.

西安交通大學

硕士学位论文

多线程并行程序动态映射机制研究与实现

学位申请人：鲁晨欣

指导教师：张兴军 教授

学科名称：计算机科学与技术

2020 年 05 月

Research and Implementation on Dynamic Mapping Optimization Mechanism for Multi-threaded Parallel Applications

A thesis submitted to
Xi'an Jiaotong University
in partial fulfillment of the requirements
for the degree of
Master of Engineering

By
Chenxin Lu
Supervisor: Prof. Xingjun Zhang
Computer Science and Technology
May 2020

摘 要

近年来高性能计算机的计算资源显著增加，应用程序的并行规模逐步扩大，使得程序计算任务与系统计算资源的合理匹配成为高性能计算领域的一个关键问题。传统解决这一问题通常采用静态映射优化方法，它们很难适应部分并行程序运行时状态变化大的特点，也不具备通用性和可移植性。此外，当 OpenMP 多线程程序运行在非一致性内存访问（NUMA, Non-Uniform Memory Access）架构的处理器节点上时，多线程间的隐式通信模式使得线程访问共享内存数据的效率并不均匀，这间接导致了程序运行时 Cache 失效率增加，进而导致了程序运行时间增加。

因此，本文首先提出了一种针对 OpenMP 多线程并行应用程序的动态的线程到 CPU 核心的映射优化机制。本文分别对映射机制中的线程间通信量检测模块、分组计算模块和映射执行模块进行了设计，实现了在映射过程中执行线程间通信量检测、线程映射的分组计算和线程绑定的方法，并结合线程创建退出检测和映射的周期性执行控制，使映射机制成为一个整体的系统。本文提出的映射机制最终被实现为 Linux 操作系统的可加载内核模块。

其次，本文在保证映射机制整体优化效果的基础上，提出了一种复杂度更低，额外开销更小的映射分组算法 LHMapping（Loop-based Hierarchical Mapping Algorithm）。该算法结合系统硬件对象的共享特性，根据线程间通信量的数据，对线程逐层分组，实现了线程到具体处理器核心的一一对应。

最后，本文分别在 Intel Xeon 处理器集群和 8 核心的单机上对 NPB 基准测试程序等应用程序进行了 LHMapping 算法的性能测试和映射机制整体优化效果测试。实验结果表明本文设计的 LHMapping 映射算法实现了比其他映射分组算法快 2-4 倍的计算开销和接近的映射精确度。本文设计并实现的动态线程到核映射优化机制对大部分多线程并行程序均有良好的优化效果，在运行时间和 Cache 命中率指标中分别实现了最大约 20%和 25%的性能优化。

关 键 词：多线程并行应用程序；共享内存通信；动态映射优化；映射分组算法

论文类型：应用研究

* 本研究得到“十三五”国家重点研发计划课题“面向 E 级计算机的大型流体机械并行计算软件研制”（编号 2016YFB0200902）资助。

ABSTRACT

Recently, the computing resources of high-performance computers have increased significantly, and the parallel scale of applications has gradually expanded, making the reasonable matching between computing tasks and system resources a key issue in high-performance computing. Lots of traditional solutions are static mapping methods. It is difficult for them to optimize some programs with large changes during runtime. and they don't have compatibility and portability. Furthermore, when OpenMP multi-threaded programs are running on the computers with non-uniform memory access (NUMA, Non-Uniform Memory Access) architecture, the overheads that threads accessing shared memory are unbalanced under implicit communication mode. It indirectly leads to an increase of Cache miss rat, which in turn leads to an increase of running time.

Therefore, the thesis firstly proposed a dynamic thread-to-core mapping mechanism for OpenMP applications. We designed the communication detection module, grouping computation module and mapping execution module in our mechanism respectively, and implemented methods of the inter-thread communication detection, the grouping computation and the thread binding. Combined with the detection of thread creation and exit, the control of periodic execution, our mechanism became a whole system. Finally, our proposed mechanism was implemented as a loadable kernel module for the Linux operating system.

On the basis of reducing the computation overload, we secondly proposed a grouping algorithm LHMapping (Loop-based Hierarchical Mapping Algorithm) with lower complexity and similar accuracy. The algorithm combines the sharing characteristics of hardware objects, grouping the threads layer by layer according to the communication between threads, and finally achieving a one-to-one correspondence between threads and specific processor cores.

Finally, we did the performance tests for the LHMapping algorithm and the overall optimization effect of the mapping mechanism on the Intel Xeon cluster and the 8-core PC. The results show that the LHMapping mapping algorithm achieves 2-4 times optimization on computational time, and close mapping accuracy than other mapping algorithms. The dynamic thread-to-core mapping mechanism has great effect on most multi-threaded parallel programs, and achieves a maximum performance optimization of about 20% and 25% in the running time and the Cache hit rate, respectively.

* 本研究得到“十三五”国家重点研发计划课题“面向 E 级计算机的大型流体机械并行计算软件研制”（编号 2016YFB0200902）资助。

KEY WORDS：Multi-threaded Parallel Applications；Communication in Shared Memory；Dynamic Mapping Optimization；Mapping Algorithm
TYPE OF THESIS：Application Research

* 本研究得到“十三五”国家重点研发计划课题“面向 E 级计算机的大型流体机械并行计算软件研制”（编号 2016YFB0200902）资助。

目 录

* 本研究得到“十三五”国家重点研发计划课题“面向 E 级计算机的大型流体机械并行计算软件研制”（编号 2016YFB0200902）资助。

CONTENTS

1 绪论

1.1 研究背景及意义

随着高性能计算平台和高性能计算技术的不断发展，并行应用程序也普遍具有了规模大和可扩展性强的特点，其中最具代表性的是 CFD（Computational Fluid Dynamics，计算流体力学）并行应用程序，它们的功能通常是求解复杂的流体力学控制方程。在实际应用场景中，由于需要仿真的流体系统相当复杂，因此用 CFD 技术对模型进行区域分解后，其网格规模可达数百亿[1]。因此 CFD 应用程序不仅有较大的并行规模和计算负载，还存在不同网格块计算负载不同的情况。

同时近年来，随着体系结构的研究者们不断用增加处理器核数的方法致力于突破计算性能的瓶颈（神威太湖之光采用 的国产 SW26010 处理器，包含 4 个具有 64 个计算核心的计算单元 CPE[2]；天河三号原型机也全面使用了国产众核处理器，包括飞腾 FT-2000+和迈创 Matrix-2000+，每个处理器节点都集成了 32-64 个 ARM 架构处理器核心[3]），计算机规模在逐步扩大，因此应用程序计算任务与计算平台的计算资源的匹配问题也逐渐成为了高性能计算领域研究的难题。映射优化成为研究者解决计算任务与计算资源相匹配的问题的主要途径。因此，针对以大规模 CFD 应用程序为代表的复杂应用，研究对程序的映射优化方案是进一步优化程序运行性能的一大重要方向，有时甚至能达到优化系统运行效率、降低机器能耗的效果。

进一步地，为了加强对应用程序的并行优化，除进程间采用 MPI 并行外，在更细粒度还会应用多线程并行优化，使计算任务能被更细粒度地分配和调度。在软件层面，常见的多线程并行方式有使用 OpenMP、OpenACC 等并行编程模型，或直接使用 Linux 系统支持的 pthreads 多线程库等。但是在以 OpenMP 为代表的多线程编程环境下，不同的线程之间借助共享内存进行数据交换与共享，也就是通信。在非一致性内存访问（NUMA, Non-Uniform Memory Access）架构的并行计算机中，基于共享内存通信的模式容易使程序的运行产生很大的通信开销不均匀问题。因此，进一步在多线程级研究程序的线程映射优化可以有效改善上述线程间通信的不均匀性，也为针对大型应用程序的总体映射优化研究增加了层次性和弹性。

此外，许多并行应用程序的计算负载与访存负载在程序运行时所占比例变化较大。因此如何能在设计一种动态的运行时的线程映射优化方案，灵活适应程序动态地在计算任务密集和访存任务密集中不断变化的特点，这具有很大的挑战。

本文依托于十三五国家重点研发计划课题“面向 E 级计算机的大型流体机械并行计算软件系统及示范”（编号：2016YFB0200902），主要针对课题中“面向 E 级计算机系统的分层弹性映射机制”子任务，研究既适用于大型流体机械真实流动精细模型 OpenMP 并行应用程序，也适用于大多数通用多线程并行程序的动态的线程与节点内

处理器核心之间的可扩展映射优化方案。本文结合部分并行计算平台的体系结构特点，针对 OpenMP 多线程并行应用程序，研究并实现了运行时动态的线程到处理器核心的映射优化机制。此外，为进一步提升映射机制的优化效果，本文提出并实现了一种额外开销更小，计算精确度不亚于传统算法的映射分组算法。总体上，本文研究实现的多线程并行程序动态线程到核映射机制，以及提出的映射分组算法 LHMapping，为最终实现面向 E 级计算机系统的可扩展分层弹性映射方案奠定了基础。

1.2 国内外研究现状

近年来，高性能计算平台的规模及其体系结构的复杂程度都随着大规模科学计算的需要而不断增加。大规模应用的计算任务与计算机系统的计算资源的匹配问题，使得映射优化这项庞大的研究课题一直处于国内外广大学者的讨论与关注之中。现阶段具有一定影响力的映射优化研究也是从很多不同的角度出发解决了具体的实际问题。

在所有与映射优化相关的研究工作中，按照映射的对象可以将映射优化研究工作分为：对任务执行映射（将作业调度到不同的计算节点上），例如 Nilesh 等[4]在云计算系统上基于作业的能效数据进行动态的调度决策；对应用程序的数据执行映射（将应用程序数据分布到合适的处理器内存上），例如 SCID 算法[5]在 Spark 环境下实现了基于采样的数据放置策略；对进程执行映射（将程序的进程映射到并行计算机的不同处理器或计算节点上），例如 OPP 方法[6]通过进程映射将 MPI 集中式通信转换为一系列点对点通信操作；以及对线程执行映射（将程序同一进程的不同线程映射到计算节点内的不同处理器核心上）。此外，针对不同特点的应用程序，按照映射优化所解决的具体问题，还可以分为：维持负载均衡的映射优化，减少系统硬件资源竞争的映射优化和解决程序进程/线程之间通信开销不平衡问题的映射优化等。

由上述的分类可知，映射优化是一项概念庞大，种类复杂的研究工作，涉及多个领域和多种技术。但国内外仍然存在一些有影响力的研究成果，其研究问题与背景和本文提出的研究问题相似，即研究多线程并行应用程序中线程到处理器核心的映射优化方法。

早期并行计算机的处理器核数较少，应用程序的线程数也较少。因此线程到核映射方法的种类不多，特点也不鲜明。研究者们通常使用近似枚举的方法，对应用程序执行全部可能的线程映射情况，再使用不同的性能评价指标评测程序在多种映射方式下的性能表现，最后根据实际需要从中选出最佳的映射方式，随后按照所选择的映射方式执行相应的程序。

AutoPin[7]通过观测 IPC（Instructions per Cycle，每时钟周期运行的指令数），从不同映射方式中选择具有最高 IPC 值的方式来执行应用程序；但显然借助 IPC 的指标来评价程序运行性能是不全面的，对于 Autopin 中的应用背景来说有效，但不适用于其他应用背景。

BlackBox[8]有着类似的思想，不过它主要面向加入了超线程技术的 SMP 架构的高

性能处理器，且只针对双线程的经典生产者-消费者程序；通过构造线程在同一节点的不同处理器上、同一处理器的不同核心上以及同一物理核心的不同逻辑核心上，分析程序的总体性能来选择最佳映射；BlackBox 实验采用的处理器平台具有的独特性，这使得线程若都分布在同一物理核心上也会出现硬件资源竞争的情况，因此不见得是性能最佳的一种情况。

此外，随着并行应用程序线程数的不断增加和机器学习技术的发展，研究者们开始使用机器学习方法执行的映射优化也比较有成效，如 Wang Zheng 等[9]用机器学习方法预测多核平台上程序运行的最优线程数和最优线程调度策略，并将该映射优化方法实现为基于编译器的平台可移植的自动化方法。

以上这些线程映射方法都属于静态映射，尽管他们非常全面地比较或者训练了各种可能的映射方案，且对某些应用程序优化效果显著；但静态映射方法只能满足核数较少的情况，且需要预先多次运行程序，因此总体上讲他们的通用性不强，实验的可扩展性和可移植性较差。为了解决上述限制，研究者开始考虑动态的映射优化方法。

而要实现在程序运行时动态映射，首先要解决的问题是如何在运行时对程序已执行部分的运行时性能进行测试与分析。使用 Linux 系统内核提供的 Pref 性能分析工具，实时访问系统性能测试单元 PMU，提取运行时状态信息是一种便捷的方式[10][11][12][13]。Suleman 等[14]利用程序运行时信息，设计了可反馈调节的动态线程数映射框架，并且针对数据同步受限和片外带宽受限的两种性能问题，分别设计了同步感知线程机制（SAT）和带宽感知线程机制（BAT），预测程序所使用的最佳线程数。

Ju T 等[15]通过改变 Linux 系统内核模块中的内核控制寄存器，实现了直接在用户空间使用 rdpmc 指令读取系统性能数据信息，最终在 Intel MIC 众核处理器平台实现了基于数据局部性分析的线程映射机制。

Acosta 等[16]提出了一种用于引入了 SMT 技术的 CMP（Chip Multi-Processors，单芯片多处理器）处理器的线程分配算法，该算法不仅考虑了工作负载特征，还考虑了底层的指令提取策略。Moursy 等[17]同样关注于 CMP 结合 SMT 的处理器，提出了一种使用硬件性能计数器来描述线程行为并在同一超线程物理核心上分配兼容线程的算法。

而 Cruz 等[18]与本文同样针对 NUMA 架构的多核处理器平台和解决共享内存通信模式下产生的通信开销不均匀问题。该研究通过在处理器每个核心私有的 L1 或 L2 Cache 上设置硬件计数器，检测并统计缓存一致性协议产生的写失效消息，间接统计到了多个线程访问相同 Cache Line 内容的信息。但是一般情况下用户无法访问并读取具体 Cache line 的内容以及处理器内在 Cache 上传递的一致性消息，因此实验测试时需要搭建缓存模拟器，无法将映射机制实现在系统中。此外，在每个核心上设置一个硬件计数器，在处理器核心数特别多的情况下，开销比较大。

实现动态映射的另一大问题是在执行映射前如何确定线程与处理单元的对应关系。研究者们一般会根据程序性能分析的结果，对线程进行分组划分，以此开发出许多映射算法。但是无论采取哪种策略，分组划分问题都是 NP-hard 问题。

经典的 Edmond[19]采用的图划分算法有效解决了线程组划分的问题，但仅限于线程和处理单元的数量均为 2 的倍数。此外，目前有 Sandia Nation Labs[20]开发的已开源并行软件包，Sandia Nation Labs 用超图（Hypergraph，图中一个边可能连接超过两个顶点）对通信量进行建模，再实施划分；尽管用超图建模出的通信量更加准确，但是繁琐的运算和建模过程不适用于动态的映射优化模式；此外，并行的超图划分软件包相比于普通的并行稀疏矩阵矢量乘的运算，优化效果并不明显。

另一种开源的分组算法软件包是 Scotch[21]，算法将待分组的顶点逐步拆解成多个元素较少的小模块，对他们进行逐层分组；每一层的分组采用基于启发式的双递归的策略；Scotch 双递归的映射计算过程仍然具有较高的时延，但它层次化分组的思想则恰好与并行计算机平台的硬件架构中层次化的特点相匹配，在部署映射时具有一定的应用价值。

综上所述，尽管研究者们尝试用多种途径研究并设计解决映射过程中各部分存在的问题，并不断进行优化，但目前仍缺乏一种能良好结合程序运行时状态且通用性较强的有效解决线程通信不平衡问题的动态映射策略。

1.3 论文研究工作

本文主要依托十三五国家重点研发计划课题“面向 E 级计算机的大型流体机械并行计算软件系统及示范”，涉及的子任务是解决该课题中“面向 E 级计算机系统的分层弹性映射机制”。

在以本项目模拟轴流压气机转子气体流动模型的 CFD 应用程序为基础，用 OpenMP 对计算任务进行细粒度并行优化时，发现程序在处理网格数据通信时访存开销占比较大，也使得早期程序在 intel_x86 架构的同构多核处理器上运行时，性能下降明显。经分析，不同线程之间数据访问速率不均匀，因此存在较大的通信不平衡问题。结合项目提出的优化目标，由此本文考虑在多线程级研究一种有效的映射优化方案。

实际上尽管操作系统的进程线程调度已经在不断升级优化，达到使用者可接受的状态，然而很多应用程序仍然因为计算负载不平衡、通信不平衡、处理器资源竞争等问题需要得到映射优化。因此本文考虑研究一种能不依赖于具体应用程序的，且在程序运行中能动态调节的线程级映射优化机制；并且映射优化机制对项目中的轴流压气机转子模型程序 rotor35，同样具有适应性。

结合以上考虑，本文的主要工作如下：

1）针对 OpenMP 应用程序，在 NUMA 访存模式的高性能计算机平台上，解决程序运行时由于线程之间数据交换和共享不平衡导致的程序运行性能下降的问题，设计并实现合理的具有动态性的线程到计算核心的映射优化机制，从而提升了程序总体性能。

2）基于 1）中设计的映射优化机制，设计并实现一种保证精确度且开销更小的映射分组算法，进一步提升了执行映射优化后程序的优化效果。

3）分别在 Intel Xeon 同构处理器集群 C80 和 Intel Core i7 系列八核处理器的 PC 机搭建的虚拟机上，对本文研究设计并实现的映射优化机制和映射算法均进行了性能测试。

1.4 论文组织结构

本文共分为 6 章，各章的主要内容如下：

第一章 绪论。本章介绍了论文的研究背景及意义、国内外研究现状，并概述了论文研究的项目背景、主要研究工作和论文的组织结构。

第二章 主要理论及方法。本章介绍了本文研究工作中涉及到的相关技术，首先介绍了论文研究内容需要的应用环境，即多线程并行应用程序和 OpenMP 并行编程模型。随后简要介绍了并行计算机体系结构和其中的 NUMA 访存模型。接着介绍了本文对线程间通信和线程映射优化的定义。最后介绍了作为检测线程间通信量的理论基础：操作系统分页和缺页错误。

第三章 动态线程到核映射机制的设计与实现。本章首先概述了设计动态映射机制需要解决实现的问题，针对这些问题，设计了整体映射机制的模块结构和执行流程，然后逐步给出各个问题的实现方案，并介绍了实现映射机制的方法，最后本文分析并介绍了动态映射机制引入的额外开销。

第四章 映射分组算法 LHMapping 的设计与实现。本章首先概述了早期设计的映射机制在映射分组过程中存在的开销问题，针对这一问题设计了简化的、保证算法精度的映射分组算法，文中对算法每一部分的实现细节进行了介绍，最后分析了算法的时间和空间复杂度。

第五章 映射机制的性能测试与分析。本章对所设计实现的动态线程到核映射优化机制在两种平台上基于两个多线程并行应用程序和程序集进行了测试，测试内容包括映射机制整体的优化效果、LHMapping 算法本身的性能、引入 LHMapping 算法后映射机制整体的优化效果和映射中部分控制变量的影响测试，并对实验结果进行了详细分析。

第六章 结论与展望。本章对论文的所有研究工作进行了总结，在给出相关结论的同时分析存在的不足，提出下一阶段工作的展望。

2 相关理论及技术分析

2.1 多线程并行与 OpenMP 编程模型

随着单核 CPU 的性能增长逐渐陷入瓶颈，处理器的体系结构逐渐转变成多核、众核结构。硬件的不断升级，既使并行体系结构更为复杂多样，也持续对研究软件的并行技术提供支持。在多核处理器平台上，为保证所有的 CPU 核都能处在非空闲状态，必须同时申请不少于 CPU 核数的进程或线程，这样才能充分利用多核处理器的性能优势。又因为对线程的操作（线程的创建，删除等）比较简单，因此在多核处理器平台上，运用多线程技术提升程序运行性能是一种主要方式。主流操作系统 Windows, Unix/Linux 等均为多线程并行程序的实现提供了相关接口。

多线程实际上就是一组并发的线程存在于同一个进程的上下文环境中，因此每个线程都会和其他该进程下的线程一起共享很大一部分进程上下文，包括整个进程虚拟地址空间。这便是多线程形成的共享机制。然而一旦存储资源可以被各线程共享，就需要一个规范和标准，使得线程能顺利访问共享存储单元且得到正确的共享数据。即程序员设计的应用程序需要能在共享存储环境下顺利运行。为适应这样的规范和标准，程序员通常借助编程接口实现对应用程序的改进。OpenMP 就是为共享存储模式下编写并行应用程序提供的编程接口。

OpenMP 是由 OpenMP Architecture Review Board 推出的针对共享内存并行的应用程序接口。它本身由编译指令、环境变量及运行时库函数三部分组成。目前 OpenMP 支持 Fortran, C/C++这几种编程语言。

用 OpenMP 编写的多线程应用程序采取 fork-join 的并行执行模式，其中 fork 表示创建新线程或处于阻塞状态的线程被唤醒，join 表示多线程汇集为一个线程。一个简单的 fork-join 执行模式的示意如图 2-1 所示。在 fork-join 模式下，程序执行伊始只有一个线程存在并运行，称为主线程；如果在运行过程中遇到需要并行的任务，主线程会派生出其他子线程来共同执行并行任务，即在这时执行 fork 操作派生多个子线程；并行任务执行完毕后，派生的子线程会退出或者阻塞，因此不再工作，程序控制流程又变为由单独的主线程执行[22]。

此外，OpenMP 还提供了共享内存模型。共享内存模型中，数据变量被分为私有（private）变量和共享（shared）变量。处于共享内存中的共享变量是对处于共享内存外部同名变量的引用；处于共享内存中的私有变量则是处于共享内存外部、由线程私自创建的同名变量的一个拷贝。默认情况下，一个进程内的数据为共享变量，他们对同一进程中的所有线程可见。若多个线程在共享内存区域中实际操作私有变量，由于共享内存中的私有变量都是线程自己持有的某个变量的副本，不对其他线程可见，因此访问私有变量不存在数据竞争；然而如果多个线程同时访问共享内存中的共享变量，

则会产生数据竞争，并且结果不可预测。因此使用 OpenMP 编程模型也需要对数据加锁保护或者用原子操作实现同步。OpenMP 对需要加锁保护的共享数据提供了单线程执行（critical）和原子操作（atomic）等编译制导语句，帮助实现同步和互斥；还提供了 API 中的互斥函数。

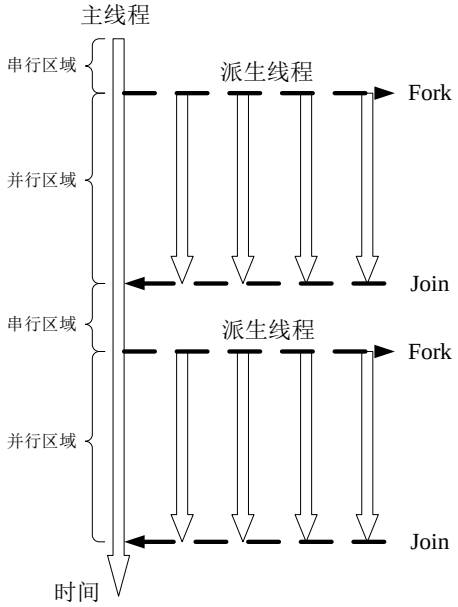

图 2-1 OpenMP 的 fork-join 并行执行模式

OpenMP 使用轻量级的线程实现任务并行，对线程进行操作其本身的开销比较小，适合细粒度的并行。OpenMP 设定的基于共享内存的通信模式也使得只能在具有可共享访问内存的线程之间实现 OpenMP 多线程并行，因此单独使用 OpenMP 时可扩展性不强。

用户使用 OpenMP 编译制导语句实现细粒度并行。编译制导语句由命令和子句组合而成。由于构成编译制导语句的命令、库函数等总共只有数十个，因此 OpenMP 也被普遍称作一种简单的多线程编程模型，应用相当广泛。

2.2 并行计算机体系结构与 NUMA 架构

并行计算机是由多个处理单元/处理器组成的计算机系统，处理单元之前可以相互通信，可协作处理复杂计算任务。并行计算机的提出完全相对于早期只有一个处理单元且只能顺序执行计算任务的串行计算机。由从最早的顺序标量处理计算机，逐步演变为向量流水线计算机；由单指令流单数据流计算机，演变为多指令流多数据流并行机。

当前，并行计算机从结构模型方面可划分为六大类：

单指令多数据流机 SIMD（Single-Instruction Multiple-Data），并行向量处理机 PVP（Parallel Vector Processor），对称多处理机 SMP（Symmetric MultiProcessor），大规模并行处理机 MPP（Massively Parallel Processor），工作站机群 COW（Cluster of Workstation）

以及分布式共享存储多处理机 DSM（Distributed Shared Memory）。其中 SIMD 并行计算机目前多为专用，其余并行机均属于多指令流多数据流计算机[23]。

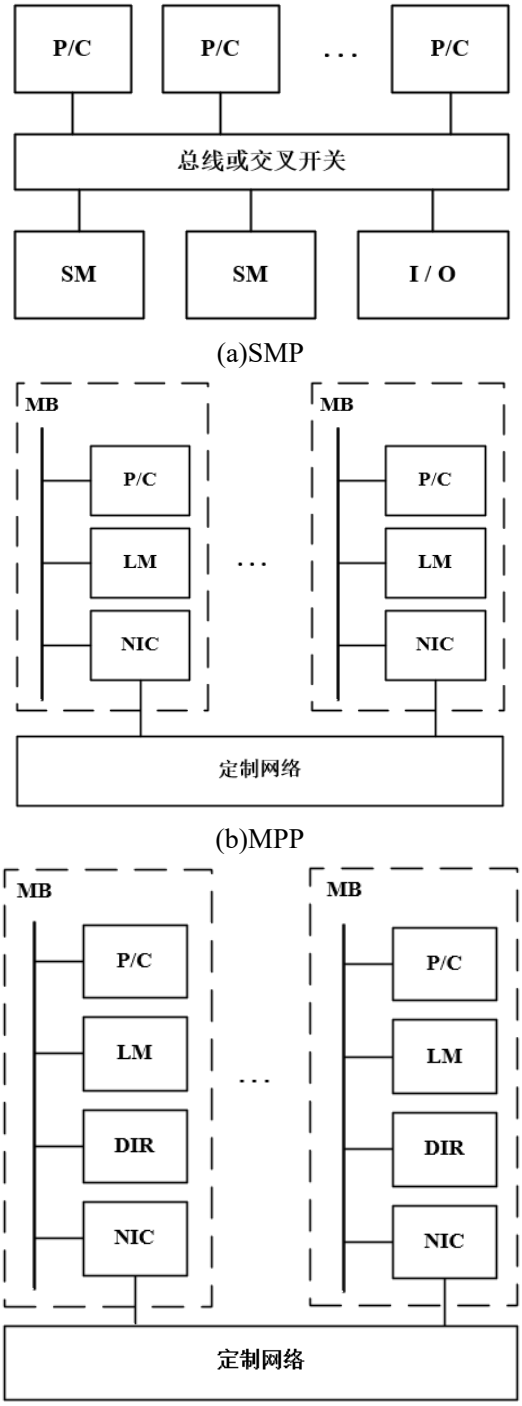

图 2-2 几种并行计算机结构模型示意图

图 2-2 展示了比较常见的 SMP、MPP 和 DSM 这几种并行计算机的模型。其中 P/C（Microprocessor and Cache）是微处理器和高速缓存，LM（Local Memory）是本地存储器，SM（Shared Memory）是共享存储器，MB（Memory Bus）是存储器总线，NIC（Network Interface Circuitry）是网络接口电路，DIR（Cache Directory）是高速缓存目

录。SMP 系统是完全对称的，每个处理器都可以等同地访问共享存储器、I/O 设备等系统资源，以及接受操作系统管理。也正因为 SMP 完全共享存储，会产生内存带宽不足等资源竞争问题，这限制了处理器数目的上限。MPP 一般指超大型的计算机系统，它是一种异步的 MIMD 机器，MPP 平台上运行的并行程序一般由多个进程组成，他们之间采用消息传递机制通信。DSM 系统相比于 MPP，增加了高速缓存目录 DIR，它用来支持 Cache 一致性。且 MPP 和 DSM 与 SMP 的最主要区别是它们有分布在各 socket 中的局存。随着体系结构发展，SMP、MPP 等结构逐渐趋向一致，DSM 也是 SMP 和 MPP 优点的相互结合。

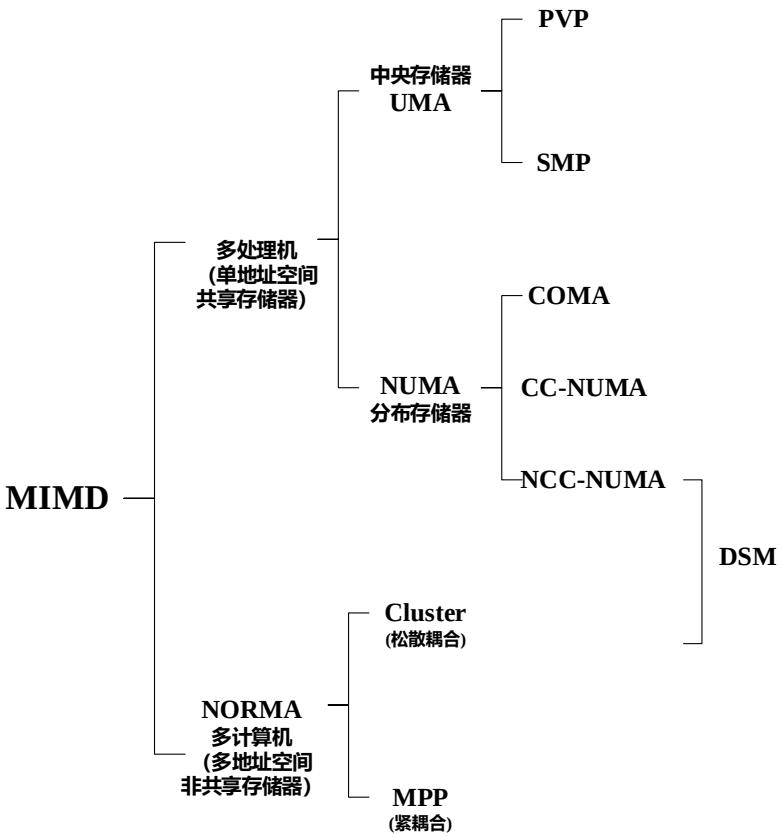

图 2-3 并行计算机系统的不同存储结构汇总

并行计算机的结构模型和访存模型的相互关系如图 2-3 所示[24]。按访问存储器的模式划分，又可以将 MIMD 并行机分为均匀存储访问模型（UMA）计算机、非均匀存储访问模型（NUMA）计算机和非远程存储访问模型（NORMA, No-Remote Memory Access）计算机。其中，NUMA 架构具有的显著特点是可共享存储器物理上分布在系统中的所有处理器中的，所有分布在本地处理器上的存储器共同构成全局地址空间。由此导致 NUMA 系统具有的另一大特点是不同处理器访存的开销是不同的，访问本地存储器或访问某一范围内的共享存储器速度更快，而访问远端的存储器或全局共享存储器速度更慢[25]。NUMA 模型的这一特点也是本研究设计映射优化机制最主要解决的问题。

2.3 线程映射优化

2.3.1 线程映射优化的定义

广义上认为，线程映射是指在多线程并行的应用程序中，根据应用程序的运行特点和性能特征，结合程序运行所在计算机平台的架构特点，找到线程到处理器处理单元的映射（对应关系），最大程度提升应用程序的性能（可以是多方面的：负载平衡、减少硬件资源竞争、减少通信开销等等）[26]。

在 OpenMP 的共享内存通信模式中，线程之间通过读写访问共享内存区域的数据变量实现通信，这种通信方式是隐式的。同时如本文第 2.2 节所述，在 NUMA 这样的并行体系结构中，内存层次结构中的某些存储单元（内存，高速缓存等）可以由一个以上的线程/处理单元共享，这也使得线程之间的通信性能有所不同。图 2-4 简要描述了这一情况。

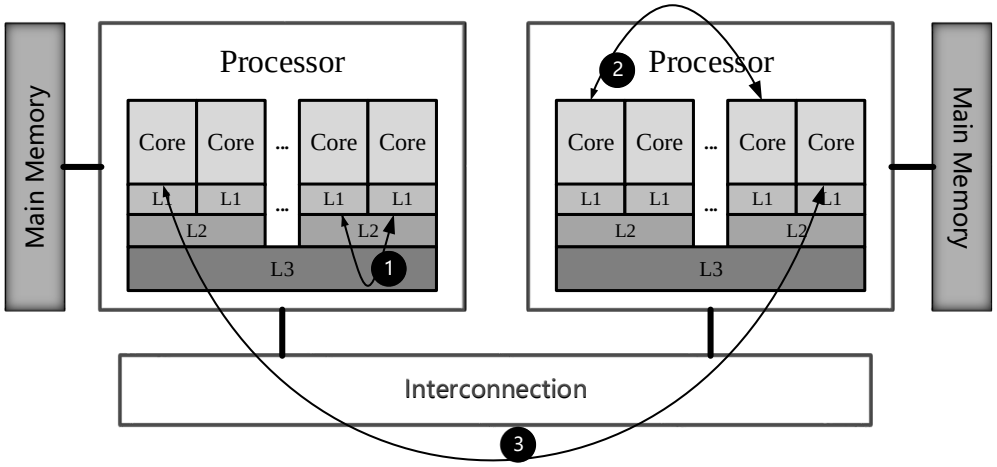

图 2-4 处理器不同核心之间通信性能不平衡示意图

从图中可以看到，在 NUMA 架构的计算机上，通信情况 1 表明线程处于两个共享同一个 L2 Cache 的 CPU 核心上，他们之间最优情况下可以访问共享 L2 Cache 直接通信；通信情况 2 表明线程处于两个不共享同一 L2 Cache，但处于同一个 Processor 上共享同一 L3 Cache 的 CPU 核心上，他们之间最优情况下可以访问共享 L3 Cache 直接通信；但是通信情况 3 表明线程处于两个不在同一 Processor 的 CPU 核心上，他们之间不共享任何 Cache，需要借助片上互连访问到另一 Processor 的内存单元，才能获取数据，进行通信。因此显然情况 1 和情况 2 的核心之间通信要比情况 3 通信更快。

因此本研究考虑在共享内存通信模式下，利用线程映射优化来解决线程之间通信不平衡的问题。这样，线程映射优化的实质是根据线程之间通信量的大小，使相互之间通信量大的一些线程尽量分布在物理位置更接近的 CPU 核心中，使之能共享相同的内存单元。其中，本文将线程之间的通信量定义为：在程序运行时，若两个不同线程依次访问了相同内存单元的数据内容，则称这两个线程之间产生了一次通信事件；通信量就是线程与线程之间产生通信事件的数量。

2.3.2 线程映射的理论优化效果

基于本文第 2.3.1 节对问题的描述和对映射优化的定义，线程映射优化的主要目地是使频繁通信的线程分布在物理位置相近的核心，使其共享相同的 Cache 或处理器局部内存，进而使线程访问共享内存数据的效率更高。因此映射优化减少了程序运行时线程间的通信开销，进而减少了程序的总运行时间。

另外地，将通信频繁的线程映射到位置相近的核心还有一个好处是减少了 Cache 未命中的数量。Cache 未命中可以分为三种：首先是无效未命中（Invalidation miss），当用于线程共享访问的某个缓存行是无效的，则该缓存行在被替换到内存之前均是无效的，则对该缓存行的访问会产生无效未命中，本文定义的线程映射优化对这种情况的缓存未命中并无实质效果。第二种缓存未命中是容量未命中（Capacity miss），在基于共享内存的并行应用程序中，当某个线程所需要访问的缓存行被其他可访问该共享 Cache 的线程替换掉，发生缓存容量未命中[27]。也就是多个共享相同 Cache 的线程所需读写访问的总数据量大于该缓存所能容纳的数据量，因此会经常发生缓存行替换。因此，通过映射优化，更多不同线程都需要的缓存行数据可以存放在这几个线程所共享的 Cache 中，减少了容量未命中的发生[28]。另一种缓存未命中是复制未命中（Replication miss），当同一个处理器上有多个 Cache 都含有相同的缓存行，则这减少了 Cache 的有效容量，容易造成缓存未命中。因此部署有效的映射优化，避免同样的数据分布在多个 Cache 中。

2.4 分页机制与缺页

由上述对研究问题中线程映射的定义和描述，映射机制根据线程间通信量的大小指导映射的选择与执行，且映射的执行发生在程序运行时。因此在运行时统计线程间的通信量成为本研究中解决的关键问题。

本研究选择采用基于缺页错误的通信检测和统计方法，为映射决策提供指导。支持这一方法的关键理论依据如下。

2.4.1 虚拟存储器与页表

一个系统中的进程是与其他进程共享 CPU 和主存资源的，然而随着系统中进程的增加，越来越多的存储资源被程序所需要。现代计算机系统为有效管理存储器且避免出错，引入虚拟存储器的概念。虚拟存储器为每个进程都提供了一个大小一致且由进程私有的地址空间，也就是虚拟地址空间；虚拟存储器保证了进程的虚拟地址空间不会被其他进程破坏。

这样，程序运行时读写数据访问的内存地址是一个虚拟地址，而不是实际的物理内存地址。操作系统再将虚拟地址映射到合适的物理地址上，使不同进程访问数据的实际物理内存地址位于内存中的不同区域，且彼此不重叠。这其中的关键环节就是虚拟地址到物理地址的映射，操作系统需要处理好这一过程。

在采用分页机制内存管理的操作系统中，虚拟地址到物理地址的映射是通过 MMU（Memory Management Unit，内存管理单元）借助页表共同完成的。页表是一个数据结构。MMU 在每次地址转换时都会读取页表，在页表中检索给定的虚拟地址，通过读取对应的页表项得到相应的物理地址信息。操作系统负责维护页表中的内容，以及在内存与磁盘之间来回置换页。图 2-5 是一个单级页表的基本组织结构。

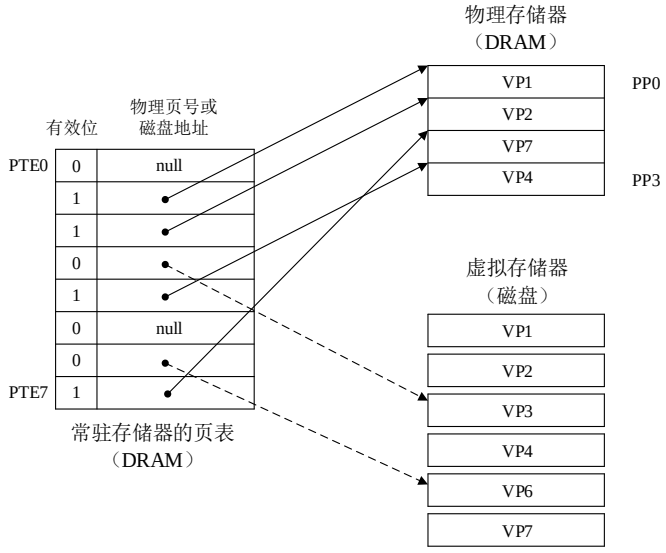

图 2-5 单级页表的基本组织结构

图中 PTE 表示页表项（Page Table Entry, PTE）；VP 表示系统将虚拟存储器（一般是磁盘）分割成的大小固定的块，称为虚拟页（Virtual Page, VP），每个虚拟页大小为假设为$P = 2^P$个字节，一般情况下 Linux 系统内存页的默认大小是 4KB；类似地，系统将物理存储器（例如 DRAM 缓存）分割成大小相同的块，称为物理页（Physical Page, PP）[29]。由图 2-5 可以看出，页表是包含了多个页表项的数组。每个页表项包含两部分，第一部分是一个有效位（valid bit/present bit），记录此页是否在内存中；第二部分用来记录物理地址。若有效位为 1，表示此页在内存中，则页表项后部直接记录内存页地址。若有效位为 0，一种情况是这个虚拟页还未被分配，则页表项中记录一个空地址；另一种情况是这个虚拟页此时没有在内存中，则页表项中记录该虚拟页在磁盘上的起始地址。

2.4.2 缺页

根据上文所述，当有效位为 0，此页可能从未被分配入内存，也可能由于页替换被调出了内存。在这种情况下，就会发生缺页（Page Fault）。发生缺页时，由内存管理单元发出中断，操作系统调用内核中的缺页中断处理程序。系统会根据某个算法（不同系统可运行不同的页面置换算法）选择内存中已存在的某个页作为牺牲页，将这个牺牲页调回磁盘，并修改该页的页表项。随后，系统将当前发生缺页中断的虚拟页由磁盘调入内存，并根据调入此页的内存位置，更改页表项的内容。基于图 2-5 中的页表结构图，一个完整的缺页中断及恢复过程如图 2-6 所示。

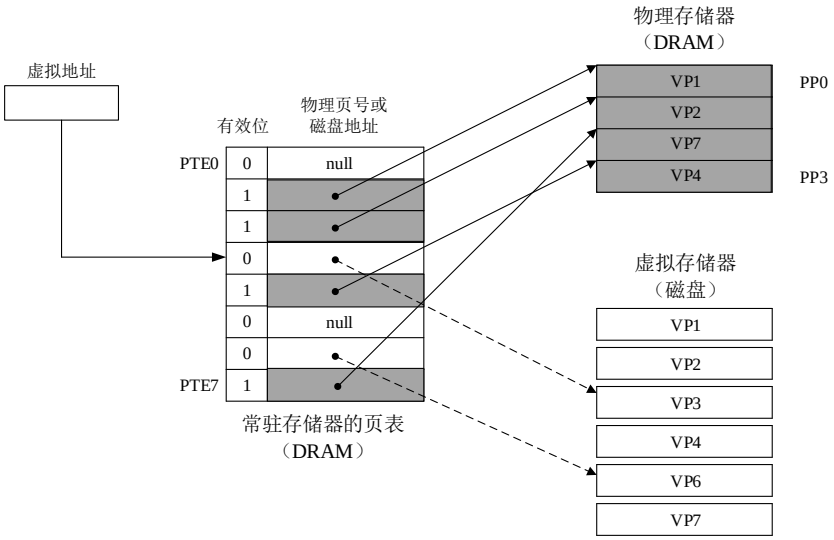

图 2-6 触发缺页中断时的页表结构示意图

图 2-6 显示，MMU 在拿到一个虚拟地址并进行地址转换的过程中，找到了相应的页表项 PTE3，但是 MMU 发现 PTE3 中的有效位是 0，于是触发缺页中断并将消息传送给 CPU，CPU 开始执行缺页中断处理程序。缺页中断处理程序查找产生缺页的原因并进行恢复，图 2-7 反映了产生缺页异常后的恢复处理情况。

由图 2-7 可以发现，缺页处理程序选择虚拟页 VP4 作为牺牲页，用磁盘上 VP3 的拷贝取代了 VP4 在内存和页表中的位置。最后缺页处理程序重新执行之前导致缺页中断的指令，页面置换后该指令就可以正常执行了[29]。

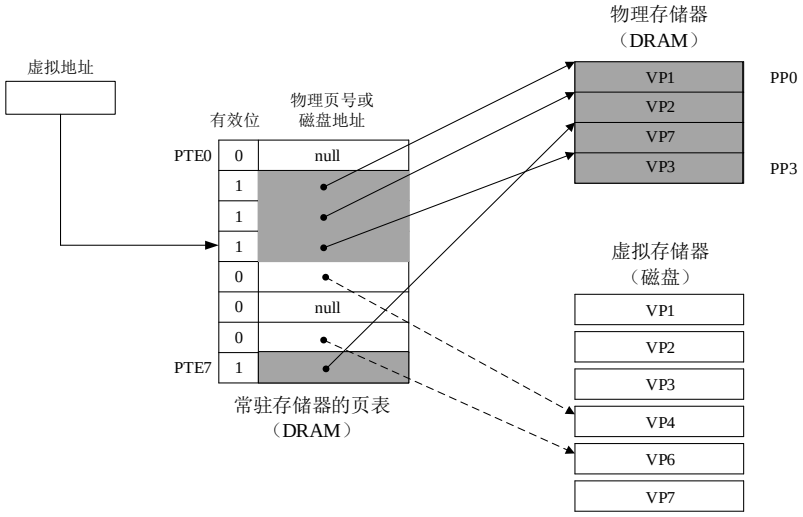

图 2-7 缺页异常处理的页表结构示意图

由上文所述，操作系统对内存管理采用的分页或段页式机制以及缺页错误机制为本文的研究中以页大小为粒度进行线程访问数据的检测提供了基础。

2.5 本章小结

本章首先介绍了多线程并行应用程序的基本概念和特点，以及 OpenMP 并行编程

模型的主要概念、执行模式和使用方法；其次简要介绍了并行计算机体系结构的结构模型和访存模型，叙述了 NUMA 模型的特点，由此表明了本文线程映射主要解决的问题；然后对本文问题背景下本文对线程映射的定义和线程间通信的定义进行了介绍，以及对执行线程映射优化的优势进行了理论分析；最后介绍了线程间通信量检测使用到的关键理论基础——操作系统内存分页管理和缺页中断。

3　动态线程到核映射机制的设计与实现

根据本文第 2.3 小节的描述，本研究把程序运行中两个不同的线程依次对相同数据的访问操作称为线程间的通信。因此当 OpenMP 程序在 NUMA 计算平台上运行时，为了解决共享内存通信模式下线程之间通信开销不平衡的问题，本文对映射的定义是：根据计算平台上体系结构的硬件信息和程序运行时线程间通信量情况，把通信比较频繁的线程映射到处理器中物理位置相近的核心。

由此本研究在设计动态映射机制时首先提出三个问题：如何在运行时统计线程间的通信量信息；如何根据线程间通信量的数据，获得线程分配给具体处理器核心的结果；如何根据分配的结果将线程部署在系统中。再对这三个问题研究并实现有效的解决方法。最后通过一些关键性的控制将本研究对这三个问题的解决方法进行有机结合，最终形成一个完整的系统。本章的后面几个小节将将分别对这几部分的研究内容与结果进行叙述。

3.1 映射机制的模块设计与流程设计

根据上述提出的三个问题，本研究分别实现对应于它们的解决方案，并封装为三个不同的模块——通信量检测模块，分组计算模块和映射执行模块。并引入控制映射周期性执行的周期性控制模块，和控制映射机制在多线程环境下执行的线程创建退出检测模块，使它们构成整体控制模块。

本研究最终设计实现的动态映射优化机制的流程图和模块图如图 3-1 和图 3-2 所示。

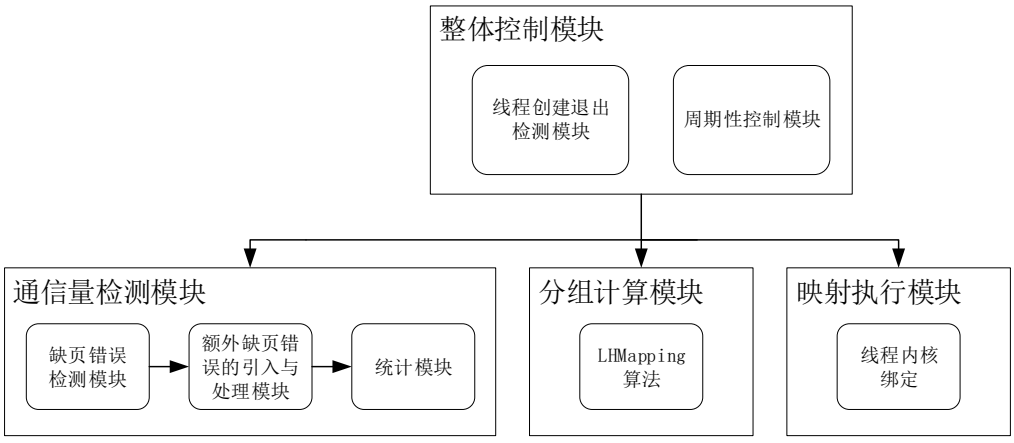

图 3-1 线程动态映射机制模块图

具体执行流程描述如下：

多线程并行应用程序开始运行时，映射机制持续检测线程的创建。一旦程序运行于多线程环境，映射机制随着程序多线程地执行，持续执行通信量检测模块，即下面流程描述中的步骤 1；同时映射机制以一定时间间隔周期性地重复依次执行分组计算模块

和映射执行模块，即下面流程描述中的步骤 2 和步骤 3。如此运行，直至检测到所有线程退出，程序终止或程序退出多线程运行环境。若应用程序以 OpenMP fork-join 的这种模式执行，则映射机制仅在多线程运行状态下工作。步骤 1-3 的具体内容描述如下。

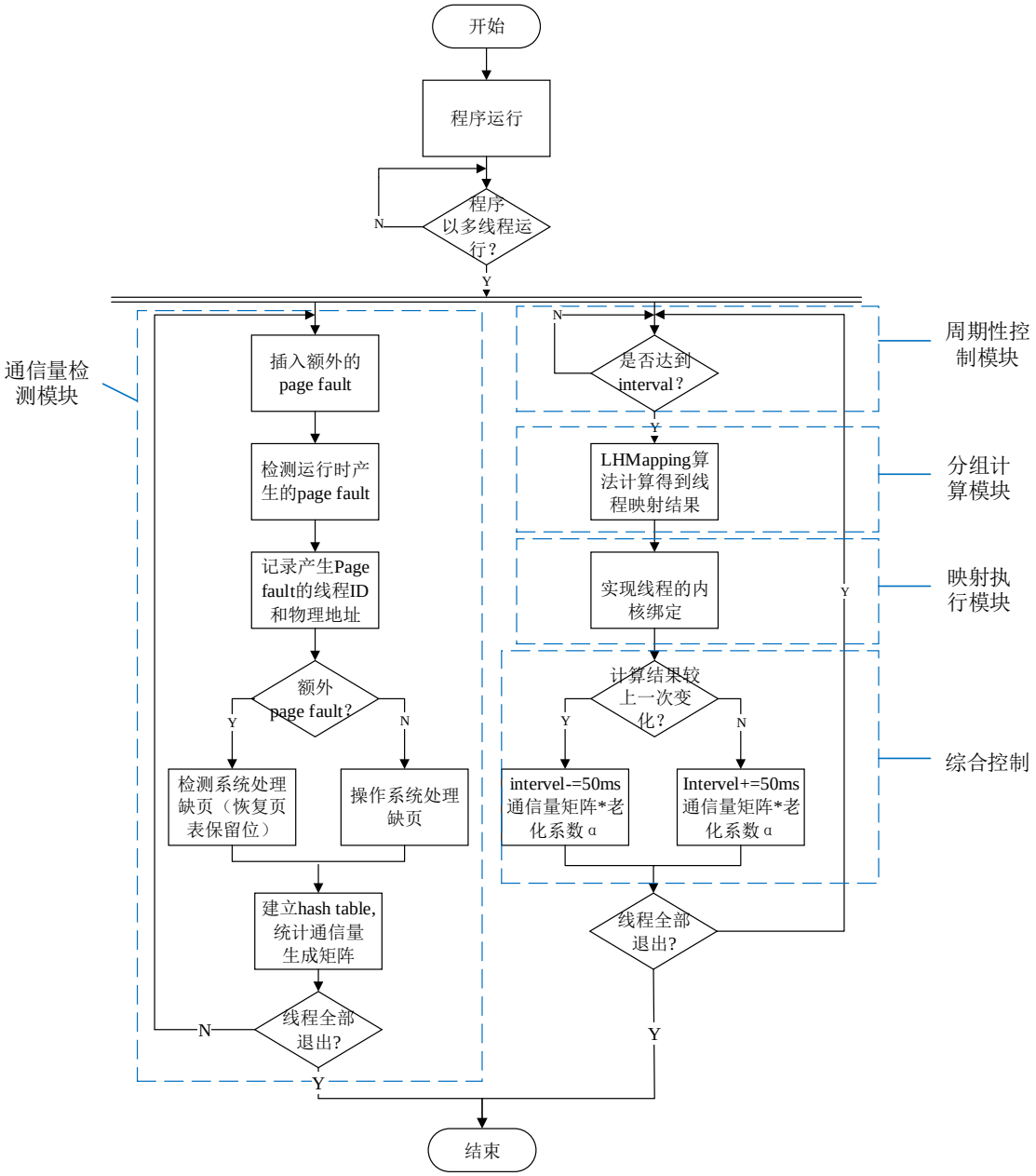

图 3-2 线程动态映射机制流程图

1）步骤 1，程序在多线程运行时的环境下，检测并统计线程间通信量。借助 KMAF[30] 中的基于缺页错误的通信检测方法，在运行时持续地检测每个线程产生的缺页错误信息，记录一段时间内不同线程在相同内存页上产生的缺页错误。并维护一个大小为线程总数的通信量矩阵$A \in R^{n\times n}$（其中，n表示程序运行的线程总数，矩阵元素$a(i,\ j)\ (i \in n, j \in n)$表示线程i和线程j的通信量数值），持续性地把一段时间内在相同页面产生缺页错误的线程之间计为一次通信过程，以此更新通信量矩阵A的值。不论在程序运行的哪个时刻计算并执行线程映射，当时通信量矩阵中元素的数值都是执行映射的唯一参考

依据。详细的通信量统计与检测实现方法将在第 3.2 小节进行介绍。

2）步骤 2，根据步骤 1 中维护的通信量矩阵，计算线程的分组。使用本研究提出并设计实现的 LHMapping 算法，对所有线程进行分组，分组结果将指定哪个线程映射到哪个 CPU 核心中。算法输入表示线程间通信量的矩阵 *A*，以及程序运行所在平台的系统硬件架构信息（借助 Hwloc[31]工具实现），输出用 map 数据结构表示的分组结果。分组计算的原则是尽可能将相互间通信量较高的线程分到同一个组中。组数及组内元素数量由系统硬件架构决定。详细的 LHMapping 算法将在第 4 章进行介绍。

3）步骤 3，根据线程分组结果，执行映射。由于步骤 2 中线程分组的结果用 map 表示，它表示出了线程与 CPU 核心的一一对应关系。因此直接依次调用 Linux 内核函数 sched_setaffinity()为每个线程设置其 CPU 亲和度，暂时将线程绑定到对应的 CPU 核心上。在之后的一段时间内，系统不会对程序运行中的任何线程进行迁移。直至下一次分组计算的结果产生，才再一次重新执行线程的绑定。详细映射执行与周期性控制的方式将在第 3.3 和 3.4 小节进行介绍。

3.2 线程间通信量检测模块

3.2.1 线程间通信量的检测

整体上，本研究设计的动态映射机制使用了基于缺页错误的通信量检测方法，下面对它的具体实现方法进行详细描述。

根据本文第 2.4 节所述，多线程并行应用程序运行时，某线程访问某存储页面的数据时产生了缺页错误，反映了该线程需要访问该页面的数据，而该页面却不在内存中，或者该页面由内存管理单元控制的页表项失效了。利用这种特征，检测机制只要能检测到两个不同线程先后在访问相同页面时产生了缺页错误，则表示他们需要访问相同页面的数据，于是检测机制认为这两个线程之间产生了一次通信事件。因此通过检测程序运行时操作系统中产生的缺页中断，记录每次缺页中断时的线程 ID 和内存物理地址，映射机制可以间接检测两个线程之间产生的通信事件。

然而，在程序正常运行状态下，一旦某个页面被调入内存，仅有极小的可能该页在内存中长时间未被访问，然后由于内存空间不够被换到磁盘空间上，则线程再次访问该页面数据时会在该页面再次产生缺页错误；其他情况下这种情况下其他线程不太可能继续在访问该页面时产生缺页错误。因此，为了使通信量检测机制有效，需要出现检测出多个不同线程均访问了同一个页面的数据，且都在访问该页面使产生缺页错误的情况。因此，需要在通信量检测机制中人为地引入一些额外的缺页错误。

引入额外缺页错误，在程序运行过程中，fork 一个线程，不断遍历应用程序进程内存空间中的页表，周期性地随机更改某一页表项。更改时将该页表项中的 present 保留位置为 0. 则当下次有线程访问该内存页的数据时，尽管页面在内存中，也会因 present 位无效而产生缺页错误。尽管由于引入了额外缺页错误，使系统中增加了额外的处理缺页中断的开销，但保证了通信量检测的精确度。实际执行时根据应用程序的内存消

耗情况，将引入的额外缺页错误的数量占总缺页数量的比例作为一个变量并维持在一个范围内。

由通信量检测机制检测出的缺页错误分为两种情况：一是由于线程访问数据的页面不在内存中，这是由程序运行时自身产生的缺页中断，由操作系统调用缺页中断处理程序完成中断处理；二是由通信量检测机制额外引入的缺页错误，检测机制自行处理这种缺页中断。自行中断处理的程序操作是：直接恢复产生缺页错误页面的页表项的 present 位，并由产生缺页错误的线程访问该页面中线程所需的数据。

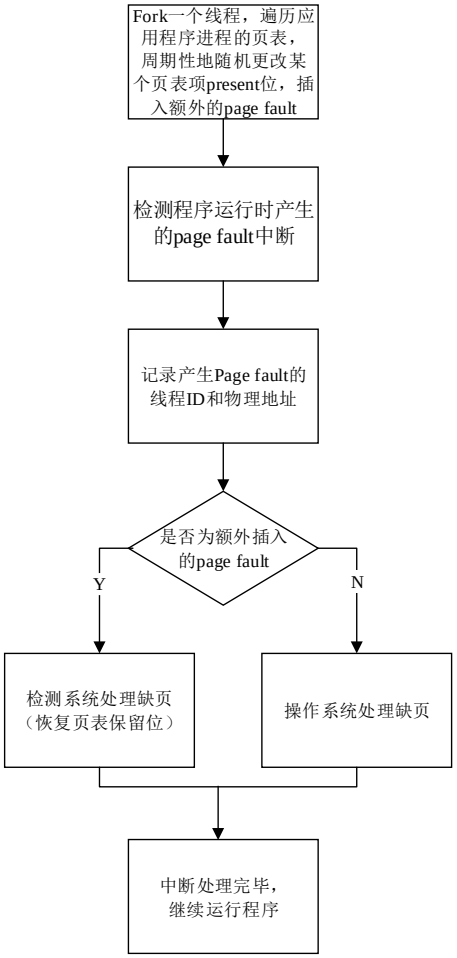

图 3-3 动态映射机制线程间通信量检测过程执行流程

线程间通信量检测的处理流程如图 3-3 所示。

3.2.2 线程间通信量的统计

通信量检测过程每检测出一个缺页错误，统计机制就会收集产生这个缺页错误的线程 ID 以及发生缺页错误页面所在的物理地址。

以此为基础，通信量统计机制定义一个哈希表，完成通信量统计过程。哈希表以程序运行所在平台内存页大小的某个倍数作为统计粒度，以除留余数法设置哈希函数。每个哈希表项可存放数量相同的队列，这些队列均表示不同物理地址的内存块，队列与具体的内存块呈一一对应的关系。队列中记录了在执行信量统计过程中，在该队列

对应的页面上发生了缺页错误的线程 ID，且队列中只记录 ID 不同的线程。同时，为每个队列都设置一个长度上限 l，l 保证了通信量统计机制只能记录同一个页面在某一段时间内产生过缺页错误的不同线程。因此合适的 l 值既能有效记录所有可能的通信事件，也能减少因为程序运行时间不断推进引起的统计误差。在本研究的映射机制中，设置 l=4. 每当队列中记录入一个新的线程号TID_{new}，认为线程TID_{new}与此队列中已存在的所有线程均产生了一次通信事件，相应地使通信量矩阵中对应的元素自增。程序运行时随时更新通信量矩阵。

对线程间通信量进行统计的示意如图 3-4 所示。

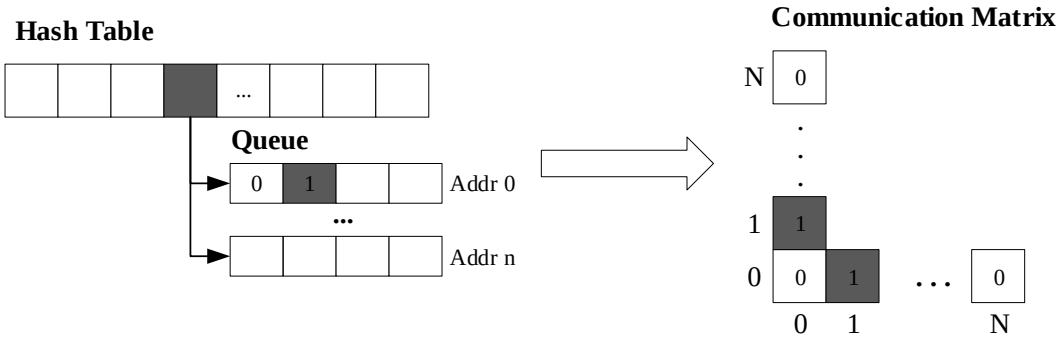

图 3-4 线程间通信量统计示意图

由图 3-4 可以看到，Hash Table 用来存放所有物理页面的内容，其表项中的每个队列表示物理地址不同的页面。在图示中，地址 Addr0 对应的页面先后由线程 0 和线程 1 产生了缺页错误，统计机制将这两个线程的 ID 记录在表示 Addr0 所在页面的队列中。因此映射机制认为线程 0 和线程 1 之间产生了一次通信事件，于是将图 3-4 线程间通信量统计示意图 右边表示的通信量矩阵中 A[0][1]和 A[1][0]位置的元素自增 1. 如此随着应用程序的不断运行，统计操作也在不断执行。

3.3 分组计算模块

映射分组计算的目的是把通信量相对较大的线程分到同一个组中，实现线程与硬件对象的一一对应。这样在后续映射执行时，会把同一组内的线程映射到共享同一个硬件存储单元的 CPU 核心上。可以直接根据通信量矩阵的数值信息，将相互通信量大的线程构建成一组；或先将通信量矩阵转化为表示线程间通信量的带权无向图，再对通信量图进行图划分。不论采取上述哪种策略，对分组的计算都是一个 NP-hard 的问题[32]。通常算法研究者们采用启发式方法，贪心策略，甚至是机器学习的相关方法解决这种分组或图划分问题。

在本研究早期设计实现的映射优化机制中，使用了 Scotch Mapping Library 提供的映射分组算法作为映射机制中分组计算模块的实现方法，其算法执行的示意如图 3-5 映射计算的层次分组执行示意图所示。Scotch 分组算法采用双递归的层次分组方法。算法双递归的思想是指，假设对 N 个元素进行分组，每组元素个数是 K，对第一个待分组元素，则算法考虑将问题分解为两个部分：若把它分进组内，则问题变为从剩余的 N-

1 个元素中选取 K-1 个元素进入组内；若不把它分进组内，则问题变为从剩余的 N-1 个元素中选取 K 个元素进入组内。将这两个部分取得的分组方法进行比较，选择组内通信量最大的一种分组；且比较两端的计算都使用递归的方法。因此 Scotch 映射分组算法的时间复杂度大约是$O(n^3 \cdot logn)$. 此外，Scotch 算法采用这样的执行流程：先从最底层开始分组，到中间层分组，直到对顶层完成分组，生成拓扑树的根节点；紧接着，算法将计算结果向中间层进行反馈，最后向最底层反馈。

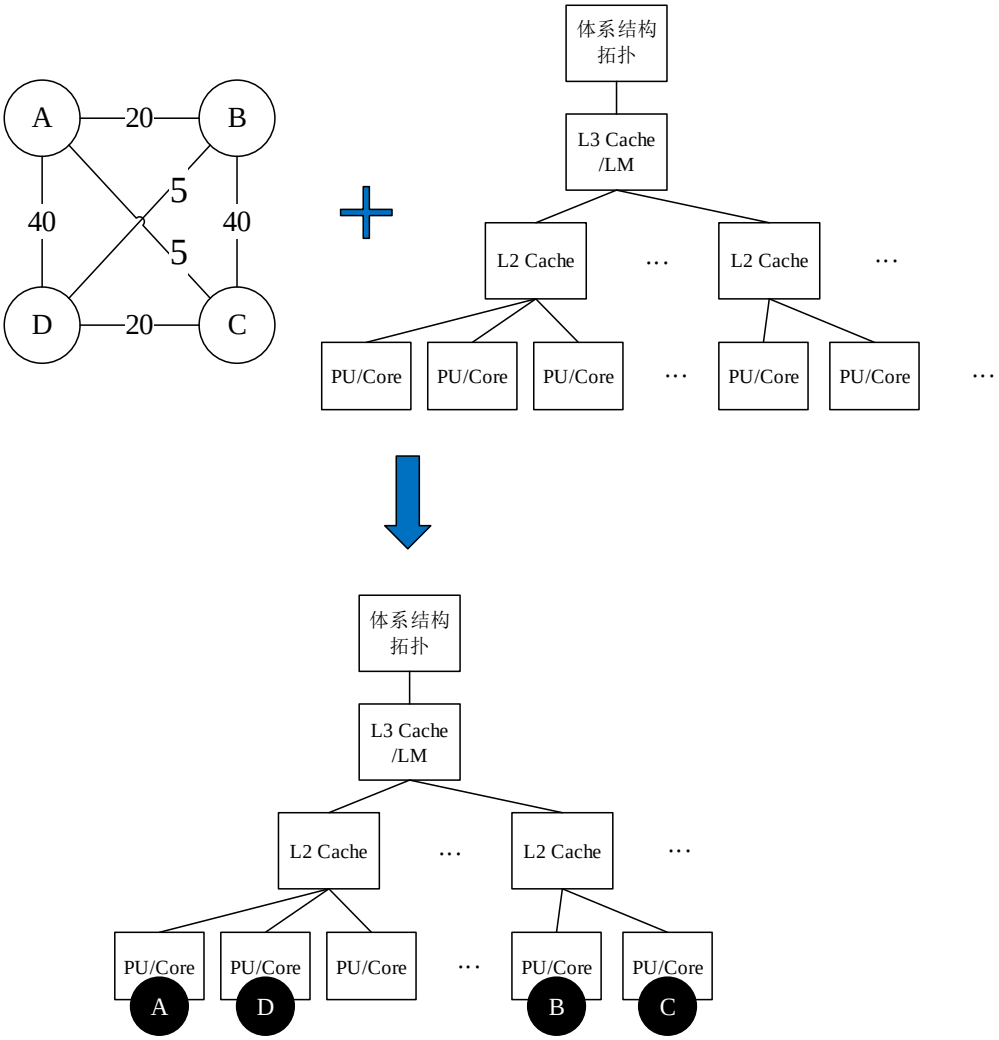

图 3-5 映射计算的层次分组执行示意图

然而这样的算法复杂度和算法执行流程尽管分组精度比较高，但大量的运算时间不适合本研究所设计的动态映射优化机制。这样无疑使映射机制引入的额外开销随着线程数的增加越来越大，所以当程序规模扩大，程序线程数增加后，本研究实现的映射机制显得亟需进一步优化。由于对程序运行时线程间通信量检测的实现其本身就是一个难题，因此无法迅速而有效避免通信量检测过程产生的额外开销；而由于动态映射的执行会不可避免地导致线程的迁移映射，由此引发额外 Cache misses 造成的额外开销也不可避免。故本研究选择设计一种开销更少的映射分组算法，最终提出了 LHMapping（Loop-based Hierarchical Mapping Algorithm, 基于循环层次化分组的映射算

法）算法。本文将在本章第 3.6 节对映射机制额外开销进行更详细的分析。

3.4 映射执行模块

为了执行映射分组计算的结果（即将计算好的线程与 CPU 核心一一对应的关系，实现在系统中），需要用到 CPU 亲和性的相关概念。CPU 亲和性是指使进程或线程在某个给定的 CPU 上尽量长时间的运行而不被迁移到其他处理器的倾向性。

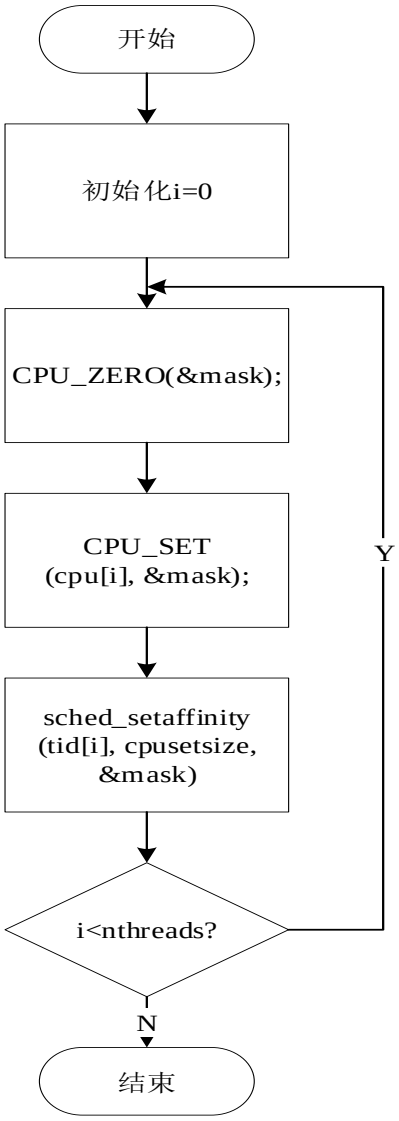

图 3-6 映射执行流程图

在 Linux 内核中，表示 CPU 集的结构体 cpu_set_t 含有与 CPU 亲和性相关的 *mask*（位掩码）变量。它与系统中的 CPU 逻辑核心（在非超线程处理器中是物理核心）相对应，每个 *mask* 位掩码中值为 1 的位都代表一个可以被正常调度的 CPU，而值为 0 的位则对应一个不可调度的 CPU. 通常的系统状态下位掩码中的所有位都会被置为 1，即操作系统可以在所有硬件平台的 CPU 核心中做调度[33]。但是如果人为地设置指定线程的指定关联位掩码，就能控制线程运行在相关联的 CPU 核心上。因此本研究借助 CPU

的硬亲和性，在每次执行映射时强制地将应用程序的线程绑定到系统中某个指定的CPU 核运行。

具体的对映射执行模块的实现流程如图 3-6 映射执行流程图 所示，其实现步骤如下：

1）输入映射分组计算后保存的映射结果 *map*，它是一个拓扑树，树的最底层根节点保存的是线程 ID 与具体的 CPU 核心 ID 的对应信息。初始化临时变量 *i*，令 *i*=0.

2）访问 *map* 结构最底层的第 *i* 个叶子节点，选取这个节点上保存的线程，即其编号为 *tid[i]*，其对应的 CPU 核心编号为 *cpu[i]*.

3）在 Linux 系统内核中，内核代码为每个线程都保存了一个变量位掩码 *mask*.因此执行映射时首先调用 Linux 内核定义的宏 void CPU_ZERO(cpu_set_t **mask*)，清除系统或用户为该线程之前设置的 CPU 位掩码。

4）其次调用 Linux 内核定义的宏 void CPU_SET(int *cpu[i]*, cpu_set_t **mask*)，将编号是 *cpu[i]*的 CPU 核心加入该线程关联掩码表示的 CPU 集中，*mask* 关联掩码随即更改为设置了 *cpu[i]*后的值。

5）可以选择调用 Linux 系统多线程接口 pthread 库中提供的线程的 CPU 亲和性设置函数 int pthread_setaffinity_np (pthread_t *tid[i]*, size_t *cpusetsize*, const cpu_set_t **mask*)，把编号为 *tid[i]*的线程限制在步骤 3-4 中设置的 *mask* 所代表的 CPU 集中运行；或者直接调用进程的 CPU 亲和性设置函数 int sched_setaffinity(pid_t *pid*, size_t *cpusetsize*, cpu_set_t **mask*)进行相同的设置，这个函数中 *pid* 使用线程的编号 *tid[i]*. 至此完成对线程 *tid[i]*的核绑定。

6）选取 *map* 的最底层叶节点中下一个未被绑定的线程，即令 $i = i + 1$. 重复步骤 2-5，直至所有线程完成绑定，映射执行完成。

3.5 动态映射机制的其他控制

3.5.1 通信量矩阵的处理

为了提升映射的优化效果，在每次线程的分组计算过后，需要对通信量矩阵 *A* 做一定的处理。即每次执行分组计算后，将通信量矩阵的所有元素值 $a(i,j)\ (i \in n, j \in n)$乘以一个系数$\alpha\ (0.5 < \alpha < 1)$，称为削减系数$\alpha$. 由于通信量矩阵的统计贯穿线程生命周期的始终，因此这样处理减少了某个时间点之前线程间通信量对后续通信量矩阵数据的影响，进而减少了对分组计算的影响。

本研究实现的映射机制暂置$\alpha = 0.75$，即令

$$a_{new}[i][j] = a_{old}[i][j] - (a_{old}[i][j] \gg 2) \qquad (3\text{-}1)$$

由于矩阵中的每个元素都需要乘以削减系数，因此设置为 0.75 使得计算时仅使用一条右移指令和一条加减运算指令，避免了因复杂的乘除指令过多而引起较大的额外开销。在本研究的实验评估中，将削减系数α作为影响映射优化性能的一个变量，具体的测试结果将在第 5.4 小节详细说明。

3.5.2 周期性控制

映射机制使用周期性的方法运行分组计算和映射执行这两个模块。因此相邻两次分组计算和映射执行之间有一个时间间隔，记为$interval$. 为了在尽可能提升映射优化效果的前提下避免过多的额外开销， $interval$ 的取值必须有一个合理的范围。

本研究实现的映射机制暂设初始时$interval = 200ms$. 在相邻两次分组计算中，若后一次的分组计算结果与前一次结果一致或仅存在很小偏差，即程序的运行特性和线程间的通信模式没有发生明显变化，则将 *interval* 增加 50ms，即经过更长的时间再次进行分组计算和映射执行。反之，若前后两次分组计算结果相差很大，说明线程间通信模式变化很大，需要及时调整，故将 *interval* 减少 50ms.

在多线程程序运行的全过程中，本研究设计的映射机制都将保证$50ms < interval < 1s$. 在本研究的实验评估中，将时间间隔 *interval* 作为影响映射优化性能的一个变量，具体的测试结果将在第 5.4 小节详细说明。

3.6 动态映射机制的实现

上述动态映射优化机制在处理时需要更改 Linux 操作系统内核的缺页中断处理机制以辅助完成通信量检测过程；同时映射执行时需要调用 Linux 内核提供的线程 CPU 亲和性设置函数。因此在执行映射时，无法基于具体应用程序，单纯地在用户态下完成映射优化。

又因为 Linux 操作系统的内核是一个整体的、综合的以及庞大的程序。它把所有的模块：进程管理，内存管理，中断管理，文件系统，网络协议等都融合在一起，因此系统本身的可扩展性和可维护性比较差。为解决这一问题，Linux 系统使用了内核模块机制。内核模块，即动态可加载内核模块（Loadable Kernel Module, LKM），是一组能扩展操作系统功能的程序。只不过它是只能独立编译，不能独立运行的程序。它在运行时会被链接到内核中，并在内核空间中运行，而不是像应用程序那样在用户空间运行。内核模块机制的存在，使用户可以根据自己的需要，在不用对内核重新编译的情况下，动态地装入或移出模块[34]。

经过分析本研究最终将所设计的映射优化机制实现为 Linux 操作系统的内核模块，并装载入内核。只要在加载过线程动态映射优化的内核模块的操作系统中运行多线程应用程序，系统就会执行所设计实现的映射优化机制。

由于本研究使用了不同的平台进行性能测试，因此在程序移植时需要保证多个平台的操作系统具有相同内核版本。本研究在实验时使用内核 3.10.0 版本。

在超级用户模式下，首先执行 “make” 命令编译，得到一个 “.ko” 目标文件；再执行 “insmod” 命令，将内核模块装载入内核中。

3.7 动态映射机制的额外开销

由于本研究设计的映射优化机制是在程序运行时动态执行的：动态地在程序运行

时进行通信量检测与统计，在程序运行时周期性地进行映射分组计算和线程 CPU 亲和性设置。因此在解决程序通信不平衡问题的同时，不可避免会或多或少引入额外开销。但是如果控制额外开销在可接受的范围内，程序性能还是可以得到有效优化。

经过分析，所设计的映射优化机制的额外开销主要来源于以下几个方面：

1）线程间通信量的检测与统计。首先在使用基于缺页错误的线程间通信量检测方法时，由于所访问页面未被调入内存中而由操作系统本身产生的缺页错误，其中断由操作系统本身的中断处理程序进行处理；而由检测机制通过修改页表项 present 位而增加的额外缺页错误，其中断由检测机制自行处理。这样一来，既产生了额外缺页中断引入的额外开销，又产生了处理额外缺页中断过程引入的额外开销。但是这样的开销很难定量表示。其次，统计线程间通信量的过程可以在缺页中断恢复后与应用程序并行完成，因此其额外开销可忽略不计。因此通信量检测模块产生的额外开销主要由额外缺页中断的处理引起。

2）映射分组的计算。每经过 *interval* 时间，动态映射机制根据实时统计的通信量计算合理的线程分组。在多线程环境下，线程保存之前程序运行的上下文，转而执行映射分组计算过程。这使得分组计算过程产生了额外开销。

3）映射的执行。每经过 *interval* 时间，动态映射机制进行分组计算后根据这个计算结果依次为每个线程调用 CPU 亲和度设置函数把线程绑定到对应的核上。执行 CPU 亲和度设置函数的过程产生了额外开销。

4）此外，由于每次映射执行后，大多数线程根据分组计算的结果被迁移到了其他 CPU 核心上，其迁移原核心的 Cache 中保留着该线程所需的一部分数据，而迁移目标核心的 Cache 中很有可能并无这些数据，由此引发了额外的 Cache 失效，产生了额外开销。但由于操作系统本身对正在运行的程序的线程也可能进行迁移操作，因此这部分额外开销影响不大。

额外开销是否影响着映射机制的优化效果，本文会在后续的实验测试部分进行介绍。

3.8 本章小结

本章首先根据第 2.3 节线程映射优化的定义和描述，提出了合理的动态映射优化需要解决并实现的三个问题：即通信量的检测与统计问题、线程的映射计算问题和映射的执行问题。随后根据这三个方面的问题，依次寻找解决方案，并构建出动态映射机制的整体模块构成和机制的执行流程。本章详细介绍了使用基于缺页错误的线程间通信量检测与统计的方法及其操作步骤；详细介绍了使用基于 CPU 亲和性设置的映射执行的方法及其操作步骤；详细介绍了本研究在映射机制中引入的关键性控制变量；最后介绍了动态映射机制引入的额外开销。

4　映射分组算法 LHMapping 的设计与实现

本文在第二章对本研究线程映射优化的定义是：将多线程并行程序运行时通信量大的线程映射在物理位置相近的 CPU 核心中，减少线程间通过片间互连访问其他处理器局存的通信过程，解决通信开销不平衡的问题。因此在本文第三章提出的动态线程到核映射机制中，需要根据线程间通信量的大小，通过计算尽量将通信量大的线程划分在一个组内，并映射到可以共享相同硬件对象（如可共享 Cache）或处于同一个 Socket 上的处理器核心中。本研究实现的动态映射优化机制最初使用 Scotch 分组算法，计算线程的分组结果。但根据本文第 3.3 节分组计算模块的介绍和第 3.7 节对额外开销的分析，本研究选择自己设计一种计算时延更低但对计算精度影响很小的分组算法——LHMapping 算法。

4.1 LHMapping 算法描述

LHMapping 算法沿用了 Scotch Mapping Library 的映射算法中利用硬件平台上可共享硬件对象（多核共享的 Socket，共享 Memory，共享 Cache 等）的层次化分组方式。因此算法需要输入由 Hwloc 工具[31]检测得到的体系结构硬件对象信息，Hwloc 工具会将这种信息表示成拓扑数结构，根节点对应计算平台的一个节点，依次向下的中间层树节点对应构成计算机节点的 socket 或 processor、共享的片上内存或 L3 Cache、共享的 L2 Cache 等，最底层的叶子节点对应 CPU 核心（具有超线程功能的处理器最低层叶子节点可以对应 CPU 逻辑核心）。在进行分组时，算法以自底向上的方式，先将需要分布到每个 CPU 核心的线程，根据共享 L2 Cache 或共享 L3 Cache 可以共享的核数 K，每 K 个线程分为一组；底层分组完毕，接着处理到再上一层共享硬件对象的分组，即把最低层分好的线程组再分为容纳若干小线程组的大线程组；依此类推，直到分组到达根节点。

而分组算法的内核，即如何确定每个分组，考虑采用贪心策略。处理最底层的分组时，算法输入本文第 3.2 节方法统计得到的通信量矩阵，先向组内放置入第一个未被分组的元素，然后对每个待分组的元素，计算他们与组内已有元素的通信量值之和，选择具有最大通信量和的元素进入组内；依此循环进行元素的选择，直到一个组的元素已被占满；继续循环产生一个组的步骤，直至当前分组层的所有分组都已生成，没有待分组元素剩余。

除了处理最底层的分组，处理其他层分组时都需要对通信量矩阵进行更新，把线程与线程之间的通信量矩阵通过线性加和转化为线程组与线程组之间的通信量矩阵，作为计算下一层分组的算法输入。算法输出一个与体系结构硬件对象拓扑树工整对应的线程组拓扑树，用变量 map 表示，这就是分组的结果。

以上就是本研究设计 LHMapping 算法的背景和主要方法，以及本研究将

LHMapping 算法称为基于循环层次化分组的映射算法的原因。下面对 LHMapping 算法各部分的具体细节进行详细介绍，并给出算法的伪代码和执行流程。对 LHMapping 算法的实验测试将分为两个部分进行：一是算法本身的开销和分组映射精度测试，通过与 Scotch 算法进行对比来体现；二是算法加入本研究设计实现的映射机制中时，映射机制整体的优化效果。这一部分将在第五章进行详细介绍。

4.1.1 顶层算法

顶层算法的伪代码和流程图如图 4-1 所示，它展示的是 LHMapping 算法顶层的执行过程。由于是多层次分组，因此首先定义两个全局变量。整型变量 nlevels 表示一共需要分组的层数。另一个全局变量是 nObject，它是一个具有 nlevels 个元素的整形数组，表示每一层中共享的硬件对象的数量。例如在引入超线程技术的 intel 处理器下，nObject[0]表示某一个节点的处理器容纳的 CPU 逻辑核心总数；nObject[1]表示这个节点的物理核心总数；而 nObject[2]表示节点内所有可共享 L2 Cache 的数量，依次类推。

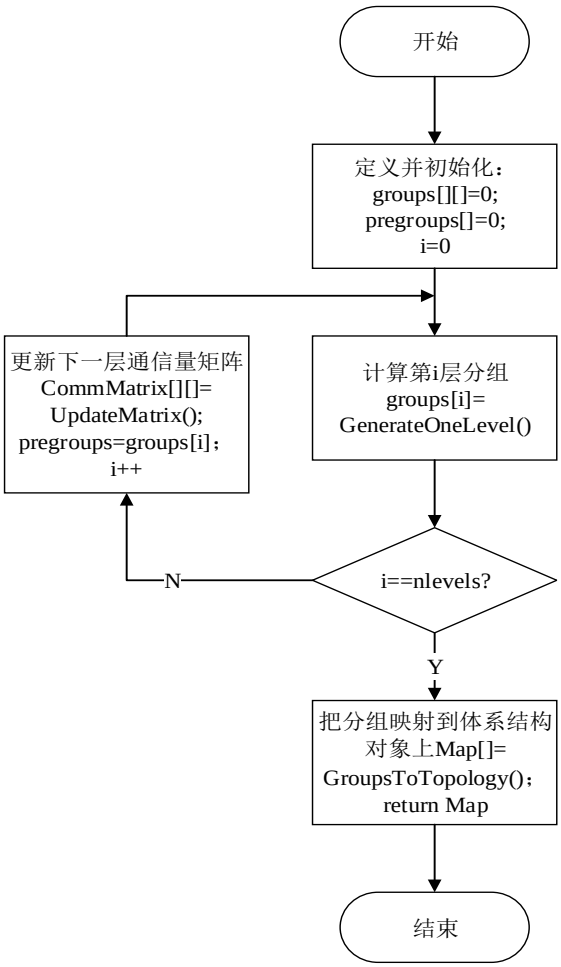

图 4-1 LHMapping 顶层算法 MappingAlgorithm 流程图

同时用局部变量 level 标识正在执行分组的层级。例如 level=1 表示当前计算的是一个线程对应一个 CPU 核心并决定与哪些线程共享同一个 L2 Cache 时的分组计算过程；level=2 表示当前计算的是一个线程组对应一个共享 L2 Cache 并决定与哪些线程组

共享同一个处理器芯片（socket）时的分组计算过程。依次类推。若处理器应用了超线程技术，则用 level=0 表示当前计算的是一个线程对应一个 CPU 逻辑线程并决定与哪个线程共享同一个 CPU 物理核心时的分组计算过程。

算法一 MappingAlgorithm

输入：最初的线程间通信量矩阵 CommMatrixInit[][]，线程数 nThreads

输出：映射结果 map[]

局部变量：待分组单元总数 nElements，i，组数 nGroups，最顶层分组 rootGroup []，通信量矩阵 CommMatrix[][]，分组结果 groups[]，前一层分组结果 pregroups[]

全局变量：硬件拓扑信息 HardwareTopoRoot，分组层总数 nlevels，各层硬件对象数量 nObjects[]

```
1  begin
2      for i = 0 ; i < nThreads ; i = i + 1 do
3          groups[i].nElements = 0;
4      nElements = nThreads;
5      CommMatrix = CommMatrixInit;
6      for i = 1 ; i <= nlevels ; i = i + 1 do
7          pregrouops = groups;
8          [nGroups, groups[i] ] = GenerateOneLevel(CommMatrix, nElements,
                                       nObjects[i], i, pregroups);
9          if i < nlevels then
10             CommMatrix = UpdateMatrix(CommMatrix, groups, nGroups);
11         nElements = nGroups;
12     rootGroup.nElements = nElements;
13     for i = 1 ; i < nElements ; i = i + 1 do
14         rootGroup.element[i] = groups[nlevels][i];
15     GroupsToTopology(HardwareTopoRoot, rootGroup, map);
16     return map;
17 end
```

算法的流程描述如下：

1）定义并初始化保存分组结果的数据结构，除了 level 变量初始化为 0 之外，还需要定义初始化两个数组 groups 和 pregroups. groups 代表分组的总结果，groups[i]表示第 i 层的分组结果，即此时 level = i. pregroups 表示前一层的分组结果，显然数组 pregroups 实际上就是 groups[i] (0 <= i <= nlevels) 中的某个值。将 groups 和 pregroups 全部初始化为 0，如算法第 3 行和第 7 行所示。

2）循环地分别调用函数 GenerateOneLevel 和 UpdateMatrix，他们分别执行在某一层中进行分组计算的任务和进入到下一层计算前的更新矩阵操作，如算法第 8-10 行所示。同时，在每次循环中更新 pregroups 和 nElements 变量，他们作为实参传递给下一

层 GenerateOneLevel 函数。循环的退出条件是分组到达了最顶层。当完成最顶层的分组后，所有线程和线程组都映射到了具体的核心和共享硬件对象上，则这一层循环不用再执行更新矩阵的操作，完成分组后直接记录结果。

3）调用 GroupsToTopology 函数，将记录的分组结果映射到体系结构中的具体硬件对象上，如算法第 14 行所示。返回得到的 map，映射算法至此结束。

4.1.2 生成一层分组

图 4-2 描述的是在 LHMapping 算法中生成某一层分组的流程。算法分别输入通信量矩阵 CommMatrix[][]，待分组的元素数（线程或线程组）nElements，当前分组所在的层级 level，上一层的分组结果 pregroups[]和本层分组要匹配的硬件对象数 nObject. 输出分得的组数 nGroups 和分组结果 groups[level].

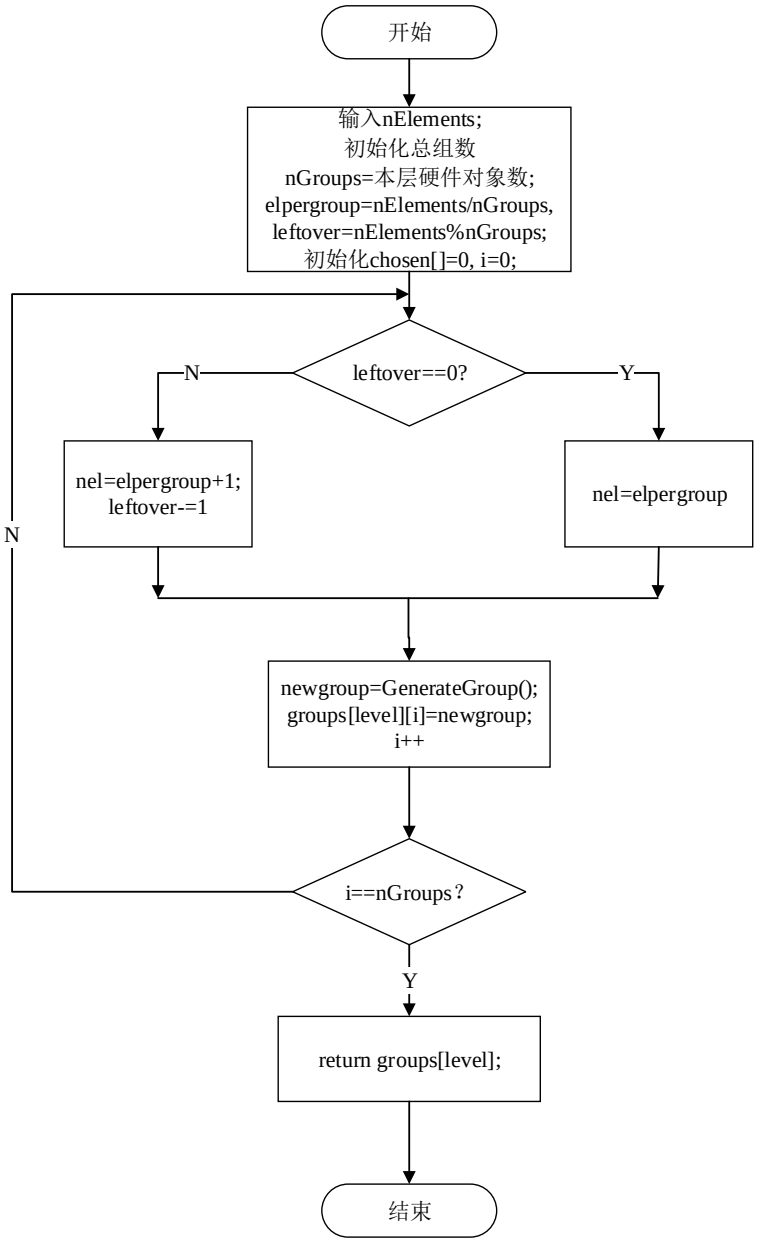

图 4-2 LHMapping 算法子函数 GeneratOneLevel 流程图

算法二 GenerateOneLevel

输入：通信量矩阵 CommMatrix[][]，待分组元素数 nElements，硬件对象数 nobject，正在分组的层级 level，上一层分组结果 pregroups[]

输出：分得的组数 nGroups，分组结果 groups[level]

局部变量：标记元素是否被划分的数组 chosen[]，每组的元素数 elpergroup，nel，分组余量 leftover，一个新的组的结果 newgroup，i，j

```
1   begin
2       if nElements > nobject then
3           nGroups = nobject;
4       else
5           nGroups = nElements;
6       elpergroup = nElements / nGroups;
7       leftover = nElements % nGroups;
8       for i = 0 ; i < nElements ; i = i + 1 do
9           chosen[i] = 0;
10      j = 0;
11      for i = 0 ; i < nElements ; i = i + nel do
12          nel = elpergroup;
13          if leftover > 0 then
14              nel = nel + 1;
15              leftover = leftover – 1;
16          newgroup = GenerateGroup(CommMatrix, nElements, nel, chosen,
                        pregroups);
17          newgroup.elements = nel;
18          newgroup.id = j;
19          groups[level][j] = newgroup;
20          j = j + 1;
21      return [ nGroups, groups[level] ];
22  end
```

算法的具体执行流程描述如下：

1）类似算法一的执行流程，算法首先定义及初始化必要的变量。根据映射计算的实质，一般情况下组的数量 nGroups 应是可匹配的硬件对象的数量；但是如果待分组元素的数量比系统中可匹配的硬件对象数量还少，则组数直接等于待分组元素数，如算法第 2-5 行所示。随后根据 nElements 和 nGroups 计算每组分配的元素数目以及是否有余量，在本函数中分别用局部变量 elpergroup 和 leftover 表示。此外算法定义一个标志元素是否被分配的整型或布尔型（都占用4个字节，所以没有区别）数组变量 chosen[]，把数组中的所有项初始化为 0.

2）循环进行分组，每循环一次，会得到一个完整的组。循环中首先处理有 leftover

的情况，使得前若干次分组均多分得一个元素，直至消除掉余量，如算法第 12-15 行所示。然后调用函数 GenerateGroup 进行分一个组的计算。一个组的结果用结构体变量 newgroup 表示，包含组的序号 id，组内元素总数 elements 和一个数组 element[]. 由此知道表示分组结果的数组 groups 是一个二维数组，而每一个元素 groups[i]都是一个结构体数组。每分好一个组，将得到的 newgroup 存入 groups[level]中。循环的结束条件是所有待分配元素都已被分到某个组内。算法返回当前层组的总数和分组结果构成的二元组。

4.1.3 生成一个组

子函数 GenerateGroup()在算法中用于执行分每一个组时的操作，其执行流程如图 4-3 所示。本算法主要采用贪心策略对待分组元素进行比较选择。算法输入执行本层分组需要的通信量矩阵 CommMatrix[][]，本层分组中待分组元素总数 nElements，一个组内应容纳元素数 nel，标记元素是否已被访问到的数组 chosen[]，和前一层的分组结果数组 pregroups[]. 输出的是一个组的分组结果，用结构体 newggroup {int id; int elements; int element[];}表示。

算法三 GenerateGroup

输入：通信量矩阵 CommMatrix[][]，同一层中待分组元素总数 nElements，一个组要分的元素数 nel，标记元素是否被划分的数组 chosen[]，上一层分组结果 pregroups[]

输出：一个组 newgroup

局部变量：标记 i，j，m，元素 j 与组内已有所有元素的通信量W_j，当前与组内已有元素通信量的最大值W_{max}

```
1  begin
2      for i = 0 ; i < nel ; i = i + 1 do
3          W_max = -1;
4          for j = 0 ; j < nElements ; j = j + 1 do
5              if chosen[j] = 0 then
6                  W_j = 0;
7                  for k = 0 ; k < i ; k = k + 1 do
8                      W_j = W_j + CommMatrix[j][newgroup.element[k]];
9                  if W_j > W_max then
10                     W_max = W_j;
11                     m = j;
12         chosen[m] = 1;
13         newgroup.element[i] = pregroups[m];
14     return newgroup;
15 end
```

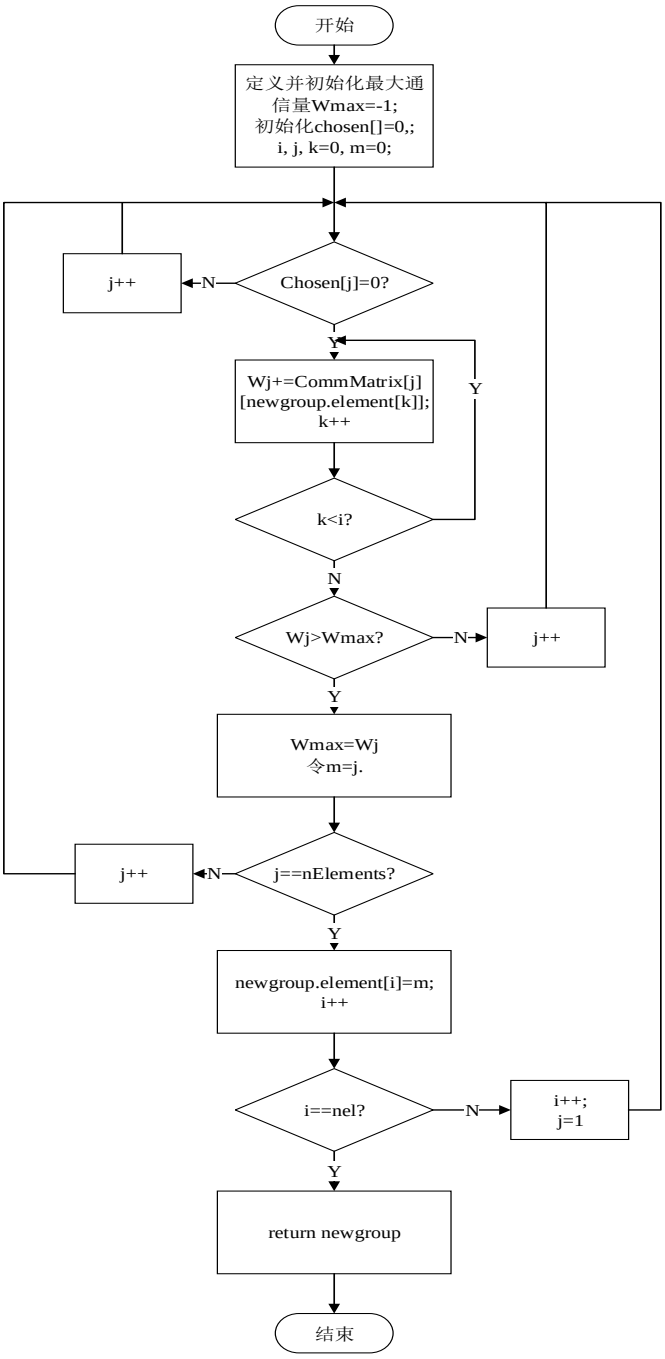

图 4-3 LHMapping 算法子函数 GenerateGroup 流程图

下面是 GenerateGroup 算法的流程描述：

1）循环对即将分成的组的每个元素进行挑选，每次循环的结果是组内增添了一个元素（算法第 2-13 行的循环）。在每次循环时，算法首先定义一个局部变量W_{max}，表示各元素之间通信量的最大值，把它初始化为 0 或-1. 这一层循环的退出条件是组已分满。

2）在 A 中的循环内部，循环遍历本层中所有元素（算法第 4-11 行的循环），选择最合适的一个元素加入组中。也就是说每挑选一个元素，就遍历一次所有元素。挑选时采用贪心策略，组内第一个元素是选择第一个没有被分配的元素直接进组，其他元素

通过计算选择与组内已有元素的通信量之和最大的元素进组。因此在遍历元素时，每评价一个元素 j，需要计算元素 j 与组内已有元素的通信量之和的值W_j，若W_j大于W_{max}，则令$W_{max} = W_j$，同时把这个当前具有W_{max}的 j 记录起来，即令$m = j$. 遍历一次之后，总有一个 m 具有最大的W_m，因此本轮选择这个 m 元素进入组中。已在组内的元素存入 newgroup.element[i]中。由于本层待分组的所有元素的序号是按照上一层分组结果 pregroups 中的序号排列的，所以这里 m 就是 pregroups[m]表示的线程或线程组。内层循环的退出条件是遍历到最后一个元素。

3）外循环退出，GenerateGroup 结束，返回分组结果 newgroup.

4.1.4 更新矩阵

该算法是顶层算法 MappingAlgorithm 调用的子函数，它执行某一非最顶层的分组完成后通信矩阵的更新操作。矩阵由表示线程与线程或较小线程组之间的通信量，转化成为更大的线程组之间的通信量。

算法四 UpdateMatrix

输入：原始通信量矩阵 CommMatrix[][]，分组结果 groups[level]，组的总数 nGroups

输出：新的通信量矩阵 newCommMatrix[][]

局部变量：标记 i，j，k，t，通信量大小 W

```
1  begin
2      for i = 0 ; i < nGroups - 1 ; i = i + 1 do
3          for j = i + 1 ; j < nGroups ; j = j + 1 do
4              W = 0;
5              for k = 0 ; k < groups[level][i] ; k = k + 1 do
6                  for t = 0 ; t < groups[level][j] ; t = t + 1 do
                       W = W +
7                      CommMatrix[groups[level][i].element[k].id]
                       [groups[level][j].element[t].id];
8              newCommMatrix[i][j] = W;
9              newCommMatrix[j][i] = W;
10     return newCommMatrix;
11 end
```

算法采用简单的代数加和的方法进行通信量矩阵的更新。例如在本层分组中，算法将 0, 1 号线程分成了一组，将 2, 5 号线程分至另外一组，在下一层计算时他们的组 id 分别为 0 和 1. 则原来线程 0 与线程 2 和 5 之间的通信量以及线程 1 与线程 2 和 5 之间的通信量转化为这几个通信量相加和的结果，即组 0 与组 1 之间的通信量。设计在代码中使用两层循环完成此运算。该算法的伪代码如上所示。

4.1.5 将分组映射至拓扑树

该算法也是顶层算法 MappingAlgorithm 调用的子函数，它执行分组结束后，将分组的结果对应到系统中具体的硬件对象上的操作。

算法采用递归的思想。类似于树结构的相关操作，从根节点出发向下遍历各硬件对象。如果遇到 CPU 核心这一层的对象，则递归到达终点。定义一个 map 数据结构，存放本层的所有被分组单元（即线程）与硬件对象（即 CPU 核心）的映射关系。如果遇到中间层（其他可共享的硬件对象），则对本层的所有硬件对象，再次调用 GroupsToTopology. 这个过程类似于以当前层的节点为根节点，再次遍历当前层节点及其的子节点共同构成的子树。此外，若硬件对象既不是 CPU 核心，也不是可共享的对象，则不考虑将线程或线程组映射到这些映射对象上。所以具体操作是调用 GroupsToTopology 将这个对象和其子节点代表的对象在 map 中置为 0. 该算法的伪代码如下所示。

算法五 GroupsToTopology

输入：硬件对象拓扑树的根节点 HardwareRoot，分组树的根节点 RootGroup，映射计算的最终结果 map

局部变量：标记 i

```
1  begin
2      if HardwareRoot.type = CPUCore then
3          for i = 0 ; i < rootGroup.nElements ; i = i + 1 do
4              map[rootGroup.element[i].id] = HardwareRoot.id;
5      else if HardwareRoot.shares > 1 then
6          for i = 0 ; i < rootGroup.nElements ; i = i + 1 do
7              GroupsToTopology(HardwareRoot.linked[i], RootGroup.element[i],
               map);
8      else
9          GroupsToTopology(HardwareRoot.linked[0], RootGroup, map);
10 end
```

4.2 LHMapping 算法的复杂度分析

4.2.1 时间复杂度分析

根据 LHMapping 算法的基本结构，算法的时间消耗主要体现在三个方面：分组的计算、矩阵的更新和拓扑的映射。

首先分析分组计算部分的时间复杂度。LHMapping 算法采用分层分组的方式，顶层计算调用 GenerateOneLevel 分一层的组，每层分组不断调用 GenerateGroup 函数产生一个分组。在评估算法的复杂度时，假设 E 表示某一层待分组元素的总数，K 表示每一组内需要分得的元素总数，n 表示映射时待分组的线程数。函数 GenerateGroup 中每

添加一个元素进入组内，需要遍历其他所有的元素并计算所有组内元素与当前搜素到的元素间的通信量。则对函数 GenerateGroup 的时间复杂度的表示如公式（4-1）所示。

$$\sum_{i=1}^{K}\sum_{j=1}^{E} i = E \cdot \sum_{i=1}^{K} i = E \cdot \frac{K(K+1)}{2} \leq E \cdot K^2 \tag{4-1}$$

因此函数 GenerateOneLevel 的时间复杂度如公式（4-2）所示。

$$\sum_{i=1}^{E/K} GenerateGroup \leq \sum_{i=1}^{E/K} E \cdot K^2 \leq \frac{E}{K} \cdot E \cdot K^2 \leq E^3 \tag{4-2}$$

采用双循环累加策略的更新矩阵 UpdateMatrix 函数的复杂度是$O(E^2)$. 假设 L 表示分组的层数，则由算法一 MappingAlgorithm 的过程描述，对算法分组计算和矩阵更新部分的时间复杂度表示如公式（4-3）所示。

$$\begin{aligned}&\sum_{i=1}^{L} GenerateOneLevel + \sum_{i=1}^{L-1} UpdateMatrix \\ &\leq \sum_{i=1}^{L} (GenerateOneLevel + UpdateMatrix) \\ &= \sum_{i=1}^{L} (O(E_i^3) + O(E_i^2))\end{aligned} \tag{4-3}$$

又因为在第一层分组时，E_1的值就是线程总数 n；其他分组层次中，$E_i \leq \frac{E_{i-1}}{2}$. 所以分组计算时间复杂度最坏的情况就是每一级硬件对象都只能共享两个上一级的单元，此时可以将分组拓扑树看作二叉树。故公式（4-3）可以改写为公式（4-4）所示。

$$\begin{aligned}\sum_{i=1}^{L} \left(O(E_i^3) + O(E_i^2)\right) &= \sum_{i=1}^{L} \left(O\left(\frac{n^3}{2^{i-1}}\right) + O\left(\frac{n^2}{2^{i-1}}\right)\right) \\ &< 2\left(O(n^3) + O(n^2)\right)\end{aligned} \tag{4-4}$$

完成分组结果到硬件对象的映射函数 GroupToTopology 实际上采用的是图的深度优先遍历策略，故复杂度可以表示为$O(V+E)$，这里 V 表示图的顶点，E 表示边。由于硬件对象的拓扑结构是一个树，所以边$E = V - 1$，复杂度为$O(V)$. 将分组结果用拓扑树表示后，最底层叶子节点的数量是 n. 由于可共享硬件对象在分组过程中可容纳的元素数一定大于 1，所以分组拓扑树中其他层节点总数一定小于等于 n，故$n \geq V/2$，n 与 V 处于同一数量级。所以函数 GroupToTopology 的复杂度最终可表示为$O(n)$.

综上所述，LHMapping 算法的时间复杂度如公式（4-5）所示。

$$\begin{aligned}&\sum_{i=1}^{L} GenerateOneLevel + \sum_{i=1}^{L-1} UpdateMatrix + GroupToTopology \\ &\leq 2\left(O(n^3) + O(n^2)\right) + O(n) = O(n^3)\end{aligned} \tag{4-5}$$

4.2.2 空间复杂度分析

根据第 4.2 小节算法的描述，保存分组结果的数组 groups 是一个二维数组，第一维是分组的层级，总数为 levels；第二维是每层中的分组个数，其元素总数可表示为

$\sum_{i=1}^{levels} nGroups_i$. 而第二维中的数组元素是一个结构体，包括两个整型变量和一个标识 id 对应关系的数组 element[]. 由于参与映射的硬件对象都是可共享对象，因此对空间占用最大的情况是：除最上层外，每一层的分组都只有两个元素，而最上层一定只有一个组。这样所有分组中 element[]数组占用空间的总大小如公式（4-6）所示。

$$2\sum_{i=1}^{levels-1}\frac{n}{2^i}=\sum_{i=0}^{levels-2}\frac{n}{2^i}\leq 2n \tag{4-6}$$

其中 n 是参与映射的线程总数。故表示分组结果的 groups 的空间复杂度为$O(n)$. 同理表示上一层分组情况的数组 pregroups 的空间复杂度为$O(n)$，故表示分组情况的所有变量的总空间复杂度为$O(n)$.

矩阵更新操作中，由初始的通信量矩阵，中间生成了若干行列数均为线程组的数量的矩阵。最差情况下除最上层外每层的分组只包含两个元素，故矩阵占用的空间如公式（4-7）所示。

$$\sum_{i=0}^{levels-1}O((\frac{n}{2^i})^2)=O(n^2) \tag{4-7}$$

综上所述，LHMapping 算法的空间复杂度为$O(n^2)+O(n)=O(n^2)$.

4.3 本章小结

本章首先在动态映射的条件下对早期使用的 Scotch 映射分组算法提出了额外开销过大的问题。通过设计直接利用统计出的通信量矩阵进行分组，而非将矩阵转化为图的数据结构再进行分组，避免了矩阵到图之间转换的步骤；通过设计基于循环和遍历的分组方法，而非双向递归的分组方法，简化了计算的复杂度；并通过设计基于硬件平台硬件对象的分层分组方法，保证了分组精度，以及明确了线程和线程组与硬件对象的一一对应关系，为映射的执行提供了便利。最后本章介绍的 LHMapping 映射分组算法进行了复杂度分析，从理论上得到算法开销随线程数增多的变化规律。

5 动态映射机制的性能测试与分析

本章将在两个不同的多核处理器平台上，评估线程动态映射机制整体对程序的优化效果。此外，本章分别从 LHMapping 算法本身的性能，和使用该算法后映射机制整体优化效果这两个维度对比评估所设计的映射分组算法。实验的具体细节如下。

5.1 实验环境

5.1.1 实验平台

本文分别在两个平台上运行程序进行实验：由 Intel Xeon E7-8850 处理器构成的 80 核并行计算机（以下简称 C80）和由 8 个 Intel Core i7-6700 处理器核心构成的 PC 机搭建的虚拟机环境（以下简称 PC）。两种实验平台硬件的详细参数如表 5-1 所示。

表 5-1 两种实验平台的详细参数

平台名称	属性	详细数据
C80	Architecture	2 nodes, 4 sockets/node, 10processors/socket
	Processor	Intel Xeon E7-8850, 2.0GHz
	Cache	10*(256KB+256KB) L1, 10*8MB L2, 16MB L3
	Memory	128GB DDR3, 页大小 4KB
PC	Architecture	2nodes, 4processors/node
	Processor	Intel Core i7-6700, 3.4GHz
	Cache	4*(16KB+16KB) L1, 4*256KB L2, 4MB L3
	Memory	8GB DDR4, 页大小 4KB

C80 机器是一个典型的 NUMA 架构平台，它包含两个 NUMA 节点，每个系节点包含 4 个处理器，每个处理器拥有 10 个完全相同的 Intel Xeon E7-8850 处理器核心。每个核心上分别有 256KB 大小的私有的 L1 指令和数据 Cache，8MB 大小的私有 L2 Cache；10 个 CPU 核心所在的处理器包含一个大小为 16MB 的 L3 Cache. 该集群的总内存大小为 128GB，操作系统内存管理采用分页，页大小为 4KB. 处理器中不使用超线程技术。

对含有 8 个处理器核心的 PC 机搭建虚拟机环境，使其包含 2 个处理器，每个处理器有 4 个核心,每个核心分别含有一个大小为 16KB 私有 L1 指令和数据 Cache，2 个核心共享同一个大小为 256KB 的 L2 Cache，4 个核心共享一个大小为 4MB 的 L3 Cache. 整机的内存大小是 8GB，操作系统内存管理采用分页，页大小也为 4KB. 处理器中不使用超线程技术。

在 PC 机中，至多只能运行 8 个线程的多线程并行应用程序；而 C80 机器至多能运行 80 线程的多线程并行应用程序。两个平台分别安装了 Linux 的 CentOS 和 Ubuntu 操作系统，统一内核后他们的内核版本均为 3.10.0.

5.1.2 应用程序

分别使用两套多线程并行应用程序评估映射优化机制的效果。

1）rotor35-omp 程序

本文使用“面向 E 级计算机的大型流体机械并行软件系统及示范”项目中的 rotor35 压缩机转子方腔流动模型算例程序（以下简称 rotor 程序），对本文提出的映射机制在该项目程序的优化效果进行测试。rotor 程序针对 36 个流道的轴流压气机转子模型，依次执行 CFD 工程应用的三个模块：前处理、数值求解和后处理[35]。数值求解过程对流体模型离散点之间的场物理量构建如公式（5-1）所示的控制方程组。

$$\frac{\partial \boldsymbol{U}}{\partial t}+\frac{\partial \boldsymbol{F}}{\partial x}+\frac{\partial \boldsymbol{G}}{\partial y}+\frac{\partial \boldsymbol{H}}{\partial Z}=\boldsymbol{I}+\frac{\partial \boldsymbol{F_v}}{\partial x}+\frac{\partial \boldsymbol{G_v}}{\partial y}+\frac{\partial \boldsymbol{H_v}}{\partial z} \qquad (5\text{-}1)$$

其中，*U* 为守恒型求解向量；*F*、*G*、*H* 别为沿着 *x*、*y*、*z* 三个坐标方向的对流向量；F、G、H 分别为沿着三个坐标方向 x、y、z 的粘性向量；I 为源项。算例对公式中求解向量 U 的 6 个物理量进行求解。使用 FMG（Full Multi-Grid）三重网格循环，依据多重网格法[36]的标准在粗网格、中网格和细网格之间进行残差限制和插值，其处理流程示意如图所示。直至程序中残差收敛或达到最大的迭代步数，程序数值求解部分结束运行。

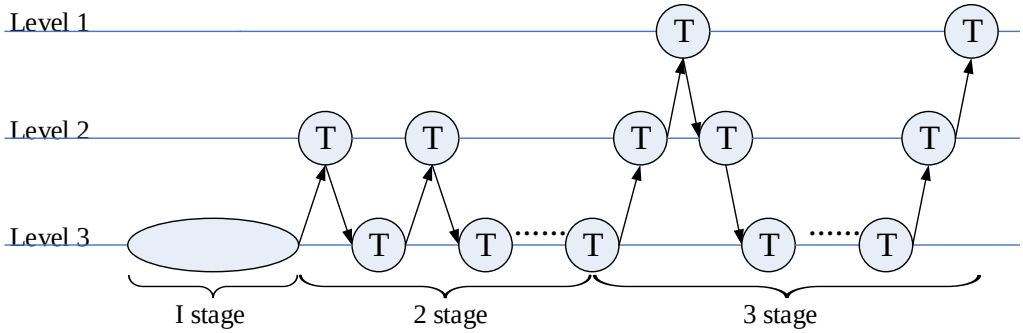

图 5-1rotor35 程序多重网格法执行示意图

2）NPB-OMP 程序集

进一步地，使用 NPB-OMP 的多线程程序集。它是 NAS 并行基准测试程序[37]（NAS Parallel Benchmark，NPB）的 OpenMP 实现[38]。NPB 是一套以流体动力学计算为主的应用程序集。目前 NPB 程序集已经是一套成熟的、完整的评测大规模并行机和超级计算机性能的标准测试程序。标准版本的 NPB 由 8 个程序组成，包括 5 个核心程序和 3 个模拟程序，他们分别是 IS、EP、FT、MG、CG、LU、SP 和 BT. 每个基准测试程序都分为 A, B, C 和 S, W，代表五个不同种类的问题规模。其中 A 类的规模最小，C 类的规模最大，W（Workstation）类通常用于工作站，S（Sample）类是样例程序。8 个程序的基本功能和运行时特点描述如下。

五个核心程序分别是：

1）IS（Integer sort）用于求解基于桶排序的二维大整数排序，程序中有大量全交换通信和大量随机性的访存操作，因此通信模式相对比较复杂，规律性不强。

2）EP（Embarrassingly parallel），用于计算 Gauss 伪随机数，主要执行浮点数计算。是计算量庞大、并行度极高的并行应用程序。EP 程序在运行时不执行任何进程间或线程间通信。

3）CG（Conjugate Gradient）程序近似地求解大型稀疏对称正定矩阵的最小特征值。CG 程序中使用了不规则的长距离通信，结合稀疏矩阵向量乘法，因此属于非结构化网格的计算。

4）MG（MultiGrid）是代表多重网格计算的最典型的基准测试程序，它的通信模式兼具短距离和长距离通信，属于访存密集型程序。

5）快速傅里叶变换（FT），用快速傅里叶变换方法求解三维偏微分方程。FT 也属于计算任务较为密集的程序。

三个流体力学模拟程序分别为：

1）LU（Lower upper triangular）使用对称超松弛法求解块稀疏方程组。LU 程序中大多数 MPI 通信是阻塞模式通信，因为程序并行度不高，仍然待优化。其性能主要受节点间网络互连的影响。

2）SP（scalar penta-diagonal）和 BT（Block Tri-Diagonal）分别用于求解 5 对角线方程组和 3 对角线方程组。它们倾向于检测计算与通信之间的平衡。SP 比 BT 有更高的通信强度。

实验评估时选择 NPB-OMP 作为待测应用程序，不仅能利用程序集属于大规模流体计算程序的特点，而且有助于分析与总结映射机制对不同特点应用程序的优化效果，体现映射优化机制的通用性。

5.2 LHMapping 算法性能测试

本节将对比评估本文提出的 LHMapping 算法本身的性能。

本文使用算法运行时间和映射的精确度两项指标评估 LHMapping 算法本身的性能。并将所设计的 LHMapping 算法与 Scotch Mapping Library 提供的映射分组算法进行对比。实验时首先运行应用程序，同时执行线程间通信量统计，并借助 Hwloc 工具分别得到两个算法需要的输入数据：程序的线程间通信量矩阵，通信量图和体系结构中硬件对象数量的层次信息等。随后单独运行这两种分组算法，统计他们执行分组计算的时间和分组的结果。

实验时选择 C80 实验平台，用 NPB 程序集中的 SP-OMP 程序完成实验，比较两种算法分别在用 4 线程、8 线程、16 线程以及 64 线程运行 SP 程序时的分组计算时间。其结果如图 5-1 所示。

由图 5-1 的实验结果，在进行实验的全部线程数规格的程序中，LHMapping 算法

的运行时间都比 Scotch 快 2-4 倍。这说明本文提出的 LHMapping 算法具有更快的分组计算效率。且根据图中的曲线，在线程数不多的情况下，两种算法的运行时间都是随着线程数的增加呈现超过线性增长、接近指数型增长的趋势。这与本文第 4.2 小节分析算法时间复杂度给出的结论一致。

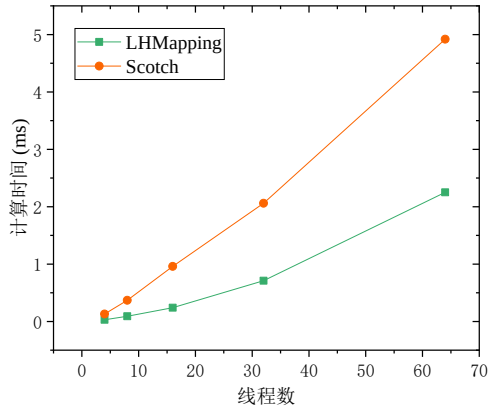

图 5-2 LHMapping 算法和 Scotch 算法的计算时间对比结果

此外，为了评价算法的计算精度，从理论上说明 LHMapping 映射分组算法的有效性，在设计实验时引入映射精确度的指标，称为 *MappingAccuracy*. 它是 TreeMatch[39] 中提出的一种评价映射精度的指标。其公式表示如公式（5-2）所示。

$$MappingAccuracy = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n}\frac{Comm[i][j]}{latency[map[i]][map[j]]} \qquad (5\text{-}2)$$

其中，n 是线程的数量，$Comm\ [i][j]$表示线程 i 和线程 j 之间的通信量值，而 $latency[map[i]][map[j]]$表示线程 i 映射到的核心与线程 j 映射到的核心之间的数据传输时延。$latency[map[i]][map[j]]$的取值大致可分为三种情况：

1）若线程 i 和线程 j 处于共享同一个 L2 Cache 的核心中，则该值代表它们访问该共享 Cache 的时延。

2）若线程 i 和线程 j 在同一个处理器上，但只能共享访问 L3 Cache 和处理器本地的 Local Memory，则该值代表他们访问这两个硬件对象的时延。

3）若线程 i 和线程 j 分布于同一个计算节点内不同的处理器上，则该值代表线程 i 访问线程 j 所在处理器 Local Memory 的时延。

latency 的值可以通过 Imbench[40]中提供的程序计算得到。

理论上，映射精确度的值越高，映射的效果越好。实验中选择对 64 线程 SP-OMP 程序，根据两种映射算法的分组结果，分别计算根据两种算法的分组结果执行映射后的映射精确度 *MappingAccuracy*. 结果如表 5-2 所示。

由表中数据可知，LHMapping 算法分组的映射精确度略低于 Scotch Mapping Library 中的算法。但二者处于同一数量级，相差不大，因此对映射决策没有实质性的影响，且若将 Scotch 算法引入本研究提出并实现的动态映射机制中，不但没有更好的优化效果，反而造成过多的额外开销。经过对比实验，LHMapping 算法整体上具有比

Scotch 算法更好的性能，且更适合于本研究设计实现的动态映射机制。

表 5-2 分组算法的映射精确度对比结果

分组算法	MappingAccuracy
LHMapping	1.21×10^7
Scotch	1.44×10^7

5.3 动态映射优化机制的整体优化效果测试

本节将对比测试本文提出的=动态线程到核映射优化机制的整体优化效果。本文分别使用 rotor 程序和 NAS Parallel Benchmark 程序集的 OpenMP 程序作为评估对象，根据本文第 2.3.2 小节对映射机制理论优化效果的介绍，选择程序运行时间和 Cache misses 这两项指标的性能增益，对整体优化效果作出评价，并给出结论。

5.3.1 映射机制对 rotor35-omp 程序的优化性能评估

在本节将对本文提出的映射优化机制应用于 rotor35-omp 程序的优化效果进行测试，验证映射机制的优化效果。在实验时选择将程序总运行时间作为评测指标，分别在映射机制的内核模块加载到系统中的前后运行应用程序，观测其性能增益。

由于本研究最终将映射机制实现为 Linux 操作系统的内核模块，因此实验时选择在向系统中加载该内核模块的前后分别重复多次运行多线程并行 rotor 程序，分别实验得到加载内核模块前后运行相同程序的平均单次运行时间数值，在计算执行映射后的运行时间比执行映射前程序运行时间减少的百分比（可以是负值）。由于 C80 平台具有每 10 个 CPU 核心分布于同一个处理器 socket 上的特点，即 10 个核心共享一个 L3 Cache 和处理器本地局部存储器，因此本实验分别选择少于 10 线程数和多于 10 线程数的设置参数进行对比。且 rotor 程序是计算 36 个流道的物理量，通过每个流道之间的通信传递并迭代残差值，这要求对网格进行更细粒度的划分前，多线程并行程序中每个线程至少处理一个流道的计算，因此本实验选择几种 36 的约数作为线程数配置。最终本实验选择在 C80 机器上分别运行 6 线程、12 线程、24 线程和 36 线程版本的 rotor 纯 OpenMP 并行程序。实验结果如图 5-2 所示。

根据图 5-2 中的结果显示，在 C80 平台上，映射优化机制为 12、24、36 线程的 rotor 程序均带来运行时间的明显减少，具有一定优化效果；而对 6 线程 rotor 程序无实际优化效果。原因是 6 线程 rotor 程序其线程在 C80 平台上均分布在同一个 socket 上，由于 10 个核心共享同一个 L3 Cache，因此不存在数据访问效率的不平衡性，故映射机制不能针对这一情况进行优化。然而由图 (b)，映射机制的优化效果随着线程数的增加呈线性增长的趋势，原因是 6、12、24、36 线程的 rotor 程序在 C80 平台上运行时分别占用 1、2、3、4 个 socket，其占用的 socket 越多，线程间通信的不平衡性越大，映射优化机制的优化效果也越好。

实验证明，映射机制对于标准的 36 线程 rotor 程序在 C80 平台上是有效的。

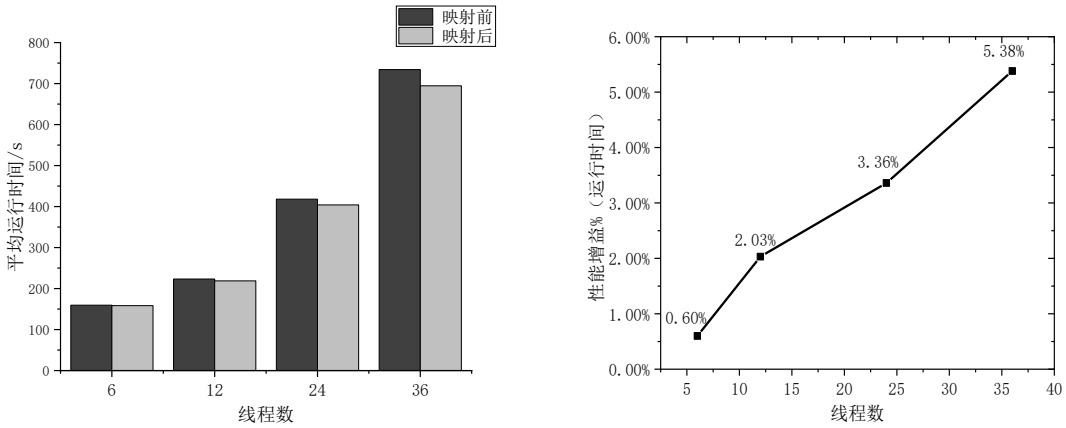

图 5-3 不同线程数 rotor 程序执行映射优化前后运行时间的性能增益对比

5.3.2 LHMapping 算法应用于映射机制的性能评估

其次，分别将本文提出的 LHMapping 算法和 Scotch 映射算法的代码实现到本研究总体设计的映射优化机制中，生成两个具有不同映射分组算法的映射优化机制，且均实现为 Linux 系统的内核模块。依次加载这两种内核模块，对比执行映射优化后与未执行映射优化的程序运行时间和Cache Misses这两项指标的性能增益。选择LHMapping算法、Scotch 算法与 OpenMP 提供的 Compact 映射模式作为横向对比。实验时分别选择运行 8 线程问题规模为 A 类别和 64 线程问题规模为 B 类别的 NPB-OMP 测试程序集，分别在 PC 平台和 C80 平台上进行实验。其结果如图 5-3 和图 5-5 映射优化机制在 PC 和 C80 上对缓存失效的优化效果对比所示。

根据图 5-3 和图 5-5 映射优化机制在 PC 和 C80 上对缓存失效的优化效果对比所示的实验结果，整体上看，使用 LHMapping 算法的映射优化机制对 SP, BT, CG, LU, MG 等程序具有明显的优化效果，对程序运行时间的加速最高达到约 25.4%，Cache misses 的最高减少约 23.5%。

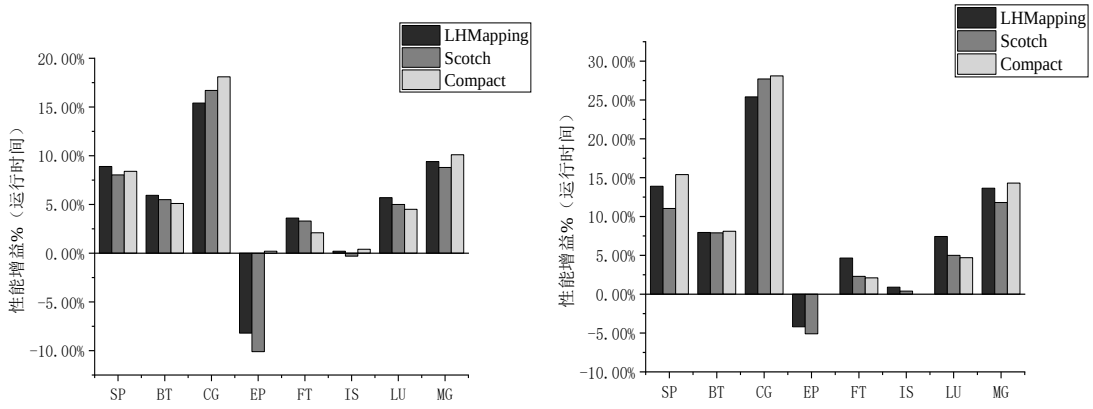

图 5-4 映射优化机制在 PC 和 C80 上对程序运行时间的优化效果对比

此外，通过对比，使用 LHMapping 算法的映射优化机制对于通信量极大、访存密集型的程序 CG 等，优化效果不如使用 Scotch Mapping Library 提供的映射分组算法。

这体现出使用 LHMapping 算法的映射机制对于完全访存密集型程序虽然具有不错的优化效果，但计算精度的细微缺失导致执行映射后程序的访存效率提高和 Cache 失效率减少并不能完美抵消映射机制带来的额外开销。而对于通信量不太大，即通信与计算任务占比相对平衡的 FT，LU，MG 等程序，LHMapping 的优化效果更好。

映射机制对完全没有处理器间通信的纯计算密集型程序 EP 没有优化任何效果，反而由于引入了额外开销使程序性能变差。因此针对计算负载平衡的映射优化方案，而非本研究采用的映射方法，对此类程序会更有效果。对于全交换通信程序 IS，即每个线程之间的通信量差不多，映射机制几乎没有优化效果。事实上，对此类应用程序针对通信不平衡问题上进行映射优化，无论采取什么方法，均不能达到很好的优化效果。

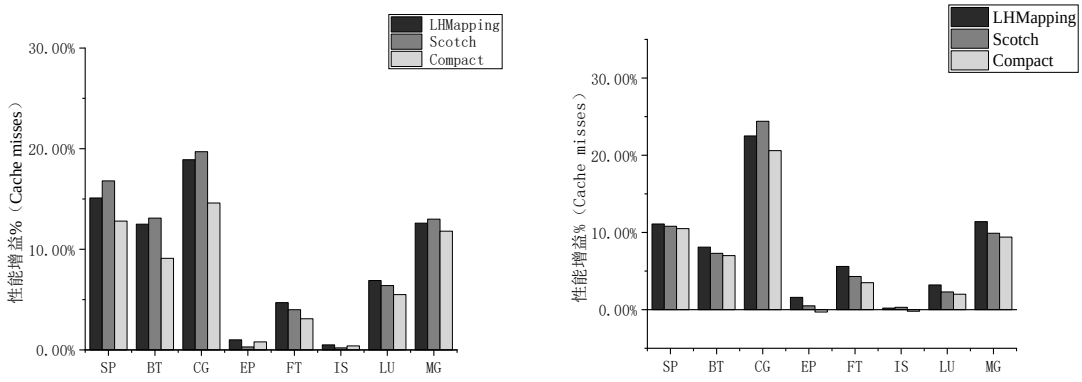

图 5-5 映射优化机制在 PC 和 C80 上对缓存失效的优化效果对比

5.4 其他影响变量的测试

根据文章第 3.4 小节的介绍，本文提出的映射机制引入削减系数α和周期性时间间隔$interval$这两个映射机制的控制变量。本节将评估不同的α和$interval$取值下映射机制的优化效果。

5.4.1 削减系数α

首先对削减系数α的取值进行实验测试，本文分别取α的值为 0，0.25，0.5，0.75 和 1，固定$interval = 200ms$. 分别测试在这几种α取值下，使用了 LHMapping 算法的动态线程到核映射优化机制对 NPB-OMP 程序集中 SP 程序的运行时间的影响。在 C80 平台上进行实验，使用 64 线程 SP-OMP 程序。具体的实验结果如图 5-6 削减系数α对映射机制优化效果的影响所示。

根据图 5-6 削减系数α对映射机制优化效果的影响中显示的结果，当$\alpha = 0$ 或 0.25时，线程间通信量的统计值只取决于最近 interval 时间内的线程间通信情况。由于某一段时间的线程间通信模式不能充分说明未来一段时间内线程间通信情况与原来一致，因此依靠这一段时间内的通信情况进行映射决策并不理想。所以当α取值很低时，映射机制优化效果不佳，与实验数据相吻合。

当$\alpha = 1$时，线程间通信量的统计值完全取决于多线程并行执行以来所有时间内的线程间通信情况，但此时应对程序通信量迅速变化的能力比α值更小时略差，这与实验数据相吻合。

当$\alpha = 0.75$时，动态映射机制对 SP-OMP 程序的优化效果最好。当$\alpha = 0.5$时，动态映射机制的优化效果也很明显，实际上与$\alpha = 0.75$的影响差不多。由于$\alpha = 0.75$时计算更为简便，因此本研究的映射机制中选择$\alpha = 0.75$为最佳。

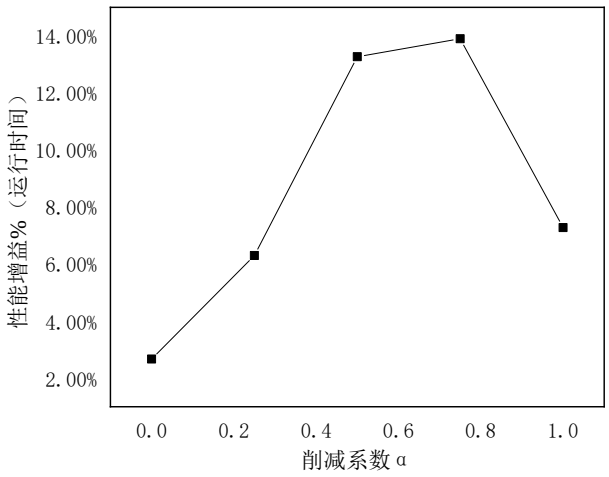

图 5-6 削减系数α对映射机制优化效果的影响

5.4.2 时间间隔 interval

对时间间隔 *interval*，分别取初始的 *interval* 的值为 100*ms*、200*ms*、300*ms*、400*ms* 和 500*ms*. 固定削减系数$\alpha = 0.75$.

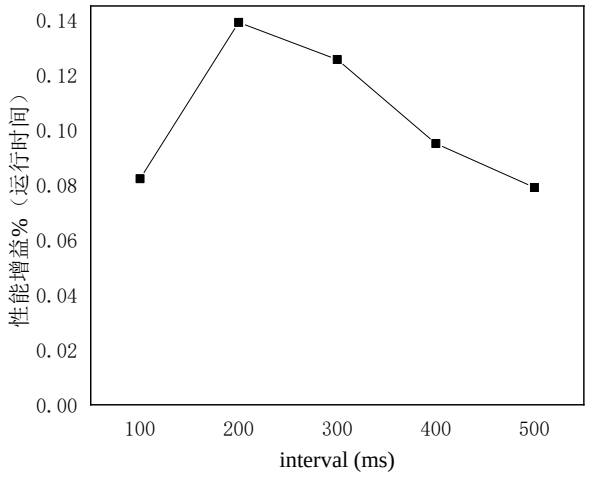

图 5-7 时间间隔 *interval* 对映射机制优化效果的影响

实验时仍然保持原始的设定，即每次分组计算结果与之前一次计算变化不大，*interval* 增加 50*ms*；否则，*interval* 减少 50*ms*；且 *interval* 始终在 50*ms*-1*s* 之间变化。比

较这几种 *interval* 取值下，使用 LHMapping 算法的映射优化机制对 NPB-OMP 程序集中 SP 程序的运行时间的影响。在 C80 平台上进行实验，使用 64 线程 SP-OMP 程序。具体的实验结果如图 5-7 时间间隔 interval 对映射机制优化效果的影响所示。

根据图 5-7 时间间隔 interval 对映射机制优化效果的影响中显示的结果，当 *interval*=200*ms* 时，映射机制对 SP-OMP 程序具有最佳的优化效果。由于映射机制中已经加入对 *interval* 随映射分组计算的结果变化的设定，因此映射机制保证了在动态执行映射的过程中，*interval* 的取值会逐渐趋向于最优。所以实验中 *interval* 的这几种取值下程序运行时间减少量的差距并不大。根据实验数据，本研究的动态线程到核映射机制最终选择 *interval*=200*ms* 为最佳。

5.5 本章小结

在 C80 集群和 PC 单机上的实验结果均表明，本文设计实现的动态线程到核映射机制对除了无处理器间通信程序（纯计算密集型程序）和全交换通信程序（随机内存访问程序）以外的 OpenMP 多线程并行应用程序均有不同程度的优化效果，其程序运行时间的性能增益最高可达 25.4%，程序 Cache 失效数的性能增益最高达到 23.5%. 本文提出并实现的 LHMapping 算法再多种线程数规格下均具有比传统 Scotch 算法快 2-4 倍的计算开销，使用了本文研究与实现的 LHMapping 算法的动态线程到核映射机制，在通用性上整体优于映射机制早期使用的 Scotch 映射分组算法和 OpenMP 本身提供的 Compact 映射模式。总体而言，在解决多线程程序共享内存通信模式下线程间通信量不平衡的问题时，本文设计实现的动态映射优化机制是非常优秀的解决方案。

6 结论与展望

6.1 结

本文以“十三五”国家重点研发计划课题为依托，结合课题流体机械多线程并行应用程序开发和弹性动态映射优化目标，在 OpenMP 多线程并行一级，为解决线程间数据访问与共享不平衡问题，设计并实现了融合线程间通信量统计、周期性动态线程分组计算、线程的 CPU 亲和度设置的线程到核的映射优化机制。并进一步结合线程间通信量和硬件平台的架构特征，设计了基于循环层次化分组的映射分组算法 LHMapping。映射优化机制整体上改善了线程间通信不平衡的问题，减少了由于数据分布不均等访存问题引起的 Cache 失效，并有效缩短了并行程序的运行时间。本文主要工作包括：

1）分析了多线程并行和 OpenMP 多线程编程模型的基本特性。详细介绍了多线程环境下，线程间通过共享内存通信的方式和特点，以及针对线程间通信问题的线程映射优化的问题背景、解决思路和理论优化效果。最后介绍了利用操作系统缺页中断机制进行线程间通信量检测的基本条件和原理。

2）针对并行应用程序通信模式变化大的特点以及传统线程映射优化方案通用性差的问题，设计并实现了一种动态的线程到核映射优化机制。详细介绍了映射机制包含的整体控制模块、通信量检测与统计模块、分组计算模块和映射执行模块的功能。详细描述了在多线程并行应用程序运行时，映射优化机制的动态执行流程。使用基于缺页错误的线程间通信量检测与统计方法，实现通信量检测模块，并通过更改页表项引入额外的缺页错误，提升通信量检测的精度。使用 Scotch Mapping Library 提供的经典的映射算法或者本文研究设计的简化的基于循环的 LHMapping 映射分组算法，实现分组计算模块，根据统计到的通信量矩阵和硬件平台的体系结构信息，得到线程与具体的 CPU 核心的一一对应关系，即映射结果，将其用 map 数据结构表示。使用 Linux 内核提供的 CPU 亲和度设置函数，为应用程序中每一个并行执行的线程设置对应的 CPU 亲和度，将他们绑定到 map 中对应的核心上，实现映射执行模块。使用周期性的动态方法完成分组计算模块和映射执行模块，并加入对通信量矩阵和周期性时间间隔进行控制的约束变量。最后利用 Linux 内核模块机制将所研究设计的映射优化机制实现为内核模块。

3）从减少映射分组计算带来的额外开销的角度出发，在保证分组精度的前提下，设计了一种基于循环层次化分组的映射分组算法 LHMapping。算法对硬件平台上的硬件对象构建拓扑树，以自底向上的方式将最初待分组的线程依次分为较小的线程组、以线程组为元素的线程组等等；在处理每一层分组时采用广度优先遍历的策略，依次计算未被分组元素和组内已有元素之间的通信量值，选择具有最高通信量数值的元素分配进组，直至当前层所有元素都被分配；最后将分组的拓扑数映射到硬件对象拓扑

树中。最后详细分析了 LHMapping 算法的时间复杂度和空间复杂度。

4）本文在 80 个 Intel Xeon 处理器核心构成的集群上和 8 核 Linux 系统单机上，对映射优化机制和 LHMapping 算法均进行了相关测试和评估。通过分别测试映射优化机制对 rotor35 轴流压气机转子 CFD 程序和 NPB-OMP 程序集的优化效果，表明本文设计实现的映射优化机制对大多数流体机械多线程并行应用程序是有效的；并且对 NPB 程序集中某些访存操作较多的程序，本文研究设计的动态线程到核映射机制具有比其他映射优化方案更好的优化效果。通过对比测试 LHMapping 算法和 Scotch Mapping Library 提供的映射分组算法，表明本文研究设计的 LHMapping 算法本身具有更快的计算效率和不亚于 Scotch 的映射精确度；并且使用了 LHMapping 算法的映射优化机制对某些应用程序具有更好的优化效果。

综上所述，本文设计的射优化机制和映射分组算法能有效解决多线程共享内存通信存在的不平衡问题，尽可能在程序运行过程中适应程序运行时特点的不断变化，且具有很强的通用性和可移植性，以及具有比其它映射优化方法更小的额外开销。本文的研究为应用程序充分匹配硬件平台计算资源提供了一种解决方向，也为进一步研究独立于应用程序的系统级映射优化甚至是异构平台下的映射优化奠定了基础。

6.2 下一步工作展望

本文针对线程间共享内存通信不平衡问题提出的线程到核的动态映射机制以及改进的映射分组算法，尽管对大多数多线程并行应用程序有良好的优化效果，但由于时间关系仍有一些问题有待研究和改进。后续工作将从以下几个方面展开：

1）本文提出的动态映射优化机制，针对的是 NUMA 节点上线程间通过共享内存方式进行通信，存在的通信不平衡问题。但由于并行应用程序在系统中实际运行时，不仅存在通信不平衡的问题，还可能存在负载不平衡，访存带宽等硬件资源竞争的问题。因此如何全面考虑解决上述多种维度的问题来进行动态性的映射优化，是一项比较困难但也非常具有意义的研究问题。

2）本文提出的动态映射优化机制中，由于在实现通信量检测模块时修改了系统的缺页中断处理，以及在映射执行模块调用了 Linux 系统内核中的 CPU 亲和性设置函数，故最终实现为 Linux 操作系统的内核模块。但是在这种情况下，受限于用户权限问题，无法在超大规模的超算平台上或 E 级原型机上进行测试，因此本文提出的映射机制在更大平台上是否仍然具有良好的优化效果还有待进一步测试。

3）结合上一条的描述，当程序规模扩大并运行在超算平台上，必须考虑结合 MPI 和多线程两级并行，甚至异构处理器的三级并行。因此如何使动态映射优化工作能够高可扩展成为了一个系统问题。在相同问题背景下亟需考虑进程基于共享内存通信时的映射机制设计以及它的通信量检测机制的设计。

4）结合本节上文所述以及本文第四章 LHMapping 算法的复杂度分析，当程序规模扩大并运行在国产神威太湖之光或天河三号上，程序高度并行所需的进程数以及进

程内的线程数增加，则映射分组算法的时间开销呈指数型增长。如何进一步优化映射分组算法，使之在进程、线程数增多的条件下，能有更少的计算时间开销以及差不多的计算精度，还有待进一步解决。

致　谢

光阴荏苒，转眼又是三年，我也即将完成我的研究生学业，过完令人难忘的研究生生活。三年的生活中，我经历了最初参与科研时感受到的的无助与挫败感，经历了努力参与科研项目讨论却难以产出阶段性成果时感到的迷茫，经历了代码调试到焦头烂额的艰苦，也经历了深夜加班加点完成系统和实验的繁忙。回首过往，用一个词语形容我觉得是平淡，也许这就是科研生活的本来面目。在此我也想写下这段文字，感谢一路走来不论是在学业中还是思想与心态上给予过我支持与帮助的人，是你们给了我直面困难的勇气和坚持到底的动力，谢谢你们！

首先，我要衷心感谢我的导师，张兴军教授。本论文是在张老师指导之下完成的。尽管张老师平时比较忙碌，却仍然以严谨认真的态度指导与关注着我，并时常与我探讨科研与生活中遇到的种种问题。是他在我反复质疑与思考如何理解文献中的方法并有效寻找创新时，告诉我科研时不能忘记以解决问题为出发点，思考研究内容如何有效解决什么实际问题；也是他在我处于迷茫期缺乏工作动力时告诫我研究生与科研工作者要具备学院修养，要时刻保持如临深渊、如履薄冰的心态，学会规划自己的时间，不断取得进步。同时也感谢在科研项目组中每周都能耐心听取学生们汇报与讨论，给予同学们很多指导的董小社老师、魏恒义老师、伍卫国老师和陈衡老师。我明白在我完成学业的过程中仍然存在很多问题，给老师们增添了不少的麻烦，在此也向老师们表达我深深的歉意与感谢！

其次，也要感谢在研究生期间提供给我很多帮助的同学们。首先感谢我的项目组组长李靖波师兄和何锋学长，是他们一直以来把握着项目的全局，不断地对我的研究和工作内容提出新颖且意义的问题，指导着我不断推进。其次要感谢我的师兄师姐雷鸣、于博成、雷雨、赵文强、安伟华和周剑锋、武旭瑞、付哲等同学，他们都是与我在同一个实验室或项目中的好朋友们，感谢他们不论是在科研工作、实验室中的生活还是人生方向的一些指引和帮助，让我感受到实验室像一家人一样的温暖。

最后，需要特别感谢我的父母，你们是我求学路上的坚实后盾。无论我作出什么决定，你们都能坚定地支持我；无论我遇到什么困难，你们也都不断鼓励并安抚着我，推动着我一路向前！

参考文献

[1] 赵兴艳,苏莫明,张楚华,苗永淼. CFD 方法在流体机械设计中的应用[J]．流体机械，2000(3)：22-25．

[2] Haohuan FU,Junfeng LIAO,Jinzhe YANG, et al. The Sunway Taihu Light supercomputer:system and applications[J]. Science China(Information Sciences),2016,59(07):113-128.

[3] You, X., Yang, H.,et al. Performance Evaluation and Analysis of Linear Algebra Kernels in the Prototype Tianhe-3 Cluster[C]//Asian Conference on Supercomputing Frontiers. Springer, Cham,2019: 86-105.

[4] Jain, Nilesh K., et al. Method, system, and device for dynamic energy efficient job scheduling in a cloud computing environment[P]. U.S. Patent No. 9,342,376. 2016-05-17.

[5] Tang, Zhuo, et al. An intermediate data placement algorithm for load balancing in Spark computing environment[J]. Future Generation Computer Systems. 2018: 287-301.

[6] Zhang J, Zhai J, Chen W, et al. Process mapping for mpi collective communications[C]//European Conference on Parallel Processing. Berlin, Heidelberg: Springer, 2009: 81-92.

[7] Klug, Tobias, et al. autopin–automated optimization of thread-to-core pinning on multicore systems[J]. Transactions on high-performance embedded architectures and compilers III. Springer, , 2011: 219-235.

[8] Radojkovic, Petar, et al. Thread assignment of multithreaded network applications in multicore/multithreaded processors[J]. IEEE Transactions on Parallel and Distributed Systems, 2012: 2513-2525.

[9] Wang Z , O'Boyle M F P . Mapping parallelism to multi-cores: a machine learning based approach[C]//Acm Sigplan Symposium on Principles & Practice of Parallel Programming. ACM, 2009.

[10] Weaver, Vincent M. Linux Perf_event Features and Overhead[C]//The 2nd International Workshop on Performance Analysis of Workload Optimized Systems, FastPath, 2013.

[11] Eyerman S, Eeckhout L, Karkhanis T, et al. A performance counter architecture for computing accurate CPI components[J]. ACM SIGOPS Operating Systems Review, 2006, 40(5): 175-184.

[12] Knauerhase R, Brett P, Hohlt B, et al. Using OS observations to improve performance in multicore systems[J]. IEEE Micro, 2008, 28(3): 54-66.

[13] Yasin A. A top-down method for performance analysis and counters architecture[C]//Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). IEEE, 2013.

[14] Suleman MA, Qureshi MK, Patt YN. Feedback-driven threading: power-efficient and high-performance execution of multi-threaded workloads on CMPs[C]//Proceedings of the ACM 13th International Conference on Architectural Support for Programming Languages and Operating Systems(ASPLOS). 2008: 227-286.

[15] Ju T , Zhang Y , Zhang X , et al. Energy-Efficient Thread Mapping for Heterogeneous Many-Core Systems via Dynamically Adjusting the Thread Count[J]. Energies.

[16] Acosta C , Cazorla F J , Alex Ramírez, et al. Thread to Core Assignment in SMT On-Chip Multiprocessors[C]// International Symposium on Computer Architecture & High Performance Computing. IEEE, 2009.

[17] A. El-Moursy, R. Garg, et al. Compatible phase co-scheduling on a CMP of multi-threaded processors[C]// Proceedings 20th IEEE International Parallel & Distributed Processing Symposium.

Rhodes Island, 2006.
[18] Cruz E H M, Diener M, Alves M A Z, et al. Dynamic thread mapping of shared memory applications by exploiting cache coherence protocols[J]. Journal of Parallel & Distributed Computing, 2014, 74(3): 2215-2228.
[19] Osiakwan K, AKL Selimg. The Maximum Weight Perfect Matching Problem for Complete Weighted Graphs IS in PC[C]//IEEE Second Symposium on Parallel & Distributed Processing. IEEE Computer Society, 1990.
[20] Devine K D , Boman E G , Heaphy R T , et al. Parallel hypergraph partitioning for scientific computing[C]// International Parallel & Distributed Processing Symposium. IEEE, 2006.
[21] F. Pellegrini. Static mapping by dual recursive bipartitioning of process architecture graphs[C]//Proceedings of IEEE Scalable High Performance Computing Conference. Knoxville, TN, USA, 1994: 486-493.
[22] 周伟明．多核计算与程序设计[M] ．华中科技大学出版社，2009．
[23] Hwang, Kai, Xu, Zhiwei. Scalable parallel computing: technology, architecture, programming.[J]. scalable computing practice & experience, 1998.
[24] 陈国良．并行计算：结构 算法 编程[M] ．高等教育出版社，1999．
[25] M. Awasthi, D. Nellans, K. Sudan, R. Balasubramonian and A. Davis. Handling the problems and opportunities posed by multiple on-chip memory controllers[C]//2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT). Vienna, 2010: 319-330.
[26] Bokhari. On the Mapping Problem[J]. IEEE Transactions on Computers, 2006, 30(3): 207-214.
[27] Zhou X , Chen W , Zheng W . Cache Sharing Management for Performance Fairness in Chip Multiprocessors[C]//PACT 2009, Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques. Raleigh, North Carolina, USA. IEEE Computer Society, 2009.
[28] Z. Chishti, M. D. Powell and T. N. Vijaykumar. Optimizing replication, communication, and capacity allocation in CMPs[C]//32nd International Symposium on Computer Architecture (ISCA'05). Madison, WI, USA, 2005: 357-368.
[29] Bryant R E, O'Hallaron D R. Computer systems - a programmers perspective[M]. 机械工业出版社，2003.
[30] Diener M , Cruz E , Alves M , et al. Kernel-Based Thread and Data Mapping for Improved Memory Affinity[J]. IEEE Transactions on Parallel and Distributed Systems, 2015, 27(9): 1-1.
[31] François Broquedis, Jérôme CletOrtega, Stéphanie Moreaud, et al. hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications[C]//2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing. IEEE, 2010.
[32] 巨涛，张兴军，陈衡等．面向众核系统的线程分组映射方法[J] ．西安交通大学学报，2016，50(10)：57-63．
[33] Love R . CPU Affinity[J]. Linux Journal, 2003(111): 18-22.
[34] D. P. Bovet, M. Cesati. Understanding the Linux Kernel[M]. 中国电力出版社，2005.
[35] 肖兮，刘闯，何锋等．面向流体机械仿真的层次化并行计算模型[J] ．西安交通大学学报，2019，53(02)：121-127．
[36] 张楚华，琚亚平．流体机械内流理论与计算[M] ．机械工业出版社，2016．
[37] Bailey, David H, Barszcz, Eric, Barton, John T, et al. The Nas Parallel Benchmarks[J]. international journal of supercomputer applications, 2010, 2(4):158-165.
[38] Jin, H. & MA, Frumkin. The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance[J]. 1999.
[39] Jeannot E , Mercier G , Tessier F. Process Placement in Multicore Clusters:Algorithmic Issues and

Practical Techniques[J]. IEEE Transactions on Parallel and Distributed Systems, 2014, 25(4): 993-1002.

[40] Mcvoy L W , Staelin C. lmbench: Portable Tools for Performance Analysis[C]// Proceedings of the 1996 annual conference on USENIX Annual Technical Conference. 1996.

攻读学位期间取得的研究成果

一、专利申请

[1] 张兴军，鲁晨欣等. 基于多线程共享内存通信的动态线程映射优化方法及装置（已受理，申请号：2019112369096，学生一作）。

二、发表论文

[2] Chenxin Lu, Xingjun Zhang, et al. LHMapping: A Dynamic Thread-to-core Mapping Method Using Loop-based Hierarchical Grouping Algorithm. (Supercomputing，已投稿)。

学位论文独创性声明（1）

本人声明：所呈交的学位论文系在导师指导下本人独立完成的研究成果。文中依法引用他人的成果，均已做出明确标注或得到许可。论文内容未包含法律意义上已属于他人的任何形式的研究成果，也不包含本人已用于其他学位申请的论文或成果。

本人如违反上述声明，愿意承担以下责任和后果：

1．交回学校授予的学位证书；

2．学校可在相关媒体上对作者本人的行为进行通报；

3．本人按照学校规定的方式，对因不当取得学位给学校造成的名誉损害，进行公开道歉。

4．本人负责因论文成果不实产生的法律纠纷。

论文作者（签名）：　　日期：　　年　　月　　日

学位论文独创性声明（2）

本人声明：研究生　　所提交的本篇学位论文已经本人审阅，确系在本人指导下由该生独立完成的研究成果。

本人如违反上述声明，愿意承担以下责任和后果：

1．学校可在相关媒体上对本人的失察行为进行通报；

2．本人按照学校规定的方式，对因失察给学校造成的名誉损害，进行公开道歉。

3．本人接受学校按照有关规定做出的任何处理。

指导教师（签名）：　　日期：　　年　　月　　日

学位论文知识产权权属声明

我们声明，我们提交的学位论文及相关的职务作品，知识产权归属学校。学校享有以任何方式发表、复制、公开阅览、借阅以及申请专利等权利。学位论文作者离校后，或学位论文导师因故离校后，发表或使用学位论文或与该论文直接相关的学术论文或成果时，署名单位仍然为西安交通大学。

论文作者（签名）：　　日期：　　年　　月　　日

指导教师（签名）：　　日期：　　年　　月　　日