西安交通大学

硕士学位论文

面向众核处理器平台多线程静态映射优化机制研究与实现

学位申请人：

指导教师：

导师团队：

学科名称：软件工程

2021 年 04 月

Research and Implementation of Multi-thread Static Mapping Optimization Mechanism for Many-core Processor Platform

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

By
Supervisor:(Associate) Prof.
Supervisor Team : Prof. (Team)
Software Engineering
April 2021

硕士学位论文答辩委员会

面向众核处理器平台多线程静态映射优化机制研究与实现

答辩人：

答辩委员会委员：

XXXXXXXXXX 大学 XXX: _____（注：主席）

XXXXXXXXXX 大学 XXX:_____

XXXXXXXXXX 大学 XXX:_____

XXXXXXXXXX 大学 XXX:_____

XXXXXXXXXX 大学 XXX:_____

答辩时间：XXXX 年 XX 月 XX 日

答辩地点：XXXXXX

摘 要

近年来随着高性能计算平台的计算资源不断增加，应用程序的并行规模逐渐扩大，如何将应用程序的计算任务与计算平台的计算资源进行合理映射是高性能计算领域的一个关键问题。当多线程并行应用程序运行在 NUMA（Non-Uniform Memory Access，非一致内存访问）架构的计算平台上时，线程间访存共享数据的效率和节点间内存带宽并不均衡，这间接导致了程序运行时跨节点的访存次数增多，以及平均内存延迟升高，最终导致了程序的总体运行时间增加，降低了程序性能。

因此，本文首先提出了一种针对多线程并行应用程序的静态线程到计算核心的映射优化机制。本文分别对映射机制中的硬件信息检测模块、访存检测模块、计算映射模块和执行映射模块进行了设计，并进一步实现了映射机制过程中执行线程间通信量检测、线程内存访问负载检测、线程映射的分组计算和线程绑定的方法，并结合硬件信息检测模块中检测得到的硬件架构信息，使映射机制成为一个整体的系统。

其次，本文提出了 CMLB（Communication-Aware and Memory Load Balance Mapping Algorithm）线程分组算法。该算法在保证跨节点访存量尽可能少的前提下，能够平衡节点间内存带宽降低内存延迟，提高程序性能。该算法根据线程间通信情况和各线程对内存的访问情况，并结合硬件架构信息对线程进行分组划分，实现了线程到计算核心的一一对应。

最后，本文在 Intel Xeon 处理器集群上对一些基准测试程序进行了映射机制的整体优化效果测试和 CMLB 分组算法的性能测试。实验表明本文设计的静态映射优化机制在不产生额外运行时开销的基础上，对大多数多线程并行程序均有良好的优化效果，在运行时间、QPI 和平均内存延迟指标上分别达到了最高约 28.29%、84.03%和 4.83%的性能优化。本文设计的 CMLB 线程分组算法相比其他映射分组算法在 QPI 值相差不多的情况下，大幅度降低了内存带宽不平衡度及平均内存延迟，分别达到了最高 89.0%和 4.9%的性能优化。

关 键 词：多线程并行应用程序；共享内存通信；内存带宽；静态映射优化；

论文类型：应用研究

本研究得到国家十三五重点研发计划课题“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：）资助。

ABSTRACT

In recent years, as the computing resources of high-performance computing platforms continue to increase, the parallel scale of application programs has gradually expanded. How to properly map the computing tasks of application programs and computing resources of the computing platforms is a key issue in the field of high-performance computing. When a multithreaded parallel application runs on a computing platform with a NUMA architecture, the efficiency of accessing and storing shared data between threads and the memory bandwidth between nodes are not balanced, which indirectly leads to the program The increase in the number of cross-node memory accesses during runtime and the increase in average memory latency ultimately lead to an increase in the overall running time of the program, which reduces the performance of the program.

Therefore, this thesis firstly proposed a static thread-to-core mapping mechanism for multi-threaded parallel applications. We designed the hardware information detection module, memory access detection module, grouping computation module, and mapping execution module in the mapping mechanism, and implemented the method of the inter-thread traffic detection, thread memory access load detection, thread grouping computation and thread binding, combined with the hardware architecture information detected in the hardware information detection module. Finally, we made the mechanism become a whole system.

Secondly, this thesis proposed the CMLB (Communication-Aware and Memory Load Balance Mapping Algorithm) thread grouping algorithm. This algorithm can balance the memory bandwidth between nodes to reduce memory latency and improve program performance while ensuring that the amount of cross-node memory accesses is as fewer as possible. The algorithm divided the threads into groups according to the communication between threads and the access of each thread to the memory, and combined with hardware architecture information, finally implemented the one-to-one correspondence between threads and computing cores.

Finally, we tested the overall optimization effect of the mapping mechanism and the performance of the CMLB grouping algorithm using some benchmark test programs on the Intel Xeon processor cluster. The experimental results show that the static mapping optimization mechanism designed in this thesis has a good optimization effect on most multi-threaded parallel programs without introducing additional runtime overhead , reaching the highest performance of about 28.29%, 84.03% and 4.83% in the running time, QPI and average memory latency indicators, respectively. Compared with other mapping grouping algorithms, the CMLB thread grouping algorithm designed in this paper greatly reduces the memory bandwidth imbalance and average memory latency when the QPI value is similar

and achieves performance optimizations of up to 89.0% and 4.9%, respectively.

KEY WORDS: Multi-threaded parallel applications; Communication in shared memory; Memory bandwidth; Static mapping optimization

TYPE OF DISSERTATION: Application Research

本研究得到国家十三五重点研发计划课题“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：）资助。

目　录

CONTENTS

1 绪论

1.1 研究背景及意义

近年来，随着高性能计算不断发展，为匹配日益增长的计算需求，体系结构研究者们在最初多核处理器计算机的基础上，扩展计算核心的数量，推出了规模更大、计算能力更强的众核处理器计算机（如神威太湖之光采用的国产 SW26010 处理器，每个计算单元包含 64 个计算核心[1]，以及天河三号原型机采用的国产众核处理器 FT-2000+和 Matrix-2000+，每个节点最多包含 64 个 ARM 架构计算核心[2]），其大规模并行计算能力甚至达到 E 级以上。如今，很多大型的科学计算问题都需要使用众核计算平台，结合高性能计算技术，以最优的性能解决复杂的计算问题。

同时，并行应用程序也普遍表现出并行规模大以及可扩展性强的特点。其中最有代表性的 CFD（Computational Fluid Dynamics，计算流体力学）并行应用程序是高性能计算的重要领域之一，很多问题根据离散化的数值计算方法，利用计算机对流体的流动特性等参数进行数值模拟。在多数 CFD 数值模拟程序中，通过前处理过程建立的网格模型复杂且网格规模庞大，其规模可达数百亿[3]。因此，针对以 CFD 应用程序为代表的并行规模大以及可扩展性强的并行应用程序程序，研究如何合理高效地映射计算任务以提高程序运行性能是高性能计算领域一大重要方向，具有深远意义。

目前常用的并行编程框架分为基于共享内存的 Pthreads、OpenMP 编程模型，基于消息传递的 MPI 编程模型，以及基于数据并行的 CUDA、Fortran 90 编程模型。上述不同类型并行编程模型亦可混合使用，如 MPI+OpenMP、MPI+CUDA 的混合编程模型。相较于 MPI 以进程粒度并行外，本研究关注更细粒度的线程并行编程模型：OpenMP、Pthreads。OpenMP、Pthreads 适用于多线程编程环境，不同线程之间借助共享内存进行数据交换与共享，即线程间的通信。在 NUMA（Non-Uniform Memory Access，非一致性内存访问）架构下的并行计算机中，基于共享内存的多线程并行应用程序容易产生线程间通信效率不均匀以及节点间内存带宽不均衡的问题。因此，本研究在线程级并行粒度下，探索合理高效的多线程映射优化方案可以改善上述两种问题，也为规模较大并行应用程序的总体映射优化研究增加了层次性与弹性。

本研究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：）,针对“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的多线程并行应用程序，同时适用于通用多线程并行应用程序的线程到计算核之间的映射优化方案。本研究总体上，研究并实现了线程级计算任务到计算核的映射优化机制，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

随着一些大型科学问题的计算需求不断增加，高性能计算平台的计算性能及其体系结构复杂程度也不断完善和增强。大型并行应用程序的计算任务如何与高性能计算平台的计算资源进行匹配，这一问题受到国内外学者广泛的讨论与关注，这使得映射优化成为高性能计算领域较为庞大且具有一定影响力的研究。

目前在映射优化的所有相关研究工作中，按照映射对象可分为：对计算任务执行映射（Task Mapping）以及对应用程序的数据执行映射（Data Mapping）。Task Mapping 是指将计算任务调度到计算机的不同节点或处理器上，而 Data Mapping 是指将应用程序的数据放置在合适节点内存上。进一步地，Task Mapping 又可按计算任务粒度分为对进程执行映射（将应用程序的进程映射到计算机的不同计算节点或处理器上），例如，Mulya Agung 等[4]使用 CLB 算法将 MPI 应用程序的不同进程映射到 NUMA 架构计算机的不同处理器上；以及对线程执行映射（将应用程序的不同线程映射到计算节点的不同核心上），例如，Matthias Diener 等[5]使用 CDSM 算法将应用程序线程映射到 NUMA 架构计算机的不同计算核心上。也有研究同时进行 Task Mapping 以及 Data Mapping，例如，Matthias Diener 等[6]使用 kMAF 算法，在 NUMA 架构计算机上动态地迁移线程以及数据，以优化应用程序的运行性能。按照映射方式对线程执行映射又可分为：静态线程映射（应用程序运行前将线程与计算核心绑定，运行时不再迁移线程），动态线程映射（应用程序运行时，根据运行时变化特点动态迁移线程，将线程绑定至别的计算核心），其中 kMAF 算法便是动态线程映射。动态线程映射虽然能根据应用程序运行时特点不断调整当前的线程绑定位置，适应程序访存特点变化，但过多的迁移线程会带来不小的额外开销，可能会抵消部分映射优化带来的性能增益且实现难度较大。因此，本研究在线程映射方式上选择静态线程映射。此外，按照映射优化所解决的具体问题，又可分为：负载均衡映射优化，缓解硬件资源竞争映射优化以及解决线程通信不均衡映射优化等。

由上可知，映射优化种类较多分支复杂，是一项庞大的且涉及多个领域及多种技术的研究工作。国内外学者也对映射优化的不同种类分支进行了大量的研究，部分研究问题和背景也与本研究相似，即针对并行应用程序的线程级任务映射优化方法。

对于多线程并行应用程序的映射优化，在早期计算机体系结构简单且核数较少时，研究者使用枚举法来寻找最优的映射策略，如 AutoPin[7]以 IPC（Instructions per Cycle，每个时钟周期运行的指令数）为观测指标，枚举出所有可能的映射策略从中选取 IPC 最高的作为最佳映射策略。但随着计算机核心数量增加特别是现代高性能计算平台一个节点便有最多 64 个计算核心，枚举法显然只适用于早期的计算机系统。后来随着机器学习的发展，研究人员便使用机器学习技术搜索最优的映射优化策略，如 WangZhang 等[8]，使用机器学习技术预测应用程序的最优线程数以及最优映射策略，并将其在编译器级实现。上述基于机器学习的线程映射优化，属于静态线程映射方法，适用于具有多个计算核心的现代计算机。但在搜索最优映射策略时，存在搜索空间大、最优解稀

疏等问题，导致算法时空复杂度较高，当程序线程数量增加时复杂度呈指数型增长。

近年来针对线程级任务映射优化，研究者们提出基于通信感知（Communication aware）的映射优化策略[9][10]。在共享内存机器架构下，不同线程依次读写同一缓存行的数据称为线程间的一次通信[11]。该策略通过将通信较频繁的线程放置在计算机同一节点，以增加线程的访存局部性（Loclity）提升程序性能。在 NUMA 架构下，计算核心访存其所在节点上的数据比访存其他节点上的数据产生的访存延迟要小一些（具体性能差异视具体机器而定）。基于通信感知的映射策略尽量使线程间通信发生在节点内部，避免了在 NUMA 架构下延迟更高的跨节点通信，从而达到优化程序性能的效果。这也是本研究所采取的映射优化策略。

为了实现基于通信感知的映射优化策略，需要统计应用程序线程间的通信情况，进一步地需要追踪剖析（Profiling）应用程序的访存行为。目前已知的应用程序 Profiling 工具分为：基于系统级（System-level），如 kMAF 中的通信检测算法，利用虚拟内存中的缺页（Page Fault）原理检测线程访存行为，并进一步统计得到线程间的通信次数，最终整理得到通信量矩阵；基于用户级[12]（User-level）。其中基于 User-level 的 Profiling 工具有两类：使用 Intel 提供的 Pin 动态二进制插桩工具[13]，如 Numalize[14]检测应用程序的线程通信情况得到通信量矩阵；使用 Linux 系统内核提供的 Perf 性能分析工具，如 ComDetective[15]一个轻量级线程通信检测工具。上述工具检测得到的均为应用程序的通信量矩阵（Communication matrix），通信量矩阵中的值表示某两个线程的通信次数，Communication matrix 反映了应用程序所有线程间的通信情况。

实现基于通信感知的映射优化策略的另一大问题是如何根据 Communication matrix 对线程进行分组划分。国内外研究者们对此问题也进行了大量的研究，并一致认为分组划分问题是一个 NP-hard 问题。Edmond[16]采用的图划分算法合理有效解决了线程组划分问题，但前提是线程数及处理单元数均为 2 的幂次积。Scotch[17]是一个开源分组算法软件包，使用双递归的策略，自顶向下逐步层次化的将所有线程拆分为较小的组。Scotch 的双递归计算仍有较高的时延，但其层次化的分组符合并行计算机系统的层次化架构。与 Scotch 相反 Eagermap[18]采用自底向上的迭代策略，运用贪心算法的思想层次化分组，相较于 Scotch 具有较小的时延并且也符合并行计算机系统的层次化架构。此外，Choicemap[19]相较于 Eagermap 的贪心策略采用更公平的线程配对算法，同时也具有层次性。

1.3 论文的研究内容

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”。

多线程并行应用程序运行在共享内存的 NUMA 架构上时，存在如下两个问题：

（1）线程间通信开销不均衡。线程间通信需要不同的线程读写同一缓存行中的数

据，由于 NUMA 架构访问内存延迟的不一致性，不同线程间的内存访问速率存在差异，导致线程间的通信开销不均衡，使得程序性能下降。

（2）NUMA 节点间内存带宽不均衡。每个 NUMA 节点有自己独立的内存，多线程并行应用程序每个线程访问内存的次数并不相同，如果把访问内存次数较多的线程均放置在一个节点，那么就会造成该节点内存访问负载增加同时内存访问延迟升高，发生内存拥塞现象[20]（Memory Congestion），使得程序性能下降。

本文针对以上两个问题，研究并实现一种 NUMA 架构下的提升通信效率平衡内存负载的多线程静态映射优化机制，并在并行基准测试程序上进行了测试。主要工作内容如下：

1） 针对多线程并行应用程序，在 NUMA 架构的计算机系统中，解决程序运由于线程间数据交换和共享不平衡和节点间内存带宽不平衡导致的程序性能下降问题，设计并实现静态线程映射优化机制，提升了应用程序的性能。

2） 提出了一种提升线程间通信效率并同时平衡节点间内存负载的线程分组方法进一步提升了映射机制的优化效果。

3）在 Intel Xeon 同构处理器平台上，对映射机制以及分组算法进行了性能测试。

1.4 论文组织结构

论文组织结构如图 1-1，本文共分为 6 章，每章的组织结构如下：

第一章 绪论。主要介绍论文的研究背景及意义、国内外研究现状、论文主要研究内容及其创新点、论文的组织结构。

第二章 相关理论技术基础。主要介绍本文研究工作涉及到的相关理论技术，包括多线程并行编程模型：OpenMP；NUMA 架构计算机的体系结构以及访存模型；描述线程映射优化具体解决的问题以及根据这些问题对线程映射优化的定义；本研究所用到的线程访存检测工具 Perf 的工作原理。

第三章 多线程静态映射优化机制的设计与实现。首先介绍整个映射优化机制总体流程，概述每个模块的作用，然后介绍各个模块的设计流程包括：硬件信息检测模块、线程访存检测模块，计算映射模块以及执行映射模块，之后介绍线程访存检测模块的实现细节，以及计算映射模块以及执行映射模块的实现细节。最后介绍静态映射机制的实现过程。

第四章 映射分组算法 CMLB 的设计与实现。映射分组算法应用于第三章映射机制中的计算映射模块。本章首先根据线程映射优化的定义以自顶向下的方式对 CMLB 分组算法进行流程设计，之后根据 CMLB 的设计流程对其进行实现，最后对比分析了 CMLB 的理论效果。

第五章 映射机制的性能测试与分析。本章对设计实现的映射优化机制在并行基准测试程序集上进行了实验，测试了映射优化机制的整体优化性能，并将其他分组算法与本研究提出的 CMLB 分组算法实验测试，测试指标包括：程序运行时间、QPI（Quick

Path Interconnect，快速通道互联）流量、节点内存带宽不平衡度、内存访问延迟，并对实验结果详细分析，同时与 kMAF 动态映射机制比较优化性能，最分析了映射机制的额外开销。

第六章 结论与展望。本章对论文的研究工作进行了总结，得出结论并且分析不足。提出下一阶段研究的展望。

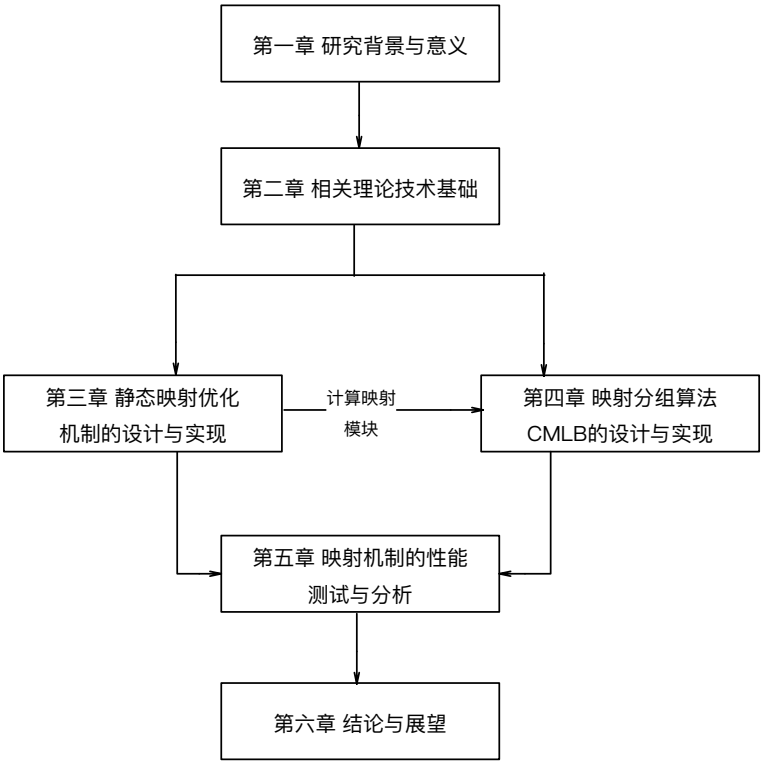


图 1-1 论文组织结构图

2 相关理论及技术分析

2.1 多线程并行编程模型

随着近年来计算机系统的处理器核心数不断增加，越来越多的计算核心被集成在单个处理器中，计算机体系结构也越来越复杂，并逐渐发展为多核、众核架构。为了匹配适应现代计算机的多核架构，研究人员对并行化技术也展开了深入研究，并行化技术可以使得应用程序在多个计算核上同时运行，在提高了程序的运行效率同时也充分利用了多核计算机的计算资源。线程作为操作系统调度的最小单位，在运行应用程序相较于进程具有轻便性与灵活性，因此运用多线程并行化技术提升应用程序运行性能是一种主要的方式。

线程相较于进程没有自己独立的资源，因此在多线程环境下每个线程都会与其他线程共享其进程所包含的资源，包括虚拟地址空间、文件描述符和信号处理等等，所以多线程并行化技术非常适合于共享内存的多核计算机架构。在目前主流的操作系统平台上（Windows，Unix/Linux）均有多线程并行编程的相关编程接口，如 OpenMP、Pthreads。这些编程接口均符合统一的规范和标准，使得每个线程能在共享内存的环境中读写到正确的数据。接下来介绍一个主流多线程编程模型：OpenMP。

OpenMP 程序设计模型提供了一组独立于平台的编译指导、指导命令、函数调用和环境变量，显示地指导编译器开发程序中的并行性。目前支持 Fortran，C/C++等编程语言。OpenMP 的执行模型采用 fork-join 模式，fork 表示创建新线程或处于阻塞状态的下场呢好难过被唤醒，join 表示多线程汇集为一个线程，如图 2-1 所示。程序开始时只有主线程在执行，当运行过程中遇到需要并行的任务，则派生出其他子线程来执行并行任务。在并行执行的时候，主线程和多个派生子线程共同工作，在并行任务执行完毕后，派生子线程退出或阻塞，因此不再工作，从而控制流回到主线程[21]。在 OpenMP 程序中，每一个线程会有一个线程 ID，用来表示不同的线程。

循环体的多线程执行采用的是静态平均调度策略，要求循环体内不存在依赖。在插入 OpenMP 编译指导之前要分析循环的依赖，确定没有数据依赖和循环体承载的依赖。如果存在依赖，就只能重构或变换循环，消除依赖关系。

此外，OpenMP 还提供了共享内存模型。共享内存模型中，数据变量被划分为私有（private）变量和共享（shared）变量。在默认情况下，一个进程内部的数据为共享变量，他们对同一个进程中的所有线程可见。如果多个线程在共享内存区域中实际操作专有变量，由于共享内存中的私有变量都是线程自己的所有权的某个变量的副本，不对其他线程可见，因此访问私有变量不存在数据竞争；而因此使用 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[22]；还提供了 API 中的互斥函数。

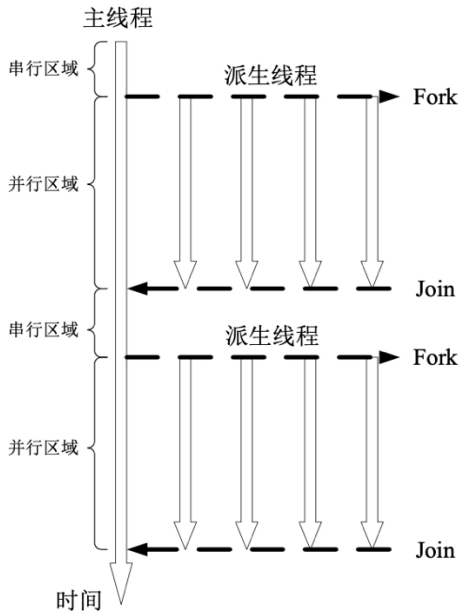


图 2-1 OpenMP 的 fork-join 并行执行模式

2.2 NUMA 架构及其访存模型

2.2.1 NUMA 架构简介

NUMA（Nonuniform Memory Access，非均匀内存访问系统），相较于 UMA（Uniform Memory Access，均匀内存访问系统）其计算核心访问主存的延迟开销并不均匀，即处理器访问本地主存时（Local Access）开销小，而访问远程主存时（Remote Access）的开销大，两者访问延迟相差较大。带有 Cache 一致性的 NUMA 系统（Cache-coherent NUMA，CC-NUMA）：即每个处理器的 Cache 间维持数据一致性的 NUMA 系统。没有 Cache 一致性的一个 NUMA 系统基本等价于一个集群，近年来出现的商用计算机系统均为 CC-NUMA 架构，和 SMP 以及集群有很大区别。一般情况下，非均匀内存访问系统都带有 Cache 一致性。

对于 SMP 架构，虽然有效的 Cache 机制能减少处理器和主存间的总线业务，但随着处理器数量增加，总线业务也会增加；同时 Cache 一致性信息也要通过总线传递，进一步加重了总线的负担。也就是说，处理器个数到达某个值后，总线就会成为性能的瓶颈。能够拥有大规模处理器又能拥有很大的全局内存的 NUMA 技术解决了 SMP 的总线瓶颈问题。因此，NUMA 架构的目标是提供一个大内存、多个 SMP 节点的透明系统，每个节点带有自己的互联系统。

2.2.2 NUMA 访存模型

图 2-2 为 Intel Boradwall NUMA 架构示意图，一个 NUMA 节点内部分为两大部分：Core 与 Uncore，并包含多个 Core 与一个 Uncore。其中 Core 为 CPU 计算核心部分包含 CPU 计算基本部件，如逻辑计算单元（ALU）、浮点计算单元（FPU），以及 L1、L2 缓存。Uncore 包含节点中除了 Core 以外的所有部件，如最后一级缓存（LLC Cache）

或 L3 缓存、IMC（Integrated Memory Controller，整合内存控制器）、QPI（QuickPath Interconnect，快速通道互联），还有其他外设控制器。NUMA 节点间通常使用 2-3 根 QPI 总线连接，带宽可达 12.8GB/s。

接下来介绍 NUMA 节点内部的计算核心是如何进行一次访存。假如位于 NUMA 节点 0 的计算核心 Core 0 需要读取一个数据，它首先访问 L1 Cache，若成功取到数据即 L1 Hit 则整个访存过程结束，否则发生 L1 Miss Core 0 继续访问 L2 Cache。与访问 L1 Cache 相同，若 L2 Hit 整个访存过程结束，否则 L2 Miss 需要访问节点 0 的 Uncore 部分 LLC（L3）Cache，若 LLC Hit 整个访存过程结束，否则发生 LLC Miss 然后 Core 0 将读请求发送到节点 0 的监视总线上。节点 0 接收到读请求后在目录里查找，确定数据位于哪个节点，若数据在本地主存上则发生 Local Dram Hit，Core 0 向节点 0 的 IMC 发送读请求，IMC 将读请求加入 RPQ[23]（Read Pending Queue，读请求待处理队列）等待从本地 DRAM 读取数据。若数据在节点 1 上，Core 0 通过 QPI 向节点 1 发送读请求，当节点 1 收到该请求后，将数据在本地主存中取得，再通过 QPI 发送到节点 0 的主存，同时送入 Core 0 及其各级 Cache 中。图 2-2 中蓝线表示 Core 0 对本地主存的访问过程（Local DRAM Access），红线表示 Core 0 对节点 1 主存的访问过程（Remote DRAM Access）。

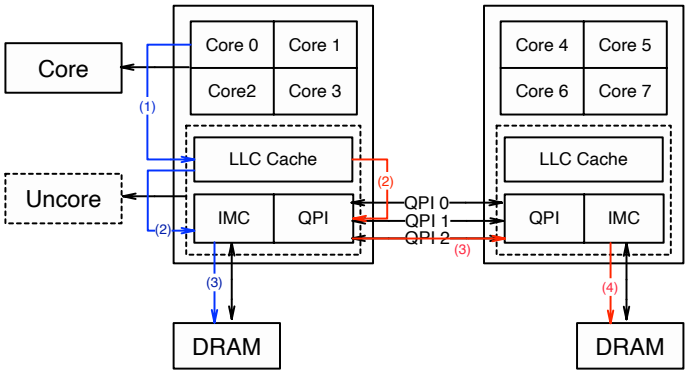


图 2-2 Intel Broadwall NUMA 架构及其访存模型

2.2.3 NUMA 架构的优缺点

CC-NUMA 系统的主要优点是相比于 SMP，无须对软件进行重大改变，就能拥有更高的并行级，从而拥有更高的性能。对于 NUMA 的任一节点，其内部总线上的业务是受限的，不超过总线的处理能力。然而，如果访问远程节点主存的操作过多，NUMA 的性能就会下降。如果应用程序具有很好的空间局部性，那么这个程序所需的数据会集中于频繁使用、有限的页上，这些页面在应用程序刚开始执行时，就会装入本地主存从而避免这种性能下降的问题。这种页面分配方式也就是 Linux 系统中默认数据放置策略（Data Placement Policy）First-touch：使内存页面放置在第一次访问自己的节点上，增加了访存的局部性。

2.3 线程映射优化

2.3.1 线程映射优化的定义

总体来说，线程映射是指对于多线程并行应用程序，根据程序的运行特征（访存行为特征、运行能耗特征、运行时长等）并结合其运行所在计算机系统的硬件架构特点，寻找出一个线程到处理器核心的对应关系，可以最大程度上提高应用程序的性能，包括平衡通信开销、提升访存效率、减少对硬件资源的竞争等等[24]。

在 OpenMP、Pthreads 为代表的多线程并行编程模型中，应用程序的线程间通过共享内存空间交换和共享数据，这一过程被称为隐式通信模式。如本文 2.2 节所述，在 NUMA 这种共享内存环境的并行计算机系统中，多线程并行应用程序正是通过这一方式实现线程间数据的交换和共享，同时却造成了如下两个问题：

首先，如图 2-3 当多线程并行应用程序运行在 NUMA 架构的计算机上时，若分布在相邻核上的线程之间交换数据，一般通过临近几个核心共享的 L2 Cache 进行；若分布在同一个节点内物理位置较远的核心上的线程间交换数据，一般通过一个节点内所有计算核心共享的 L3cache 进行；若分布在不同节点上的线程间交换数据，则只能通过节点间的 QPI 通道，从其他节点的内存中获取数据。很显然，以上列举的三种线程通信方式开销是不同的，第二种方式的开销略大于第一种方式，但它们都属于节点内部通信，开销近似相同。而由于 NUMA 架构的特性，第三种通信方式的开销属于跨节点通信，其开销远大于前两种。综上所述，可以将线程在 NUMA 架构计算上的通信分为两类：第一类是在节点内的通信，第二类是跨节点通信。因此，在 NUMA 架构上运行的多线程并行应用程序通信开销并不均衡，过多的跨节点通信会产生过多远端内存访问（Remote DRAM Access），这样会降低应用程序的访存效率。

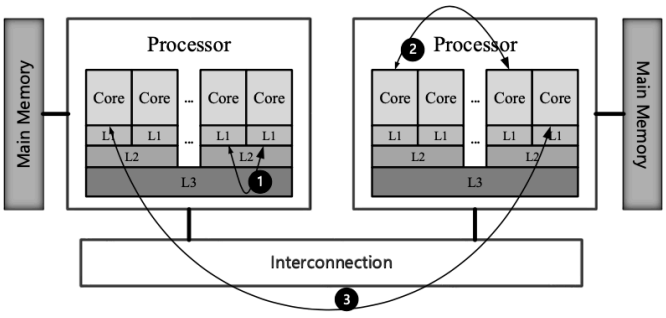


图 2-3 NUMA 计算机三种线程间通信方式

其次，现代 NUMA 架构计算机每个节点内集成的计算核心数量越来越多，这样会潜在地导致内存带宽争用问题，使得内存延迟升高。例如，程序运行时的某个时间段内，某个节点内部的核心争相访问内存，导致其内存带宽大幅升高，使得该节点的内存处于高饱和的状态，对该内存的其他访问事件需要等待更长的时间，内存访问延迟因此会升高，影响了程序整体性能。图 2-4 便展示了这种影响，使用 Intel Memory Latency Checker[25]工具测试内存带宽延迟，对 NUMA 计算机在单个节点内部进行测试。当某个节点内存带宽升高时，其内存延迟随之升高，特别地，当带宽超过最大物理带

宽时延迟会急剧升高。这种现象是由于节点间内存带宽不均衡导致某节点内存带宽过高，称之为内存拥塞问题（Memory Congestion）。

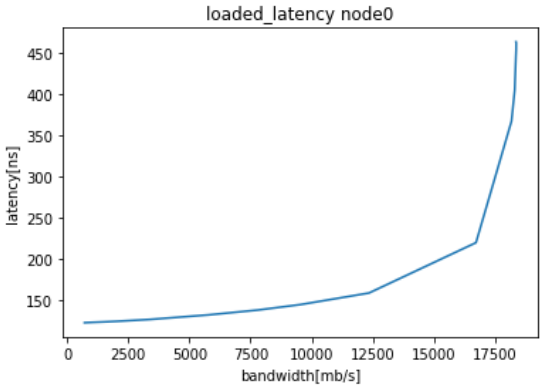


图 2-4 内存负载延迟测试

为了避免内存拥塞的发生，可以考虑将某节点的高内存带宽以线程迁移的方式分摊给其他节点。由于不同线程对内存的访问次数不一定相同，若根据所有线程的内存访问次数以一定的策略合理分配放置，将访问次数高的几个线程均摊给不同节点，则可以达到平衡内存带宽的效果。

为了同时解决上述两个问题，需要找出以上两个问题的联系，得出相应的解决策略。若只考虑第一个问题平衡线程之间的通信开销，则需要使线程间的通信尽量发生在节点内部，避免过多的跨节点通信。这样能极大减少跨节点的访存次数，但可能会将对内存访存次数多的两个线程放置在一个节点内，潜在地导致内存拥塞问题。若只考虑第二个问题，平衡节点间的内存带宽，则可能出现过多的线程跨节点通信导致跨节点访存量增多，降低程序性能。因此上述两个问题是相互影响的。

为了进一步探寻这两个问题如何影响程序的性能，进行如下测试：

分别使用 Eagermap 与 Interleave 两种策略，对 NPB-OMP 程序集中的 SP 程序从运行时间、节点内存带宽不平衡度、QPI 和内存延迟这几个指标进行测试。其中，内存带宽不平衡度计算了各节点内存带宽的标准差，QPI 表示跨节点传输的数据量。Eagermap 是一种线程放置方法，它将通信量大的线程划分至一个组内，这样可以平衡线程间通信开销避免出现过多的跨节点通信，极大地减少了过多的跨节点访存次数，属于减少跨节点访存量的策略。Interleave 是一种页面放置方法，通过在节点间交替放置数据页面保证每个节点的内存负载近乎相同，这使得每个节点内存访问量也近乎相等，属于平衡内存带宽的策略。

表 2-1 两种策略针对 SP 程序的性能测试

Policy	Exec time(s)	Imbalance	QPI(MB)	Latency(ns)
Eagermap	56.38	0.56	117008	574
Interleave	67.40	0.06	968955	632

如表 2-1 所示，虽然 Interleave 在节点内存带宽不平衡度（Imbalance）上明显低于

Eagermap，但其 QPI 值是 Eagermap 的将近 9 倍说明 Interleave 的跨节点访存量大大高于 Eagermap，因此导致在 Interleave 策略下有相当一部分的内存访问是由远端节点发起的，并由于 NUMA 架构特性使得平均内存访问延迟高于 Eagermap，最终导致总体运行时间明显高于 Eagermap。

由以上实验得知，仅考虑内存带宽的平衡的策略会使得远端内存访问次数增多，反而增加了平均内存延迟，导致程序性能不如仅考虑减少跨节点访存量的策略。因此需要在减少跨节点访存量的前提下，同时平衡内存带宽，这样才能使得跨节点访存量少的同时内存延迟得到下降，达到理想的优化效果。

综上，本文将线程映射优化定义为：通过对线程在计算核心上的的合理放置，在平衡线程间通信的前提下，使得每个节点的内存带宽保持均衡，达到优化程序性能的效果。

2.3.2 线程映射的理论优化效果

根据本文在 2.3.1 节对问题的描述以及对映射优化的定义，线程映射优化解决的问题是：线程间通信开销不均衡以及节点间内存带宽不均衡导致应用程序性能下降。因此线程映射优化理论上使得线程间的通信发生在各自节点内部并且各个节点的内存访问带宽应大致相同。

进一步来说，线程映射优化使得 NUMA 计算机系统中的远端内存访问相较映射前减少即节点间 QPI 通道的数据流量降低，并且由于映射后各节点内存带宽的平衡使得内存访问延迟相较于映射前也有所降低。上述影响都会使得应用程序的总体运行时间缩短，从而提高了程序性能。

在第 5 章的实证部分，会针对程序运行时间、QPI 流量、节点内存带宽不平衡度以及内存访问延迟这 4 个指标对线程映射的优化效果进行测试。

2.4 Perf 相关技术理论

根据之前对线程映射优化的定义与描述，整个映射优化机制是根据线程间的通信特征以及各个线程的内存访问负载特征进行线程分组和执行映射，因此如何检测并统计到这些特征是本研究需要解决的关键问题。

本研究选择基于 Perf 的程序性能分析工具，用来检测并统计线程间通信及其访问负载特征，为之后的步骤提供决策指导。接下来介绍与 Perf 相关的技术理论。

2.4.1 PMU 简介

PMU（Performance Monitor Unit，性能监视单元）存在于计算机的每个计算核心中，它提供了一种可编程的方式来对一些硬件事件（例如读写数据、CPU 时钟周期数）进行计数。对于某个想要统计检测的事件可以在 PMU 上对该事件设定一个阈值，一旦检测到该事件发生的次数超过了这个阈值，PMU 便触发一次中断，该阈值也被称为采样率（Sampling Rate）。PMU 发生中断时会有基于 Perf 的性能监视程序对发生 PMU

中断的事件进行一次记录，包括事件类型、执行的线程 id、访存的内存地址、数据大小等。每次的 PMU 中断被称为一个样本（sample），性能监视程序便根据 PMU 中断来进行事件采样（sampling）。Intel 的 PEBS（Precise Event Based Sampling，基于事件的精确采样）功能就是利用了 PMU 对某些类型的事件进行采样并能精确的提取其指令访问的有效地址。AMD 从第 10 代处理器以来，也提供了相应的功能 IBS（Instruction Based Sampling，基于指令的采样），同样 POWER 处理器通过 Marked Event 工具提供了这个功能。

2.4.2 perf_event_open

Perf 中提供了一个名为 perf_event_open 的系统调用函数可以对 PMU 进行编程，该函数可以获取 PMU 中断产生的数据记录。从 Linux2.6.39 版本开始 perf_event_open 可以访问多个 PMU。用户态的基于程序性能检测的代码可以被映射到一个循环的缓冲区，当 PMU 发生中断时 Linux 内核向其特定线程传递信号，该线程不断地将 PMU 采样得到的数据写入这个循环缓冲区，perf_event_open 从而可以从缓冲区中获取到相应事件的数据。

综上所述 Perf 是一个可以利用 PMU 进行事件采样的性能分析工具，通过 Perf 可以精确地检测并统计访存事件发生次数及其访问地址，其中 perf_event_open 便是 Perf 中获取 PMU 中断数据的编程接口，Perf 也可以设置 PMU 的事件采样率。总之，Perf 为本研究获取线程间通信特征以及内存访问负载特征提供了基础。

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特点，以及以 OpenMP 为代表的多线程并行编程模型的主要概念、执行模式和适用环境；其次介绍了 NUMA 计算机系统的主要概念、类别、访存模型以及优缺点，为后续小结的问题描述做了铺垫；然后针对本文的主要研究内容对线程映射优化的基本定义以及所要解决的问题做了详细的分析论述，并对映射后的理论优化效果做了性能指标上的分析；最后介绍了检测并统计线程间通信特征以及各个线程内存访问负载特征的相关技术理论——Perf。

3 多线程静态映射优化机制的设计与实现

根据本文 2.3 节所述，当多线程并行应用程序在 NUMA 架构的计算机上运行时，会出现线程间通信开销不均衡以及节点间内存带宽不均衡的问题，导致程序性能下降。并根据映射优化的定义，本研究设计并实现了多线程静态映射优化机制，该机制结合计算平台的硬件架构信息和程序的线程通信特征及内存访问特征，使用线程分组映射算法将每个线程放置在合适的计算核心上，达到优化程序性能的效果。

由此本研究在设计该机制时会面临如下三个问题：

1） 如何检测并统计得到应用程序的线程通信特征和内存访问特征；

2） 如何根据上述程序特征结合计算平台的硬件架构信息，设计线程分组算法；

3） 如何根据线程分组结果将线程一一绑定至计算核上。

在映射机制的设计过程中，需要寻求合适的方法将上述 3 个问题依次解决，最后形成一个完整的系统。本章的后续内容将分别介绍对上述 3 个问题的研究。

3.1 线程映射优化机制的总体设计

根据在设计映射优化机制时提出的 3 个问题，本研究设计了适合的解决方案并将其实现，最终封装为 3 个模块：访存检测模块（包括通信量检测及内存访问负载检测）、计算映射模块以及执行映射模块，同时加入硬件信息检测模块用于检测计算机的硬件架构信息。将这些模块有机组合，便构成了整体的映射优化机制。本研究最终设计实现的静态映射优化机制流程图如 3-1 所示。具体流程描述如下：

流程开始，需要输入相关参数，包括所要映射优化的应用程序以及 2.4 节提到的 PMU 采样率（默认为 100K，具体参数值在 3.2 节介绍），之后硬件信息检测模块与访存检测模块同时进行，硬件信息检测模块是调用 hwloc[26]库对计算机硬件架构检测，检测时间较短，所以会在访存检测模块之前结束。访存检测模块执行时，会在基于 Perf 的性能监视程序下运行所要映射优化的应用程序，即下面流程描述步骤 1；之后将硬件信息检测模块与访存检测模块的输出结果输入到计算映射模块，即下面流程描述步骤 2；最后，将线程分组结果输入到执行映射模块，即下面流程描述步骤 3，流程结束。步骤 1-3 具体内容如下：

1）步骤 1，检测并统计线程通信情况以及线程内存访问负载情况。定义一个大小为线程总数的通信量矩阵 $A \in R^{n \times n}$ (n 表示程序运行的总线程数，A[i][j](i∈ n,j∈ n)表示线程 i 与线程 j 的通信量)。首先启动性能监视程序然后运行所要映射优化的应用程序，性能监视程序框架与通信检测算法借助 ComDetective，性能监视程序捕获到 PMU 中断产生的访存事件信息，包括执行访存指令的线程 id、访存地址及当前时间戳（单位为时钟周期数），将其记录在以访存地址为键的哈希表中，同时将该条记录的保存在二维数组中。然后以正在运行的线程数判断应用程序是否结束，若未结束，性能监视程

序继续捕获 PMU 中断产生的信息，根据访存地址在哈希表中查找若有其他的记录则判断这两条记录的时间戳是否满足一定条件，若满足且线程 id 不同则这两条访存记录的线程发生一次通信，将通信情况更新于通信量矩阵中。同时将该记录在二维数组中。重复上述过程，当应用程序运行结束时，性能监视程序也运行完毕，得到最终的通信量矩阵 A。最后使用数据分析方法处理记录所有访存记录的二维数组，得到一个内存访问负载向量 V 保存着每个线程的内存访问负载特征。详细的通信检测方法与内存访问负载检测方法在 3.2 节介绍。

2）步骤 2，根据步骤 1 得到的通信量矩阵、内存访问负载向量和硬件架构信息，计算线程分组。本研究提出并设计实现了 CMLB 算法，该算法结合通信量矩阵和内存访问负载向量以及硬件架构信息，对所有线程进行分组，输出用 map 保存的线程到计算核心的映射关系。分组算法的原则是使通信量较高的线程放置在一个节点，同时尽量保持各节点内存访问负载大致相同。组数以及组内线程数量由硬件架构信息决定，详细的 CMLB 算法设计与实现过程将在第 4 章介绍。

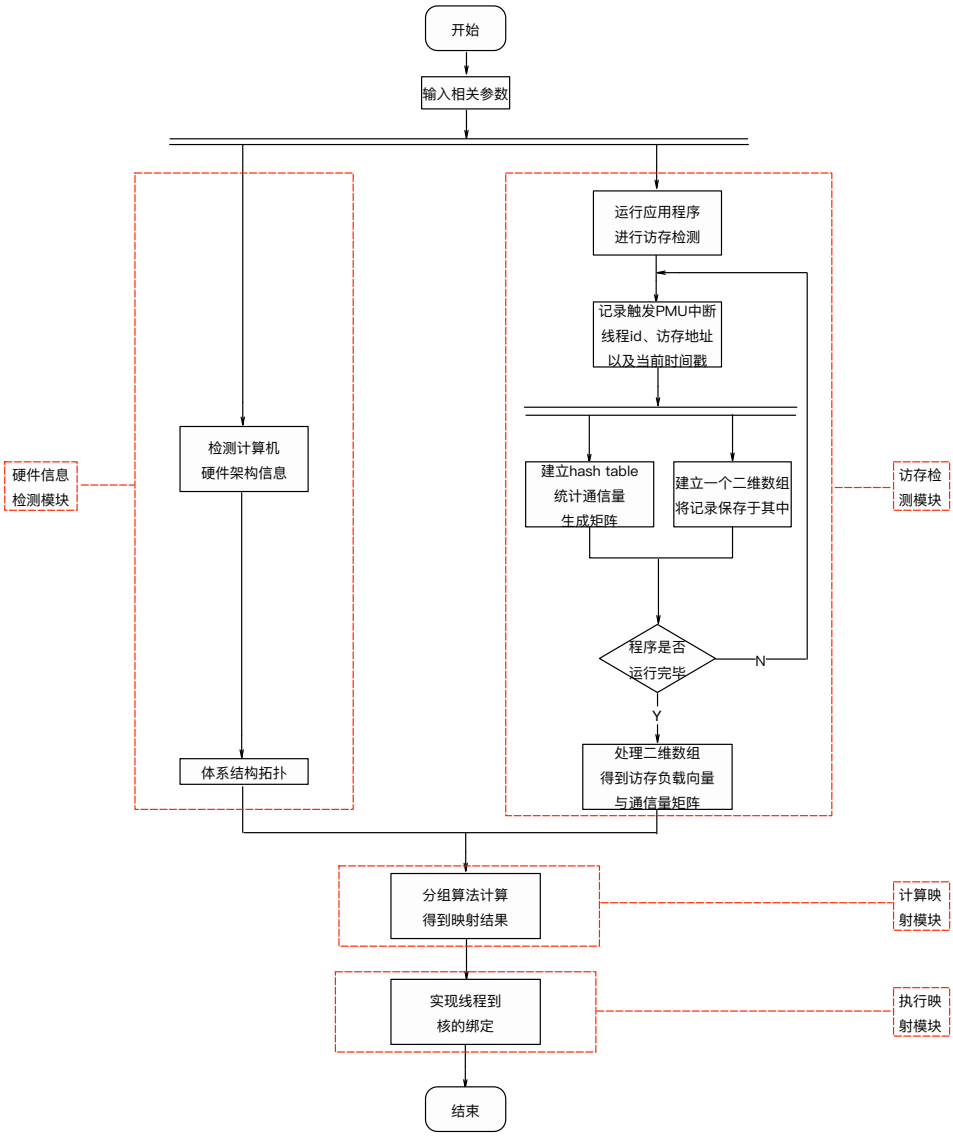


图 3-1 静态映射优化机制流程图

3）步骤 3，根据步骤 2）的分组结果，执行映射。将步骤 2）map 映射关系解析输出至映射结果文件，用 numactl[27]根据映射结果文件设置环境变量实现线程到计算核心的绑定过程。详细的线程绑定过程在 3.4 节介绍。

3.2 线程访存检测模块

在本研究中，把运行的应用程序的两个不同线程依次访存相同的数据称之为线程间的通信，因此线程通信本质上就是两次线程访存事件。所以本研究将线程间的通信检测与内存访问负载检测均归入线程访存检测模块。这里简要论述下二者之间的关系：线程间通信量检测注重于统计线程与线程之间在访存数据上的重合度，如果访存数据的重合度很高，即它们访存的大部分是相同的数据，则说明这两个线程通信量较大，通信量检测最终得到的是一个二维对称矩阵。内存访问负载检测注重统计每个线程独自的对内存访问情况，最终得到的是一个一维数组，并不同于通信量矩阵所表示的线程间交互程度。

本节的后续内容将分别介绍线程通信量检测与内存访问负载检测的方法。

3.2.1 线程间通信量检测

总体上，本研究设计的静态线程映射优化机制使用基于 Perf 的通信量检测方法，接下来将对其实现方法进行详细描述。

通过本文第 2.4 节可知，使用 Perf 中的系统调用函数 perf_event_open，可以获取 PMU 中断产生的访存事件相关数据。利用 perf_event_open 函数，检测机制便可以捕获到应用程序的所有访存事件。在 3.2 节开头提到过，两个线程间的通信本质上还是两次线程访存事件，通信检测机制根据这两个线程的访存地址且两次访存的时间戳来判断这两个线程是否发生了一次通信。若两次访存的地址位于同一访存单元且两次访存的时间戳满足一定条件，则判定发生了一次通信。这里的访存单元是指存储数据的粒度，包括内存页面单独的一个数据（Data）、内存页面的一个缓存行（Cache Line）或者是一个内存页面[28]（Page）。本研究中的通信检测机制使用缓存行粒度的访存单元，即两个线程在一定时间范围内访存了同一缓存行，便称这两个线程发生了一次通信。

通信检测机制开始前建立一个访存哈希表，其键为缓存行地址，值为访存时间的相关信息，包括执行该访存指令的线程 id、访存类型、访存数据长度、访存发生的时间戳。通信检测机制开始时，perf_event_open 函数获取到 PMU 中断产生的访存信息，并传给自己定义的 PMUSampleHandler 函数进行处理。处理方式如下：当一条访存记录进入 PMUSampleHandler 函数时，首先将访存地址 M_1 转换为其所在的缓存行地址 L_1 并根据 L_1 在哈希表中查找键相同的所有记录列表，若记录列表不为空，则遍历该列表选择列表中时间戳与当前访存时间戳的差值在一个生存周期（Expiration period）内且线程 id 不同的那一条记录，这两条访存记录的线程发生一次通信。这里生存周期定义为：一条访存记录在访存哈希表内的生存时间，若该访存记录的时间戳与当前时刻的差值超过生存周期，则该记录失效，当哈希表项容量已满时清理掉所有失效的记录。

生存周期的设置借鉴 CommDetecive 中的数值，判定发生一次通信后，将这两个线程在通信量矩阵的对应位置中更新。然后将当前访存记录插入到哈希表中。通信检测算法如算法 3-1 所示：

算法 3-1：通信检测算法

```
1  输入: 访存地址 M1，线程 id T1，时间戳 ts1.
2  全局变量：访存哈希表 ConcurrentMap，通信量矩阵 CommMatrix，生存周期 ExpirationPeriod.
3  Begin
4     L1 = getCacheline(M1);       //根据访存地址得到其所在缓存地址
5     entrylist = ConcurrentMap.AtomicGet(key=L1);       //在访存哈希表中取得键相同的记录列表
6     if entrylist != NULL then
7           for entry in entrylist:
8                 < M2 ,T2, ts2> = getEntryAttributes(entry)
9                 if T1 != T2 and ts1 - ts2 <  ExpirationPeriod then    //判断线程 id 与时间戳是否满足条件
10                      CommMatrix[T1][ T2] += 1;       //更新通信量矩阵
11                      CommMatrix[T2][ T1] += 1;
12                end if
13          end for
14    end if
15    ConcurrentMap.AtomicPut(key=L1， value = <M1, T1, ts1>);       //将当前记录插入访存哈希表
16 End
```

图 3-2 描述了整个通信检测过程。过程如下：1）线程 T_1 发生了一次访存事件，其所在核的 PMU 对其进行计数；2）当 PMU 计数器超过了阈值发生中断；3）PMU 产生中断的数据记录传递给访存哈希表，并在表中记录；4）线程 T_2 的访存事件也触发了 PMU 中断；5）PMU 产生中断的数据记录传递给访存哈希表，在哈希表中根据键查找其他记录；6）线程 T_2 所在核的 PMU 产生的数据在哈希表中找到线程 T_1 的记录，T_2 与 T_1 发生一次通信并在通信量矩阵中更新。

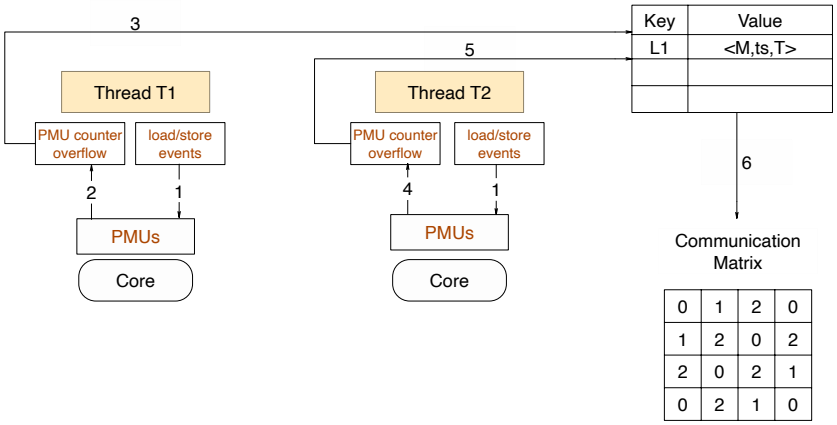


图 3-2 通信检测示意

3.2.2 线程内存访问负载检测

与线程通信量检测不同，内存访问负载检测分为两个阶段：在应用程序运行完毕前筛选出所有内存访问记录并收集于一个二维数组中；在应用程序运行完毕后使用数据分析方法对二维数组分析处理，最终得到内存访问负载向量。接下来将对内存访问

负载检测方法的两个阶段进行详细叙述。

在应用程序运行时，如 3.2.1 节所述，记录由 PMU 中断产生的所有访存信息。由于内存访问负载检测只需要统计访问内存的记录，因此需要对访存事件类型进行筛选。筛选后的每条访存记录包括访存地址、线程 id、访存时间戳、访存类型以及访存长度。为方便起见这里只提取访存地址、线程 id、访存时间戳作为一条访存记录，将其存入一个二位数组，当应用程序运行完毕后，二维数组记录了所有访存事件的信息。之后便进入二维数组处理阶段。

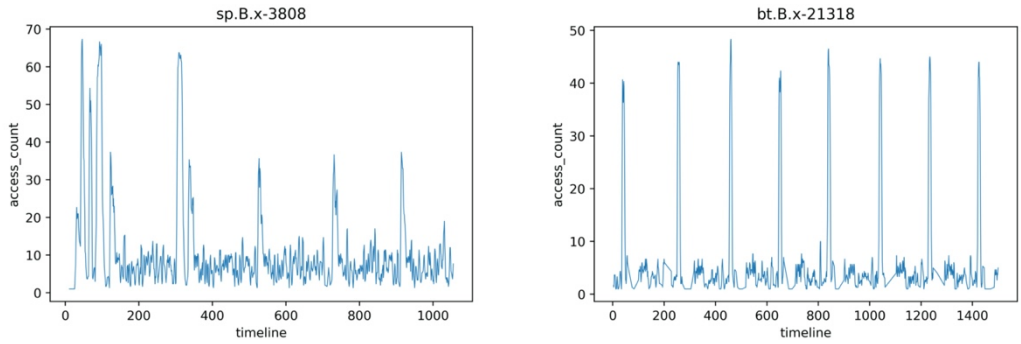
应用程序运行结束后，将对访存二维数组进行分析处理。此时性能监视程序也运行完毕，输出通信量矩阵与访存二维数组，在分析处理访存二维数组时，使用 Python 提供的 Numpy[29]以及 Pandas[30]库进行数据加载与处理。具体过程如下：

首先根据硬件信息检测模块检测到的 CPU 时钟频率，将二维数组中以时钟周期为单位的时间戳转换为以纳秒（ns）为单位的时刻。

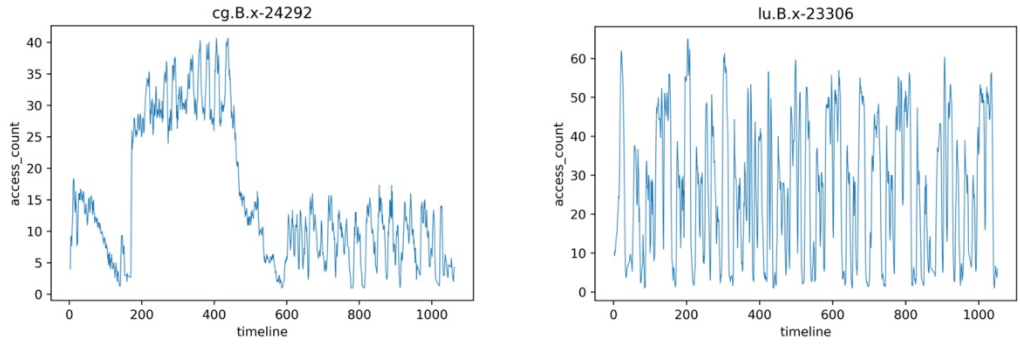
转换过程如公式 3-1:

$$tsc = \frac{cycles \times 1000000}{feq} \tag{3-1}$$

其中 tsc 表示以 ns 为单位的时间戳，cycles 表示以时钟周期为单位的时间戳，feq 表示 CPU 时钟频率，单位为 khz。转化完成后，需要设置一个时间片（timeslice），将时间戳间隔在时间片内的记录合并，时间片大小在本研究中设置为 1ms。合并后一条记录包括，时间戳信息、线程 id 列表以及合并记录的组成数（该合并记录是由多少条原始记录组成的）。合并组成数表示了在一个时间片单元内，进行了多少次内存访问，该数值越高说明这一时间片内内存访问数据流量越高，同时内存带宽延迟越高。在内存访问负载检测时，需要重点提取合并数高的记录的各个线程访存特征，因为合并数高的记录对内存访问影响大，容易造成内存拥塞问题。接下来对根据时间片合并后的二维数组进行分析，以时间戳为 X 轴，以合并数即每个时间片内的访存量为 Y 轴绘图，来查看应用程序随时间的内存访问次数变化情况，如图 3-3 为本研究针对几个 Benchmark 内存访问量随时间的变化趋势图。



(a) SP 应用程序　　(b) BT 应用程序



(c) CG 应用程序　　(d) LU 应用程序

图 3-3 NPB Benchmark 内存访问数量随时间变化趋势图

根据图 3-3 可发现，应用程序的访存过程呈一定的阶段性，如图 3-3（c）应用程序 cg 从 X 轴上来看，0-200 以及 200-600 是两个明显的访存阶段，每个访存阶段在 Y 轴上呈先先增大后减小的趋势，其他程序的访存过程也存在访存次数先增加后减小的阶段性特征。在进行程序的内存访问负载特征提取时，可以根据程序在访存过程中具有阶段性的特点，提取每一阶段的访存特征，再将所有阶段的特征综合起来。

为了实现这一思路，需要将程序的整个访存进行阶段划分，这里规定访存次数从低点上升到高点然后在下降到低点作为一个访存阶段。在划分访存阶段前需要对数据进行预处理，在使用 Perf 检测统计访存数据时，可能会存在噪声干扰，例如将一些不属于应用程序的访存事件监测统计进来，这可能是由于计算机系统环境某个时刻不稳定导致的。因此针对按时间片合并后的二维数组进行数据平滑，主要针对访存数量这样的时间序列数据进行线性平滑，这里参考 Rocka[31]中的方法：通常时间序列数据中的异常值小于 5%，因此删除与序列中平均值差异值前 5%的数据，并用线性插值来填充替换。

数据线型平滑完成后，开始进行访存阶段的划分。本研究使用滑动窗口（Slide Window）进行处理：参照数据线性平滑的方案，本研究使用整个访存数据中访存次数后 5%序列的平均值作为访存次数低点（low value）,设置滑动窗口两个指针从时间片 0 开始，首先固定左指针右指针移动，当右指针接触到访存次数低点时，将左指针与右指针包含的区间作为一个访存阶段记录下来，然后移动左指针到右指针的位置，开始下一个访存阶段的查找。

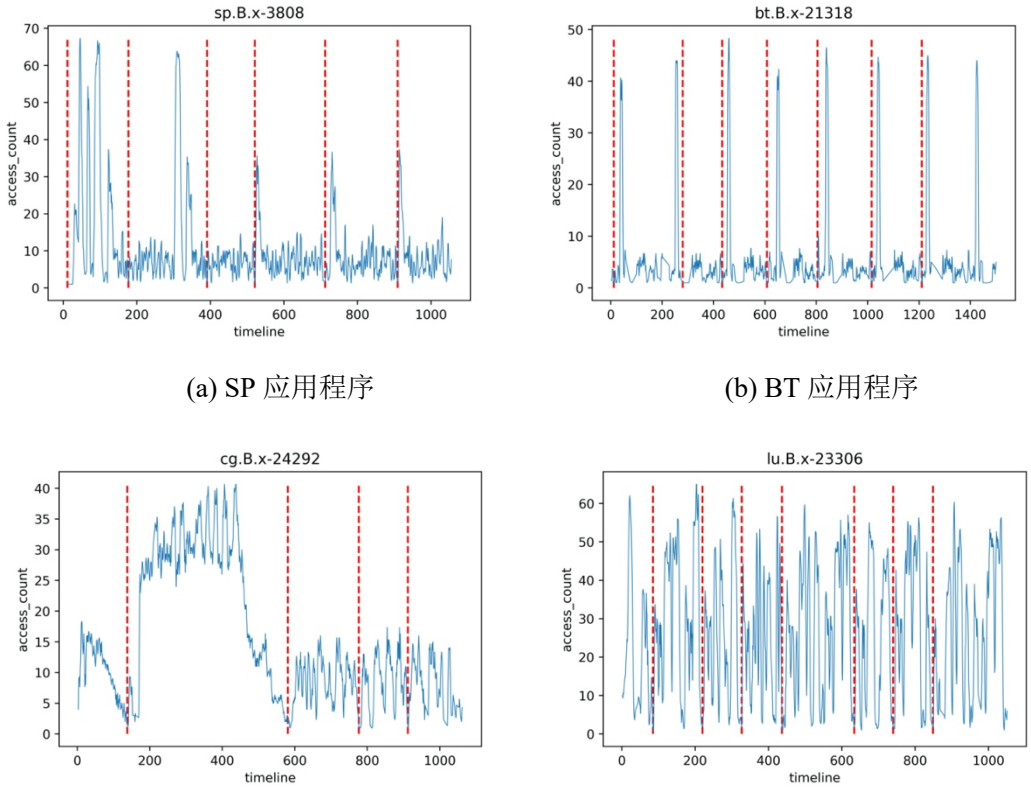
在实际实验中，由于访存次数的多个低点连续而出现一个访存阶段较小的情况，因此还需要根据访存数据的特点设定一个最小窗口范围，从而可以规避低点连续导致的访存阶段较小的问题。在对大多数应用程序进行实验测试后，发现访存次数低点连续个数基本在 100 以内，因此本研究将最小窗口范围定为 100，即一个访存阶段的跨度范围不小于 100 个时间片。

基于滑动窗口的访存阶段划分算法如算法 3-2 所示:

算法 3-2：访存阶段划分算法

```
输入:访存量的时间序列 s.
输出:所有访存阶段列表 acc_ls.
Begin
    num = len(s) * 0.05       // 占序列 s 5%的元素个数
    low_value = mean(sorted(s)[: num] )     //取序列 s 后 5%平均值作为低值点
    left = 0 ; right = 0        // 滑窗左右指针初始化
    min_width = 100       //规定最小窗口范围
    while right <= len(s) do
        if s[right] <= low_value then
            if right – left >= min_width then        //判断窗口大小是否满足条件
                acc_ls.append(s[left : right+1])      //满足条件记录一个访存阶段
            end if
            left = right
        end if
        right += 1
    end while
    return acc_ls
End
```

通过算法 3-2 便可得到程序的所有访存阶段，划分效果如图 3-4 所示。图中红线之间的区域代表一个阶段。



(a) SP 应用程序　　(b) BT 应用程序

(c) CG 应用程序　　(d) LU 应用程序

图 3-4 访存阶段划分效果图

如前文所述，程序的访存阶段经过划分后每个阶段具有自己的访存特征，为了统

计得到最终的内存访问负载特征，需要提取每个阶段的特征并综合起来。根据 3.2 节开头所述，内存访问负载特征反映的是各个线程对内存的访问负载权重，用一个数组表示，数组每个元素值代表每个线程的内存访问负载，若某个元素值比较高则说明该线程对内存产生的负载越大。

每个访存阶段的内存访问负载特征同样用一维数组来代表，为了得到每一阶段的特征，需要知道每个阶段的各线程内存访问次数，Perf 得到的每条访存记录均有线程 id 字段，按照访存记录中的线程 id 字段进行统计，便可得到每个线程的访存次数。因此，对于每个阶段的各线程内存访问次数，可以根据这一阶段所有访存记录的线程 id 字段进行统计得到。

根据本文 2.3 节所述，随着内存的访问量增加，内存访问延迟也随之增加。为了能更好地体现出这一规律，对于每个访存阶段计算出自己的内存负载权重，然后乘上该阶段各线程的内存访问次数，得到最终的内存访问负载特征。尽管每个阶段本身的访存量便可代表内存负载权重，访存量高的阶段内存负载本身就很高，但这样做并不能凸显出阶段之间的差异性。对于 2.3 节提出的内存拥塞问题，本研究设计的访存检测机制更希望在程序的整个访存过程中，识别出内存访问量高的某些时段，并重点提取这些时段的内存访问负载特征。因此引入了每个阶段的内存负载权重，使得访存量高的阶段更能凸显自己的内存访问负载特征。

得到每个阶段的访问负载向量后，将所有阶段的向量相加，便得到最终的内存访问负载向量。计算过程如公式 3-2、3-3:

$$w = \frac{1}{n} \sum_{i=1}^{n} d_i \quad . \tag{3-2}$$

$$V = \sum_{i=1}^{n} w_i \cdot p_i \ . \tag{3-3}$$

公式 3-2 描述了对每个访存阶段计算其内存负载权重，n 表示该阶段的时间片数，d_i 表示第 i 个时间片包含的访存记录数。计算得出的 w 数学意义上表示：该访存阶段内平均每个时间片包含的访存记录数。公式 3-3 描述了计算最终的访存负载向量，n 表示访存阶段数，w_i 表示第 i 个访存阶段的内存负载权重，p_i 代表第 i 个访存阶段的内存访问负载向量。

3.2.3 相关参数设置

本文 3.2.1 与 3.2.2 节分别介绍了线程访存检测模块的两大部分，现在针对线程访存检测模块涉及的一些参数的设置进行说明。

在使用 PMU 进行事件采样时，需要对事件类型以及采样率进行设置。由于本模块为线程访存检测模块，所以事件类型设置为：MEM_UOPS_RETIRED:ALL_STORES 与 MEM_UOPS_RETIRED:ALL_LOADS，这两个事件代表 Intel 处理器上内存读写的访存事件。采样率在本研究中设置为 2000，即 PMU 每 2000 个访存指令发生一次中断。

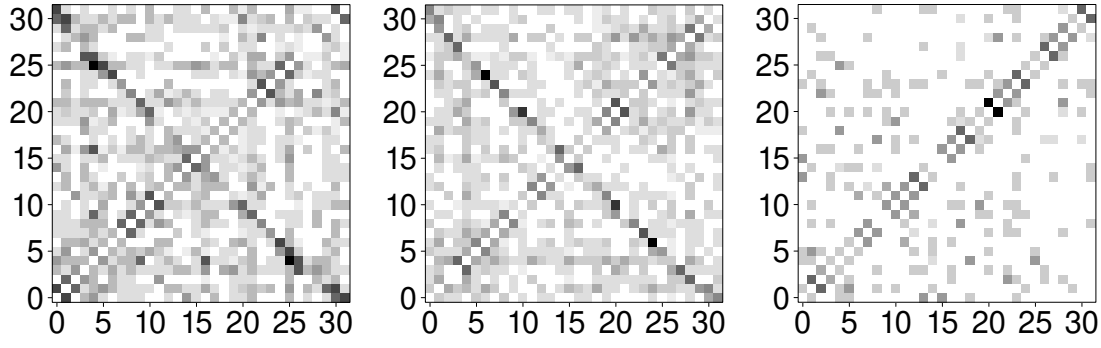
本研究在选定采样率时，分别选择了 1K、2K、10K、100K 与 500K 进行实验。不同的采样率对于统计出的通信量矩阵以及访存负载向量影响不同，若采样率过高，则 PMU 发生中断的次数会比较少，统计得到的访存事件较少，不能精确地检测到线程的访存情况；若采样率过低，则 PMU 发生中断的次数会很高，虽然能精确地把握线程的访存情况，但造成的额外开销（包括程序运行时间开销、内存开销）又过大。如表 3-1，反映了 LU 应用程序不同采样率对采样精度以及额外开销的影响。

表 3-1 不同采样率对采样精度以及额外开销的比较

采样率	访存事件数据量	额外开销
1K	3.8MB	1.98×
2K	2.1MB	1.61×
10K	346KB	1.40×
100K	38KB	1.06×
500K	6KB	1.03×

访存事件数量表示，PMU 中断产生的总访存记录数据量，该值越大对程序的访存行为统计的越精确，额外开销指，在不同采样率下比较程序使用 Perf 进行检测与程序不使用 Perf 检测的运行时间比。根据表 3-1 的信息，采样率为 2K 时，访存事件数据量为 2.1MB，大致为采样率 1K 时的一半，但远高于 10K、1000K 以及 500K 的访存事件数据量。从额外开销上来看，采样率为 2K 的程序运行时间与 10K 的运行时间差别不多，但与 1K 的运行时间相差较 10K 多一些，100K 与 500K 的运行时间相近但访存事件数量较少。综上来看，当采样率为 2K 时，访存事件数据量仍然较多，对访存事件的检测仍保持较高精度，且额外开销处于可以接受的范围，因此选择 PMU 采样率为 2K。

如图 3-5 为，LU 程序分别在 PMU 采样率 1K、2K 与 100K 下的通信量矩阵：



(a) 1K (b) 2K (c) 100K

图 3-5 不同采样率通信量矩阵比较

通信量矩阵是一个对称矩阵，矩阵中颜色越深的地方表示对应两个线程通信越多，采样率为 1K，以及 2K 的矩阵均反映了 LU 程序两条对角线对应的线程对通信较多，而 100K 的通信量矩阵只呈现出主对角线上通信频繁的情况，这反映了 100K 的访存检测精度较低，而 1K、2K 访存检测精度较高。

对于线程内存访问负载检测阶段，需要从所有的访存事件筛选出事件类型为

MEM_LOAD_UOPS_LLC_MISS_RETIRED.LOCAL_DRAM 以及 REMOTE_DRAM 的两个事件，分别表示对本地内存及远端内存的访问，这两个事件代表了线程对内存的所有访问记录。

3.3 计算映射模块

计算映射的目的是，根据先前统计得到的线程访存信息经过分组算法的计算，将线程划分为若干组。同一个组内的线程映射会到共享同一存储单元的计算核心上，实现线程到核的一一对应。本文 1.2 节提到了 3 个分组算法：Scotch、EagerMap、ChoiceMap，它们均是根据线程通信量矩阵的数值信息，将通信量大的线程划分至一个组中，或者将通信量矩阵转化为带权无向图，再对无向图进行划分。通信量矩阵或者无向图进行分组划分是一个 NP-hard[32]问题，研究者针对这种问题通常采用贪心策略、启发式算法以及强化学习构建搜索树的相关方法来解决这种线程分组问题。

本研究早期设计实现的映射优化机制，采用了 EagerMap 算法作为计算映射模块进行线程分组划分。该算法采用贪心策略并结合计算机硬件架构拓扑信息，自低向上的进行循环分组划分。该算法示意图如图 3-6 所示：

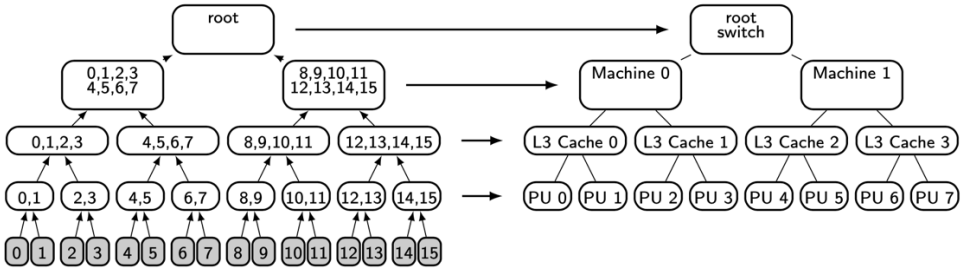


图 3-6 EagerMap 线程分组映射示意图

图 3-7 展示了将 16 个线程分组映射至 8 个核心上。该图的右半边展示了计算机硬件架构拓扑树：最底层有 8 个核心，每个核心可以放置 2 个线程，每两个核心共享一个 L3Cache，每 2 个 L3Cache 共享一个节点资源，总共有 2 个节点构成了整个计算机系统。图中左半边展示了自底向上的层次分组映射过程，根据通信量矩阵采用贪心策略选取通信量大的线程构建成一个组，从核心层开始两两分组，分到一组的两个线程放置在一个核心；然后在 L3Cache 层将上一层划分的组构建成更大的组，每个组包含 4 个线程共享一个 L3Cache 的资源；最后到节点层，所有线程被划分成 2 个组，每个组共享一个节点的所有资源，映射完毕。EagerMap 的时间复杂度大约为 $O(n^3)$。

EagerMap 充分利用了通信量矩阵中的数值信息并结合了硬件架构信息，使得绝大多数的线程间通信发生在节点内部，极大提升了访存局部性。但这样的分组方式还存在一个问题：EagerMap 在线程分组过程中仅考虑了线程间通信情况，忽视了每个线程的访存负载。仅利用通信量矩阵中的信息进行分组，可能导致访存负载高的线程被划分到一个节点内，从而造成内存拥塞问题。因此为了修正这一潜在缺陷，本研究设计并提出了 CMLB(Communication-Aware and Memory Load Balance Mapping Algorithm，

基于通信感知以及内存负载均衡的分组映射算法) 。本文将在第 4 章详细介绍该算法设计与实现的相关细节。

3.4 执行映射模块

为了根据计算映射得到的线程分组结果，将线程一一绑定至计算核上，需要用到 CPU 亲和性[33]的相关概念。CPU 亲和性是指线程或进程在某个给定 CPU 上尽量长时间运行而不被迁移到其他处理器的倾向性。

目前对 CPU 亲和性设置的方法有两类：第一类是使用 CPU 亲和性设置函数进行设置，包括 pthread 库提供的 pthread_setaffinity_np 函数，系统提供的 sched_setaffinity 函数，以及 hwloc 库提供的 hwloc_set_cpubind 函数进行亲和度设置。第二类是使用环境变量的方式进行 CPU 亲和性设置，包括 numactl 库提供方式：numactl –physcpubind，以及 gcc 编译器提供的 GOMP_CPU_AFFINITY。

本研究中在执行映射模块使用设置环境变量的方式进行 CPU 亲和性设置，具体使用 GOMP_CPU_AFFINITY 进行设置。相较于 CPU 亲和性设置函数，环境变量的设置方式更加方便。执行映射模块的具体执行步骤如下：shell 执行映射脚本首先读取计算映射模块保存在文件中的分组结果，然后将结果赋给 GOMP_CPU_AFFINITY，便完成了线程到核的绑定。另外，为了保证在执行映射脚本结束后所设置的 CPU 亲和性依然生效，需要将 GOMP_CPU_AFFINITY 设置为全局环境变量。

3.5 静态映射机制的实现

本研究所提出设计的线程映射机制是一个静态的过程，即在应用程序运行前完成线程的到计算核心的映射，在程序运行过程中不再迁移线程，因此在程序运行时不会带来任何的额外开销。

在应用程序运行前，需要对程序进行一个预运行的过程，以检测统计程序访存信息。之后根据检测统计得到的访存信息计算线程分组，并根据分组结果完成线程的绑定，该过程也就是图 3-1 中映射机制的整个执行流程。因此，整个静态映射机制是在应用程序运行前，通过检测统计程序的访存信息，结合硬件架构信息计算线程分组，最后线程根据分组结果设置环境变量对线程进行绑定，完成整个过程。并且静态映射机制在无需修改应用程序源代码的情况下，完成在用户态下对线程的绑定，具有程序通用性。

相较于静态映射机制的程序运行前完成线程的绑定，作为动态映射机制的代表 kMAF 在程序运行过程中，以一定周期对线程通信量进行检测并动态调整线程的放置。动态映射机制虽然不需要预运行应用程序对其进行检测相对减少了额外开销，但需要在程序运行时，动态检测线程通信并迁移线程，这会带来不小的额外开销，且动态剖析线程访存情况实现难度较大，所以静态映射机制的额外开销是在程序运行前，动态映射机制的额外开销是在程序运行中。为了进一步比较分析静态映射机制与动态映射

机制的优化效果，本文第 5 章将对本文提出的静态映射机制与 kMAF 的动态映射机制在优化性能上进行对比测试，并对静态映射机制的额外开销进行分析。

3.6 本章小结

本章首先在设计映射优化机制前，根据 2.3 节线程映射优化的定义以及问题描述，提出了静态映射优化机制需要解决的 3 个问题：访存检测统计问题，线程分组划分问题以及执行映射问题。之后根据这 3 个问题，提出并设计了线程静态映射优化机制的总体框架和执行流程。并简要介绍了构成映射优化机制的 3 个模块：线程访存检测模块、计算映射模块、执行映射模块。随后详细介绍了线程访存检测模块，包括通信量检测统计和内存访问负载检测统计的设计与实现方案，以及相关参数设置。然后详细介绍了计算映射模块，包括 EagerMap 算法思路以及利弊分析，并提出一种改进的算法 CMLB。然后介绍了几种设置 CPU 亲和度的方案，并选择通过设置环境变量完成对 CPU 亲和度的设置。最后简要叙述了静态映射机制的实现过程与特点，以及与动态映射机制在额外开销上的区别。

4　映射分组算法 CMLB 的设计与实现

针对本文 2.3 节提到的多线程并行应用程序在 NUMA 架构计算机上运行存在的问题，本文第 3 章设计并实现了多线程静态映射优化机制来解决上述问题。在本研究提出的线程映射优化机制中，需要根据线程间通信量以及线程内存访问负载，通过计算将通信量大的线程划分至一个节点内，同时保证每个计算节点的内存访问负载大致相同。本研究在第 3 章中设计实现的线程映射优化机制，其中计算映射模块最初使用 Eagermap 分组算法，计算得到线程与核的对应关系。然而根据本文 3.3 节计对算映射模块的介绍，以及对 Eagermap 算法的分析，本研究选择自己设计一种避免过多的跨节点通信同时保证节点间内存访问负载均衡的分组算法——CMLB 映射算法。

4.1 CMLB 算法设计

根据 2.3.1 节对线程映射优化的定义：通过对线程的合理放置，在平衡线程间通信的前提下，使得每个节点的内存带宽保持均衡，达到优化程序性能的效果。因此分组算法的设计原则为：在尽可能少的跨节点访存的前提下，保证每个节点内存访问负载均衡。根据这一设计原则，对线程分组算法进行设计，整个设计过程采用自顶向下的方法，具体过程描述如下：

（1）确定所有线程的分组数量，循环处理每个组的线程划分。为了充分利用计算资源，在本研究中将所有运行的线程分布在所有节点上，这样可以使得每个节点内部的硬件资源得到充分利用，因此线程的分组数量为计算机节点个数。节点数的获取可根据 3.1 节中的映射优化机制中的硬件信息检测模块，使用 hwloc 库检测硬件架构信息，包括计算机的节点数量、每个节点包含的各级 Cache 数、核心数。分组数量确定后，循环处理每个组的线程划分，具体一个组的划分过程在（2）中描述，每个组处理完成后，得到所有线程的分组结果，分组算法执行完毕。

（2）结合 3.2 节线程访存检测模块得到信息，生成一个组。根据（1）中的分组数得到一个组内的线程数量，然后选择一个待分组的线程放置在正在处理的组中，结合通信量矩阵以及访存负载向量从剩余待分组的线程中选取一个放入该组，具体选择过程在（3）中描述。重复上述步骤，直到该组内的线程数量达到上限。

（3）从待分组的线程中选择最合适的线程放入组中。执行流程如图 4-1 所示。主要借鉴了 Eagermap 中的贪心策略，根据在（2）中组内其他所有的线程集合 t = {t_1, t_2, t_3, ….}，结合通信量矩阵 CommMatrix，计算所有待分组的线程与 t 的通信量数值并从高到低排序，具体过程如下：遍历选取待分组的线程中的一个线程 t_w，通过 CommMatrix 分别计算 t_w 与 t 内所有线程的通信量并相加得到通信量总和 C，如公式 4-1 所示：

$$C = \sum_{i=1}^{n} CommMatrix[t_w][t[i]] \tag{4-1}$$

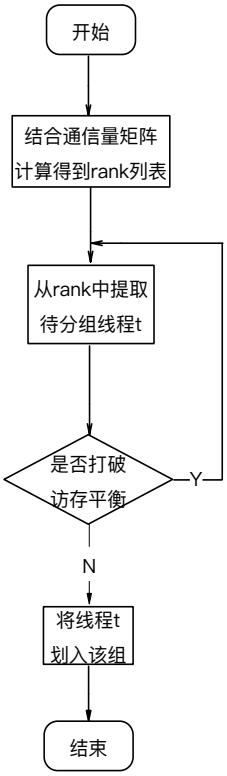


图 4-1 待分组线程的划分流程图

式中 t[i]表示已分组的线程集合 t 中的一个线程，n 表示 t 中线程个数。然后对于剩余待分组的线程重复上述步骤，分别计算得出通信量总和，计算完毕后将所有通信量总和值从高到低排序，得到通信量总和及所属线程 id 的 rank 列表: [[C_1, t_{w1}], [C_2, t_{w2}], …]。取出 rank 中的第一个元素即与线程集合 t 通信量总和最大的待分组线程 t_{w1}，结合访存负载向量判定若将 t_{w1} 放入该组是否会破坏该组内的访存平衡，访存平衡判定算法在（4）中描述。若判定为假即不会破坏访存平衡，则将 t_{w1} 加入该组；若判定为真，即会破坏访存平衡，则 t_{w1} 不加入该组，选取 rank 中的下一个待分组线程 t_{w2} 进行访存平衡判定。重复上述步骤，直到将一个待分组的线程加入组内。

（4）组间访存平衡判定。在（3）中将一个待分组的线程划分进入到一个组之前，还需判定该线程的加入是否会破坏整体的访存平衡。在（1）中确定分组数量后，需要根据线程访存负载向量计算得出每个组的 Aml（Average memory load，平均访存负载），如公式 4-2 所示：

$$Aml = \frac{1}{groups}\sum_{i=1}^{n} AccVector[i] \tag{4-2}$$

式中 groups 表示分组数量，AccVector[i]代表访存负载向量中，线程 i 的负载值，n 表示访存负载向量长度。Aml 表示了每个组的访存负载期望，访存平衡算法使用 Aml 来判断是否破坏了平衡，具体过程如下：当（3）中选定了一个待分组的线程 t_{w1}，先将其加入分组，计算该组的目前访存负载值以及达到 Aml 的剩余负载期望 lastLoad，同

时根据（2）中的组内平均线程数计算出该组的剩余线程数量 n。然后将剩余待分组的线程按照对应 AccVector 中的值从高到低排序，并计算得到前 n 个访存负载总和 maxLoad 及后 n 个访存负载总和 minLoad，若 lastLoad 值在 minLoad 与 maxLoad 之间，则判定 t_{w1} 的加入未打破整体的访存平衡，因为剩余待分组线程中部分访存负载之和，可以达到 lastLoad 值从而维持所有组的访存负载平衡。相反若 lastLoad 值不在 minLoad 与 maxLoad 之间，则剩余待分组线程无论怎么组合其访存负载之和也达不到 lastLoad 值从而打破整体的访存平衡，并将破坏访存平衡的线程 t_{w1} 放入列表 OverLoadList 中。当 t_{w1} 为将要加入该组的最后一个线程，则需在 OverLoadList 查找 t_{w1} 是否存在其中，若存在则判定打破访存平衡，否则判定未打破平衡。执行流程如图 4-2 所示。

以上 4 个过程呈自顶向下的关系，过程（1）为算法主流程，执行（1）的过程中会层层调用后面的过程。在过程（3）中使用了贪心策略将与组内线程通信量大的待分组线程准备划入该组，并在过程（4）中对新线程的加入是否打破访存平衡做了判断，因此（3）和（4）过程分别体现了 CMLB 算法使通信频繁的线程放置在一个节点同时维持节点内访存平衡的设计原则。具体的伪代码实现将在 4.3 节展示。

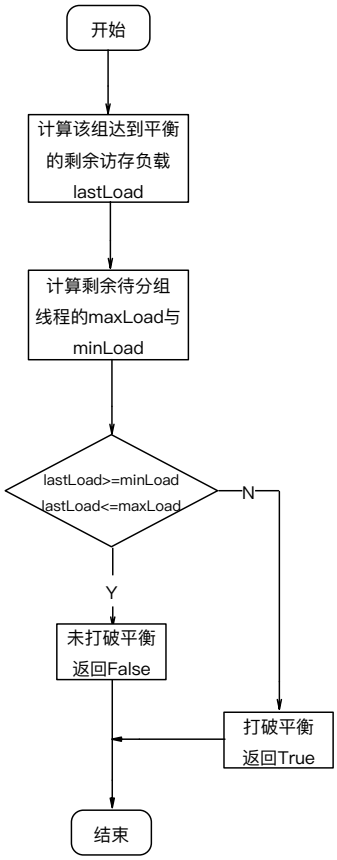


图 4-2 访存平衡判定流程图

4.2 CMLB 算法实现

根据 4.2 节的 CMLB 算法设计流程，本节将从展示 CMLB 算法的伪代码实现。

4.2.1 顶层算法

算法 4-1：MainAlgroithm

```
1  输入：通信量矩阵 CommMat，访存负载向量 AccVector，硬件架构信息 configMap
2  输出： 线程划分结果 map_res
3  全局变量：CommMat、AccVector、num_nodes、per_group_tids、map_res、overLoad_list
4  Begin
5     num_nodes = configMap['nodes']     //获取节点数量
6     num_threads = len(CommMat)         //获取线程数量
7     per_group_tids = num_threads // num_nodes    //计算每一组的线程数量
8     map_res = []; overLoad_list = []                    //初始化分组结果列表以及超载列表
9     last_threads = [x for x in range(num_threads)]     //初始化剩余待分组线程列表
10    for i in range (num_nodes) :
11          group, last_threads = GenerateOneGroup(per_group_tids, last_threads) //  生成一个组
12          map_res.append(group)     //将生成的组存入结果列表
13    end for
14    return map_res
15 End
```

算法 4-1 对应 4.2 节中的（1）过程，是 CMLB 算法的主函数。输入参数为：通信量矩阵、访存负载向量以及硬件架构配置信息，输出为分组划分结果。算法 5-9 行计算每组线程数以及初始化待分组线程列表、分组结果列表等变量，10-12 行循环处理每一组，并将每组划分结果加入最终结果列表。

4.2.2 生成一个组

算法 4-2：GenerateOneGroup

```
1  输入：每组的线程数 per_group_tids，剩余待分组线程列表 last_threads
2  输出：一个组的划分结果 group，剩余待分组线程列表 last_threads
3  Begin
4     group = last_threads[0]   //  初始化组列表，将待分组线程列表的首个线程加入组中
5     last_threads.pop(); overLoad_list.clear()   //移除剩余待分组列表的队首元素并清空超载列表
6     for i in range(per_group_tids) :
7           tar_tid = SelectOneThread(cur_grouped ,last_threads)   //  从待分组列表中选取一个线程
8           group.append(tar_tid)     //  将选出的线程加入组中
9           last_threads.remove(tar_tid)   //  从待分组线程列表中移除选出的线程
10    end for
11    return group, last_threads
12 End
```

算法 4-2 对应 4.2 节中的（2）过程，输入一个组的长度以及剩余待分组线程列表，返回一个组的划分结果和经过更新的剩余待分组线程列表。其中 4-5 行初始化组列表和更新剩余待分组列表，7-9 行以循环的方式从剩余待分组线程中选择一个加入 group，并更新剩余待分组列表，直到组内线程数达到上限。

4.2.3 选择一个线程

算法 4-3：SelectOneThread

```
1  输入：当前已分组的线程列表 cur_grouped，剩余待分组线程列表 last_threads
2  输出：将要加入组中的线程 tar_tid
3  Begin
4     rank_ls = []   // 初始化与已分组线程通信量的 rank 列表
5     for last_t in last_threads :
6           sum_ = 0
7           for grouped_t in cur_grouped:
8                 sum_ += CommMat[grouped_t][last_t]   // 计算与已分组线程列表的通信量总和
9           end for
10          rank_ls.append([sum_, last_t])
11    end for
10    rank_ls = sorted(rank_ls, reverse=True)   // 从高到低排序
11    for pair in rank_ls :
12       if not justifyLoad(pair[1], cur_grouped, last_threads)   // 判断是否打破访存平衡
13             return pair[1]
14       else
15             continue
16       end if
17    end for
18    return rank_ls[0][1]   // 若所有待分组线程均未满足条件，则返回通信量之和最大的线程
19 End
```

算法 4-3 对应 4.2 节中的过程（3），选择合适的线程加入分组。输入当前已分组的线程列表以及剩余待分组线程列表，输出将要加入分组的线程。算法 4-10 行以循环的方式计算了每个待分组线程与已分组线程列表的通信量总和，并排序后记录在一个 rank 列表中，11-15 行以循环的方式遍历 rank 列表，从中取出待分组线程并判断是否打破访存平衡。若所有待分组线程均打破访存平衡，则选择 rank 列表中排在第一位线程，如第 18 行所示。最后返回加入分组的线程。

该算法 11-15 行的过程使用了贪心策略，选择当前与已分组线程列表通信量总和最大的待分组线程，进行访存平衡判断并准备加入分组。

4.2.4 判定是否负载平衡

算法 4-4：JustifyLoad

```
1  输入：候选线程 tid, 当前已分组的线程列表 cur_grouped, 剩余待分组线程列表 last_threads
2  输出：True or False
3  Begin
4     cur_load = AccVector[tid]
5     for grouped_t in cur_grouped :
6           cur_load += AccVector[grouped_t]   // 计算当前组内访存负载
7     end for
8     last_load = sum(AccVector) / num_nodes - cur_load   // 计算达到负载上限的剩余负载
9     last_num_tids = per_group_tids – len(cur_grouped)    // 计算该组所需的剩余线程数
10    if   last_num_tids == 1:
11          if tid in overLoad_list :   // 若组内剩余一个线程的位置，判断候选线程是否在超载列表
12                return True          // 若存在表示该线程曾经打破负载平衡，则现在加入也会打破
13          else
14                return False
```

```
15          end if
16      end if
17      f = sorted(last_threads.remove(tid),reverse=True)   // 计算待分组线程最大最小负载
18      max_load = sum(f[ : last_num_tids])
19      min_load = sum(f[-last_num_tids:])
20      if last_load >= min_load and last_load <= max_load: // 判断剩余负载是否在范围内
21          return False
22      else
23          return True
24      end if
25  End
```

算法 4-4 对应 4.2 节中的过程（4），判定新线程的加入是否打破组间平衡。输入候选线程 tid、已分组的线程列表和待分组的线程列表，输出判定结果 True 表示会打破组内访存平衡，False 表示不会打破。算法 4-6 行计算当前已分组的线程访存负载，8-9 行计算组内达到平均访存负载的剩余负载，以及该组的剩余线程数量。10-16 行表示：如果候选线程为组内最后一个将要加入的线程，判段该线程是否存在于 overLoad_list 中，若存在表示该线程曾经打破了访存平衡，现在若加入也会打破组间平衡，因此返回 True。17-19 行计算待分组线程能提供的最大以及最小访存负载范围，并在 20-23 行用组内剩余负载 last_load 判断其是否在 min_load 与 max_load 之间，若为真则说明候选线程的加入，剩余待分组线程能够使得该组达到平均访存负载值，并判定不会打破组间访存平衡。若为假说明剩余待分组线程不论如何，也不能使得该组达到平均访存负载值，故判定会打破组间访存平衡。

算法 4-4 体现了 CMLB 算法的访存负载平衡策略，对候选线程的加入使用最值区间来决策：计算候选线程加入后组内的剩余负载期望 last_load，并计算剩余待分组线程能提供的最大和最小访存负载值 max_load 与 min_load，最后判断 last_load 是否在 max_load 与 min_load 区间范围内。

4.3 理论效果测试

本节将对 CMLB 算法进行理论效果测试，并与 Eagermap 从跨节点通信量和节点访存负载平衡度这两个指标进行对比。

本节所进行的分组算法理论效果测试是指：在分组算法进行计算映射后，得到了确定线程分组结果，然后根据分组结果计算上文提到的两个指标。在本研究中每组对应一个 NUMA 节点，最终将每一组的线程映射到一个节点内的核心上，因此。跨节点通信量用组之间的线程通信量表示，如公式 4-3、4-4 所示：

$$RemoteComm = \sum_{i=1}^{n}\sum_{j=i+1}^{n} CommVal(map[i], map[j]) \tag{4-3}$$

$$CommVal(g1,g2) = \sum_{m=1}^{n1}\sum_{n=1}^{n2} CommMat[g1[m]][g2[n]] \tag{4-4}$$

式 4-3 中 *RemoteComm* 表示跨节点通信量，n 为节点数（组数），*CommVal* 表示计算

两个组之间的通信量，*map* 表示分组结果，包含所有组的线程。式 4-4 为 *CommVal* 的计算过程，g1、g2 表示两个组，n1、n2 分别表示两个组的线程数。*CommMat* 为通信量矩阵，*CommVal* 最终通过累假两个组的线程之间的通信量计算得出。*RemoteComm* 对应于第 5 章实验中的 QPI 指标，即跨节点访存量，该指标用于评价分组算法，*RemoteComm* 越小程序性能收益越高。

节点访存负载平衡度也是评价分组算法的指标之一，该值用每个组访存负载的标准差计算得来，每个组的访存负载为该组内所有线程的访存负载之和得到，计算过程如公式 4-5、4-6:

$$Load_{std} = \sqrt{\frac{\sum_{i=1}^{n}(L_i - L)}{n}} \tag{4-5}$$

$$L_i = \frac{1}{m}\sum_{j=1}^{m} AccVector[L_i[j]] \tag{4-6}$$

式 4-5 中 $Load_{std}$ 表示组之间访存负载的标准差，n 为组数，L_i 表示第 i 组的访存负载值，L 表示所有组的访存负载平均值。式 4-6 为一个组访存负载值的计算过程，m 表示该组的线程数，*AccVector* 为访存负载向量。$Load_{std}$ 对应于第 5 章实验中的 imbalance 指标，表示内存访问的不平衡度，$Load_{std}$ 越小程序性能收益越高。

在对应用程序分别使用 CMLB、Eagermap 进行计算映射得到各自的线程分组结果后，根据两种算法的分组结果计算上述两个指标，进行对比分析。

本测试使用 NPB 中 SP、BT、LU 三个应用程序进行测试，结果如下：

表 4-1 SP 程序理论测试结果

分组算法	RemoteComm	$Load_{std}$
CMLB	1.99×10^8	139.37
Eagermap	1.90×10^8	33455.01

表 4-2 BT 程序理论测试结果

分组算法	RemoteComm	$Load_{std}$
CMLB	3.51×10^7	462.69
Eagermap	3.40×10^7	29310.60

表 4-3 LU 程序理论测试结果

分组算法	RemoteComm	$Load_{std}$
CMLB	2.27×10^8	968.89
Eagermap	1.89×10^8	36440.19

通过表 4-1、4-2、4-3 的测试结果可以看出，CMLB 在 3 个应用程序的 RemoteComm 指标上略高于 Eagermap，但均处于一个数量级，而对于 3 个应用程序的 $Load_{std}$ 指标

CMLB 相比 Eagermap 均有大幅度降低。综合来看，CMLB 在跨节点通信量与 Eagermap 相差不大的情况下，能够大幅度减少节点间的访存负载差异，维持了节点访存负载的平衡。因此从理论效果来看，CMLB 的分组效果整体上优于 Eagermap。

在本文的第 5 章中，将对这两个算法在更多的 Benchmark 上进行实际效果测试，并对测试效果进行对比分析得出最终的结论。

4.4 本章小结

本章首先根据 2.3.2 节中线程映射优化的定义，制定了 CMLB 分组算法的设计原则：在尽可能少的跨节点访存的前提下，保证每个节点内存带宽均衡；并基于 CMLB 的设计原则与目标，以自顶向下的方式设计出 CMLB 的执行流程；之后根据算法的执行流程，将每个步骤使用代码实现，并介绍了代码的相关实现细节。最后从 RemoteComm 和 $Load_{std}$ 这两个指标对比分析了 CMLB 与 Eagermap 的理论优化效果，得出 CMLB 的理论效果整体上优于 Eagermap 的结论。

5　映射机制的性能测试与分析

本章将在多核处理器平台上，使用相关多线程并行应用程序 benchmark 评估多线程静态映射优化机制的整体优化效果。实验的具体细节如下。

5.1 实验环境

5.1.1 实验平台

本研究使用由 Intel Xeon E7-4809 处理器构成的多核计算机平台进行实验。实验平台的参数配置如表 5-1 所示。

表 5-1 计算节点参数配置

属性	配置参数
Architecture	2 nodes, 1 socket/node, 8processors/socket
Processors	Intel Xeon E7-4809
Cache	8*(32KB+32KB)L1, 8*256KB L2, 20MB L3
Memory	32GB DDR3, 页大小 4KB

该计算节点是一个典型的 NUMA 架构平台，它包含 2 个 NUMA 节点，每个节点包含 1 个 socket，每个 socket 拥有 8 个同构 CPU 核心。每个核心上有 32KB 的 L1 指令 Cache 与 L1 数据 Cache，以及 256KB 的 L2Cache。8 个核心共享一个大小为 20MB 的 L3Cache，每个节点拥有大小 16GB 的内存，可被节点内的所有核心共享。计算平台的总内存为 32GB，页大小为 4KB。处理器中使用超线程技术，即每个核心可以运行 2 个线程。计算平台安装了 Linux 的 Ubuntu 系统，内核版本为 5.7.2。

5.1.2 应用程序

本研究分别使用 rotor35-omp 程序、NPB 程序集以及 PARSEC 程序集进行实验评估。

1）rotor35-omp 程序

本研究使用“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，对本研究提出的映射优化机制在该项目程序的优化效果进行测试。rotor35 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、数值求解以及后处理[34]。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。

$$\frac{\partial Q}{\partial t}+\frac{\partial \mathbf{F}_c}{\partial x}+\frac{\partial \mathbf{G}_c}{\partial y}+\frac{\partial \mathbf{H}_c}{\partial z}=\frac{\partial \mathbf{F}_v}{\partial x}+\frac{\partial \mathbf{G}_v}{\partial y}+\frac{\partial \mathbf{H}_v}{\partial z}+\mathbf{I} \tag{5-1}$$

其中，Q 为守恒型求解向量；F_c、G_c、H_c 分别为沿着 x、y、z 三个坐标方向的对流向量；F_v、G_v、H_v 分别为沿着 x、y、z 三个坐标方向的粘性向量；I 为源项。算例对

公式中求解向量 Q 的 6 个物理量进行求解。使用 FMG（Full Multi-Grid）三重网络循环，依据多重网格法[35]的标准在粗网格、中网格以及细网格之间进行残差限制和插值，其处理流程如图 5-1 所示。直到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。

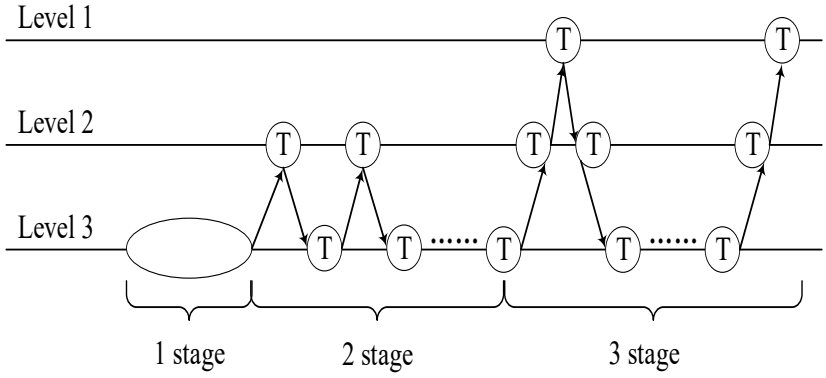


图 5-1 rotor35 程序多重网格法执行示意图

本研究中使用的 rotor35 程序为原始 MPI 版本的修改版，原始 rotor 程序使用进程并行计算 36 流道，即一个流道由一个进程计算，进程之间会发生通信。由于本研究所使用的计算机平台为共享内存环境，只允许一个 MPI 进程运行。因此为了适应实验平台，需要将原始的 rotor35 程序修改为只计算一个流道的单 MPI 进程程序，rotor35 在单个进程下执行时同时使用 OpenMP 多线程进行并行，最终将 rotor35 多进程并行的 MPI 程序，修改为单进程下多线程并行的 OpenMP 程序，简称 rotor35 程序。

2）NPB-OMP 程序集

本研究同时使用 NPB-OMP 程序集中的部分应用程序进行实验。NPB-OMP 是 NAS 并行基准测试程序[36]（NAS Parallel Benchmark，NPB）的 OpenMP 版本实现[37]。NPB 是以流体力学计算为主的应用程序集。NPB 程序集可用于评估高性能计算机以及众核处理器平台的性能，是一套至今仍在维护的基准测试程序。NPB-OMP 程序集共有 8 个程序组成，其中包括 5 个核心程序和 3 个模拟程序：IS、EP、FT、MG、CG、LU、SP、BT。8 个程序的功能和特点描述如下。

核心程序：

（1）IS（Integer sort）用于解决基于存储桶排序的整数排序，通信模式并未体现规律性。

（2）EP（Embarrassingly parallel），用于计算 Gauss 伪随机数，其通信模式为线程间不进行任何通信。

（3）CG（Conjugate Gradient），求解稀疏对称正定矩阵的最小特征值。其通信模式为全交换通信，因此其通信模式并未体现规律性，属于访存密集型程序。

（4）MG（MultiGrid）是多重网格计算的典型程序，其通信模式兼具短距离与长距离通信，属于访存密集型程序。

（5）FT（Fourier Transform），快速傅立叶变换程序。用于求解三维偏微分方程，属于访存密集型程序。

三个流体力学模拟程序：

（1）LU（Lower upper triangular），用于求解块稀疏方程组。通信模式上具有相邻线程以及远端线程之间通信频繁特点。

（2）SP（scalar penta-diagonal），用于求解 5 对角线方程组。通信模式为相邻线程通信频繁的特点。

（3）BT（Block Tri-Diagonal），用于求解 3 对角线方程组。通信模式与 SP 相似，BT 也具有相邻线程通信频繁的特点，同时兼具全交换通信的特点。

实验评估时选择 3 个流体力学模拟程序：LU、SP、BT，以及核心程序中的 CG 作为待测应用程序。NPB-OMP 剩余的 4 个程序由于其平均运行时间过短不好展示映射机制的效果，所以将其舍弃。

3）PARSEC 程序集

PARSEC[38]是一套多线程并行应用程序集，主要用于多核共享存储处理器并行应用研究的性能评测。由英特尔与普林斯顿大学合作开发，包含金融分析、数据挖掘、图像渲染、计算机视觉等众多应用领域的基准测试程序。本研究选用其中几个具有代表性的程序，程序功能和运行特点描述如下：

（1） Streamcluster ，用于处理在线聚类问题，主要针对连续产生的大量流数据，属于数据挖掘领域。输入数据量较大时具有通信受限的特点，其通信模式具有小范围线程组通信频繁的特点。

（2）Facesim，用于处理面部模拟问题，属于计算机视觉领域。其通信模式为相邻线程通信较频繁，且具有全交换通信的特点。

（3）Fluidanimate，用于模拟计算流体流动，属于流体动力学领域，具有间隔为 5 的两个线程通信频繁的特点。

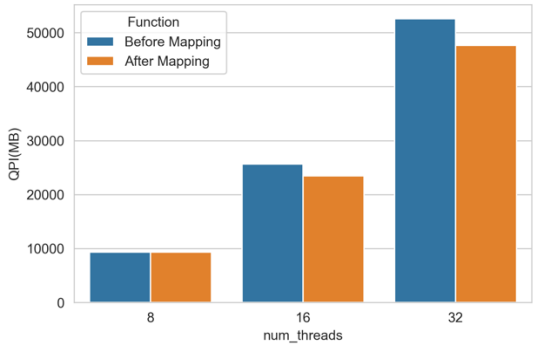
5.2 静态映射优化机制的整体优化效果测试

本节将对多线程映射优化机制的优化效果进行测试。本文使用 5.1.2 节中的 rotor35 程序和 NPB-OMP 程序集的部分程序以及 PARSEC 程序集的部分程序作为评估对象，根据本文 2.3.2 节中映射机制理论优化效果的介绍，选择运行时间、QPI、各个节点内存访问的不平衡度以及内存访问延迟这 4 个指标，对整体优化效果做出评价，并得出相关结论。

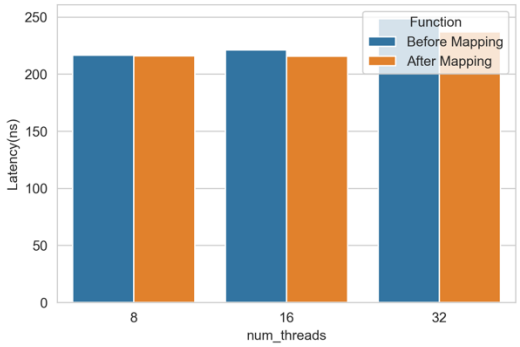
5.2.1 映射机制对 rotor35-omp 程序的性能优化测试

在本节将对本研究设计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，其中映射机制的计算映射模块选用 CMLB 分组算法。实验中，选择程序总体运行时间、QPI 以及内存延迟作为评测指标，分别比较映射前与映射后以上指标的性能差异。最终对映射机制的整体效果做出评价，并给出结论。

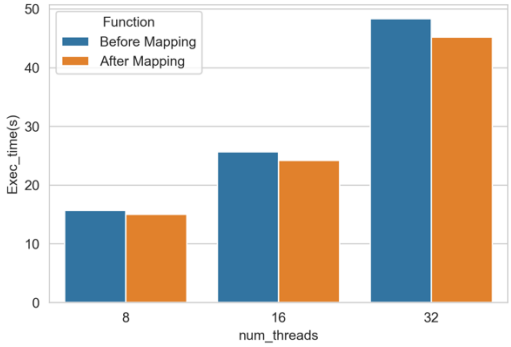
在本文5.1.2节中提到本研究使用的rotor35程序是单进程下多线程并行的OpenMP程序。结合实验平台参数配置，测试时分别选择8线程、16线程以及32线程版本的rotor35程序，不同线程版本的rotor35程序其数据规模也不同。实验时分别多次运行映射前以及映射后的程序，最终的得到结果为多次实验的平均值。实验结果如图 5-2 所示。



(a)QPI



(b) Latency



(c)运行时间

图 5-2 不同线程数 rotor35 程序映射优化前后性能对比

根据图 5-2 所示，映射机制在线程数越多时优化效果越明显。对于图（a），映射优化机制为32、16线程的rotor35程序均带来QPI值的最高9.33%的减少，具有一定的优化效果，而对于8线程rotor35程序未有明显降低；对于图（b），映射优化机制也使得32、16线程的rotor35程序的内存延迟有所降低，最高下降约4.83%，而8线程的rotor35程序也未有明显降低。这是因为程序的数据规模随着线程数增大而增大，程序的访存次数也随之增多，从而导致线程间通信的不平衡性与节点间内存带宽的不平衡性越来越大，映射机制的优化效果也就越好；从图（c）总体运行时间来看，32、16线程的rotor35程序在映射后运行时间明显减少，最高下降约6.25%，8线程的rotor35程序运行时间未有明显下降，这也应证了上述观点。

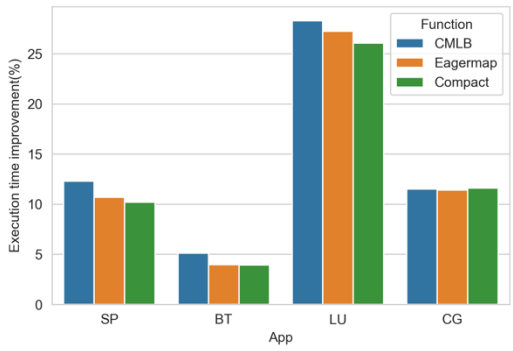
实验证明，映射机制能平衡线程间的开销使得QPI值明显降低，同时也能平衡节点间内存带宽使得内存延迟也明显降低，最终使得程序的总体运行时间明显降低，并且线程数越大时优化效果也越明显。因此，映射机制对于rotor35程序的优化是有效的。

5.2.2 CMLB 算法应用于映射机制的性能测试

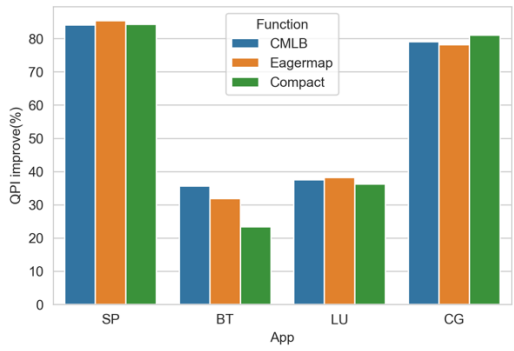
本节对 CMLB 算法的性能进行实验测试，实验时分别选用不同的分组算法实现到映射优化机制的计算映射模块中，生成几个具有不同分组算法的映射优化机制，并在 5.1.2 节中的公开程序集上进行测试，对比运行时间、QPI、节点内存带宽不平衡度以及内存访问延迟这 4 个指标的性能差异。在具体测试时，选用 CMLB、Eagermap 以及 OpenMP 提供的 Compact 映射模式作为不同的线程分组算法，同时选择 NPB-OMP 程序集中的 SP、BT、LU、CG 程序，以及 PARSEC 程序集中的 Streamcluster、Facesim 和 Fluidanimate 程序进行测试。

1）NPB-OMP 程序集的测试

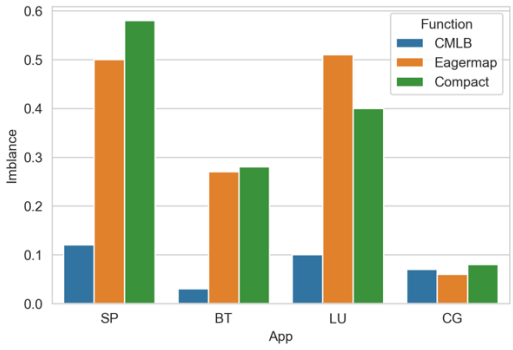
首先选用问题规模为 B 的 NPB-OMP 程序集进行测试，测试程序为 SP、BT、LU、CG，根据问题规模将线程数设置为 32。测试结果如图 5-3 所示：



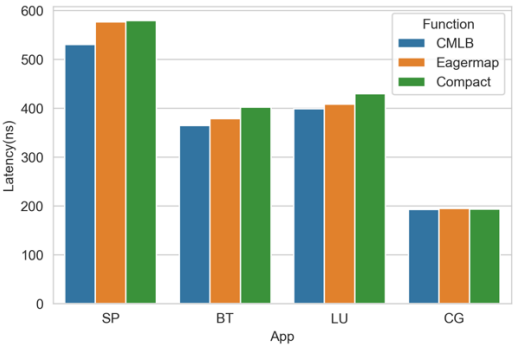
（a）运行时间加速比



（b）QPI 性能增益



（c）内存带宽不平衡度



（d）平均内存延迟

图 5-3 NPB-OMP 程序集在不同分组算法下的性能对比

对 NPB-OMP 程序集的测试指标分别为：运行时间、QPI、内存带宽不平衡度及平均内存延迟，其中对于运行时间与 QPI 指标分别计算不同分组算法相较于映射前的性能增益值，得到运行时间加速比和 QPI 性能增益，这两个值越大说明优化效果越好，内存带宽不平衡度与平均内存延迟的值越小说明优化效果越好。

根据图 5-3 所展示的实验结果，从整体上看，使用 CMLB 算法的映射机制在 SP、BT、LU、CG 程序相较于映射前具有明显的优化效果，运行时间加速比最高达到约 28.29%，QPI 最高下降约 84.03%。

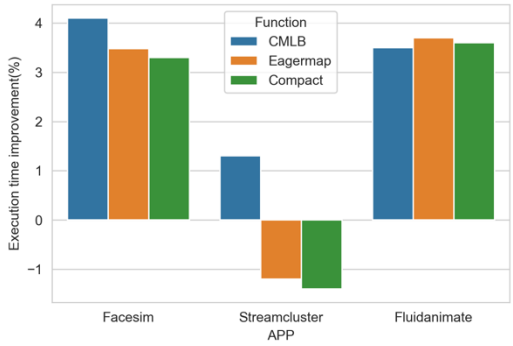
此外通过与其他算法的对比，对于 SP、BT、LU 程序，CMLB 算法在 QPI 性能增益上与 Eagermap、Compact 相差不多的情况下，内存带宽不平衡度相较于 Eagermap、Compact 大幅降低，其中对于 BT 程序下降明显，相较于 Eagermap、Compact 分别下降约 88%与 89%。这也使得平均内存延迟有所降低，其中 SP 程序平均内存延迟下降明显，较 Eagermap、Compact 分别下降 8.1%和 8.4%。以上指标的性能差异最终使得 SP、BT、LU 这 3 个程序的运行时间加速比较 Eagermap、Compact 分别获得平均 2.03%与 2.20%的提升。

然而对于 CG 程序，CMLB 算法相较于 Eagermap、Compact 并未取得任何指标上的提升，这是因为 CG 的各个线程对内存的访问量也大致相同，内存带宽始终处于较为平衡的状态使得内存延迟基本持平，并且由于在 QPI 指标中，CMLB、Eagermap、Compact 这三种方法的性能增益相差不多，导致最终的运行时间加速比也相差无几。因此对于 CG 程序直接使用 OpenMP 提供的 Compact 映射模式是最佳选择。

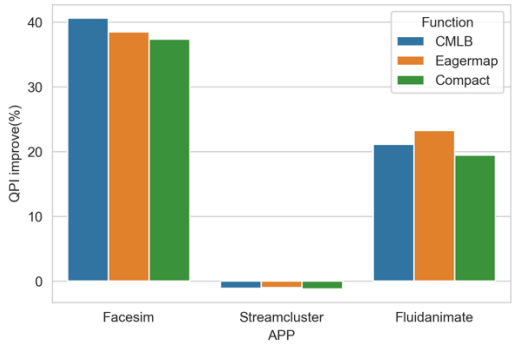
因此 CMLB 算法对于 NPB-OMP 测试集中的 SP、BT、LU、CG 程序较映射前优化效果明显，并且在 QPI 指标与 Eagermap 及 Compact 相差不多的情况下，使得内存带宽不平衡度大幅下降以及平均内存延迟降低，最终使得程序运行时间加速比有明显的提升。所以 CMLB 在 NPB-OMP 测试集的整体性能优于 Eagermap 及 Compact，这也应证了 4.4 节中的理论效果测试。

2）PARSEC 程序集测试

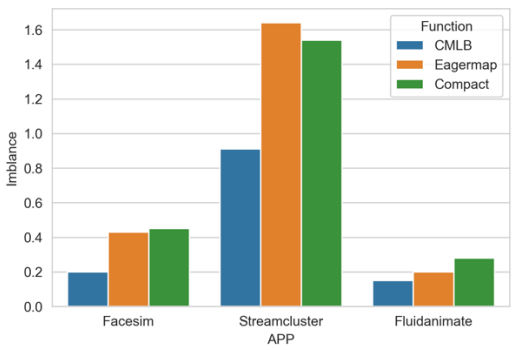
选用 PARSEC 程序集进行测试，测试程序为 Facesim 、Streamcluster、和 Fluidanimate，输入数据规模为默认数据规模，并将线程数设置为 32。测试结果如图 5-4 所示：



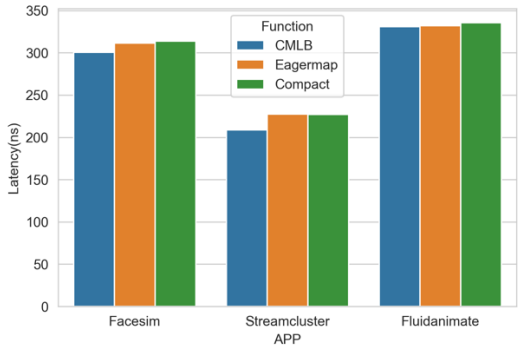
（a）运行时间加速比



（b）QPI 性能增益



（c）内存带宽不平衡度



（d）平均内存延迟

图 5-4 PARSEC 程序集在不同分组算法下的性能对比

对 PARSEC 程序集的测试指标与 NPB-OMP 程序集相同。根据图 5-4 所展示的实验结果，从整体上看，使用 CMLB 算法的映射机制在 Facesim 、Streamcluster、和 Fluidanimate 程序相较于映射前具有一定的优化效果，运行时间加速比最高达到约 4.1%，QPI 最高下降约 40.1%。

通过与其他算法的对比，对于 Facesim 程序，CMLB 算法在 QPI 性能增益上与 Eagermap、Compact 相差不多的情况下，内存带宽不平衡度相较于 Eagermap、Compact 分别降低约 50.00%和 50.02%，使得平均内存延迟相较于 Eagermap、Compact 分别下降约 3.5%和 3.6%，最终的运行时间加速比相较于 Eagermap、Compact 分别获得 0.72%与 0.91%的提升。

对于 Streamcluster 程序，3 种方法在 QPI 性能增益上均出现了少许的负增长，数值上相差不多均值为 1.1%，而内存带宽不平衡度相较于 Eagermap、Compact 下降明显，分别下降了 44.5%和 40.9%，使得平均内存延迟相较于 Eagermap、Compact 均下降了 8.3%，而最终的运行时间加速比 CMLB 较映射前提升了 1.3%，Eagermap、Compact 均分别下降了 1.2%和 1.4%。这是因为这 3 种方法在 QPI 指标上较映射前均出现了负增长，再加上 Eagermap 和 Compact 内存带宽不平衡使得内存延迟过高，因此导致这两种方法未能对 Streamcluster 程序产生优化效果，而 CMLB 平衡了节点间内存带宽，使得内存带宽不平衡性大幅下降，平均内存延迟明显降低，最终使得最终的运行时间加速比获得了提升。

而对于 Fluidanimate 程序，与 NPB-OMP 中的 CG 程序情况类似，在 QPI 与内存带宽不平衡度指标中 3 种方法在数值上相差不大，同时平均内存延迟受内存带宽不平衡度的影响也基本持平，因此总体的运行时间加速比 3 种方法相差不多。与之前结论类似对于 Fluidanimate 程序以及 CG 程序在 QPI 与平均内存延迟指标上使用 3 种方法均未得到有效提升时，直接使用开销更小的 Compact 映射方式是最佳选择。

因此 CMLB 算法对于 PARSEC 程序集中的 Facesim 、Streamcluster、和 Fluidanimate 程序较映射前具有一定的优化效果，并且相比于其他 2 种方法，CMLB 由于在 QPI 相差不多的情况下，平衡了内存带宽并减少了内存延迟，使得对 Facesim 和 Streamcluster

产生的优化效果更为明显。所以 CMLB 算法在 PARSEC 程序集上的整体性能优于 Eagermap 和 Compact。

综上所述，CMLB 算法在 NPB-OMP 和 PARSEC 的程序集上的性能测试中，其 QPI 指标在与 Eagermap 和 Compact 保持同一水准的情况下，能够大幅度降低内存带宽不平衡度使得内存延迟降低，最终使得程序的运行时间加速比得到较多的提升。因此，本研究提出的 CMLB 算法的整体性能优于 Eagermap 和 Compact，且具有很强的程序通用型。

5.2.3 映射机制与 kMAF 的优化效果对比测试

kMAF 是一种动态映射优化机制，在 3.5 节中提到，以 kMAF 为代表的动态映射优化机制需要在程序运行时以一定周期进行访存检测并迁移线程。这种根据程序运行时的变化特点进行动态调整线程的放置可以在一定程度上带来性能增益，但在程序运行时动态检测统计访存信息及迁移线程也会带来一些的额外开销。在文献[6]中，作者将 kMAF 实现为 Linux 内核模块，使用该映射机制时需要将内核模块装载入内核。

本节中，分别对比本研究提出的静态映射机制与 kMAF 动态映射机制在 NPB-OMP 与 PARSEC 程序集上的优化效果，选择运行时间加速比作为优化指标。如图 5-5 所示：

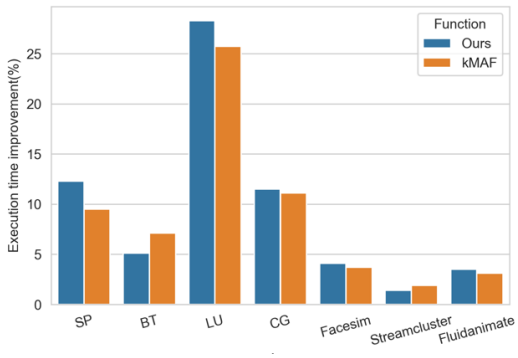


图 5-5 kMAF 与本研究的静态映射机制优化效果对比

从整体上看，本研究的静态映射机制对这 7 个应用程序平均运行时间加速比为 9.48%，kMAF 的平均运行时间加速比为 8.44%。对于 SP、LU、CG、Facesim、Fluidanimate 程序，静态映射机制优化效果优于 kMAF，而 BT、Streamcluster 程序静态映射机制优化效果不如 kMAF。这是因为 kMAF 根据程序运行时线程访存通信情况动态调整线程映射，对于 BT、Streamcluster 这种访存特征在运行时变化大的程序，具有明显的性能提升，而对于其他 5 个程序，其访存特征在运行时变化不大，由于动态访存检测及线程迁移带来的额外开销，会抵消部分性能收益。因此本研究提出的静态映射机制能够在不引入任何运行时开销的情况下，整体上达到略高于 kMAF 的映射优化效果。

5.3 静态映射机制的额外开销测试

根据本文的 3.5 节所述，静态映射机制的额外开销在程序运行前，需要对程序进行访存检测、计算映射以及线程绑定。以上这 3 个过程中，访存检测需要预运行应用程

序使用 Perf 对线程的访存行为进行检测，而计算映射以及线程绑定根据访存检测后的结果进行计算，只需要毫秒级的时间。因此主要的通信开销在映射机制的访存检测模块。

在其他的一些静态映射方法中，文献[11] [14][18][19]均使用基于 Intel 的 Pin 插桩工具进行通信检测，检测时同样需要预运行程序，通过 Pin 获取程序的访存信息。本文将对本研究中使用的基于 Perf 的访存检测方法与文献[14][18][19]中使用的基于 Pin 的 Numalize 工具的访存检测方法，使用 5.1 节中的 NPB-OMP 程序在运行时间上进行对比。如图 5-6 所示：

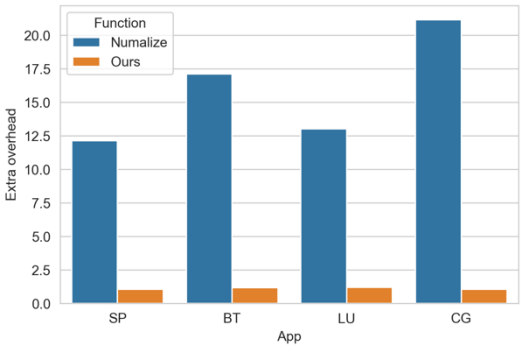


图 5-6 Numalize 与本研究访存检测的额外开销对比

在进行额外开销对比时，选择访存检测的运行时间与程序正常运行时间的比值作为实验指标。根据图 5-6 所示，Numalize 在运行时间上的额外开销明显高于本研究访存检测方法的额外开销。Numalize 运行时间几乎是程序正常运行时间的 12-20 倍，而本研究的额外开销平均为程序正常运行的 1.15 倍。因此，本文提出的静态映射优化机制相较于其他静态映射方法具有明显较小的额外开销。

5.4 本章小结

在 5.2 节中的实验结果表明，本文设计实现的静态线程映射机制对 rotor35 程序、NPB-OMP 和 PARSEC 程序集的部分程序均有不同程度的优化效果，其运行时间加速比最高可达 28.29%，QPI 最高下降达 84.03%，平均内存延迟最高下降达 4.83%，解决了线程间通信不平衡导致的跨节点访存量过多以及节点间内存带宽不平衡导致的内存延迟过高的问题。本文设计并实现的 CMLB 算法在 NPB-OMP 和 PARSEC 程序集的部分程序的测试中，通过与 Eagermap 分组算法与 OpenMP 提供的 Compact 映射模式的对比，在 QPI 指标相差不多的情况下，能够大幅度降低内存带宽不平衡度使得平均内存延迟降低，其中内存带宽不平衡度最高降低 89.0%，平均内存延迟最高降低 8.4%。最终使得程序的运行时间加速比相较于其他两种方法得到最高 2.3%的提升，因此本研究提出的 CMLB 算法的整体性能优于 Eagermap 和 Compact。同时本文的静态映射机制较 kMAF 动态映射机制有 1.04%的性能提升，能在程序运行时不引入任何额外开销的情况下，达到略高于 kMAF 的映射优化效果。最后分析比较了映射机制的额外开销，主要

存在于预运行阶段，其开销明显小于基于 Pin 的 Numalize 方法。总体而言，在解决线程间通信不平衡以及节点间内存带宽不平衡的问题时，本文设计实现的静态映射优化机制具有额外开销小且在程序运行时不引入任何开销的特点，是一个有效的映射优化方案。

6 结论与展望

6.1 结论

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静态映射优化目标，在多线程并行一级，为解决线程间数据访问与共享不平衡及节点间内存带宽不平衡的问题，设计并实现了融合硬件架构信息统计、线程间通信量统计、线程内存访问负载特征统计、计算线程分组、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矩阵、线程访存负载向量以及硬件架构信息特征，设计了基于通信感知及内存负载均衡的映射分组算法 CMLB。映射优化机制整体上改善了线程间通信不平衡的问题，同时平衡了各节点的内存带宽，减少了跨节点的访存量，降低了平均内存延迟，并有效的缩短了应用程序的运行时间。本文主要工作内容包括：

1）分析了多线程并行应用程序的基本特点，详细介绍了以 OpenMP 为代表的多线程并行编程模型的主要概念、执行模式和适用环境，以及 NUMA 架构计算机的访存模型，并对线程映射优化的问题背景、解决思路及理论优化效果进行详细分析论述。最后介绍了利用 Perf 检测并统计线程间通信特征以及各个线程内存访问负载特征的相关技术理论。

2）针对线程映射优化的问题背景提出了静态映射优化机制需要解决的 3 个问题：访存检测统计问题、线程分组划分问题和执行映射问题，并根据以上问题设计并实现了一种静态线程映射优化机制。详细介绍了映射机制中包含的硬件信息检测模块、线程访存检测模块、计算映射模块、执行映射模块的功能，并描述了映射机制的静态执行流程。使用 hwloc 工具对硬件架构信息进行检测，实现硬件信息检测模块。使用基于 Perf 的线程通信量检测统计以及线程内存访问负载检测统计的方法，实现了访存检测模块。使用基于通信感知的 Eagermap 分组算法或者本研究设计实现的 CMLB 映射分组算法，实现计算映射模块，根据统计得到的通信量矩阵以及访存负载向量结合硬件架构信息，得到线程的分组结果。使用 GOMP_CPU_AFFINITY 设置 CPU 亲和度的环境变量，根据计算映射得到的线程分组结果，将线程绑定至计算核心上，实现执行映射模块。

3）分析了影响应用程序性能的相关因素并结合访存信息的数据特点，制定了分组算法的设计原则：在尽可能少的跨节点访存的前提下，保证每个节点内存带宽均衡。基于这一设计原则结合硬件架构信息，以自顶向下的方式设计出 CMLB 分组算法的执行流程。首先从生成所有组的划分结果出发，设置所有线程的分组数为节点数并以迭代的方式生成包含所有组的最终划分结果；然后从生成一个组的划分结果出发，仍以迭代的方式选取一个线程加入组内，直到组内线程数达到上限，完成一个组的划分；之后从选择一个线程加入组出发，采用贪心策略结合通信量矩阵，选择一个当前与组

内所有线程通信量之和最大的待分组线程，准备将其加入该组，加入时需要判断该线程的加入是否会破坏整体的访存平衡，若不会则该线程加入，否则从待分组中选择通信量第二大的线程重复上述过程；最后判断一个线程的加入是否会打破坏整体的访存平衡，返回真或假。之后针对自顶向下的设计流程使用伪代码实现，并介绍相关实现细节。最后对比分析了 CMLB 与 Eagermap 的理论优化效果。

4）本文在 16 个 Intel Xeon 的处理器核心构成的 NUMA 架构计算平台上，对映射机制和 CMLB 算法均进行了相关测试和评估。通过分别测试映射机制对 rotor35 轴流压气机转子 CFD 程序，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。然后使用 NPB-OMP 以及 PARSEC 程序对比测试 CMLB、Eagermap 和 OpenMP 内置的 Compact 映射模式，表明本研究设计的 CMLB 算法整体性能优于 Eagermap 和 Compact；同时比较分析了本文的静态映射机制与 kMAF 动态映射机制在 NPB-OMP 和 PARSEC 程序集的优化效果，实验表明静态映射机制的平均运行时间加速比较 kMAF 高 1.04%，能在程序运行时不引入任何额外开销的情况下，达到略高于 kMAF 的映射优化效果；最后分析测试了映射机制的额外开销，实验表明本文的静态映射机制访存检测的额外开销明显低于其他静态映射方案使用的基于 Pin 的 Numalize 方法。

综上所述，本文设计的映射优化机制和映射分组算法，能有效解决解决多线程共享内存通信的不平衡问题减少了跨节点访存量，并平衡了各节点内存带宽减小了内存延迟，以及能优化几乎所有的多线程并行应用程序，具有很强的通用性。并且映射优化机制具有额外开销小且在程序运行时不引入任何开销的特点。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

本文针对线程间通信不平衡以及节点间内存带宽不平衡的问题提出线程到核的静态映射机制以及改进的映射分组算法，尽管对大多数线程并行应用程序有良好的游湖效果，但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）本文提出的静态线程映射优化机制中，对应用程序的执行线程数需要手动设置。据一些研究表明，多线程并行应用程序并不是并行的线程数越大运行时间越短，其中存在最优线程数，当应用程序在最优线程数下运行时，其运行效率是最高的。因此需要考虑如何预测应用程序的最优线程数，并将该部分加入到映射优化机制中，是一项比较困难但也非常具有意义的研究问题。

2）本文提出的静态线程映射优化机制，当程序规模扩大并运行在超算平台上，需要考虑结合 MPI 与 OpenMP 的两级并行。因此如何提高映射优化机制的高扩展性成为了一个重要问题。需要去考虑 MPI 进程在共享内存通信的映射机制设计以及如何检测其通信量。

3）本文的映射优化机制为静态机制，需要在程序运行前将线程绑定至计算核上并且程序运行过程中不会再迁移线程。这种映射方式不能根据程序运行时变化特点在运行时迁移线程，因此需要设计一种动态的映射机制。本文中提到的 kmaf 便是一种动态机制，它虽然能够在程序运行时以一定周期检测访存信息并调整线程位置，但这种周期性的检测会带来比较大额外开销，检测周期过短可能导致机制的额外开销超过其本身映射带来的收益，反而导致总体性能下降。并且 kmaf 为系统级的映射机制，可移植性较差。因此需要设计一种可移植性强的产生较小额外开销的动态映射机制。

4）本文的静态映射优化机制，需要在同构 CPU 平台上运行。为了提高其可扩展性，需要设计一种能在 CPU+GPU 这种异构平台上运行的映射优化机制，这里需要考虑 CPU 与 GPU 的计算性能差异、访存模型差异以及内存带宽差异，是一项比较复杂具有挑战意义的研究问题。

致　谢

参考文献

[1] Chung T J. Computational fluid dynamics[M]. Cambridge university press, 2010.

[2] 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.

[3] 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.

[4] Agung M, Anrizal M A, Kazuhiko K, et al. A Memory Congestion-aware MPI Process Placement for Modern NUMA Systems[C]//IEEE 24th International Conference on High Performance Computing, Jaipur, India, 2017:152-161.

[5] Diener M, Cruz E H M, Navaux P O A, et al. Communication-Aware Process and Thread Mapping Using Online Communication Detection[J]. Parallel Computing, 2015:43-63.

[6] Diener M, Cruz E H M, Navaux P O A, et al. kMAF:Automatic Kernel-Level Management of Thread and Data Affinity[C]//PACT:Proceedings of the 23rd international conference on Parallel architectures and compilation, 2014: 277-288.

[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] 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, Raleigh, NC, USA, 2009: 14-18.

[9] Jeannot, E., Mercier, G., and 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.

[10] Cruz E H M, Diener M, Pilla L, et al. An Efficient Algorithm for Communication-Based Task Mapping[J]. in Proc. Int. Conf. Parallel, Distrib. Netw.-Based Process, 2015: 207-214.

[11] Agung M, Anrizal M A, Egawa R, et al. DeLoc: A Locality and Memory-congestion-aware Task Mapping Method for Modern NUMA Systems[J]. IEEE Access, 2020: 6937-6953.

[12] Diener M, Curz E H M, Alives M A Z, et al. Affinity-Based Thread and Data Mapping in Shared Memory Systems[J]. ACM Comput. Surv. V, N, Article0 (2016), 38 pages.

[13] Luk C, Cohn R, Muth R, et al. Pin:building customized program analysis tools with dynamic instrumentation[J]. SIGPLAN, 2005: 190-200.

[14] Diener M, Curz E H M, Alives M A Z, et al. Communication in shared memory:Concepts, definitions, and efficient detection[C]//PDP: 24th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, 2016: 151-158.

[15] Sasongo M A, Chabbi M, Akhtar P, et al. ComDetective: A Lightweight Communication Detdction Tool for Threads[C]//ACM Supercomputering, NY, USA, 2019: 17-22.

[16] 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.

[17] F. Pellegrini. Static mapping by dual recursive bipartitioning of process architecture graphs[C]//Proceedings of IEEE Scalable High Perfmance Computing Conference.Knoxville, TN, USA, 1994: 486-493.

[18] Cruz E, Diener M, Pilla L L, et al. EagerMap: A Task Mapping Algorithm to Improve

Communication and Load Balancing in Clusters of Multicore Systems[J]. ACM Transactions on Parallel Computing, 2019, 5(4), pp.17.

[19] Soomro P N, Sasongko M A, Unat D. BindMe: A thread binding library with advanced mapping algorithms[J].Concurrency Computat Pract Exper (cpe), vol.30, no. 21, p.e4692, 2018.

[20] Gaud F, Lepers B, Funston J, et al. Challengs of Memory Management on Modern NUMA Systems[C]. Queue, 2015: 70-85 .

[21] 周伟明. 多核计算与程序设计[M]. 华中科技大学出版社，2009.

[22] 朱利, 李晨. 计算机系统结构[M]. 清华大学出版社，2012.

[23] Intel, Intel Xeon Processor E5-2600 v2 Product Family Uncore Performance Monitoring Reference Manual, 2013.

[24] Bokhari. On the Mapping Problem[J]. IEEE Transactions on Computers, 2006, 30(3): 207-214.

[25] Intel, Intel Memory Latency Checker(Intel MLC).

[26] Broquedis F, Cletortega J, Moreaud S, 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.

[27] Kleen A, SUSE Labs, numactl[CP], Version 2.0.10-rc2, 2014.

[28] Bryant R E, O'Hallaron D R. Computer systems -a programmers perspective[M]. 机械工业出版社，2003.

[29] Travis E, Oliphant's. Guide to Numpy[M]. Computing in Science and Engineering, 2007.

[30] McKinney W. Data structures for statistical computing in python[M], Proceedings of the 9th Python in Science Conference, 2010.

[31] Zhihan Li, Youjian Zhao, Rong Liu, et al. Robust and Rapid Clustering of KPIs for Large-Scale Anomaly Detection[C]//IEEE/ACM 26th International Symposium on Quality of Service(IWQoS), Banff, AB, Canada, 2018: 1-10.

[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] 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.

[35] 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.

[36] 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.

[37] Jin, H. & MA, Frumkin. The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance[J]. 1999.

[38] Bienia C, Kumar S, Singh J, et al. The PARSEC benchmark suite: characterization and architectural implications[C]//Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques(PACT). Arg. 2008: 72 -81.

攻读学位期间取得的研究成果

[1] 参与国家十三五重点研发计划课题“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：），研究弹性映射机制方案。

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