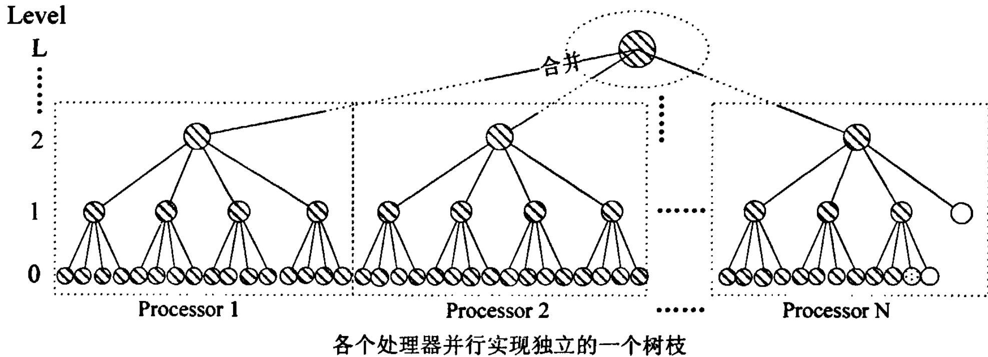
运行结果如表 5.1 所示。

表 5.1 MD6 逐级单独实现结果

CPU 核数	处理速度（MB/s）
1	52.989
2	101.297
3	142.857
4	186.289
5	242.718
6	242.718
7	267.094
8	258.264

5.2.2 串行合并实现

由 5.2.1 节提到的内存问题，可以使用参考文献[37]中提到的串行实现方式——数据驱动，收到一个输入消息块就可以调用 MD6 压缩函数，逐级深入，整个程序运行只需开辟一个很小的结构体空间，存储级数和当前需处理的数据块。利用这种实现方法，可以使用串行合并的思想实现并行，如图 5.5 所示：



各个处理器并行实现独立的一个树枝

图 5.5 MD6 串行合并实现

每个处理器单独执行一个树枝，最后将所有处理器的处理结果当作压缩函数的输入，其输出即为整个输入消息的压缩函数值。

该实现方式存在以下问题：

(1) 分配给每个线程的初始数据块数应该为 4 的幂次，这样才能保证合并后得到的压缩函数值与串行时相同；

(2) 其数据分配方式变得复杂，要使整个程序适应数据规模的变化以及处理器数的变化，必须要设计一个良好的算法给每个处理器分配任务，以及数据

合并时的处理。出于本文目的的考虑，程序实现时只考虑了处理器数的变化。

运行结果如表 5.2 所示。

表 5.2 MD6 串行合并实现结果

CPU 核数	处理速度（MB/s）
1	52.9886
2	108.766
3	163.150
4	214.748
5	266.728
6	322.123
7	369.020
8	416.502

5.2.3 程序实现结果分析

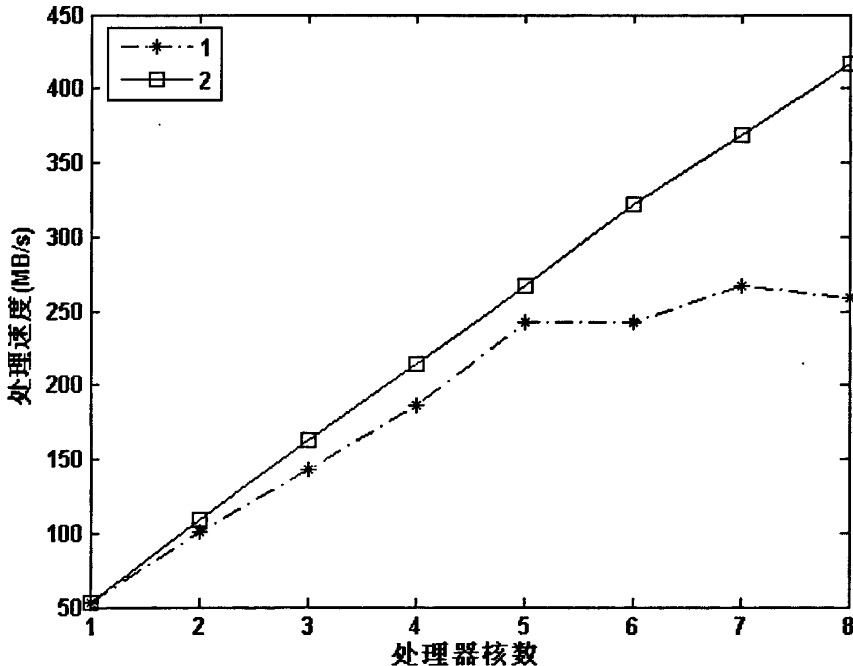


图 5.6 MD6 两种并行实现方式结果

图 5.6 中线条 1 表示了 MD6 逐级单独实现结果，由 5.2.1 中分析可知，程序的运行数据规模一定，在处理器核数为 5 的时候，并行实现达到最大效果，即充分的利用了处理器资源，处理速度与处理器核数成线性增长关系。此后增加处理器核数，并没有对处理速度有很大的提高，由图中 7-8 可知，甚至处理

速度会降低。

图中线条 2 表示了 MD6 串行合并实现结果，由于程序实现过程中并不需要开辟太大的存储空间，因此其运行数据规模可以没有限制，并行实现的效率可以达到处理速度与处理器核数成 N 倍线性增长关系，达到最大效果。

比较图中线条 1 与线条 2 横坐标 1-5 可以看出，虽然两种实现方式均表现出了最大利用处理器资源的特点，然而线条 1 的处理速度比线条 2 的会低一些。原因在于逐级单独实现方式中，每一级的实现均要重复进行线程的创建、销毁和调度操作，处理器核数越大，线程数越大，消耗的时间也越多，由此影响了处理数据的速度。

5.3 本章小结

本章主要提出了两种 MD6 并行实现方法，并对比和分析了实现结果。说明一个良好的并行实现算法可以对程序实现效果起到至关重要的作用。

通过这两种实现方式结果的对比，可以看出 OpenMP 并行实现有其局限性，受到内存空间大小的限制，其处理问题规模也受到了限制，因此，当处理器核数很大时，并行实现并不能最大利用到计算机资源，甚至降低效率。当问题规模很大时，人们一般采用 MPI 等其他方式实现并行处理。

第六章 总结与展望

随着人们对计算能力要求的不断提高、计算机硬件的高速发展和软件技术的迅速提高，对并行算法的研究已经变得相当必要和紧迫。而作为并行计算的一个重要应用分支—安全算法，对它进行更加深入的研究已是势在必行。不管是从应用效率方面或者应用安全的角度出发，并行环境下的安全算法的研究都大大难于串行情况，它不止包括串行环境下的所有问题，还引出了许多新的问题。

本文从并行实现的角度出发，针对安全算法的一般规律提出了一般的并行算法思路，给出相对应的 OpenMP 实现方式，并对可能出现的实现中的问题提出一般的解决方案。并对并行程序进行了性能优化，使得硬件资源得以被充分利用。针对计算机硬件资源的快速更新，要求程序具有可扩展性，即并行程序应该能在不同的硬件环境，表现出高效的并行性。本文也设计出了可应用于一般环境下的并行实现算法。

本文还详细介绍了现有的并行伪随机数生成器技术与优缺点，并给出并行实现的算法步骤。最后总结了现有的对并行伪随机数生成器的检测技术与其基本实现。

文章最后给出了 MD6 两种并行实现方式，通过对两个程序实现结果的对比分析，可以看出 OpenMP 有其自身的劣势，它受到内存等的限制，且不适用于分布式存储系统。当需要处理一个非常复杂且庞大的数据时，如基于物理模型的检测技术，OpenMP 是难以发挥其优点的。因此对算法的并行研究应该更加深入，研究其在不同系统环境下的设计，使并行计算在各种应用领域中发挥其重要的优势。

随着并行计算的出现，现有的安全算法面临着一个尴尬的局面，即在串行条件下行为有效的算法，并不能确保其在并行条件下也同样适用。在并行条件下，对现在安全算法的扩展有两个方向，一是扩大现在串行算法的规模，二是提出适用于并行环境的新的算法。

对随机数的研究已经经历了一个很漫长的充满曲折的历史，其中，不同的随机数生成器被提出，然后研究、测试、证明、提倡使用、广泛使用，最后在一定的环境下应用时发现生成器有着不足甚至是严重的缺陷。很遗憾的是，现今的多种生成器虽然已被发现其存在缺陷，但仍然被广泛使用。现有的所有测试方法中，没有一个测试可以证明或者确定一个生成器可以在一个新的应用中表现出良好的性能。随着并行计算的提出，对现有的随机性检测算法也提出了新的要求，无论是从其实现手法或者检测规模上都需要有很大的提高。而对并行

伪随机数生成器的检测算法的研究也只是开始，仍然面临着很多问题，需要不断的深入研究。

目前，随着计算机硬件的不断升级以及并行算法的提出，解决问题的规模、运行的速度与运行需要的存储空间不再是困难问题。在此环境下，对随机数的基于物理模型的检测技术的研究已经没有任何的硬性阻碍。相信，在不远的将来，这种检测技术将会有很大的突破并能被广泛应用。希望在不远的将来，对安全算法的研究将会有一个很大的突破。

致 谢

首先深深感谢我的导师马文平教授！在我攻读硕士学位期间马老师为我创造了良好的学习环境和学习氛围，提供了锻炼和提高自身专业技能的机会。在学习和生活上始终得到马老师的亲切关怀和悉心的指导。马老师渊博的知识、一丝不苟的工作作风、分析问题的能力以及敏锐的洞察力都给我的学习和研究以莫大的帮助和启发。马老师提倡学术自由，善于引导学生进行创造性思维，鼓励学生发表个人见解。他严谨的学风、豁达的胸襟和诲人不倦的精神令我终身难忘。再次感谢马老师对我谆谆的教导和孜孜不倦的言传身教，这一切都将让我终生受益！

感谢实验室的高胜博士、杨元原博士、何叶锋博士、陈和风博士、余旺科博士、冯赟硕士、傅佩龙硕士、徐明硕士、殷浩硕士、刘维博硕士、刘宝成硕士、冯佳硕士、郭娜硕士、朱继伟硕士等所有的师兄师姐师弟师妹们，以及张荣、陈小光、费腾、赵闻博，感谢他们在学习上对我的关心、支持和帮助、在生活上给我带来的欢乐！研究生两年的实验室生活，给我带来的不仅仅是个人专业技能的提高，更多的是大家相互关心、帮助的温暖与感动！再次深深的感谢他们！

感谢我的舍友吕晨、王丽丽、张晓、李博、刘思伯、王彦平、章悦、赵隽、杨帆、王朝，以及我所有的朋友们，感谢他们对我日常生活和学习上的关心、支持、鼓励以及帮助，感谢他们陪我度过了硕士阶段的美好时光，给予了我无微不至的关怀，使我的生活充满温暖和幸福！感谢他们！

感谢我的家人，没有他们就没有我现在一切一切的幸福与满足！

攻读硕士的两年半是我人生中的一个重要阶段，其中有成功与喜悦也有挫折与失败，我将永远怀念这段人生历程，也将永远记住那些关心、帮助过我的人们！

再次向所有关心和帮助过我的人表示衷心的感谢！

参考文献

[1] Shameem A, Jason R. 多核程序设计技术——通过软件多线程提升性能.北京:电子工业出版社,2007.

[2] 陈国良.并行计算——结构·算法·编程.北京:高等教育出版社,2004.

[3] 陈国良,吴俊敏,章锋等.并行计算机体系结构.北京:高等教育出版社,2002.

[4] 多核系列教材编写组.多核程序设计.北京:清华大学出版社,2007.

[5] 周伟明.多核计算与程序设计.武汉:华中科技大学出版社,2009.

[6] Timothy G. Mattson, Beverly A. Sanders, Berna L. Massingill.并行编程模式.北京:清华大学出版社,2005.

[7] 王剑峰,刘宝宏.Windows 环境下的多线程编程原理与应用.北京:清华大学出版社,2002.

[8] http://zhidao.baidu.com/question/117303681.html.

[9] Barry W, Michael A.并行程序设计.北京:机械工业出版社,2005.

[10] Introduction to Parallel RNGs. http://sprng.cs.fsu.edu/Version1.0/paper/index.html.

[11] Random Number Generation on Parallel Computer Systems. http://www.npac.syr.edu/projects/reu/reu94/cstoner/proposal/proposal.html

[12] A. Srinivasan, M. Mascagni and D. Ceperley. Testing Parallel Random Number Generators. Parallel Computing, 29, 69-94, 2003.

[13] Daniel V. Pryor, Steven A. Cuccaro, Michael Mascagni. Implementation of a Portable and Reproducible Parallel Pseudorandom Number Generator. 1994.

[14] Gerald P. Dwyer, K. B. Williams. Portable Random Number Generators. 2000.

[15] S. A. Cuccaro, M. Mascagni and D.V. Pryor. Techniques for testing the quailty of parallel pseudo-random number generators, in Proc. Of the 7th SIAM Conf. on Parallel Processing for Scientific Computing, SIAM, Philadelphia, 279-284. 1995.

[16] K. Entacher, A. Uhl, S. Wegenkittl. Parallel Random Number Generation: Long-range Correlations Among Multiple Processors.

[17] Chih Jeng, Kenneth Tan. On Parallel Pseudo-Random Number Generation. 2001.

[18] P. D. Coddington. Random Number Generators for Parallel Computers. Version 1.1. 1997.

[19] Paul D. Coddington, Sung-Hoon Ko. Techniques for Empirical Testing of Parallel Random Number Generators. Technical Report DHPC-025, 1998.

[20] P. D. Coddington, A. J. Newell. JAPARA-A Java Parallel Random Number Generator Library for High-Performance Computing. 2004.

[21] D. E. Knuth. The Art of Computer Programming, vol. 2: Seminumerical Algorithms, third ed. Addison-Wesley, Reading, MA, 1998.

[22] I. Vattulainen, T. Ala-Nissila, K. Kankaala. Physical tests for random numbers in simulations. Phys. Rev. Lett. 73. 1994.

[23] M. J. Durst. Testing parallel random number generators. In Computing Science and Statistics: Proceedings of the XXth Symposium on the Interface, 228-231. 1988.

[24] A. De Matteis and S. Pagnutti. Parallelization of random number generators and long-range correlations. Parallel Comput, 15:155-164. 1990.

[25] T. Filk, M. Marcu and K. Fredenhagen. Long range correlations in random number generators and their influence on Monte Carlo simulations. Phys. Lett. B165, 125. 1985.

[26] P. D. Coddington. Tests of random number generators using Ising model simulations. 1996.

[27] OpenMP. http://openmp.org/wp/.

[28] CSDN 多核软件开发社区. http://forum.csdn.net/SList/IntelMulti-core/.

[29] Ivor Horton.Visual C++ 2005 入门经典.北京:清华大学出版社,2007.

[30] Sun Microsystems,Inc.OpenMP API 用户指南.Sun Studio 11,2005.

[31] LLNL 国家实验室 OpenMP 教程. https://computing.llnl.gov/tutorials/openMP/.

[32] Tim Mattson, Larry Meadows. A “Hands-on” Introduction to OpenMP. Intel Corporation.

[33] Tim Mattson, Rudolf Eigenmann. OpenMP Tutorial Part 1:The Core Elements of OpenMP. Intel Corporation.Purdue University School of Electrical and Computer Engineering.

[34] Tim Mattson, Rudolf Eigenmann. OpenMP Tutorial Part 2:Advanced OpenMP. Intel Corporation. Purdue University School of Electrical and Computer Engineering.

[35] Intel Tread Checker 3.1, Guide to Sample Code. http://www.intel.com.

[36] Intel Tread Profiler, Guide to Sample Code. http://www.intel.com.

[37] Ronald L. Rivest. The MD6 hash function - A proposal to NIST for SHA-3. Submission to NIST, 2008.

硕士期间发表的论文和参与的科研项目

一. 参与的科研项目

国家 863 项目“密码算法和安全协议检测分析技术与测评系统”,项目编号:2007AA01Z472。

二. 发表的专利

(1) 马文平,陈秋丽,殷浩.基于随机置换的伪随机序列的随机性检测方法.专利申请号:200910218442.2;

(2) 马文平,秦好磊,陈秋丽.基于抽样的伪随机序列的随机性检测方法.专利申请号:200910024379.9。