

博士学位论文

**异构众核系统线程映射关键技术研究**

学位申请人：巨涛

指导教师：董小社 教授

学科名称：计算机科学与技术

2016年9月

**Research on Key Technologies of Thread Mapping for Heterogeneous Many-Core Systems**

A dissertation submitted to

Xi’an Jiaotong University

in partial fulfillment of the requirements

for the degree of

Doctor of Engineering Science

By

Tao Ju

Supervisor: Prof. Xiaoshe Dong

(Computer Science and Technology)

September 2016

**论文题目：异构众核系统线程映射关键技术研究**

**学科名称：计算机科学与技术**

**学位申请人：巨 涛**

**指导教师：董小社 教授**

摘 要

随着多核、众核技术的迅速发展，出现了具有不同硬件架构、不同计算模式、高并行计算能力的异构众核处理器，为构建高性能计算机系统提供了更高性能的硬件支持。基于异构众核处理器搭建异构超级计算机已经成为高性能计算领域一个重要的发展趋势，这也将成为构建E级计算机系统的主要架构趋势。针对异构系统的异构计算技术，因其可有效解决能耗、可扩展性等问题，已成为全球高性能计算领域中的一种重要新兴模式，也将会在后续的万万亿次、百亿亿次超级计算机系统中发挥更重要的作用。由于异构系统不同计算部件的异构性，进一步加剧了并行应用程序的开发难度，使并行应用的开发、并行应用程序的移植、异构众核系统高计算潜能的发挥面临严峻挑战。如何将应用程序的计算任务合理映射到异构众核系统处理器具体处理核上，使应用程序运行特点和硬件架构特征较好的匹配，对异构众核系统来说至关重要，将直接影响能否充分利用异构众核系统计算资源、能否有效提升并行应用程序的计算效能。

本文针对异构众核系统计算任务到计算资源映射时面临的挑战，主要对并行应用程序执行过程中线程到异构众核处理器映射时的线程数设置、线程内数据局部性及线程间数据相关性、应用线程到具体处理核的映射等关键问题展开研究，取得的主要创新成果如下：

1) 针对异构众核系统下如何为多线程应用程序设置合理线程数的问题，提出了一种面向众核系统的最优线程数预测模型TCPM（Thread Count Prediction Model）。通过研究多核多线程程序运行时不同线程间相互作用对程序性能的影响，分析线程数和性能变化多样性对应关系的原因，抽象影响因素；在充分考虑多线程执行时由于不同线程之间相互作用产生的额外开销对程序性能影响的基础上，扩展Amdahl定律，结合回归分析理论，利用最小二乘法构建最优线程数预测模型。通过假设检验、预测精度及额外开销测试，验证所构建的最优线程数预测模型的有效性。实验评测表明，利用TCPM可以简单、准确、有效的预测最优线程数，可在启用最小线程数的情况下，达到最优计算性能的97%。

2) 针对应用程序在执行过程中，由于运行行为发生阶段性变化而对计算资源有不同需求的问题，提出了一种基于程序阶段性变化的线程数动态调整机制DPTM（Dynamic Prediction Thread Mapping）。通过实时的检测程序运行状态及阶段性变化，根据收集到的反映程序运行状态的硬件性能计数器信息，利用所提出的最优线程数预测模型直接计算最优线程数的预测值；根据计算出的最优线程数预测值，对程序的并行性进行调整，以满足在程序不同的执行阶段对计算资源的不同需求，从而更有效的利用众核处理器计算能力来提升系统的整体计算效能；针对Intel MIC异构众核系统设计实现了基于DPTM的线程数动态调整映射框架。基准测试评测结果表明，DPTM线程数动态调整机制，相对于传统映射方法平均性能提升了48.6%，平均能耗降低了59%，引入的额外开销为2.03%。

3) 针对如何准确量化反映应用程序不同线程之间数据依赖关系的数据相关性问题，提出了一种基于数据重用距离的线程间数据相关性度量方法DRAQ (Data Reuse Distance based Data Affinity Quantifying)。通过剖分不同线程存储访问数据信息，分析不同线程之间的数据局部性及存储访问特征，设计线程数据重用距离计算方法，计算不同线程的数据重用距离；在数据重用距离的基础上，通过定义反映不同数据相关性的局部性模式，将不同线程归并为不同的模式类；在不同模式类的基础上通过比较不同线程间数据共享量，用相关性矩阵来度量不同线程间的数据相关性，并将线程相关性矩阵转换成相应的相关性图，来直观的反映不同应用线程之间的数据相关性。实验评测表明，DRAQ相对于OS默认映射方法，Cache缺失率平均降低了10.4%，达到最优映射下Cache缺失率的87%，引入的平均开销为8%。

4) 针对如何根据应用线程之间的数据相关性，将线程合理映射到众核处理器不同处理核上的问题，提出了一种基于数据相关性的线程分组映射机制DagTM（Data Affinity Grouping based Thread Mapping）。以应用程序不同线程数据相关性特点为基础，结合具体众核系统硬件架构特点，在线程数据相关性矩阵及数据相关性图的基础上，将线程的逻辑分组问题抽象成一个图的分解问题；通过设计数据相关性子树生成算法，将线程相关性图分解为满足一定要求的Ｋ棵子树，实现将具有高数据共享的线程划分到同一个线程组内，将具有较强数据访问争用关系的线程划分到不同的线程组内，从而降低因不同线程之间相互争用共享计算资源引起的高存储访问开销；通过线程组到处理核的绑定，将应用线程合理的映射到众核处理器不同处理核上高效执行，以提高众核处理器计算资源利用率，提升异构众核系统的整体计算效能。实验评测结果表明，DagTM映射机制，在不产生额外运行时开销的基础上，与传统映射机制相比计算性能平均提高了14%，能耗降低了10%。

**关 键 词**：异构众核系统；线程映射；最优线程数；数据重用距离；数据相关性

**论文类型**：应用基础

**Title:** **Research on Key Technologies of Thread Mapping for Heterogeneous Many-Core Systems**

**Discipline: Computer Science and Technology**

**Applicant: Tao Ju**

**Supervisor: Prof. Xiaoshe Dong**

ABSTRACT

With the rapid development of the multi-core and many-core technology, the heterogeneous many-core processors with the different hardware architectures, computation patterns, and high parallel computing power provide the hardware support for constructing the high performance computer system. The supercomputer composed of the heterogeneous many-core processors has emerged as a promising trend in the domain of high-performance computing. The heterogeneous computing has become as an important emerging computing pattern in the high performance computing, and will play important role in the subsequent exascale computing systems. However, the heterogeneity of different components of heterogeneous system increases the programing difficulties, and makes the development and transplant of the parallel applications, and exploitation the high computing power of the heterogeneous many-core system facing serious challenges. How to map the computing tasks to the specific processing cores to reasonably match the running behavior of the application program with the characteristic of the hardware architecture, which is important to efficiently exploit the computing resources and improve the computing performance.

To handle the challenges above, this dissertation studies the key technology problem from setting thread count, data affinity between threads, and thread mapping for heterogeneous many-core systems during the execution of a parallel application program. The brief innovations and contributions are presented as follows.

1) To handle the challenges of how to determine the appropriate thread count for multi-threaded application running on the heterogeneous many-core systems, the dissertation proposes an optimum thread count prediction model TCPM(Thread Count Prediction Model). Through studying the impact of interaction between different threads on computing performance when multi-threaded application running on the heterogeneous many-core system, analyzing the reason of diversity relationship between thread count and computing performance, abstracting the impact factors, and considering the additional overhead, an optimum thread count prediction model TCPM is established using the regression analysis on the basis of extending Amdahl’s law. The hypothesis testing and measurement results of prediction precision and overhead verify the validity of TCPM. Experimental results shows that TCPM could realize efficiently, simply, and dynamically real-time predict the thread count, and it could obtain the 97% of the optimal computing performance.

2) To handle the challenge of how to allocate the computing resource according to the different requirements of an application at different running phases, the dissertation proposes a phase changes based thread mapping mechanism DPTM(Dynamic Prediction Thread Mapping), which could dynamically regulate the hardware thread count according to the phase changes of running program during execution. DPTM real-timely detects the running status and phase changes of the running program, and uses the prediction model to estimate the optimum thread count at different phases of the application through a real-time sampling of hardware performance counter information. After that, DPTM dynamically adjusts the number of active hardware threads and processing cores during program execution according to the optimum thread count to improve the application computing performance and energy efficiency. The DPTM thread mapping framework is implemented based on the Intel MIC heterogeneous many-core system. Experimental results shows that DPTM obtains a nearly 48.6% improvement in performance, a 59% reduction in energy consumption on average compared with traditional thread mapping, and introduces about 2.03% additional overhead for benchmark programs running on an Intel MIC heterogeneous many-core system.

3) To handle the challenge of how to quantify the data affinity between threads, the dissertation proposes a data reuse distance based data affinity quantifying approach DRAQ(Data Reuse Distance based Data Affinity Quantifying). Through profiling and collecting the memory access data of different threads, analyzing the data affinity relationship between different threads of an application program, a parallel algorithm of calculating data reuse distance of different threads is designed. The different threads are merged into the different locality pattern classes by locality pattern classification based on the data resue distance. Through alculating the number of the same accessed data between different threads in the every pattern class, the data affinity between threads is analyzed and quantified by using data affinity matrix. After that the data affinity matrix is transformed to the data affinity graph to intuitive reflect the data affinity between threads. Experimental results shows that DRAQ obtains a 10.4% reduction in L2 cache miss rate compared with OS default mapping, and amounts to 87% of optimal mapping, and introduces about 8% additional overhead.

4) To handle the challenges of reasonably mapping the threads to the specific processing cores to match the data locality of application threads with the spatial locality of hardware platform, and efficiently exploit the high parallel computing power of the heterogeneous many-core processor, the dissertation proposes a data affinity based threads grouping mapping strategy DagTM(Data Affinity Grouping based Thread Mapping). DagTM categorizes threads into different groups according to their data affinity and the hardware architecture feature of many-core processors. The thread grouping is abstracted as a graph decomposition problem. By design an affinity sub-tree spanning algorithm, the data affinity graph is decomposed as K subtrees to meet the special requirements. The threads with high data sharing are merged into a same thread group, and the threads with strong memory access conflicts are merged into different thread groups to reduce memory access overhead. Finally, the thread groups are assigned to different processing cores through static binding in order to reduce conflicts of the shared memory access and additional data transmission, increase utilization of the computing resources, and improve entire system computing energy efficiency. Experimental results show that DagTM obtains a nearly 14% improvement in computing performance, and a nearly 10% reduction in energy consumption compared with the traditional thread mapping mechanism under the condition of not introducing additional runtime overhead.

**KEY WORDS**: Heterogeneous many-core systems; Thread mapping; The optimum thread count;

Data reuse distance; Data affinity

**TYPE OF DISSERTATION**: Application Fundamentals

# 

目 录

[1 绪论 1](#_Toc457205828)

[1.1 研究背景及意义 1](#_Toc457205829)

[1.2 异构系统研究现状 3](#_Toc457205830)

[1.2.1 异构系统编程模型研究现状 4](#_Toc457205831)

[1.2.2 异构系统性能优化研究现状 5](#_Toc457205832)

[1.2.3 异构系统进一步的研究方向 7](#_Toc457205833)

[1.3 异构众核系统线程映射面临的关键问题 8](#_Toc457205834)

[1.4 论文的研究内容及创新点 9](#_Toc457205835)

[1.5 论文组织结构 10](#_Toc457205836)

[2 研究工作基础 12](#_Toc457205837)

[2.1 异构众核系统概述 12](#_Toc457205838)

[2.1.1 异构众核系统架构特点 13](#_Toc457205839)

[2.1.2 异构众核系统编程特点 18](#_Toc457205840)

[2.2 GPU、MIC异构众核系统性能测试 21](#_Toc457205841)

[2.2.1 测试环境及测试程序 22](#_Toc457205842)

[2.2.2 测试结果及分析 22](#_Toc457205843)

[2.3 MIC异构众核系统线程映射及性能优化分析 23](#_Toc457205844)

[2.3.1 线程到MIC处理核映射分析 23](#_Toc457205845)

[2.3.2 MIC异构系统应用程序优化实例 27](#_Toc457205846)

[2.4 线程映射相关工作分析 31](#_Toc457205847)

[2.4.1 线程数设置 32](#_Toc457205848)

[2.4.2 线程数动态调整 32](#_Toc457205849)

[2.4.3 线程数据局部性及数据相关性 34](#_Toc457205850)

[2.4.4 基于数据局部性的线程映射 36](#_Toc457205851)

[2.5 本章小结 37](#_Toc457205852)

[3 面向众核系统的最优线程数预测模型 39](#_Toc457205853)

[3.1 引言 39](#_Toc457205854)

[3.2 线程数对计算性能的影响分析 40](#_Toc457205855)

[3.2.1 性能可扩展性测试 40](#_Toc457205856)

[3.2.2 程序性能可扩展性影响因素分析 44](#_Toc457205857)

[3.3 最优线程数预测模型TCPM 45](#_Toc457205858)

[3.3.1 性能指标定义 45](#_Toc457205859)

[3.3.2 预测模型理论基础 45](#_Toc457205860)

[3.3.3 预测模型构造 49](#_Toc457205861)

[3.4 TCPM预测模型有效性验证 51](#_Toc457205862)

[3.4.1 样本值测试 51](#_Toc457205863)

[3.4.2 假设检验 52](#_Toc457205864)

[3.4.3 预测精度及额外开销评测 54](#_Toc457205865)

[3.5 本章小结 56](#_Toc457205866)

[4 基于程序阶段性变化的线程数动态调整映射机制 58](#_Toc457205867)

[4.1 引言 58](#_Toc457205868)

[4.2 线程数动态调整机制处理流程 60](#_Toc457205869)

[4.3 基于程序阶段性变化的线程数动态调整映射框架 62](#_Toc457205870)

[4.3.1 DPTM映射框架 62](#_Toc457205871)

[4.3.2 状态信息采集 64](#_Toc457205872)

[4.3.3 动态检测程序阶段性变化 65](#_Toc457205873)

[4.4 DPTM映射框架实现 68](#_Toc457205874)

[4.4.1 OpenMP运行时库扩展策略 68](#_Toc457205875)

[4.4.2 MIC异构系统下的DPTM实现 69](#_Toc457205876)

[4.5 实验评测及分析 71](#_Toc457205877)

[4.5.1 基准测试程序及测试环境 71](#_Toc457205878)

[4.5.2 测试结果及分析 73](#_Toc457205879)

[4.6 本章小结 78](#_Toc457205880)

[5 基于数据重用距离的线程数据相关性度量方法 79](#_Toc457205881)

[5.1 引言 79](#_Toc457205882)

[5.2 数据重用距离理论 81](#_Toc457205883)

[5.2.1 数据重用距离的形式化表示 81](#_Toc457205884)

[5.2.2 数据重用距离实例 82](#_Toc457205885)

[5.3 数据重用距离计算 82](#_Toc457205886)

[5.3.1 数据重用距离计算方法 82](#_Toc457205887)

[5.3.2 线程访问数据收集方法 83](#_Toc457205888)

[5.3.3 线程访问数据统计 85](#_Toc457205889)

[5.3.4 基于平衡二叉树的数据重用距离计算 87](#_Toc457205890)

[5.3.5 算法复杂度分析 90](#_Toc457205891)

[5.3.6 线程数据重用距离计算实例 90](#_Toc457205892)

[5.3.7 线程平均数据重用距离计算 92](#_Toc457205893)

[5.4 数据相关性判定及度量 92](#_Toc457205894)

[5.4.1 线程内数据局部性模式分析 92](#_Toc457205895)

[5.4.2 线程间数据相关性度量 94](#_Toc457205896)

[5.5 数据相关性度量方法评测 95](#_Toc457205897)

[5.5.1 数据局部性分析方法比较 95](#_Toc457205898)

[5.5.2 数据相关性度量方法有效性评测 97](#_Toc457205899)

[5.6 本章小结 99](#_Toc457205900)

[6 基于数据相关性的线程分组映射机制 100](#_Toc457205901)

[6.1 引言 100](#_Toc457205902)

[6.2 DagTM线程分组映射机制 101](#_Toc457205903)

[6.3 线程逻辑分组 101](#_Toc457205904)

[6.3.1 Affinity子树生成算法 102](#_Toc457205905)

[6.3.2 Affinity子树生成实例 104](#_Toc457205906)

[6.3.3 Affinity子树生成算法复杂性分析 105](#_Toc457205907)

[6.4 线程分组映射及实现 105](#_Toc457205908)

[6.4.1 线程组到处理核的映射规则 105](#_Toc457205909)

[6.4.2 线程分组映射实现 106](#_Toc457205910)

[6.4.3 线程分组映射实例 106](#_Toc457205911)

[6.5 实验评测及分析 109](#_Toc457205912)

[6.5.1 实验环境及方法 109](#_Toc457205913)

[6.5.2 测试结果及分析 110](#_Toc457205914)

[6.6 本章小结 113](#_Toc457205915)

[7 结论与展望 114](#_Toc457205916)

[7.1 本文工作总结 114](#_Toc457205917)

[7.2 下一步工作展望 116](#_Toc457205918)

[致 谢 117](#_Toc457205919)

[参考文献 118](#_Toc457205920)

[攻读学位期间取得的研究成果 128](#_Toc457205921)

声明

CONTENTS

1 Preface 1

1.1 Research Background 1

1.2 Research Status of the Heterogeneous Many-Core Systems 3

1.2.1 Research Status of the Programming Model of Heterogeneous

Many-core Systems 4

1.2.2 Research Status of the Performance Optimization of Heterogeneous

Many-core Systems 5

1.2.3 The Further Research Direction 7

1.3 The challenges of the Thread Mapping of Heterogeneous Many-Core Systems 8

1.4 Research Contents and Contributions 9

1.5 Framework of Dissertation 10

2 Survey of the Related Works 12

2.1 The Overview of the Heterogenesous Many-Core Systems 12

2.1.1 The Architecture Characteristics of Heterogeneous Many-Core Systems 13

2.1.2 The Programming Characteristics of Heterogeneous Many-core systems 18

2.2 The Performance Measurement of GPU and MIC Heterogeneous Systems 21

2.2.1 Measurement Environment and Programs 22

2.2.2 Measurement Results and Analysis 22

2.3 Analysis of Thread Mapping and Performance Optimization for MIC

Heterogeneous Many-Core Systems. 23

2.3.1 Analysis of Thread Mapping on MIC Processing Cores 23

2.3.2 The Instance of Program Performance Optimization on MIC

Heterogeneous Many-Core Systems 27

2.4 Relatd Work and Discussion of Thread mapping 31

2.4.1 The Thread Count Setting 32

2.4.2 Dynamic Adjusting of Thread Count 32

2.4.3 Data Locality and Affinity of Thread 34

2.4.4 Thread Mapping Based on Data Locality 36

2.5 Brief Summary 37

3 Optimum Thread Count Prediction Model for Many-core Systems 39

3.1 Introduction 39

3.2 Performance Scalability Measurement and Analysis 40

3.2.1 Performance Measurement for Varying Thread Count 40

3.2.2 Analysis of Impact Factors on Scalability 44

3.3 Optimal Thread Count Prediction Model TCPM 45

3.3.1 Notations of Performance Metrics 45

3.3.2 Theoretical Basis of Prediction Model 45

3.3.3 Prediction Modle Establishment 49

3.4 Effectiveness Verification of TCPM 51

3.4.1 Sample Testing 51

3.4.2 Hypothesis Testing 52

3.4.3 Evaluation of Prediction Precision and Overhead 54

3.5 Brief Summary 56

4 Dynamic Adjusting of Thread Count Based on Phase Changes of running

program 58

4.1 Introduction 58

4.2 DPTM Mapping Mechanism 60

4.3 DPTM Mapping Framework 62

4.3.1 DPTM Mapping Framework 62

4.3.2 Sampling the Status Information 64

4.3.3 Detecting the Phase Changes of runnign program 65

4.4 DPTM Mapping Framework Implementation 68

4.4.1 The Strategy of Expending OpenMP Runtime Library 68

4.4.2 DPTM Implementation Based on the MIC Heterogeneous Systems 69

4.5 Experimental Evaluation 71

4.5.1 Experimental Evaluation 71

4.5.2 Evaluation Results 73

4.6 Brief Summary 78

5 Quantifying Data Affinity Between Threads Based on Data Reuse Distance 79

5.1 Introduction 79

5.2 Data Reuse Distance 81

5.2.1 Formal Denotation of Data Reuse Distance 81

5.2.2 The Instance of the Data Reuse Distance 82

5.3 Calculatoin of Data Reuse Distance 82

5.3.1 Calculating Data Reuse Distance 82

5.3.2 Collecting Thread Access Data 83

5.3.3 Statisticing the Thread Access Data 85

5.3.4 Data Reuse Distance Calculaton Algorithm based on the AVL Tree 87

5.3.5 The Algorithm Complexity Analysis 90

5.3.6 The Instance of Calculating Data Reuse Distance 90

5.3.7 The Average Data Reuse Distance of Thread 92

5.4 Determing Data Affinity 92

5.4.1 Definition of Data Locality Pattern 92

5.4.2 Quantifying Data Affinity Between Threads 94

5.5 Evaluation of Data Affinity Quantifying Approach 95

5.5.1 Comparasion of Different Analysis Approaches of Data Locality 97

5.5.2 Evaluation of Effectiveness of DRAQ 97

5.6 Brief Summary 99

6 Threads Grouping Mapping Based on Data Affinity 100

6.1 Introduction 100

6.2 Threads Grouping Mapping Framework DagTM 101

6.3 Thread Gropuing 101

6.3.1 Affinity Subtree Spanning Algorithm 102

6.3.2 The Instance of the Affinity Subtree Spannng Algorithm 104

6.3.3 The Algorithm Complexity Analysis 105

6.4 DagTM Implementation 105

6.4.1 Mapping Rules 105

6.4.2 Thread Grouping Mapping Implementation 106

6.4.3 The Instance of the Thread Gropuing Mapping 106

6.5 Experimental Evaluation 109

6.5.1 Experimental Environment 109

6.5.2 Evaluation Results 110

6.6 Brief Summary 113

7 Conclusions and Suggestions 114

7.1 Conclusions 114

7.2 Suggestions 116

Acknowledgements 117

References 118

Achievements 128

Declarations

# 绪论

随着多核、众核技术的迅速发展，出现了具有不同硬件架构、不同计算模式、高并行计算能力的异构众核系统，为构建高性能计算机系统提供了更高性能的硬件支持。由于异构系统不同计算部件的异构性，进一步加剧了并行应用程序的开发难度，使并行应用程序的开发、并行应用程序的移植、异构众核系统的高计算潜能的发挥面临严峻挑战。如何提高异构众核系统的可编程性，充分利用异构众核系统的计算资源，在保证异构众核系统高计算性能的同时降低系统的整体能耗是异构众核系统面临的关键科学问题。而在具体执行并行应用程序时，如何将应用线程合理的映射到异构众核系统的具体处理核上，使应用程序执行特点和具体硬件架构特征较好的匹配，对发挥异构系统计算能力，提高应用程序的计算效能起着至关重要的作用。本文针对异构众核系统线程映射机制，从线程到计算资源映射层面来研究异构众核系统的性能优化方法。

## 研究背景及意义

高性能计算技术的不断发展，使高性能计算机系统的性能一直在稳步持续地提高，性能平均每10年增长3个数量级。尽管现在的高性能计算机已超过千万亿次的运算速度，但许多重要的应用领域，如：环境科学、航空航天、生物信息、材料科学、高能物理等，都对使用更高性能的计算机提出了迫切的需求，预计2018年就会出现峰值速度超过每秒百亿亿次运算（Eflops，1018flops）的系统。百亿亿次系统已经成为各国激烈竞争的解决国家挑战性问题的重要手段，以及解决制约国家经济发展瓶颈问题的重要工具，具有重要的战略意义，也是一个国家国防、经济与科技综合势力的重要体现[1]。

随着多核、众核技术的迅速发展，出现了具有不同硬件架构、不同计算模式、高并行计算能力的异构众核系统，这不但为构建高性能计算机系统提供了更高性能的硬件支持，同时也为高性能计算面临的Power Wall问题提供了一种有效的解决方案。多核处理器通常指将多个处理器核（10个以下处理核数）集成在一个芯片上，各处理器核之间主要通过共享存储器进行通信，通过更多的处理单元同时工作来提高处理器的性能；众核处理器通常指将大量的处理核心（10个以上处理核心）集成在一个芯片上，处理核心之间主要通过片上互连网络方式互连在一起，提供高效的并行处理能力。

新型异构众核系统为了达到高性能低功耗的目的，通常采用主处理器+协处理器的体系架构，该架构下主处理器，通常采用多核处理器，负责具有复杂控制逻辑的任务计算，协处理器负责计算密度高、逻辑分支简单的大规模数据并行任务的计算。为了满足复杂控制逻辑计算的需要，主处理器采用了复杂的控制逻辑和分支预测，以及大量的缓存来提高执行效率，导致主处理器的功耗很高；又因缓存和控制逻辑要占用很大的芯片面积以及具有复杂控制逻辑的处理核的散热问题，限制了在一块芯片上可集成的处理核数量。由于没有足够的处理核数，使主处理器在处理数据密集型并行任务时性能无法满足应用的需求。为了弥补主处理器的不足，用协处理器来专门处理数据密集型的并行任务。结合所处理的数据密集型任务的特点，在设计协处理器时对其硬件控制单元进行了简化，减少了不必要的复杂逻辑控制单元，加大芯片内执行单元的数量，使数量庞大而功能单一的执行单元运行更多相对简单的线程，来处理数据密集型的并行任务，同时由硬件管理轻量级线程，实现零开销的线程切换，保证在有限芯片面积上集成更多的处理核来实现更高的计算能力。采用主处理器+协处理器异构架构，可以使异构系统从整体上达到高性能和低功耗的目的，为具体的应用提供高效的计算平台。

以异构众核为特征的新型异构系统已经成为构造现代高性能计算机系统的主要技术，且应用越来越广泛。针对异构系统的异构计算因其可有效解决能耗、可扩展性等问题，已经成为全球高性能计算领域中的一种重要新兴模式，将会在后续的万万亿次、百亿亿次超级计算机系统中发挥更重要的作用[2]。

由于异构系统不同计算部件的异构性，进一步加剧了并行应用程序的开发难度，使并行应用的开发、并行应用程序的移植、异构众核系统的高计算潜能的发挥面临严峻挑战，主要表现在以下方面：

1）异构系统并行程序编程困难

新型异构并行系统由于其异构性，大大增加了用户开发并行应用程序的难度。新型异构众核系统为了达到高性能低功耗的目的，通常采用主处理器+协处理器的体系架构，该架构下主处理器（CPU）负责具有复杂控制逻辑的事务计算，协处理器如GPU（Graphics Processing Unit）、Intel MIC（Intel Many Intergrated Core）、FPGA（Field-Programmable Gate Way等，负责计算密度高、逻辑分支简单的大规模数据并行任务计算。为了实现不同的计算目的，异构系统主处理器和协处理器采用了不同的指令系统，各自具有独立的存储访问空间。而当前针对异构众核硬件架构缺乏较好的统一抽象模型，没有提供一种有效的完全透明的编程语言，没有封装存储系统的异构性。在这种新型异构架构下，用户在开发应用程序时面临如下问题：必须显式管理协处理单元与主处理单元的协作；需要对各种存储器的特点和局限性都要有清楚的认识并有选择的使用；显式管理数据的层次化分布、访存等工作；要根据具体硬件特点将任务进行合理的划分、进行具体的线程配置；在编程时要对数据传输、存储器访问及指令流进行优化。这些大大的增加了用户开发程序的难度，限制了异构并行系统的使用。

2）面向异构系统的软件移植困难

已有的应用软件很难直接在异构系统上运行，软件移植存在很大的问题。由于传统编程模型没有考虑新型异构系统下协处理器的使用及针对它的程序开发方法，为了有效利用和发挥大规模协处理器来提高传统应用程序的计算性能，大量已有的应用程序需要针对新型异构系统的架构特点，进行重新设计和开发，这从根本上增加了应用软件的移植难度和系统使用成本；又由于不同厂家异构众核硬件架构的不同，编程模型也多种多样，虽然学术界和工业界对统一异构众核并行编程进行了多年的研究和探索，至今并未形成一个统一有效的标准体系，这更进一步加剧了应用程序移植的难度。而应用程序移植却要求尽可能少地修改源程序、保持原并行算法不变、保持原并行程序结构不变，充分利用新系统的计算能力来提高并行应用软件的并行效率，所以如何在新型异构众核系统下实现已有应用软件可靠高效的移植是充分发挥异构众核系统性能的关键。

3）异构系统计算资源得不到充分利用

针对新型异构系统，由于缺乏较好的调度机制和优化策略，造成了具体应用程序不能充分利用异构众核系统计算资源，导致异构系统整体计算性能不高。新型异构众核系统包含大量的并行协处理器单元，而传统使用模式通常针对具体架构及应用进行专门设计与优化，优化机制大多只侧重某一方面的性能优化问题，多数情况是用手工编码的方法实现优化，存在使用困难、性能不能动态地自动优化等弊端。特别是对于不规则应用，由于其复杂的数据依赖关系，负载不均衡等问题，不合理的映射和优化策略将使通用处理器和协处理器协同计算时计算资源得不到充分利用，导致系统整体执行效率不高。所以如何将各种优化技术进行合理的整合，设计优化工具及函数库，从任务分解、线程配置、存储访问、指令执行方面综合对系统性能进行优化，使异构系统的计算资源得到充分利用，这是从整体上提高异构系统计算性能必须要解决的问题。

4）异构系统可靠性差，功耗不可控

异构众核系统包含了不同硬件架构的处理器，同时协处理器又包含了众多的处理核心，在传统的异构众核运行模式下，由于没有很好的能耗控制机制，在程序运行的过程中众核处理器持续运行在全速模式下，系统功耗会持续增加，导致芯片温度过高，最终影响到系统的稳定性和可靠性。目前还没有能在保证程序计算性能的同时，根据程序运行状态，对异构众核处理器的电压和频率进行动态调整，以降低系统能耗、提高系统可靠性的有效机制。

综上所述，如何提高异构众核系统的可编程性，如何充分利用异构众核系统的计算资源，如何在保证异构众核系统高计算性能的同时降低系统的整体能耗、提高系统的可靠性是异构众核系统面临的关键科学问题。异构并行系统的可编程性与性能优化作为高性能计算软件理论的重要核心，现已成为高性能计算机系统不可分割的一部分。它不仅是硬件系统充分发挥实际效能和经济效益的关键，同时也是我国在信息技术领域增强自主创新能力与核心竞争力的重要契机。

本文针对上述研究背景，主要针对新型异构众核系统高计算能力不能充分发挥的问题，通过研究线程到处理核资源的合理映射机制，从计算任务到处理单元的映射层面来优化异构众核系统的计算效能。

## 异构系统研究现状

采用主处理器+协处理器的异构架构，可以使异构系统从整体上达到高性能、低功耗的目的，从而为具体的应用提供高效的计算平台。Owens等[3]对GPU通用计算相关的硬件及软件、编程模型、编程语言及相应的编译调试环境、性能优化技术及相关的GPGPU（General Purpose GPU）应用作了综述。Brodtkorb等[4]对GPU硬件架构和传统的优化技术进行了综述。王海峰等[5]从应用的角度对GPU体系架构、编程模型、开发语言、存储模型、通信模型、负载均衡、可靠性、低功耗优化等方面进行了综述，同时对GPU当前的应用领域及进展进行了分析，最后总结了GPGPU研究中存在的问题及面临的挑战。由于异构众核系统采用了数量众多的处理核，同时又采用了不同的存储层次及硬件架构，所以使异构众核系统比传统的同构多核系统编程更为复杂，如何充分利用异构众核计算资源来提升应用程序的计算性能已经成为当前的研究热点。面临的主要问题有可编程性问题，性能优化问题。

目前针对异构众核系统硬件架构缺乏较好的统一编程模型，没有提供一种有效的完全透明的编程语言，现有的编程模型没能较好的封装底层硬件架构的异构性。用户开发异构系统程序的难度较大，同时由于缺乏较好的性能优化机制，异构系统的计算资源得不到充分利用，不能充分发挥异构众核系统高效计算能力。基于以上异构众核系统存在的问题，学术界对异构系统下如何降低编程难度，如何提高编程效率及性能优化方面做了大量的研究工作。

### 异构系统编程模型研究现状

并行编程模型是底层体系结构与上层应用程序之间的桥梁，向上隐藏并行处理器的细节，提供给程序员并行表达的方法；向下充分利用硬件资源、高效正确地实现应用[6]。并行编程框架是对某一类问题解决方案的最高抽象形式，它由一套应用程序接口API和系统运行时函数组成。用户需要完成的工作是根据具体的应用问题和该框架的并行处理方式，设置接口函数的参数值，调用相应的接口函数。接口函数根据用户指定的参数值，对操作数据集进行划分，然后调用相应的任务处理函数和系统运行时函数自动完成问题的并行求解。异构系统下同样是按编程模型或编程框架的方式来解决可编程性问题。

当前异构系统编程模型一种主要的设计方法是通过对已有的语言进行扩展，提供一种高级的异构多核编程抽象机制，让用户利用较通用的并行编程语言或更接近高级语言的简单语言编写具体的应用程序，然后通过特定的编译器生成异构系统底层目标代码。用户在开发应用程序时不用考虑异构系统具体的硬件特点，主要从算法层面考虑应用程序的开发，而具体的任务分配、线程配置、数据传输、存储访问、指令执行的优化工作由编译器来完成，相应的编译器将用户代码自动转换成适合异构系统特点的目标代码，在异构系统上运行。

这方面的研究大多针对CPU+GPU异构众核系统，代表性的工作主要有：

Lee S等[7]提出了一种将通用并行编程语言OpenMP编写的标准应用程序转换成GPGPU代码的自动编译和优化框架。该编程框架的主要思路是让用户利用已有的通用并行编程模型来编写基于异构系统的并行程序，而由源到源编译器来完成具体的代码转换、性能优化及代码到具体硬件的映射工作，从而提高异构系统的可编程性。利用编译器来屏蔽异构系统底层的编程和存储访问的实现细节，代价是系统性能受到限制，要进一步提升性能需要额外手动的对编译器产生的底层代码进行调节，所以整个系统的扩展性、编程灵活性、系统性能都会受到影响。Han TD等[8]提出了一种基于指导语句的高级语言hiCUDA，它利用源到源编译技术为用户提供了一种简单的方式完成一般应用程序到CUDA（Compute Unified Device Architecture）程序的移植。该类研究通常只针对特定的异构系统进行编译指导支持，如果底层硬件架构发生变化，就要重新设计编译和指导机制，系统的可扩展性有限。研究能够提供一套标准的指导语句规范，根据异构系统不同的底层架构，使用不同的编译选项进行不同的编译，以适应异构系统底层硬件架构不断发展变化的要求，是解决系统可扩展性一个重要的研究方向。Baskaran MM等[9]实现一种通用串行C程序到CUDA程序的自动转换系统。该方法与前两种方法相比，对用户来说可编程性更好，是一种比较理想的解决异构系统可编程性的思路，用户直接用以前熟悉的串行编程模型进行程序开发，而由编译器将通用的串行程序直接转换成满足异构系统的并行代码，整个过程中无需用户参与，但这种方法只针对特定的硬件架构，程序的并行性受系统编译和自动并行转换技术的限制，系统的性能和可扩展性受到制约。Linderman M.D等[10]提出的Merge框架，为异构多核平台提供了并行编程模型、编译器和运行时支持；Dubach C.等[11]专门针对GPU的异构系统，设计了一种兼容JAVA的面向异构系统的高级编程语言Lime，这也是一种通过语言扩展的方式来提高异构系统可编程性的策略；Liu W等[12]提出了能够在CPU和加速器之间平衡计算任务，充分利用两类处理器计算能力的异构系统编程模型；Gelado I等[13]提出了一种非对称的分布式共享存储的异构计算编程模型，该编程模型通过轻量级方式实现虚拟的共享存储，可以克服对称分布式共享存储的不足。虚拟共享存储是异构计算的一个重要研究方向。

以上通过设计编程模型或编程框架来解决异构系统可编程性问题的主要特点是：侧重于如何降低异构系统的编程复杂性，用户在编程时主要从算法层面考虑，不用考虑具体的底层硬件架构特点；具体性能优化的工作由编译器完成，对程序员透明。

总体上，提高异构系统的可编程性可以从以下方面进行深入的研究：

（1）将原有的通用并行编程模型编写的并行程序通过源到源编译技术转换成满足特定异构系统要求的目标代码；

（2）通过扩展现有的通用编程语言，在原有的程序中加入一些指导语句，结合编译技术，实现通用程序到异构系统目标代码的转换；

（3）提供一种和异构系统底层硬件架构完全独立的共享虚拟存储的编程模型和语言，结合编译器和相应的运行时来提高异构系统的可编程性、可扩展性及性能；

（4）实现完全自动的将通用串行程序转换成满足异构系统要求的目标程序。

### 异构系统性能优化研究现状

由于异构众核系统和传统的多核系统在硬件架构上有很大差异，在具体应用时，用户要对异构系统计算资源进行显式管理，如果缺乏较好的性能优化机制，将导致异构系统的计算资源得不到充分有效的利用。研究已表明优化和未优化的应用程序性能可能相差数十倍或更大[14]。Ryoo S等[15]分析了GPU异构平台上的性能优化问题，总结了具体的优化原理，同时对具体的优化策略进行了测试验证，为异构系统性能优化提供了参考。目前针对异构系统性能优化主要通过设计编译器、提供优化工具及函数库、设计运行时系统三种方式实现。

1. 通过设计编译器进行性能优化

通过设计编译器的方法对异构系统进行性能优化，主要使用静态的方法，结合一些抽象的数学变换来抽取出程序中的并行部分，分析各并行部分的依赖关系，从任务划分、存储访问、数据传输、负载均衡和向量化等方面进行优化。但这些优化大多只针对具有规则并行特征的应用程序，或对某一类专用的应用程序进行优化，整个优化的通用性有限，要使通用应用程序在异构系统上获得理想的性能，目前仍然需要借助一定的手动优化方法。Baskaran MM等[16]设计了一个对具有规则循环嵌套的GPGPU程序进行优化的编译器框架，利用具有抽象数据依赖分析和程序转换特点的多面体模型编译器对全局和共享存储访问进行优化，通过实验探测的方法确定任务分割及循环展开参数，使程序在具体执行时线程数和相应的计算资源能够合理的匹配，从而保证异构系统的计算资源得到有效的利用，提高系统的性能。Jang B等[17]针对异构系统存储子系统具有的异构性和分布式的特点，建立能反映循环嵌套中存储访问模式的数学模型，使用数学的方法对程序的存储访问特点进行建模，具有较强的抽象性和通用性。Sundaram N等[18]提出了一个在GPU上运行专用应用程序的软件框架，该框架主要考虑了GPU编程中的两个关键问题，一是所要处理应用程序数据远远超过GPU存储容量限制的程序运行问题；二是最小化CPU和GPU之间数据传输问题，为异构系统下处理和优化大规模专用应用程序提供了借鉴。Liu Y等[19]通过探测程序输入参数对GPU程序优化的影响，提出了一个基于编译器的自适应优化框架G-ADAPT。

1. 通过优化工具和函数库进行性能优化

通过提供优化工具和函数库的方式帮助程序员对编写好的满足具体异构系统要求的初始并行应用程序进行优化。程序员在编写具体应用程序时不用过多考虑底层硬件特点和具体的优化，只要编写出基本满足异构系统要求的代码，利用所提供的优化工具对源程序进行检测和分析，找出程序中影响性能的瓶颈部分，然后调用相应的函数库按特定的策略进行优化，从而充分利用异构系统的计算资源，提高应用程序的计算性能。同时针对具体的异构系统，将一些科学计算中常用的典型数据密集型应用经优化和并行化后封装成完整的核心函数库，用户在开发具体程序时直接通过API调用对应的函数库就可以充分利用异构处理器来提升整个应用程序的计算性能。如NVIDIA的CUBLAS数学库[20]和Intel MIC MKL核心函数库[21]等。Lee J.等[22]提出了一种多线程感知的预取机制来优化GPGPU性能。YangY等[23]提出了一种GPGPU新的优化编译器框架。同时针对GPU异构系统，为了充分挖掘系统的处理能力，的发挥主处理器和协处理器的计算性能，文献[24]从如何通过共享存储复用技术来提高可同时运行在GPU上的线程数方面对系统的整体性能优化进行了研究。文献[25]针对将应用移植到CPU-GPU异构并行系统上时优化策略各自分散、没有一个全局优化策略的问题，提出了一种基于剖分的优化策略库、剖分工具库和策略配置模块组成全局性能优化方法，通过三个模块的相互配合以渐近的方式完成应用的全局优化过程。

优化工具和函数库大多数只侧重某一方面的性能优化问题，或针对某种具体应用进行优化，多数情况用手工编码的方法实现优化，如何将各种优化技术整合，实现动态自适应优化，从整体上提高计算性能仍存在一定的困难。

1. 利用运行时支持技术实现性能优化

通过为异构多核平台提供运行时支持实现应用程序的自动优化和软件到硬件的映射，达到降低异构多核系统编程难度、提高性能的目的。Wang等[26]提出了一种将异构加速器表示成基于ISA的MIMD应用级计算资源的体系架构EXOCHI；Luk等[27]提出了一种自动将计算任务映射到异构多核处理器运算单元上的自适应映射（Adaptive mapping）技术；Jablin TB等[28]提出了一种新的完全自动的CPU-GPU通信运行时管理系统。

以上针对异构多核架构，从如何提高异构系统的通用性，如何降低编程难度及性能优化方面所做的工作仍然要依赖于底层具体的硬件架构和对应的软件支持，如果底层硬件架构一旦发生变化，就要根据具体硬件架构特点重新设计整个编译及运行时系统、性能优化库和工具。所以系统的可扩展性和软件可移植性仍然不能较好的适应新型异构系统快速发展的趋势。

### 异构系统进一步的研究方向

由于目前缺乏一个标准化编程环境来统管异构系统内呈多样化发展态势的各种资源，异构系统的发展和普及受到了限制。在提高异构系统通用性方面，OpenCL创造了一个独立于硬件的软件开发环境[29]，它支持不同层次的并行，并能将应用程序高效映射到由CPU、GPU、FPGA和其他潜在的未来设备组成的单一或多设备的同构或异构系统。但与CUDA类似，OpenCL提供了一种底层语言抽象层接口，使用者需要对整个设备的硬件架构有相当的了解，用户编程难度依然很大，业界仍需要一种能够有效支持跨平台且低学习门槛的统一编程规范的出现。OpenACC是最新出现的一种跨平台、基于指令的隐式编程模式[30]，致力于为不同的异构硬件平台提供统一的编程环境。与OpenCL相比，OpenACC保持了良好的跨平台性并且拥有更高层次的抽象。目前已经有多家厂商提供设备和编译器支持[31][32]。

通过以上分析可知，为了适应快速发展的异构系统硬件架构的要求，通过设计与异构系统底层硬件架构和对应的支持软件完全独立的编程框架来解决异构系统的可编程性、可扩展性和软件可移植性问题是一种发展趋势。该框架下，用户只从程序算法层面进行并行程序开发，而不用考虑具体异构系统底层硬件特点。编程框架会根据应用程序特点将程序转换成通用的中间代码，然后再由相应的运行时系统负责将中间代码映射到具体的异构系统上运行。而运行时系统在实现具体映射时，根据在操作系统中设置的系统信息感知模块获取必要的硬件信息来完成计算任务到具体硬件上的映射执行。当底层硬件架构发生变化时，对整个编程框架来说，不用重新设计运行时系统和前端通用程序到中间代码的转换部分，只根据具体的硬件架构特征改变或增加相应的运行时库，就可以完成软件到新的异构系统的移植，保证整个编程框架具有较好的可扩展性，以适应不同的异构系统。

## 异构众核系统线程映射面临的关键问题

在主处理器+协处理器架构的新型异构众核系统下，主处理器主要负责处理复杂的逻辑控制任务，协处理器负责处理计算密度高、逻辑分支简单的大规模数据并行任务，主处理器和协处理器协同计算任务的不同部分，为具体的应用程序提供高效的计算平台。在具体运行程序时，如何将线程合理的映射到异构众核系统具体的处理核上，实现程序应用特点和异构众核系统计算资源相匹配，对程序性能、异构系统整体能耗都将产生重要的影响。在具体实现线程到处理核映射时主要面临以下关键问题：

（1）首先面临的关键问题是，如何合理设置多线程应用程序的线程数。异构众核系统下，主处理器一般采用通用的片上多核处理器（CMP），因处理器核数有限，通常在CMP上将线程数设置为具体的处理器核数，就能达到较好的加速效果。协处理器通常采用新型众核处理器（如GPU，MIC），由于处理器核数众多（通常几十、上百个核），同时处理核内部又引入了同时多线程（SMT），这使得影响因素更加复杂，如果设置的线程数过多，会因线程间争用共享资源导致计算性能下降，同时启用太多的处理核会造成过高的能源消耗；设置的线程数过少，则因异构系统处理核资源得不到充分利用，影响计算性能的提升。

（2）因异构系统下主处理器和协处理器具有不同的硬件架构，同时应用程序在具体执行过程中，运行行为可能会发生阶段性变化，要根据程序的运行行为，结合具体运行平台架构特征动态的调整线程数，才能较好的实现应用特点和计算资源的合理匹配。根据应用程序运行时不同阶段对计算资源不同需求的特点，设计线程数动态调整机制对充分利用不同处理器的硬件资源、高效发挥异构众核系统的计算能力及提高异构众核系统整体效能具有重要意义。

（3）在将不同线程映射到具体的处理核上执行时，如何将应用程序线程的数据相关性和异构众核系统物理架构特点结合，实现应用线程数据局部性和应用平台架构空间局部性较好的匹配，减少存储访问延迟及由于共享资源竞争而造成的过高额外开销，在保证充分利用处理核计算资源的同时尽量降低系统能耗，对程序性能、异构系统整体能耗都具有重要意义。根据不同线程之间的数据局部性特点，结合异构众核架构特征将线程合理分配到相应的处理核上，最大程度的降低线程之间的数据通信开销，提高程序的整体计算性能，是线程到处理核映射时要解决的一个关键问题。

## 论文的研究内容及创新点

本文针对异构众核系统线程映射面临的以上关键问题，在分析异构众核系统相关技术及研究现状的基础之上，对异构众核系统线程映射时的线程数预测、线程数动态调整、数据局部性及线程数据相关性度量、基于线程数据相关性的映射机制进行了深入的研究，取得的主要创新之处包括：

1）构建了一种面向众核系统的最优线程数预测模型

为了达到高效、简单、精确的预测最优线程数的目的，结合应用程序本身特点及其在具体MIC异构众核系统上运行时的动态阶段性变化特征，分析不同线程数对程序性能影响，抽象影响因素，在充分考虑多线程执行时不同线程之间的相互作用对程序性能影响的基础上，通过扩展Amdahl定律，根据反映程序执行性能的主要性能指标IPS（Instrpucitons Per Second），结合回归分析理论，利用最小二乘法构造最优线程数预测模型TCPM（Thread Count Prediction Model），并利用假设检测理论对TCPM预测模型的有效性进行了验证。TCPM预测模型可以在保证预测精度的情况下，实现高效、简单的最优线程数预测，可作为动态线程映射机制的理论基础实现对应用程序线程数实时动态的调整。

2）提出了一种基于程序阶段性变化的线程数动态调整映射机制

针对已有的动态调整线程数映射机制存在额外开销高、通用性差、依赖于特定硬件、不能实时调节异构众核系统下运行程序并行性等问题，提出了一种面向异构众核系统，基于程序阶段性变化的线程数动态调整映射机制DPTM（Dynamic Prediction Thread Mapping）。以最优线程数预测模型为基础，通过设计程序阶段性变化动态检测机制，实时的检测程序运行状态信息，根据检测到的状态信息动态的预测程序不同执行阶段的最优线程数，同时根据最优线程数对程序执行过程中的线程数进行动态调整，使程序在执行过程中并行性达到最佳状态，同时结合操作系统对计算资源的管理机制，在充分利用众核系统计算资源提升程序计算性能的同时，降低系统的整体能耗，提升异构众核系统的整体效能。

3）提出了一种基于数据重用距离的线程间数据相关性的度量方法

为了能准确的定量分析反映应用程序不同线程之间数据局部性关系的数据相关性，将应用程序线程数据相关性和异构众核系统物理架构特点相结合，实现应用线程数据局部性和应用平台架构空间局部性较好的匹配，研究了不同线程之间的数据局部性及存储访问特点，通过剖分不同线程的存储访问数据，计算线程的数据重用距离，基于数据重用距离设计了不同线程之间的数据相关性的度量方法DRAQ（Data Reuse Distance based Data Affinity Quantifying）。根据数据重用距离信息，分析线程内部数据局部性特点及不同线程之间的数据相关性，该数据局部性和数据相关性和具体的运行平台无关，反映了程序固有的数据依赖关系。具体采用模式分类的方法将不同的线程归并为不同的局部性模式，根据线程不同的局部性模式，用相关性矩阵（Affinity 矩阵）和相关性图来度量不同线程间的数据相关性。

4）提出了一种基于数据相关性的线程分组映射机制

为了能根据不同线程之间数据相关性特点，结合异构众核架构特征将线程合理分配到相应处理核上，减少存储访问延迟及由于共享资源竞争而造成的过高额外通信开销，实现高能效的线程映射，提高程序的整体计算效能，提出了基于数据相关性的线程分组映射机制DagTM（Data Affinity Grouping based Thread Mapping）。根据线程数据相关性图，对应用线程进行逻辑分组，然后结合具体异构众核系统硬件架构特点，通过线程组到处理核静态绑定实现线程到处理核的合理映射，将具有较好数据相关性的计算任务分配到同一个处理核的相邻硬件线程之上，使程序的数据相关性和运行平台的架构特点较好的匹配。具体分组时，将线程的逻辑分组问题抽象成一个图的分解问题，即将线程相关性图分解为满足上述要求的Ｋ棵子树，实现将具有高数据共享的线程划分到同一个线程组内，将具有较强数据访问争用关系的线程划分到不同的线程组内，以降低因不同线程之间相互竞争共享计算资源或存储访问延迟而引起的过高额外通信及同步开销。

## 论文组织结构

本文针对异构众核系统的性能优化问题，主要从计算任务到处理单元的映射层面来优化异构众核系统的计算效能。研究主要集中在最优线程数预测、线程数的动态调整、线程间数据相关性度量、线程的分组映射四个方面。全文共分七章，具体组织结构如下。

第一章 绪论。主要介绍论文的研究背景及意义，综述了异构众核系统编程模型及性能优化的研究现状，分析了线程映射面临的挑战，概括了论文的主要研究内容及创新点。

第二章 研究工作基础。主要从异构众核架构特点、编程特点、计算性能、发展趋势等方面对异构众核系统进行了概述；对两种典型的异构众核系统的计算性能进行测试，验证了两种异构系统不同的应用特点，为用户选择具体异构系统提供参考；利用具体的应用实例，对MIC异构众核系统线程映射及性能优化进行分析；针对线程映射面临的关键问题分析了相关的研究工作和存在的问题，阐述了本文的研究动机。

第三章 面向众核系统的最优线程数预测模型。针对众核系统下如何设置合理线程数的问题，研究多核多线程应用程序运行时不同线程之间相互作用对程序性能的影响，分析出现多样性对应关系的原因，抽象影响因素，通过扩展Amdahl定律，结合回归分析理论，提出了一种众核系统下最优线程数预测模型。

第四章 基于程序阶段性变化的线程数动态调整机制。给出了线程数动态调整的映射机制DPTM，通过实时的检测程序运行状态及阶段性变化，根据收集到的反映程序运行状态硬件性能计数器信息，利用所提出的数学模型直接计算出最优线程数的预测值，然后根据计算出的最优线程数对程序的并行性进行调节，以提升应用程序的计算性能，降低系统的整体能耗。

第五章 基于数据重用距离的线程数据相关性度量方法。通过剖分不同线程存储访问数据信息，设计并行线程数据重用距离计算方法，计算不同线程的数据重用距离；在数据重用距离的基础上，通过定义反映不同数据相关性的局部性模式，将不同的线程归并为不同的模式类；在不同模式类的基础上通过比较不同线程间数据共享量，用相关性矩阵来记录、度量不同线间的数据相关性。

第六章 基于数据相关性的线程分组映射机制。针对众核系统下线程到处理核的映射问题，以应用程序不同线程数据相关性特点为基础，结合具体众核系统硬件架构特征，根据线程相关性图，利用最小生成树实现不同线程的逻辑分组；通过线程组到处理核的绑定实现将应用线程合理的映射到众核处理器不同的处理核上，达到减少不同线程间的共享资源冲突，提高共享资源的利用率，降低数据传输开销，提高系统的整体计算效能的目的。

第七章 结论与展望。总结了本文的研究工作，对进一步的研究工作进行了展望。

# 研究工作基础

本章对异构众核系统相关研究领域的发展趋势、相关技术、面临的问题进行分析和总结，作为整个论文研究工作的依据和支撑。异构众核系统已经成为当前高性能计算领域重要的发展趋势，针对异构众核系统，从架构、编程、所支持的应用三方面分析对比当前不同异构系统的特点，揭示了异构系统的发展趋势及异构系统相对于传统多核并行系统的优势；针对两种典型的异构众核系统CPU+GPU和CPU+MIC进行不同应用类型的Benchmark测试，验证了两种异构系统不同的应用特点，为用户选择具体异构系统提供参考；为了研究线程到集成众核处理核心不同映射方式对性能的影响，测试对比不同映射方式下MIC集成众核的计算性能，总结了合理的多线程映射应考虑的因素；在MIC异构众核系统下，通过对具体的k-means应用程序进行移植和并行优化，讨论了MIC异构系统下应用程序的移植和优化过程，分析了具体移植过程中影响并行程序性能的各种因素，从不同的方面考虑了具体的优化方法，对具体的优化性能进行测试和分析，为本文具体的研究提供理论基础和技术支撑；针对应用程序线程映射面临的关键问题，分析了相关的研究工作及存在的问题，阐述了本文的研究工作动机。

## 异构众核系统概述

以异构众核为特征的新型异构系统已经成为构造现代高性能计算机系统的主要技术，且应用越来越广泛。异构众核系统因其可有效解决能耗、可扩展性等问题，已经成为全球高性能计算领域中的一种重要新兴模式。2012年11月发布的TOP500榜单[33]中，已有62套系统使用了主处理器+协处理器（加速器）混合架构，其中排名榜首的Titan和排名第八的我国Tianhe-1A都采用了NVIDIA GPU作为加速计算部件。而同年6月份的榜单中只有58套系统使用了主处理器+协处理器（加速器）架构，这一变化充分说明在高性能计算领域异构系统发展之迅速。首次进入该榜单TOP10，排名第七的Dell公司的Stampede系统使用了最新的Intel MIC Knights Corner协处理器，同时TOP500整个榜单中还有其它6个系统也同样使用了最新的Intel MIC Knights Corner协处理器。而仅仅六个月前的上次TOP500排名中，只有一套系统使用了Intel MIC Knights Corner协处理器，且位列TOP500第150位。2013年6月17日发布的TOP500榜单中，我国Tianhe-2首次以每秒33.86千万亿次的浮点运算速度荣登TOP500榜首，其采用的也是主处理器+协处理器的混合架构，且使用了最新的Intel MIC Knights Corner协处理器，且自2013以来至2015年11月连续6次位列世界超级计算机TOP500之首。这一变化进一步说明新型异构系统因其高性能低功耗的优势，在高性能超级计算机领域的地位越来越重要。图2-1是根据TOP500榜单统计出的最近6年内12次排名中的异构系统数量变化趋势图。

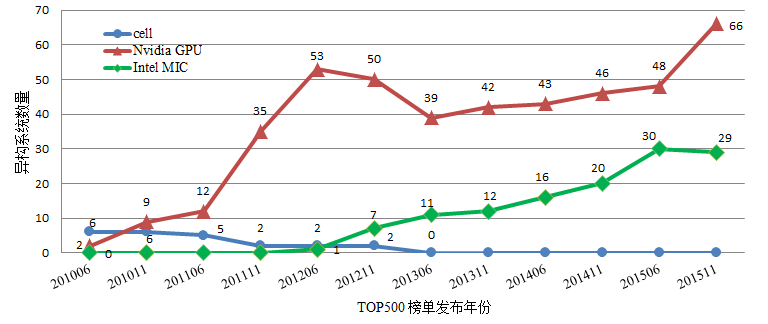


图2-1 TOP500榜单中近12年中主要异构系统数量变化趋势

TOP500中异构众核系统架构的整体发展趋势，反映出异构系统将是未来高性能计算机系统重要的发展方向。随着新的协处理器技术的出现，这种混合架构将会在后续的万万亿次、百亿亿次超级计算机系统中发挥更重要的作用。

本节针对当前高性能计算领域三种典型的异构系统架构CELL、CPU+GPU、CPU+MIC，具体分析各种架构的特点、编程特点、各自的应用范围。重点对两种典型的片间异构系统CPU+GPU和CPU+MIC从编程特点方面进行论述和对比，分析它们处理应用时不同特点，总结它们不同的应用范围。

### 异构众核系统架构特点

当前异构系统主要有以下几种架构：Cell/B.E、CPU+GPU、APU、CPU+MIC及CPU+FPGA。其中FPGA主要用在嵌入式系统中的高性能需求中，而传统意义上的高性能领域使用的异构系统架构主要有Cell/B.E、CPU+GPU、APU和CPU+MIC。本文主要针对传统意义上的节点级异构，故对FPGA不作详细讨论，对其详细的论述可参考文献[2]。

#### Cell/B.E.异构系统架构

Cell/B.E.(Cell Broadband Engine)是依据CBEA (Cell Broadband Engine Architecture)技术所实现的处理器，即通常所说的Cell处理器。就技术而言，Cell/B.E.是以原有PowerPC架构为基础，再针对娱乐用3D绘图运算的需求添加协处理单元设计而成。

标准的Cell处理器内置一个通用PPE(PowerPC Processing Element)和8个SPE(Synergistic Processing Element)，它们通过高速互连总线EIB (Element Interconnect Bus)和存储流控制器(MFC, Memory Flow Controller）、内存接口控制器（Memory Interface Controller）、总线接口控制器（BIC，Bus Interface Controller）相连。PPE内置32KB一级指令缓存和数据缓存及512KB二级缓存。Cell处理器的计算能力主要来自SPE，SPE由协处理单元SPU和256KB本地存储LS (Local Storage)构成，协处理单元SPU只能直接访问LS中的数据和代码，如果应用的数据和代码超过了LS容量的限制，就必须显式地通过效率远低于访存指令的DMA操作，从片外系统主存中获取。在3.2GHz主频下，SPU提供对LS高达51.2GB/s的高带宽访问，而通过DMA由高速片上互连总线从系统主存获取数据的带宽为25.6GB/s。异步DMA操作可与SPU计算重叠，实现通过数据预取来隐藏访存延迟。在Cell架构下，每一个处理核都可以按MPMD的方式运行独立的程序。Cell处理器架构如图2-2所示[2]。

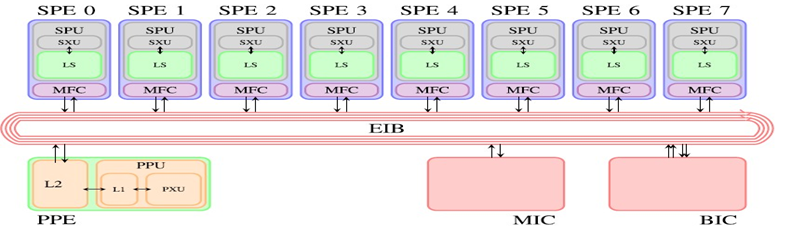


图2-2 Cell处理器架构

Cell处理器是一种完全独立的异构处理器，使用成本较高，现在只用在一些高端服务器上[5]。由于其程序开发难度较大，限制了Cell处理器在通用高性能领域的普及，同时也制约了它的进一步发展，它在异构系统中的优势逐渐衰退。

#### GPU异构系统架构

GPU异构系统主要采用CPU+GPU架构。GPU的主要设计目标是以大量的线程实现面向高吞吐量的数据并行计算，适合于处理计算密度高、逻辑分支简单的大规模数据并行任务。而CPU具有复杂的逻辑控制单元和大容量的缓存，有低的数据传送延迟，能够适应各种不同类型的运算，尤其擅长复杂的逻辑运算。采用CPU+GPU架构，将CPU和GPU各自优点结合起来，让GPU处理数据密集型的并行任务，而由CPU进行复杂的逻辑事务处理，从而充分发挥CPU和GPU各自优势，最大限度的利用异构系统的处理能力，降低计算成本和能耗。图2-3为GPU异构系统架构[5]。



图2-3 GPU异构系统架构

CPU和GPU一般经北桥芯片通过PCI-E或AGP总线连接，各自有独立的外部存储器，分别为内存（Main memory）和显存（Device memory）。如图2-3所示，每个GPU包含多个SIMD（Single Instruction Multiple Data）的Stream Multiprocessor（SM），GPU支持成百上千个线程的同时处理，并将所有的线程分成相应的线程组，每个线程组分成一定数量的调度单元，并分派到一个流多处理器SM（Stream Multiprocessor）上进行处理。每个SM内部又有若干个流处理器SP（Stream Processor），如图2-2中的sp2,sp2…spn，每个SP执行每一个具体的线程。每一个线程组中线程共享每个SM中的计算资源，比如寄存器（Register），Share Memory，Local Memory。GPU的Device Memory（设备存储器）主要由Global Memory，Constant Memory，Texture Memory组成，这些存储器可以被所有的线程共享，同时可以直接和CPU主存进行通信，从而完成CPU和GPU之间的数据交互。

在CPU+GPU架构下，具体处理计算任务时，CPU负责逻辑性较强的事务计算，GPU则负责数据密集型的并行任务计算。因事务计算在处理时，要求执行单元要以很低的延迟获得数据和指令，因此采用了具有复杂控制逻辑和分支预测、以及大量的缓存来提高执行效率的CPU来处理事务计算；而为了达到高吞吐率的数据密集型并行任务计算的目的，GPU必须在有限面积上实现很高的计算能力和很高的存储器带宽，因此需要大量的执行单元来运行更多的相对简单的线程，在当前线程等待数据时就切换到另一个处于就绪状态、等待计算的线程。GPU使用由硬件管理的轻量级线程，可以实现零开销的线程切换，线程切换在这种情况下对实现高吞吐率的数据计算是非常有利的。当一个线程因为访问片外存储器或者同步指令开始等待以后，可以立即切换到一个处于就绪状态的线程，从而达到用计算来隐藏延迟的目的。当线程中的计算指令需要的时间较多，而访存相对较少，即计算密度较高时，就可以很好的用计算来隐藏访存延迟，而且线程越多，延迟隐藏的越好。

#### APU 异构系统架构

APU（Accelerated Processing Unit）是AMD推出的整合了x86/x64 CPU处理核心和GPU处理核心的新型“融聚”（Fusion）处理器。具体架构如图2-4所示[35]。APU将通用x86架构CPU核心和可编程矢量处理引擎相融合，把CPU擅长的精密标量运算与传统上只有GPU才具备的大规模并行矢量运算结合起来。AMD的APU设计综合了CPU和GPU的优势，为软件开发者带来前所未有的灵活性，可采用最适合的方式开发新的应用。APU通过一个高性能总线，在单个硅片上把一个可编程x86 CPU和一个GPU的矢量处理架构连为一体，CPU和GPU分别拥有自己独立的L1级和L2级Cache，同时双方共享片上L3级Cache且能直接读取片外高速内存。APU中还包含其他一些系统成分，比如内存控制器、I/O控制器、专用视频解码器、显示输出和总线接口等。

APU的主要特点是它内含由标量和矢量硬件构成的全部处理能力，由于将两种计算核心整合在一起，受芯片空间和制造工艺及散热的制约，处理器的主频及处理核数目受到限制，同时当前APU中X86 CPU和GPU矢量处理器还没有实现无缝集成，所以其整体计算性能与主处理器和协处理器分离的架构相比稍有不足。



图2-4 AMD APU架构

#### MIC异构系统架构

MIC(Many Integrated Core)，是Intel 2011年推出的新一代集成众核协处理器，它通过对传统的微处理器进行向量扩展，将多个扩展后的核心整合在一起，进一步提升计算能力。MIC协处理器主要面向HPC领域，旨在引领行业进入百亿亿次计算时代，但其在计算机体系中，并非是用来取代现有处理器产品，而是作为提高HPC服务器运算性能的协处理器[35]。Xeon Phi（Intel推出的基于MIC的高性能计算协处理器的系列产品代号）必须和现有服务器的处理器搭配使用，而无法独立存在，所以Xeon Phi以PCI-E适配卡的形式进行设计。服务器上安装适配卡之后，Xeon Phi可以和服务器上搭载的Xeon处理器组成运算集群来执行并行程序，或者作为服务器中的内部子系统，就像是系统中的系统一样，服务器可以派发任务给Xeon Phi适配卡中的子系统来运算，从而达到协处理器和主处理器协同计算来提升计算性能的目的。

MIC协处理器的独特之处在于：它不像传统的加速器，它本身具有独立的微操作系统μOS，所以更像是一个能被访问、被编程、功能全面的高性能计算节点，在应用程序看来，它就如同一个运行着自身的基于Linux的操作系统（与主操作系统无关）的计算机。图2-5为MIC异构系统架构[36][37]。

MIC协处理器主要有以下部分组成：

（1）计算核心（VECTOR IA CORE）。每个微处理器核心是一个功能齐全、彼此独立、能够按序执行指令的X86计算核心。这些核心支持硬件多线程，每个核心支持四个硬件线程。四个硬件线程由硬件线程调度器统一调度后以顺序轮询的方式并行执行。每一个计算核心拥有一个512bit线宽的向量处理器（VPU），它能以16\*32bit（整型或单精度浮点类型）或8\*64bit（双精度浮点类型）的组合模式工作。同时有一个标量处理部件X86单元（Scalar Unit），它是x86架构的标量微处理器单元。除此之外还有L1 Code Cache及L1 Data Cache、L2 Cache、Cache未命中处理单元、计算核心与内核环形总线接口CRI（The Core Ring Interface）等。

（2）全局共享的一致性二级缓存（coherent cache）。MIC中的每个核拥有一个全局可见的L2 Cache，将每个核的L2 Cache组合起来构成了一个全局共享的大容量一致性的二级缓存。在具体使用一致性二级缓存时，为了减少应用程序热点部分各计算核心之间的数据访问竞争，MIC实现了一套分布式标签目录，使得MIC处理核心所能访问的物理地址都通过一个可逆的一对一的地址变换哈希函数，映射到该标签目录内。该哈希函数不仅把每个物理地址都映射成一个索引标签，而且还提供了一套多核一致性框架，实现比单核体系更复杂的一致性协议，从而保证高效正确的使用共享的二级缓存。

（3）片上环形互连网络（the core ring interprocessor network），包含MIC上各构件的接口、环站点、环轮转控制单元、寻址单元、流量控制单元。MIC协处器有两个这样的环，分别对应两个方向。在某些情况下，目的端（数据接收端）没有足够的空间来接收信息时，可以将信息暂时留在环上，并在下个周期接收信息。计算核心、内存控制器、PCI-E接口及其私有的L2 Cache 通过CRI链接到片上环形网络，实现高速通信。

（4）存储控制器。MIC协处理器总共有8个存储控制器，用来读/写外部存储设备，每个存储控制器都基于GDDR5规范，有2个数据传输通道控制单元，每个通道控制单元控制一个32bit宽度的访存通道，使MIC协处理器最高传输速率可达到5.5GT/s, 计算核心与GDDR5存储之间的数据传输理论带宽高达352GB/S。

（5）I/O接口。I/O接口由终端逻辑单元进行控制，它是MIC系统与Host端CPU或其它PCI-E设备通信的接口，支持PCI-E x8或PCI-E x16线宽配置，双向通道带宽可分别达到8GB/S和16GB/S。



图2-5 MIC异构系统架构

综合以上分析可知，CELL和APU是一种片内的异构系统，而GPU和MIC则是要和主处理器（CPU）配合以对应用进行加速的片间异构系统。根据目前的发展趋势，GPU和MIC这种以加速卡的运行方式更具通用性，同时这种架构的使用成本相对较低。因为采用GPU和MIC加速不需要从物理上对已有的硬件系统进行较大的改动，只需要将GPU卡和MIC卡通过PCI-E接口和已有的通用处理器相连，就可以构成异构系统，可以在保留原有硬件投资的基础上，相对容易的以较低成本实现原有计算系统向异构系统的扩展，为普通高性能计算用户提供一种经济、可行、实际有效的计算、仿真和验证平台。

### 异构众核系统编程特点

目前主流的异构众核系统从本质上来说，就是利用协处理器高的并行处理能力配合主处理器完成一些高数据密集型的并行计算任务来对整个处理任务进行加速，提升应用的整体运算能力。如何较好的利用协处理器的高并行数据处理能力，如何保证协处理器很好的和主处理器协作是异构系统下编程所要解决的关键问题。目前的计算模式主要有：SISD(Single Instruction Single Data)、SIMD(Single Instruction Multiple Data)、MIMD(Multiple Instruction Multiple Data)三种方式，具体的计算模式示意，如图2-6所示。

(a) SISD 计算模式 (b) SIMD计算模式



（c）MIMD计算模式

图2-6 不同的计算模式

GPU的并行计算方式属SIMD（Single Instruction, Multiple Data），单一指令多重数据的计算模式，所有运算核心都执行同一个指令，只是作用在不同的数据上。Intel MIC 运行方式则属于MIMD计算模式，Intel MIC Xeon Phi 处理器中的60个核心，每个核心都可以处理不同的指令和不同的数据。本节针对通用高性能计算领域，以易扩展和有效实用性为基础，重点对目前通用的GPU和MIC两种典型的节点级异构系统从编程特点方面作进一步讨论。

#### GPU编程特点

在GPU异构系统上最有代表性的编程模型是NVIDIA的CUDA（Compute Unified Device Architecture），也是当今最流行的GPGPU(General Purpose Computing on Graphics Processing Unit)语言。与以往GPU编程语言不同的是，CUDA不再面向图形计算，它提供编译器和开发工具，让开发人员用一种类C的编程语言来开发GPGPU程序，并同时运行于CPU和GPU上[34]。

CUDA C提供给开发人员一个完整的接口，可以访问NVIDA GPU的本地命令集、存储器等并行计算元素，使它们变成像CPU一样的开放式的架构。CUDA C是含有NVIDIA扩展和限制的类C语言，它支持大多数C语言的特性和语法（除了某些特性，如函数指针、递归函数），并加入了能使程序在GPU上进行多线程计算的语言扩展，CUDA提供的nvcc编译器能自动区分代码中的C语言部分和扩展部分，并将C部分交由本地编译器编译，而nvcc负责编译CUDA的扩展部分，最后链接所有的目标文件，生成可以直接运行在NVIDIA GPU上的机器代码。程序的具体执行流程如图2-7所示[38]。



图2-7 CUDA程序执行流程

图中CUDAfe负责分离源代码中的主机端和设备端代码，分离后的主机端C代码由通用的c/c++编译器进行编译；设备端代码由nvcc编译器进行编译后生成\*.ptx形式的汇编代码，然后再经ptxas编译器将\*.ptx格式的汇编代码编译成满足GPU指令要求的CUDA二进制代码Cubin.bin；最后由标准的c/c++编译器将主机端代码和CUDA二进制代码Cuda.bin结合，并连接CUDA运行时API库Libcuda.a生成可执行代码，再由CUDA驱动调用CUDA运行时API，设置和分派计算kernel到具体GPU设备上运行，而主机端可执行代码直接在CPU上运行。

CUDA为GPU通用计算提供了统一的编程接口，提供近似高级语言的语法特性，突破了以往GPU编程时必须要深入了解底层硬件和图形处理接口限制，简化了GPU编程难度。但CUDA并不是一种完全透明的语言，它没有封装存储系统的异构性，在进行CUDA程序开发时，程序员还是需要对各种存储器的特点和局限性有清楚的认识，要根据硬件特点将任务进行合理分解，进行具体的线程配置，并在编程时对数据传输、存储器访问及指令流进行优化，这从根本上增加了编程难度和程序调试难度，且容易出错。从而影响了GPGPU程序的开发效率，阻碍了GPGPU向更广泛的具体应用领域的普及。

#### MIC编程特点

MIC架构的优点之一是可运行已有的x86应用程序，而不用将程序完全移植到一个新的编程环境中，可以比较方便的利用协处理器的高计算性能。MIC架构处理器能够支持标准化、现有的编程工具，大大方便了开发人员。MIC架构在单个芯片中融合了众多核心，这些核心都能够通过使用标准的C、C++和FROTRAN进行编程。而为MIC编写的这些源代码同样可以应用和运行在标准的至强处理器平台之上。MIC众核架构，在兼容性、高度并行运算和编程可控方面和GPU相比更具优势。

MIC卡拥有灵活的编程方式，即可以作为一个协处理器，又可以被看作是一个独立的计算节点。当MIC卡和主机端CPU配合时有以下五种应用模式，分别为：纯CPU计算模式；以CPU为主，MIC为辅的协同计算模式；纯MIC计算模式；以MIC为中心，CPU为辅的协同计算模式；CPU和MIC对等模式。较常用的方式是将MIC作为协处理器和主机端CPU协同来进行任务计算。

（1）以CPU为主MIC为辅的运行模式编程方法

这种模式是MIC编程中最常用的模式，适合于在串行程序中有部分高并行度的程序。这种模式又被称为offload模式，通过编译指导语句，如offload语句标示出需要在MIC卡上运行的程序代码段，但在MIC端要利用其强大的并行处理能力，必须要结合传统的OpenMP、pthread等并行编程语言才能在MIC设备上实现程序的并行化执行。Offload模式是以CPU端为基础，将MIC端作为协处理器运行的，CPU端作为控制端，完成任务分派和数据传输，MIC端作为计算端完成CPU分派的计算任务。

Offload模式通常用于结构是单节点，且已经被OpenMP改造过的程序。Offload模式支持单机单MIC卡和单机多MIC卡，但不支持使用offload语句控制多节点。如果要在多节点进行并行应用，需要在各节点并行程序的外围框架之间结合消息传递并行模式（如MPI），而节点内部（如MPI进程内）使用offload+OpenMP模式。

要将现有的串行程序使用offload模式移植成MIC程序，首先确定要并行化的部分，将串行程序改写为OpenMP并行程序，然后确定输入输出变量，在改写后的并行程序中，用offload语句标示出需要在MIC端运行的代码段，便完成并行代码段到MIC上的移植。接下来的编译，并行执行等操作完全由编译器完成。编译器隐藏了大量的细节使得MIC程序无论从用户友好角度还是从编译角度，都与现有的CPU程序，尤其是并行程序非常类似。

（2）CPU和MIC对等的运行模式编程方法

该运行模式下，CPU和MIC执行同样的任务，此时MIC卡可以看作是与CPU对等的一个计算节点。MIC之所以能和CPU对等的使用是因为MIC卡有属于自己的基于Linux的微操作系统μOS，且拥有自己的IP地址，可以执行与CPU端类似的程序。这种运行模式下，运行跨节点的并行程序时，需要使用MPI进行节点间的数据通信，此时不需要用offload语句进行标示，而是将全部程序的一个副本放在MIC端执行。在使用MPI方式运行程序时需要编译两份代码，在启动MPI程序时，在CPU端和MIC端分别启动对应的代码，以正确运行程序。该模式适用于对各计算节点并行代码段间的负载平衡要求较高的情况。

（3）MIC原生运行模式编程方法

MIC拥有自己的基于Linux的操作系统，且拥有自己的IP地址，支持卡上直接运行模式，可以将程序和所需要的数据手工传输到MIC卡上，直接在MIC卡上运行程序。在MIC卡上直接运行程序可以节省传输数据的时间，适合多次处理同一块数据的情况，通常用于程序整体算法是并行的，或部分代码虽然可以使用CPU执行，但传输、同步开销过大的情况。

MIC编程基于X86指令系统，能够支持现有的标准化编程工具和手段，但并行程序执行时由于存在多个任务/线程之间信息交互，使得应用程序行为变得复杂，因此需要从不同方面进行优化，才能使MIC计算性能达到最优。

## GPU、MIC异构众核系统性能测试

为比较两种典型异构众核系统CPU+GPU和CPU+MIC处理任务时的不同特点，选取了具有不同应用特征的Benchmark测试程序对两种异构系统进行了实测，通过对不同类型测试程序进行测试对比，比较两种异构系统性能差异，分析引起性能差异的原因，得出两种异构系统处理不同类型应用的特点，指出两种异构系统各自擅长的应用，从而为用户在实际应用时根据自己的应用程序特点选择具体的异构计算平台提供参考。

### 测试环境及测试程序

测试平台分别采用浪潮TS10K集群中的两个独立的MIC计算节点和GPU计算节点。两个计算节点配置为：

MIC节点有两块Xeon Phi 7110P MIC卡。GPU节点有两块Nvidia的kepler架构的k20m GPU卡。MIC节点和GPU节点都配备了两颗E5-2670 CPU，主存大小64G，CPU主存和加速卡之间采用通道带宽为x16的PCI-E总线进行数据传输。操作系统均为RedHat Linux 5企业版，MIC节点安装Intel parallel\_studio\_xe\_2013\_update3\_intel64软件开发环境，GPU节点安装了CUDA SDK 5.0软件开发包。

基准测试程序选取了以下六个具有不同应用特点的典型基准测试程序：SPGEMM、Monte Carlo、Histogram、FFT、N-body、K-means[39]。

### 测试结果及分析

图2-8和2-9给出了不同基准测试程序在CPU、GPU、MIC三种平台下的执行时间及相对于CPU串行运行时的加速比。从图中可以看出，SPGEMM、Histogram和K-means三类应用在GPU异构系统上的计算性能优于MIC系统，而Monte-Carlo、FFT和N-body三类应用在MIC异构系统上表现出了较好的计算性能。这是因为SPGEMM、Histogram和K-means属于细粒度高密集的数据并行类应用，而GPU是以大量的线程实现面向高吞吐量的数据并行计算，擅长处理计算密度高、逻辑分支简单的大规模数据并行任务，故以上三种应用在GPU上表现出了较好的性能。其它三类应用属于粗粒度并行类应用，该类应用是对同一批数据反复进行大量的运算，在MIC异构系统上运行则能发挥出MIC处理器处理粗粒度并行任务的优势，获得相对较好的计算性能。

总体来说，GPU适合处理细粒度大规模的数据并行型的应用，这类应用属于SIMD运算模式，所有的运算核心都执行同一个指令，只是作用在不同的数据上；MIC更适合处理粗粒度的任务并行类应用，它的特点是对一批数据反复进行大量运算，属MIMD运算模式。用户在具体应用时可以根据应用本身的特征，结合GPU和MIC不同的计算特点，选择合适的异构平台，以获得较好的计算性能。

图2-8 CPU/GPU/MIC三种运行平台下的计算时间对比

图2-9 GPU和MIC相对于CPU的加速比

## MIC异构众核系统线程映射及性能优化分析

由于通用多线程并行编程模型没有针对集成众核处理器(MIC, Many Integrated Core)设计专门的线程到处理核心的映射方式。在具体编程时，仍然要根据传统的OpenMP运行时内置的映射机制。不合理的线程映射，会导致MIC协处理器部分计算资源闲置，制约了MIC协处理器高计算性能的有效利用。为了研究线程到集成众核处理核心不同映射方式对性能的影响，分析引起性能差异的原因，以充分发挥集成众核处理器的高并行处理能力，通过测试对比不同映射方式下的计算性能，总结了MIC集成众核下实现合理的多线程映射时应考虑的因素。为了进一步提升性能，通过对具体的k-means应用程序进行移植和并行优化，讨论了MIC异构系统下应用程序的移植和优化过程，分析了具体移植过程中影响并行程序性能的各种因素，从不同方面考虑了具体的优化方法，并对具体的优化性能进行了测试和分析，从而为本文进一步深入研究异构众核系统下的具体性能优化策略提供理论基础和技术支撑。

### 线程到MIC处理核映射分析

在实现多线程并行编程时需要从两方面考虑计算负载的映射，一方面考虑计算负载到线程的映射；另一方面要考虑线程到具体处理核的映射。本节主要分析不同的线程到MIC处理核心上的映射对计算性能的影响。由于MIC卡通常集成了60个以上的处理核心，每个处理核心可以支持四个硬件线程，如Intel Xeon Phi处理器具有61个处理核心，总共可支持244个硬件线程。如何将不同的计算负载合理的映射到具体处理核心的硬件线程上，对充分利用集成众核计算资源，发挥其高计算性能至关重要。

#### 不同映射方式及其特点

在MIC卡上实现具体应用时，要结合相应的并行编程模型才能充分利用众核处理器的高并行计算能力，目前广泛使用的是OpenMP编程模型。OpenMP提供了专门的环境变量来控制线程到不同处理核心的映射方式，主要提供以下三种映射方式：

（1）紧密方式（compact），该方式以充分利用每一个单独的计算核心为主，在具体映射线程时首先保证一个硬件核心获得足够的线程，然后再将剩余线程分配到下一个核心，直到将所有的线程都分配出去，这种映射方式容易导致负载不均衡，但如果相邻线程间有数据交换或公共数据，则可能由于线程处于同一个计算核心而大大减少额外开销，从而有利于性能的提升。具体映射过程如图2-10所示，图中白色部分代表闲置资源，假定最大线程数为2n。

图2-10 Compact 映射方式

（2）分散方式（scatter），该方式下线程优先被分配到负载最轻的计算核心上，尽量保证计算核心的负载均衡，这种方式对相邻线程间没有数据依赖的情况有利。具体映射过程如图2-11所示。

图2-11 Scatter映射方式

（3）平衡方式（balanced），该方式同时兼顾负载均衡和数据局部性，保证将线程尽量均匀分配到不同处理核心的同时，将相邻的线程尽量安排在同一个计算核心之上。具体映射过程如图2-12所示。

图2-12 Balance映射方式

#### 不同映射方式下的性能测试

（1）测试环境

测试平台由2路8核的E5-2670 CPU和Xeon Phi 7110P MIC卡组成，CPU主存和加速卡之间采用通道带宽为x16的PCI-E总线进行数据传输，最高传输速度可达到16GB/s。采用Red Hat Enterprise Linux Server release 6.3操作系统。

具体的开发环境为：Intel parallel\_studio\_xe\_2013\_update3\_intel64软件包。

（2）测试方法

利用MIC的两种计算模式offload和native 分别模拟直线方程y=mx+b的计算，来测试在不同线程到处理核的映射方式下，整个处理器所能达到的最大计算性能。

在native模式下，测试compact，scatter，balanced三种映射方式下，线程数为：61，122，244，305时的程序计算性能。MIC卡总共有61个处理核，每个处理核最多可支持4个硬件线程，为了更好的反映线程到具体处理核心的映射及处理核的实际使用情况，测试时线程数设为处理核数的整数倍。

在offload模式下，总共可以用于计算的MIC处理核数为60，其中1个处理核用来运行微操作系统，负责主处理器和协处理器间的数据管理和任务控制，测试时线程数分别设置为实际用于计算的处理核数60的整数倍：120，180，240，300。测试程序计算规模随着所设置的线程数目的增加而相应的增加，而单个线程的计算任务量是固定的。

（3）测试结果及分析

图2-13给出了按以上测试方法，测得的在不同的OpenMP映射方式、不同线程数的情况下，利用MIC两种并行计算模式native和offload时的异构众核系统计算性能测试结果。

（a）native计算模式下的性能

（b）offload计算模式下的性能

图2-13 不同计算模式的性能

从图2-13(a)中可以看出，在native模式下balanced方式取得了较好的性能，因该模式下线程被均匀的分配到了各个处理核上，各处理核之间获得了较好的负载平衡。同时可以看到当线程数为244时取得了最好的计算性能，达到理论峰值（2.130TFlops）的94.6%，此时启用了全部处理核上的所有硬件线程。图2-13（b）中在offload模式下，balanced方式也取得了较好的性能，最高达到了峰值性能的93.3%。和native模式相比offload模式整体性能略差一点，因为该模式下，在运行过程中要将程序中加速部分的代码及相关数据加载到MIC卡上，计算完成后又要将计算结果传回到CPU端，引入了额外的传输开销。

#### MIC性能可扩展性分析

为了评测MIC卡的可扩展性，在native模式下，通过进一步增加计算规模和线程数，对MIC卡的性能变化情况进行测试，测试结果如图2-14所示。从图2-14中可以看到，当线程数从61、122、183增加到244时，三种映射方式性能都随着线程数的增加而增加，其中compact模式下性能几乎随线程数线性增加，表现出了很好的可扩展性。Compact模式在线程数小于244的情况下，部分处理核由于闲置导致性能较低，随着线程数的增加，所利用的处理核越来越多，当线程数达到244时，所有处理核得到了充分的利用，此时三种方式获得了基本相同的计算性能。当线程数超过244时计算性能开始下降，因为当线程数超过MIC卡所支持的最大硬件线程数（244）时，多个线程之间开始争用硬件线程上下文，争用硬件线程的线程数越多，引起的额外开销越大，对性能的影响也越大。随着线程数的继续增加，MIC卡的整体计算性能趋于最优性能，当线程数超过1000时三种映射方式性能逼近最好的性能，性能不再有明显的变化，基本达到一个较稳定的状态。同时可以看到当线程数超过244后，scatter和balanced模式几乎没有什么大的差异，因为在完全占满MIC卡硬件线程的情况下，由于测试程序中不同线程之间没有数据依赖，两种映射方式对程序的局部性没有大的影响，所以图中两条线几乎重叠。整体上来看，MIC表现出了较好的性能可扩展性，随着计算规模的扩大，在不断增加线程数目的情况下，性能随着MIC卡上物理线程数的增加而持续提升，当利用了MIC卡的所有的硬件线程后，性能达到理论峰值性能的93%以上。再继续增加计算规模和线程数时性能变化不大，保持在实际的最好性能附近。

图2-14 native模式下性能随线程数的变化情况

#### 线程映射规则总结

通过以上测试和分析，总结出以下线程映射规则：（1）在具体进行线程到处理核映射时要结合应用程序本身的应用特征，使用合理的映射方法来尽量充分利用集成众核的所有计算资源，降低通信等额外开销；（2）如果应用程序各线程之间有较强的数据依赖特征，使用compact映射方式将相关的线程映射到同一个处理核上，可充分利用时间和空间局部性来减少额外的通信开销；（3）如果应用程序各线程之间没有明显的数据依赖，而线程数较多时，利用scatter映射方式可以较好的实现各处理核间负载平衡，可以充分利用所有处理核的计算能力，提高整体计算性能；（4）如果要同时考虑线程间的数据依赖和负载平衡，使用balance映射方式可以较好的兼顾两方面的平衡，实现较好的计算性能。

### MIC异构系统应用程序优化实例

应用程序要充分利用MIC异构系统的高计算性能，不仅要考虑以上具体线程到MIC处理核心上的映射方式，同时在具体进行程序移植和优化时还要考虑其它影响性能的因素。为进一步分析MIC异构系统下影响程序性能的不同因素，以广泛应用在数据挖掘中的k-means聚类算法为例，通过具体的应用程序并行移植、优化、测试，讨论在MIC异构系统下进行具体应用程序的移植及优化方法。

#### 利用native计算模式直接并行化

利用native模式将OpenMP并行化的k-means程序[41]移植到MIC卡上，测试不同数据规模、不同线程情况下程序在CPU和MIC上执行时的性能差异。

图2-15给出了k-means程序在不同的映射方式下，分别在CPU和MIC上的运行时间及不同数据规模下的加速比对比。

图2-15（a），（b）分别表示了程序在数据规模为819200，CPU和MIC端使用不同线程时的运行时间。可以看到在CPU端，当启用所有的线程数32（2路8核CPU，每核支持2个硬件线程）时性能达到最优。在MIC端当所启用的线程数小于32时，程序的执行时间远远大于在相同线程时CPU端的执行时间，当线程数超过32时MIC端表现出了较好的性能，随着线程数的增加，程序的运行时间不断减小，当线程达到120后，再继续增加线程时执行时间变化不大。

图2-15（c）表示了在不同数据规模情况下MIC端和CPU端相对于串行程序时的最大加速比。总体上程序在MIC卡上的加速比优于CPU端。当数据规模为61440000时，在CPU端的加速比达到了10.45，说明当问题规模不断扩大时，并行所带来的收益显著高于相应的开销，从而表现出大规模数据集加速效果优于小规模数据集的情况。在MIC卡由于受MIC卡内存的限制，采用native模式时，61440000数据规模时程序无法直接在MIC上运行，所以在数据规模为61440000时，图中没有给出相应的MIC上的加速比。

(a) 819200数据规模下cpu端程序运行时间

(b) 819200数据规模下mic端程序运行时间

(c) 不同数据规模下的加速比

图2-15 k-means程序运行情况比较

#### 利用CPU和MIC协同计算模式优化性能

由于受MIC卡内存容量的限制，在native 模式下不能直接处理大规模数据集的应用；另在native模式下，当MIC卡进行计算时，CPU处于闲置状态，不能充分利用CPU的处理能力。为了充分利用MIC异构系统中的计算资源，同时又可以处理大规模数据，使用offload模式让CPU和MIC以协同的方式来共同完成计算任务。采用MPI和OpenMP两级并行对程序进行优化，主机端使用MPI并行，MIC端使用offload模式，由主机端的MPI进程将计算任务加载到MIC端，MIC卡内部使用OpenMP进行线程级的并行。同时为了克服MIC端内存不足的限制，采用任务分块的方式将整个大规模数据任务进行分块，由CPU和MIC卡分别执行任务的不同部分。

#### 综合性能优化方法

为了充分利用MIC来提升程序的计算性能，结合MIC架构特性，具体从以下方面对程序性能进一步优化。

（1）内存管理优化：除了利用上面的任务分块的方法解决MIC内存空间受限的问题外，为了更好的利用MIC内存，在进行局部聚类时对程序中被MIC端多个线程都用到的数据以共享变量的形式让多个线程共享，不需要为每个线程都保存独立的数据复本，从而提高内存的利用率。

（2）数据传输优化：在CPU和MIC卡协同计算的模式下，两处理器之间的数据传输占了程序执行时额外开销的主要部分，降低通信开销可以有效的缩短并行程序的执行时间。CPU和MIC之间通过PCI-E通信，传输速度较慢，减少CPU和MIC间的通信次数，可以有效的降低通信开销。

Offload编程模式中控制数据传输的in、out语句默认为每次offload开始时在MIC端申请空间，结束时释放空间。但在具体的应用程序中，无需每次offload时都先申请空间，然后传输数据，计算结束时再释放空间。相当一部分数据和空间是可以重复利用的。如果考虑对这些数据和空间的复用，不仅可以大大减少不必要的数据传输，同时也可以减少MIC端分配和释放内存的开销，提升程序的整体性能。本文程序中利用offload模式提供的nocopy()，in()，out()语句结合相应的alloc\_if()，free\_if()控制子句对程序中迭代部分可重复利用的数据和空间进行控制[21]，对整个迭代过程中重复使用的空间只在迭代开始时分配一次，在整个迭代过程中被不同的迭代语句共享，同时对整个迭代过程中涉及到的常量也只在迭代开始时只分配、传输一次，最后在整个迭代结束时一次性将最终结果返回到主机端，并释放各中间变量在MIC端所占的存储空间，从而大大减少不必要的中间结果的来回传输及不必要的空间分配与释放，从整体上减少CPU和MIC的通信次数，大大降低了通信开销。

offload异步传输：通过流水线的方式实现MIC端异步的数据传输与计算，使MIC端的数据传输和计算重叠，用计算来隐藏数据传输，以充分利用计算资源，达到提升应用程序计算性能的目的。

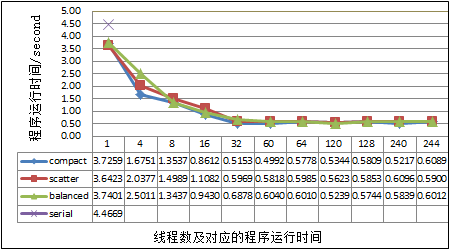
（3）向量化：MIC处理器支持512位宽的knights Corner指令，支持16\*32bit或8\*64bit处理模式，一次可以完成8个双精度或16个单精度数的向量操作。MIC向量化主要有两种方式：自动向量化和SIMD指令优化。通常SIMD指令可以比自动向量化取得更好的性能，但针对不同的硬件平台编写的SIMD指令也不同，且SIMD指令易读性差，所以本文为了降低编程难度在不改变原程序结构的基础上，只在需要向量化的地方插入对应的指导语句自动实现向量化，充分利用MIC强大的向量处理能力来提升性能。

（4）负载均衡：在具体实现程序时考虑两个层面的负载均衡，一是MIC和CPU之间的负载均衡，另一方面是各处理器内部处理核心之间的负载均衡。根据CPU和MIC卡的处理能力，CPU端启用16个进程，每个进程对应CPU的一个处理核心，同时利用P0和P1两个进程来控制两个MIC卡上的计算任务的分发和计算结果的归约，当计算任务分派到MIC卡上后，两个进程继续执行CPU端的计算任务。本文程序中将整个计算任务分成三部分，CPU和两个MIC卡分别执行任务的1/3。由于采用了两路八核的CPU，单个CPU计算核心的计算能力要强于每个MIC卡上的单个处理核的计算能力，所以当完全启用CPU端的16个处理核及每个MIC卡上的60个处理核来分别处理1/3的任务时能达到较好的处理器间的负载平衡。同时在CPU和MIC处理器核内部，按照前面总结出的线程映射规则，通过采用合理的映射方式来保证将各个计算任务尽量均衡的分配到每个处理核上，从而充分利用所有处理核来提升程序性能。

（5）线程可扩展性优化：线程扩展性是指在同样的算例、同样程序的前提下，将线程扩展到更多时，程序计算效能提高的程度。可扩展性优化主要通过改进算法本身来提高程序的并行性；通过改善硬件性能瓶颈来提高可扩展性，如内存大小、内存带宽、存储带宽等方法来实现。可扩展性优化是一个综合优化的过程，要同时结合以上所述的优化方法，才能达到最终提高可扩展性的目的。为了验证本文程序的线程可扩展性，分别在不同数据规模下，在不同的线程情况下，测试了程序的计算性能。

#### 测试结果及分析

为了验证对k-means程序的性能优化效果，在以上具体优化策略的基础上，分别利用不同规模的测试数据集对优化程序进行测试，具体测试平台为：两块Xeon Phi 7110P MIC 卡，2路8核的E5-2670 CPU，64G内存，300G硬盘，组成的异构系统计算平台。具体测试结果如图2-16所示。

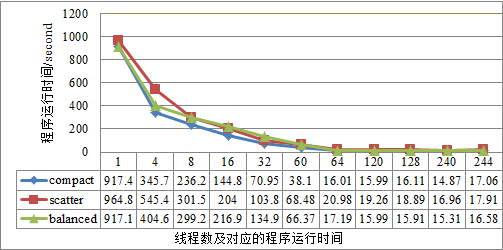


(a) 程序运行时间 (b) MIC加速比

图2-16 409600数据规模下的K-means程序运行时间及加速比

图2-16（a）是针对409600规模的数据集进行聚类时的程序执行时间，图2-16（b）是在不同的线程到处理核的映射方式下并行程序相对于串行程序获得的最大加速比。由于计算规模较小，使用MIC后并没有显著提升程序性能，最大加速比只达到7.4。如果利用MIC卡获得的加速收益不能显著抵消引入的额外传输开销时，有些情况可能使用MIC卡后反而程序执行时间更长。从图中可以看到，compact模式下性能优于scatter和balanced模式，同时随着MIC端线程数的增加计算性能也随之提升，当线程数达到120后程序性能基本稳定，此时在每个处理核上平均启用了两个线程，每个处理核都得到了应用，但由于受问题规模的限制，继续增加线程数时，性能基本没有什么变化。

图2-17（a）是对61440000规模的数据集进行测试的结果，从图中可以看出，随着线程数的增加，程序加速效果明显。当线程数超过120时加速效果趋于稳定。图2-17（b）是程序进行综合优化后相对于串行程序在该大数据规模情况下所达到的最大加速比，达到了85.56，而同样规模数据只在CPU端并行优化所能达到的加速比只有10.45。



（a） 程序运行时间 (b) 加速比

图2-17 61440000数据规模下的K-means程序运行时间及加速比

本节通过对比不同线程到Intel MIC处理器不同处理核心上的映射方式对异构系统性能的影响，分析了在具体应用时进行合理线程映射应考虑的因素；同时通过在MIC异构系统上对具体应用程序进行移植、优化，讨论了在集成众核异构系统上进行程序移植的过程，分析了在MIC异构系统下影响程序性能的各种因素，有针对性的给出了具体的优化策略，并通过测试，验证了移植和优化策略的有效性。本节讨论的线程映射和程序移植及性能优化策略可作为进一步深入研究基于MIC异构系统性能优化的理论基础和参考依据。

## 线程映射相关工作分析

如何将线程合理的映射到异构众核系统具体的处理核上，实现程序应用特点和异构众核系统计算资源相匹配，对充分发挥异构众核高计算能力，提升应用程序的计算性能，降低异构系统整体能耗都有着重要的意义。并行应用程序执行过程中线程到处理器映射时主要考虑的关键问题是线程数设置、线程内数据局部性及线程间数据相关性、应用线程到具体处理核的映射等问题。针对上述线程映射方面的关键问题，已有大量的相关研究工作。本节从线程数设置、线程数的调整、线程数据相关性、线程到计算资源映射四个方面分析已有的研究工作及存在的问题，阐述本文的研究动机。

### 线程数设置

在多线程并行应用程序执行过程中，如何设置合理的线程数，对程序性能、异构系统整体能耗都将产生重要的影响。确定最优线程数方面主要有以下相关研究工作。Suleman等[43]提出了一种利用程序运行时信息动态控制线程数的反馈驱动的线程映射框架，针对数据同步受限和片外带宽受限的两类应用程序，分别设计了用于预测最佳线程数的同步感知线程机制（SAT）和带宽感知线程机制（BAT）。但两种机制利用分析模型预测线程数时，没有考虑共享cache竞争、线程上下文切换、线程迁移等因素对性能的影响，预测是在一种有约束的较理想的情况下进行的，当处理问题规模不大、处理器核数不多、每核单线程模式下可以预测到近似最优的线程数，但不能较好的适应处理器核数众多、同时处理核内引入同时多线程（SMT）的异构众核系统下的线程预测。Pusukuri等[44]通过动态监控影响程序性能的操作系统级状态信息，搜索能够产生程序最佳性能的理想线程数。该方法通过在程序运行过程中不断改变线程数来监测性能，只有找到了最佳线程数，整个搜索过程才结束，然后利用搜索到的最佳线程数，重新运行程序。该方法本质上是一种通过探测来搜索最佳线程数的方法，会引入过高的额外开销。Sasaki等[45]针对多程序多线程的众核处理器运行环境，通过动态检测应用程序可扩展性，预测不同程序对处理核资源的需求，利用组合优化的方法为不同的应用程序分配合理的处理核数，同时设计了一种空闲处理器供让机制来充分利用处理器资源，并在此基础上实现了一个用户级的调度器原型系统SBMP。该工作主要针对多程序运行时如何为不同程序分配合理处理核的问题，主要在处理核粒度上进行调度，没有考虑具体运行平台架构特点及共享资源冲突对性能的影响。Liu等[46]针对Intel试验型的众核处理器SCC（Single-Chip Cloud Computer），测试不同应用程序在不同参数配置下的性能及能耗，通过分析得出针对不同应用程序选择不同的处理核数对程序性能和能耗有着重要影响，表明只有根据不同应用程序的特点选择合适的线程数及处理核数才能使程序性能和系统能耗达到较好的平衡。这也充分说明了程序运行过程中设置、启用合理的线程数及处理核数对程序性能及众核系统能耗的重要性。

本文针对如何预测最优线程数的问题，在分析以上相关研究工作的基础上，为了达到高效、简单、精确预测最优线程数的目的，结合应用程序本身的特点及其在具体MIC异构众核系统上运行时的动态阶段性变化特征，在充分考虑多线程执行时不同线程之间相互作用引入的额外开销对程序性能影响的基础上，通过扩展Amdahl定律，根据反映程序执行性能的主要性能指标，结合回归分析理论，利用最小二乘法构造最优线程数预测模型TCPM。从而在保证预测精度的情况下，实现高效、简单的最优线程数预测，作为动态线程映射机制的理论基础，实现对应用程序线程数实时动态的调整。

### 线程数动态调整

在多线程应用程序执行过程中，根据应用程序不同执行阶段对计算资源的不同需求，通过调整应用程序的并行性，动态分配计算资源，使计算负载并行特征和计算资源较好匹配，对充分利用不同处理器的硬件资源、高效发挥异构众核系统的计算能力及提高异构众核系统整体效能具有重要意义。

已有的根据程序运行特点调整程序并行性方面主要有以下工作。Curtis-Maury等[69]在2008年提出了一种基于预测的动态并行性调节机制，通过收集程序不同执行阶段多个不同的性能计数器信息，利用具体的性能预测模型，推测程序在不同执行阶段不同线程资源配置情况下的性能，选择在不同阶段产生最优性能的线程配置来运行程序，从而实现动态阶段性感知的并行性调节，以提高整体效能。但该调节机制使用过多的性能计数器，性能预测模型计算过程复杂，主要通过比较在不同线程配置情况下的性能来选择最优线程数，没有直接针对线程数进行预测，不能很好的适应具有不同硬件架构特征、处理核数众多、异构众核并行系统下的多线程应用程序的动态并行性调节。美国佐治亚理工学院的Lee等[70]在2010年提出了一种调节应用程序线程数的动态编译系统，首先对程序进行静态剖分，分析程序特点，得到反映程序本身特点的线程间通信和同步模式，建立线程通信图；在程序动态编译过程中，通过分割通信图，动态组合线程，并产生最终组合线程后的代码。以减少原来线程之间不必要的同步及通信所引起的额外开销和共享资源冲突，提高系统性能、降低能耗。但通过动态编译调节线程数，实现复杂，同时依赖于特定的编译系统，通用性受到限制。另在动态编译完成后，程序的并行性状态不能再改变，不能反映程序在运行过程中动态及阶段性变化的特性。Heirman等于2014年在文献[71]中针对cache冲突和片外共享带宽对性能的影响，分析认为启用最大的线程数运行程序并不能得到最好的计算性能，提出了一种CRUST线程调度方法，通过动态调整线程数，提高共享cache的利用率，使应用程序负载与可用的片上cache容量、程序的片外带宽需求与实际的片外带宽更好的匹配，从而有效的提升程序性能和能效。在动态调整线程数时，从最大线程数开始，通过逐次减少线程数来比较程序性能的变化，最后选择达到最好性能的线程数作为最优线程数来执行后续的程序。同时通过检测程序运行行为的阶段性变化，根据程序阶段性变化，按照以上方法对线程数进行相应的调整。该工作所采用的确定最优线程数方法，本质上是一种穷举的方法，当处理问题规模变大时，由于要预运行程序且进行多次迭代比较后才能确定最优的线程数，会引入过高的额外开销，这种开销可能会远远大于优化线程数所能带来的性能收益，同时该方法是建立在多处理核共享最后一级cache架构的前提下，所以不能适应采用同时多线程众核系统下的大规模计算任务的处理。在文献[72]中Heirman等对CRUST进行扩展后，结合同时多线程特性，提出了一种基于Intel Xeon Phi众核系统、根据程序阶段性特点和硬件性能计数器信息、自动进行程序并行性调节的最优线程数设置方法。在具体设置线程的过程中，仍然需要事先在不同的线程数下预运行程序，通过比较不同线程数情况下的性能，从中选择产生最优计算性能的线程数，作为后续执行程序时的线程数设置值。由于仍然采用比较选择的方法确定最佳线程数，所引入的额外开销对程序计算性能的影响仍然很大。同时在程序运行过程中启用了Xeon Phi MIC协处理器的所有处理核，只对核内的线程数进行调节，没有考虑在充分利用每一个处理核的情况下，对处理核数进行调节，所以能耗优化的空间受到限制。

本文针对已有的线程数调节机制存在的额外开销高、通用性差、不能实时调节异构众核系统下运行程序并行性的问题，提出了一种面向异构众核系统，基于程序阶段性变化的线程数动态调节映射机制DPTM。通过设计动态检测程序阶段性变化机制，实时的检测程序运行状态信息，根据检测到的状态信息，利用最优线程数预测模型动态的预测程序不同执行阶段的最优线程数，同时根据最优线程数对程序运行过程中的线程数进行动态调整，使程序在执行过程中的并行性达到最佳状态。同时结合操作系统对计算资源的管理机制，在充分利用众核系统计算资源提升程序计算性能的同时，降低系统的整体能耗，提升异构众核系统的整体效能。

### 线程数据局部性及数据相关性

程序局部性在大规模并行计算环境下变得越来越重要，在算法设计、编译器设计、操作系统及整个系统架构设计时都要考虑数据局部性，只有这样才能获得较好的计算性能[117]。准确的定量分析反映应用程序不同线程之间数据局部性关系的数据相关性，对实现应用线程数据局部性和应用平台架构空间局部性较好的匹配、减少存储访问延迟及由于共享资源竞争而造成的过高额外开销、提高异构众核系统的整体能效具有重要意义。数据重用距离由于能直观准确的反映程序的数据局部性，被广泛的应用在性能分析、程序优化、虚拟存储及高速缓存管理、网络数据传输分析等方面。

根据数据重用距离分析数据局部性特点方面，已有大量的相关研究工作。Ding C等[125][126]提出了一种结合重用距离近似分析、数据局部性模式、高效采样的通用程序局部性预测方法。通过重用距离近似分析来计算重用距离，在重用距离的基础上通过一定的剖分训练标识出程序的局部性模式，利用基于距离的采样方法来预测不同程序在不同输入数据时的局部性。主要根据所定义的通用数据局部性模式，包括常数、线性、次线性的模式，来预测不同程序的局部性。Jiang YL等[127]在扩展传统数据重用距离的基础上提出了并发性数据重用距离的概念，设计了一种概率模型，通过剖分或模拟，将并发重用距离和应用程序的数据局部性关联起来，简化了并发性重用距离的计算，预测共享cache在同时多线程时引起的cache缺失率，实现了在CMP多线程环境下利用数据重用距离分析程序的数据局部性。Xiang XY等[124]提出了一种高阶的数据局部性理论HOTL，通过对影响cache局部性的不同指标进行综合分析，根据不同指标间的关联关系，对这些局部性指标进行整合后，推导出一个覆盖不同局部性指标的综合指标模型。在所提理论的基础上，设计了无需硬件支持的实时数据局部性测试技术及多核cache冲突预测技术来预测cache 缺失率、分析cache局部性及优化cache性能。

Niu QP[128]提出了一种通过分析存储访问数据来快速计算数据重用距离的并行算法。主要通过在剖分存储访问数据过程中标识出并行性，并结合这种并行性来加速重用距离分析。为了实现并行性，将不同的访问数据分成不同的访问块，然后通过MPI并行方法，利用不同的处理器并行的计算不同数据分块内的数据重用距离，最后通过归约完成整个程序全局数据重用距离计算。Wu MJ[129][130]提出了一种通过预测数据重用距离来获取基于循环并行程序数据局部性的方法，用来评测和分析超大规模片上多核处理器存储性能，该方法克服了传统通过采样和模拟，抽取数据局部性方法不能适应超大规模片上多核处理器存储性能分析和评测的问题，为选择最优cache配置提供依据。所提出的预测数据重用距离的方法，主要基于循环并行程序线程对称性，预测在不同cache及处理核配置下存储访问数据的并发重用距离（Concurrent Reuse Distanc，CRD）及私有cache栈的数据重用距离（Private-LRU-stack Reuse Distance，PRD）来获得反映cache性能评测指标。首先分析了CRD及PRD如何随核数的扩展而变化；然后设计了能预测CRD及PRD变化的技术；通过预测CRD及PRD来分析多核cache性能，相比直接通过采样获取CRD及PRD的方法，可以利用较低的开销实现高效的多核cache性能分析。该工作主要基于循环并行程序的线程对称性，利用少量剖分得到的重用距离信息，预测大规模处理核时的数据重用特性来评测不同多核及cache配置时的性能，为大规模片上多核处理器下的高效存储性能评测提供有效的解决思路，为设计合理的cache结构提供指导，但不直接针对通用多线程并行应用程序的数据重用距离计算及线程映射问题。

Nugteren G[131]在扩展传统重用距离理论的基础上，建立了GPU cache行为分析模型，通过重用距离计算，分析程序的片内存储访问行为，利用数据局部性来提升GPU片上cache利用率，提升应用程序性能。Moore RW[132]在同时考虑应用程序和目标机器特点的基础上，提出了利用机器学习的方法自动产生反映程序数据局部性的AutoFinity策略，为实现将不同应用程序不同线程合理分配到处理核上提供指导，以充分利用计算资源，减少额外通信开销，提升应用程序的计算性能。Schuff DL[133]提出了一种基于随机采样和并行化技术的多线程程序数据重用距离测试方法，通过快速的采样模式及不同的测试方法加速多核感知的重用距离分析，为私有或共享cache建模。Fauzia N[134]提出了一种描述程序固有数据局部性及改变执行顺序后程序潜在数据局部性的动态分析方法。Pericas M[135]提出了一种粗粒度的基于kernel的重用距离计算方法，该方法通过检测kernel输入和输出特征，可以快速的评测数据局部性，从而克服传统数据重用距离计算方法面临的插桩及高开销挑战。使用基于kernel的重用距离计算方法可以降低插桩开销，减少重用数据的探测规模，实现共享cache下快速重用距离的计算。

在异构众核系统下，由于复杂的存储结构及数量众多的异构处理核，程序局部性对计算性能的影响更为重要，必须要将程序局部性的特点和异构众核系统具体物理架构特点较好的结合，将程序合理的映射到具体的处理核上，才能充分发挥众核处理器的高计算能力。实现以上目标的前提是必须要有一种简单、准确、高效的程序数据局部性的量化方法，作为定量分析应用程序不同应用线程之间数据局部性的基础。本文针对在异构众核系统下已有的数据局部性分析方法存在的问题，通过计算数据重用距离，定量分析线程内部数据局部性特点及不同线程之间的数据相关性，提出了一种应用程序不同线程之间数据相关性的度量方法DRAQ。

### 基于数据局部性的线程映射

根据应用程序不同线程数据局部性及线程之间数据相关性特点，结合具体运行平台硬件架构特征，将线程合理的分配到相应的处理核上，降低由于存储访问延迟、共享资源竞争、数据传输而造成的过高额外开销，实现高效的线程映射，对提高程序的整体计算效能具有重要意义。

根据数据局部性实现应用程序计算任务到处理核的映射，已有大量的相关研究工作。Zhang Y等[154]提出了一种同时考虑核间和核内数据重用的应用程序到处理核的映射机制，将计算任务和数据分成不同的计算块及数据块，通过比较不同计算块在核内和核间的数据重用量，将数据重用量最大的计算块调度到当前时间槽的当前处理核上，以此实现计算任务到处理核的有效映射。在具体实现时，通过分析不同计算块之间的数据依赖关系得到数据依赖图，通过比较不同计算块之间的共享数据量，得出不同计算块之间的数据共享图，结合数据依赖图及数据共享图，计算每个时间槽内可调度计算块的数据重用量，最后将数据重用量最大的计算块作为当前调度块，不断循环直到所有的计算块都调度完成。该方法本质上是一种穷举比较的调度方法，要提前计算反映数据重用量的权重信息、数据依赖图、数据共享图，要在性能和额外开销之间做较好的平衡，同时该方法只针对多核单线程方式设计，没有考虑同时多线程时的情况。Muralidhara SP等[155]提出了一种基于cache感知的应用分组算法，实现应用程序到核的映射。该方法通过采样，计算出不同应用的重用距离分布，分析应用的存储访问模式，结合不同cache层次，将应用分为不同的应用组，再根据不同的存储层次将不同的应用组逐层映射到不同的处理核上。该方法只根据在模拟器上采样得到的数据重用距离分布情况，分析不同应用程序之间的存储访问关系，没有考虑同一程序内部不同线程之间的数据相关性特点；同时该方法主要根据不同层次cache共享特点在较粗的程序级粒度进行应用程序的分组，不能充分反映相同应用不同线程之间数据交互性特点。在多应用程序情况下，该方法可获得较好的映射性能，由于没有考虑不同线程之间的数据相关性，不能很好的适应具有较强数据相关性的单应用程序多线程情况。

Tam D等[156]设计和实现了一种在线的线程调度机制。使用系统性能计数器信息检测线程的不同共享模式，根据不同的共享模式对线程进行分类，然后设计线程迁移策略，将属于同类簇的线程尽量调度到相同处理核上运行，以减少不必要的存储访问延迟，提高程序的计算性能。该方法在程序运行时，通过读取程序性能计数器信息来分析不同线程间的共享模式，然后根据不同的共享模式进行线程分类，再通过线程迁移实现线程到处理核的调度，整个过程会引入一定的额外开销。对片上众核处理器来说，这种在线调度机制引入的额外开销对性能的影响会更大，甚至引入的额外开销会大于所获得的性能收益。Diener M等[157]提出了一种在内核级自动管理数据和线程局部性的机制kMAF，通过减少多级存储并行架构系统中远程cache访问量的方法来提高存储访问局部性。同时考虑线程之间局部性和所访问数据的局部性特点，利用页失效率分析运行程序的共享存储访问行为，进而调整线程和数据的相关性，将具有共享数据的线程迁移到相同存储层次的处理核上，同时将线程所访问的存储页迁移到线程所在的处理节点上，以提高存储访问性能，该方法在程序运行过程中，通过动态迁移线程来调节线程之间、线程和数据之间的相关性来提高存储访问性能，会引入相应的开销，只有在所获得的性能收益远高于所引入的开销的情况下，这种动态调整方法才能获得好的效果。

Ding W等[158]提出了一种面向多核架构cache层次感知的代码映射及调度策略。映射机制利用数据重用和数据依赖特性将循环迭代分配到多核处理器不同的处理核上；调度决定了分配到处理核上的不同循环迭代的执行顺序，将数据重用转换成数据局部性来提高程序的性能。通过两种机制，将核内及核间数据访问模式和片上多核处理器存储层次特点结合起来，充分利用数据局部性来提升程序计算性能。在实现映射时，将循环迭代、数据重用、处理器存储层次分别抽象成迭代向量、重用向量、核向量，通过代数的方法实现循环迭代到处理核的映射。这种将存储架构和循环迭代完全抽象成向量的形式，通过比较向量来计算数据局部性的方法，主要考虑了存储架构的特点，对计算密集型的循环迭代应用能获得较好的计算性能，但对存储密集型及通信密集型的应用，单纯通过向量比较不能很好的抽取出这类应用线程间相对复杂的数据相关性。Fu W等[159]提出一种通过预测存储访问模式，在程序运行过程中动态迁移线程，以减少片上分布式存储系统上数据传输量的方法。Eduardo HM[160]等提出了利用cache一致性协议提供的信息，识别共享存储应用中不同线程之间的通信模式，根据不同的通信模式设计了线程迁移算法和动态映射机制，在一定的硬件支持下实现线程到处理核的动态映射，以降低线程之间的通信开销。

以上不同的线程映射方法要么需要编译器或额外的硬件支持，限制了其通用性；要么会引入一定的额外运行时开销，只有在额外开销和性能收益达到较好平衡时才能获得理想的效果。针对已有线程映射机制存在的以上问题，本文在同时考虑应用程序本身应用特点及运行平台硬件架构特点的基础上，提出了一种基于数据相关性的线程分组映射机制DagTM。通过计算不同线程数据重用距离来分析应用程序本身的数据相关性，根据不同应用线程内部和不同线程之间的数据相关性进行线程分组，以实现应用线程数据局部性和处理核空间局部性的较好匹配，使处理核内部线程间的数据局部性最大化，使处理核之间的数据相关性最小化，从而合理利用处理器片上cache层次，减少共享存储访问冲突，降低不必要的数据传输开销，在不增加程序运行时额外开销和无需特殊硬件支持的前提下提高程序的计算性能，降低系统能耗。

## 本章小结

异构系统已经成为高性能计算领域一种重要的发展趋势，伴随着计算机性能的快速提升，计算机体系结构变得越来越复杂，呈现出“系统包含成千上万个节点，节点包含多个或数十个CPU，CPU内包含数十上百个核，核内包含多级存储和多运算器，结点内配置异构加速器”的多层嵌套和异构加速的典型特征。已有的基于加速器的大规模异构系统结构基本一致，都是由同构的计算节点通过高速互联网络连接，而计算节点内部异构，由一个或多个多核CPU与一个或多个众核加速器通过PCI-E总线互连。

提升系统性能，改善系统可编程性、可移植性是新型异构系统亟待解决的关键问题。同时提高可编程性，提升性能作为高性能计算软件理论的重要核心，现已成为高性能计算机系统不可分割的一部分。它不仅是硬件系统充分发挥实际效能和经济效益的关键，同时也是我国在信息技术领域增强自主创新能力与核心竞争力的重要契机。

在新型异构众核系统下，利用协处理器对应用程序进行加速过程中面临的具体问题是：如何充分发挥协处理器的高效处理能力；如何使主处理器和协处理器很好的协作；如何通过较简单的编程方法最大程度的利用异构系统高效的处理能力，提高应用程序计算性能的同时降低系统能耗。

本文通过以上调研和分析，主要针对新型异构众核系统高计算能力不能充分发挥的问题，围绕系统运行时软件与硬件映射及性能优化机制，确定研究异构众核系统下线程映射和计算性能的关系，通过研究线程到处理核资源的合理映射机制来提升异构众核系统的计算效能，从计算任务到处理单元的映射层面探索新型异构众核系统架构下的性能优化方法。对具体众核系统互联网络等不同硬件架构对计算效能的影响不作讨论，主要从异构众核系统线程映射时的线程数预测、线程数动态调整、数据局部性及线程数据相关性度量、线程到具体处理核的映射等方面，研究如何将应用程序的特点和具体的运行平台硬件架构特点相结合，实现应用并行特征和运行平台计算资源合理匹配，以高效执行更高级别的细粒度并行任务，从而充分利用异构众核系统的计算资源来提升应用程序的计算效能。

# 面向众核系统的最优线程数预测模型

异构众核系统已经成为高性能计算领域一种重要的发展趋势，如何充分利用异构系统高计算能力，降低系统的能耗是异构众核系统亟需解决的一个关键问题。在异构众核系统下运行多线程应用程序时，主处理器和协处理器端设置不合理的线程数将对异构众核系统计算性能和能耗产生重要的影响：线程数设置的过多会因线程间争用共享资源，导致计算性能下降，同时增加系统能耗；设置的线程数过少，则因异构系统处理核资源得不到充分利用，影响计算性能的提升。本章针对在众核系统多线程应用情况下如何设置最优线程数的问题，通过分析多线程应用程序线程数和计算性能之间的关系，在扩展Amdahl定律的基础上，利用回归分析理论，构建了一种最优线程数预测模型TCPM（Thread Count Prediction Model）。

## 引言

随着多核、众核技术的迅速发展，异构系统已经成为高性能计算领域一种重要的发展趋势，如何在充分利用异构系统高计算能力提升程序性能的同时，降低系统的能耗是异构系统亟需解决的一个关键问题[2][42]。在主处理器+协处理器架构的新型异构众核系统下，主处理器主要负责处理复杂的逻辑控制任务，协处理器负责处理计算密度高、逻辑分支简单的大规模数据并行任务，主处理器和协处理器协同计算任务的不同部分，为具体的应用程序提供高效的计算平台。在具体运行程序时，如何在主处理器和协处理器端设置合理的线程数，对程序性能、异构系统整体能耗都将产生重要的影响。

如何通过合理设置多线程应用程序的线程数，来充分利用处理核计算资源、降低能耗、提高异构系统的整体效能方面已有大量的研究工作。总体上来说，已有的工作在确定具体线程数时主要有根据经验值静态设置、迭代探测、动态预测三种方法。根据经验值静态设置的方法，由于受所处理数据及具体运行平台的变化等因素影响，通常不能正确的反映程序具体运行时的特性，准确性和性能较低，但简单、不引入额外开销；迭代探测的方法在程序运行过程中通过不断的测试对比不同线程数对性能的影响，最终找到近似最优的线程数，然后以此为标准进行线程映射，该方法由于要不断的测试对比后才能确定最终的线程数，会引入较高的额外运行时开销，进而影响程序的整体计算性能，同时也不能反映程序动态变化的特性；动态预测的方法介于前两种方法之间，通过采集程序运行时动态变化的状态信息，利用一定的数学方法或机器学习的方法估计出最优的线程数，既能反映程序动态特性，又不会引入过高的额外开销，是一种较理想的方法。

已有的动态预测线程数工作主要通过机器学习或多元回归统计分析的方法来确定最优线程数。基于机器学习方法在进行预测线程数时要通过大量的数据训练及学习过程，要耗费一定的训练及学习时间。尽管机器学习的方法可以获得较精确的预测值，但由于引入了过高的额外开销，同时当程序的输入、运行特征、运行平台发生变化时，模型需要重新进行训练，所以基于机器学习的预测方法不能很好的适应实时线程数调整的情况；已有的基于多元统计回归分析的方法，可以获得比机器学习更好的预测效率，但在进行线程数预测时，需要采样大量不同的性能指标值，同时模型计算复杂，限制了其在动态线程数调整应用中的通用性。

本章针对如何预测最优线程数的问题，通过测试不同基准测试程序在具体Intel MIC异构众核系统上运行时不同线程数对程序性能的影响，分析出现多样性对应关系的原因，抽象影响因素。在多核多线程并行程序运行时，线程数、cache miss率、带宽利用率、线程上下文切换率、线程迁移率等变化都会是影响程序性能的重要因素，又因为其他程序性能影响因素都间接的随线程数的变化而变化，由主成分分析理论可知线程数是影响程序性能的主要因素。根据以上的分析结论，在充分考虑因线程数变化而引起的共享资源争用、信息交互、数据传输所造成的额外开销对程序计算性能影响的基础上，通过对Armdahl定律扩展后得出了本文最优线程数预测模型的理论基础；之后根据反映程序执行性能的主要性能指标IPS（Instrpucitons Per Second），结合回归分析理论，利用最小二乘法构造最优线程数预测模型TCPM。在利用本文线程数预测模型进行线程数预测的过程中，不会引入过高的额外开销，可为实现基于程序动态阶段性变化的线程数调整映射机制提供理论依据，从而高效的利用异构众核系统的计算能力来提升应用程序的性能，提高整个异构众核并行系统的整体效能。

## 线程数对计算性能的影响分析

本节对不同应用程序在不同线程数情况下的计算性能进行测试，通过对比应用程序在不同线程数执行时的性能变化，分析线程数和应用程序计算性能之间的关系，总结程序计算性能随线程数变化的规律。

### 性能可扩展性测试

为了分析应用程序线程数如何影响应用程序性能及应用程序计算性能随线程数的变化规律，选用具有不同应用特点的Benchmark程序，测试其在不同线程数下执行时的程序计算性能，分析应用程序性能是如何随线程数的变化而变化的。

#### 基准测试程序选取

为使所选的基准测试程序具有较广泛的代表性，能反映不同类型应用程序计算性能随线程数变化的关系，选用PARSEC[47][48][49][50]基准测试集中的基准测试程序。

PARSEC基准测试集是为了适应片上共享存储的多核及众核处理器技术的发展，由Intel联手普林斯顿大学开发的开源的多线程并行应用基准程序测试集，主要用于多核共享存储处理器并行应用研究的性能评测。该基准程序测试集涵盖模式识别、数据挖掘、图像处理、金融分析、大规模应用模拟等广泛应用领域的最新应用，从数据集大小、局部性、数据共享、同步、数据传输等方面反映应用的不同的特征。PARSEC基准程序测试集自2008年发布第一个版本以来，根据多核处理器架构及应用发展变化在不断的完善升级，现在最新的版本为PARSEC3.0。

本文所选取的八个基准测试程序的具体应用特点如下：

*Balckscholes*基准测试程序[51]使用Black-Scholes偏微分方程(PDE)进行期权定价，分析期权值如何随底层资产价格的变化而变化，主要应用在金融分析领域。该基准测试程序属于计算受限的规则应用，主要采用数据并行的处理方式；

*Canneal*基准测试程序使用基于cache感知的模拟退火（simulated annealing，SA）算法来最小化芯片设计的路由成本[52]，使用具有无锁同步的细粒度并行算法及伪随机最差情况下的存储访问模式，在实现时使用了基于数据争用恢复的数据同步策略。该基准测试程序属于计算受限的规则应用，主要应用在工程应用领域；

*Freqmine*基准测试程序使用了一个基于数组的频繁增长模式方法来实现频繁项集数据挖掘[53]。频繁项集数据挖掘是关联数据挖掘的基础，是一个通用的数据挖掘问题，常用在蛋白质序列分析、市场数据分析、日志分析等领域。该基准测试程序属于通信受限的规则应用，主要应用在数据挖掘领域；

*Ferret*基准测试程序主要针对具有丰富数据特征的应用如音频、图像、视频、及三维图形等，实现基于内容的相似性搜索。该基准测试程序属于通信受限的规则应用；

*Raytrace*基准测试程序主要实现三维动画场景的渲染。该基准测试程序属于计算受限的非规则应用；

*Swaptions*基准测试程序用于期权定价，主要应用在金融分析领域，该基准程序属于计算受限的规则应用；

*Streamcluster*基准测试程序主要处理在线的聚类问题，主要针对连续产生的大量流数据，要求实时的对其进行处理的场景，比如网络入侵检测、模式识别、数据挖掘等领域。该基准程序在低维数据时表现为存储受限的应用特点，而在高维数据时又表现出通信受限的应用特点；

*Bodytrack*基准测试程序根据采集得到的多个摄像头的图像序列来跟踪人体活动，主要应用在计算机视觉领域，属于计算受限的非规则应用。

#### 可扩展性测试方法

在MIC异构众核系统上对以上八个基准程序进行测试，具体测试环境见4.6节。每个MIC处理器每个处理核最多支持4个硬件线程，为了保证每次能够使用一个完整的处理核，具体测试时线程数以4的倍数增加。测试程序使用CPU和MIC协处理器协同计算的offload方式实现。计算性能用加速比指标度量。加速比指基准程序单线程执行时间和多线程执行时间的比值。实验中对每个基准程序在不同线程数时相对单线程的加速比结果，经过10轮测试，取平均值后得出。

#### 测试结果分析

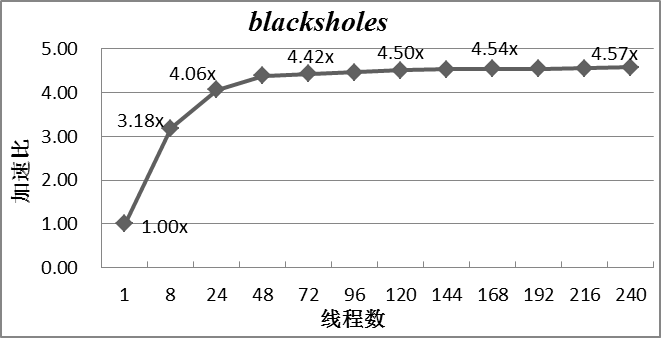
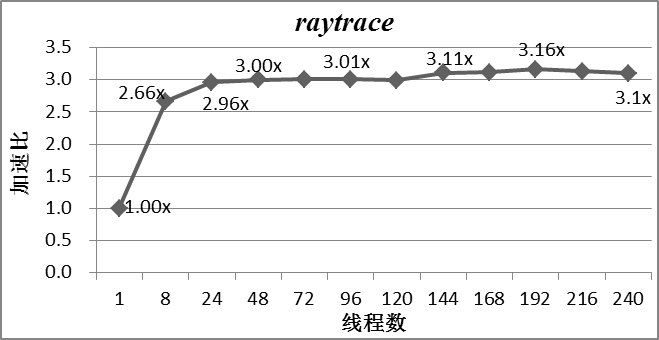
根据线程数和程序性能不同的对应关系，将八个基准测试程序按照程序计算性能随线程数变化规律，分为四组，每组对应两个基准测试程序的运行结果。测试结果如图3-1所示。具体测试结果分类如下：

第Ⅰ类情况：程序的性能随线程数的增加变化缓慢，当超过一定的线程数时，继续增加线程，程序的性能几乎没有明显变化。如图3-1(a)所示，对*balcksholes*和*raytrace*基准测试程序，当所启用的线程数超过24时，加速比分别达到4.06和2.96，继续增加线程数，两个基准程序的加速比曲线开始趋于平滑，加速比分别在4.5和3.1附近波动，加速比随线程数增加没有明显的变化。总体上*blacksholes*表现出了比*raytrace*较好加速效果。这类应用的可并行执行部分有限，当达到最大执行线程数时再继续增加线程性能不会继续增加；

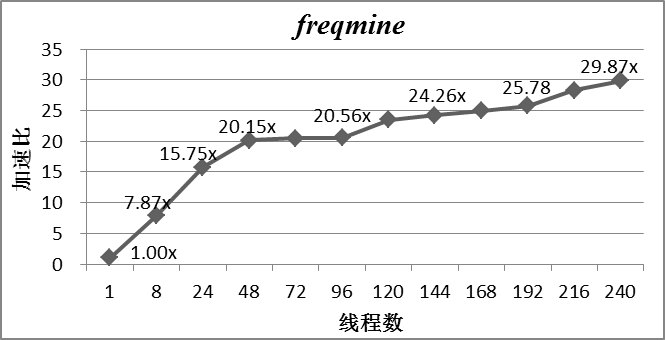
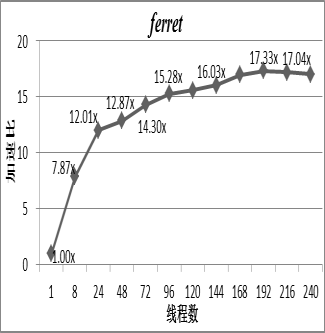
第Ⅱ类情况：程序性能几乎和线程数呈线性关系，当线程数增加时程序的加速比随之增加，表现出了很好的可扩展性。如图3-1(b)所示*，freqmine*和*ferret*基准程序加速比随着线程数增加持续增加，加速比分别从24线程时的15.75和12.01持续增加，当达到最大支持的硬件线程数240时两基准程序的加速比分别达到了29.87和17.04。同时*freqmine*表现出了比*ferret*更好的性能可扩展性。这类应用属计算密集型应用，具有很好的性能可扩展性；

第Ⅲ类情况：当线程数达到一定数目以后，程序性能会随着线程数的继续增加而下降。如图3-1(c)所示，*bodytrack*基准程序在线程数从1到120变化时，程序加速比随线程数增加持续增加，从8线程时的4.03增加到120线程时的最大加速比17.94，然后继续增加线程数时加速比开始急剧下降，当达到最大线程数240时，加速比降低到9.02。*streamcluster*基准程序同样表现出了类似的变化趋势，从8线程时的3.06增加到72线程时的最大加速比7.35，然后继续增加线程数时加速比开始下降，当达到最大线程240时，加速比下降到3.01。总体上*bodytrack*表现出了比*streamcluster*较好的加速效果。这类应用属访存密集型应用，当线程数达到一定值后，继续增加线程会引起不同线程间对共享存储、共享带宽的竞争，从而导致程序性能下降；

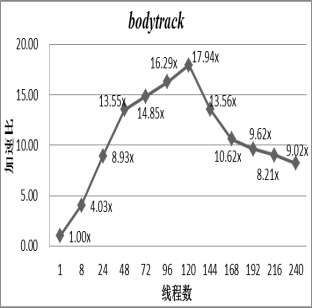
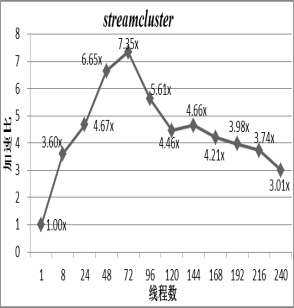
第Ⅳ类情况：程序的性能和线程数之间没有规则的对应关系，在不同线程数情况下表现出了不同的可扩展性，有时性能随线程数增加而增加，有时性能随线程数增加而下降，有时又出现性能不随线程数变化而变化的情况。如图3-1(d)所示，*cannel*基准程序，当线程数从1到24变化时程序的加速比是持续增加的，加速比从3.73增加到5.51；线程数从24变化到120时，程序加速比开始下降，从5.51下降到4.18；线程数增加到144时，加速比又表现出了增加的趋势；线程数从144增加到240时，加速比又开始降低，从线程数144时的加速比5.66下降到线程数为240时的2.73。*swaption*基准程序加速比同样表现出了不规则的变化趋势。线程数从8到96变化过程中，加速比从8线程的6.12增加到96线程时的39.10， 加速比表现出了线性增加的趋势；线程数从96到144范围，加速比表现出了下降的趋势；从144到192范围加速比又开始增加，达到192线程时的最大加速比54.35后，随着线程数的增加，加速比又开始下降。当达到最大线程240时加速比下降为48.02。整个线程数变化过程中，*swaption*基准程序的性能加速比表现出了阶段性变化的趋势，整体加速效果优于*canneal* 基准程序。这主要因程序不同的并行执行部分有不同的计算特性，在不同的线程数情况下，程序在不同的阶段表现出了不同的加速效果。

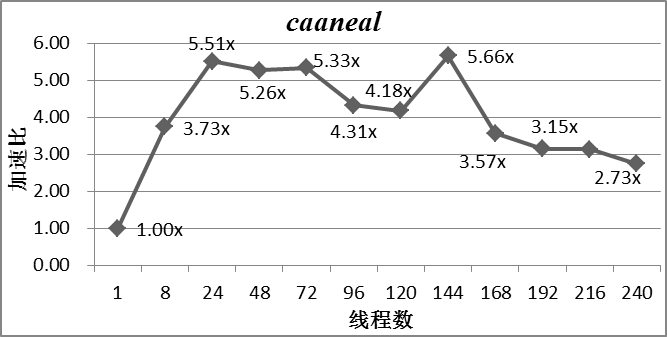
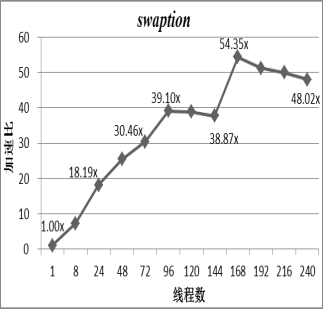
(a)Ⅰ类

(b)Ⅱ类

(c) Ⅲ 类

(d) Ⅳ 类

图3-1 程序性能与线程数的关系

### 程序性能可扩展性影响因素分析

上述不同测试程序表现出不同的性能可扩展性，主要有以下方面的原因：

1）程序本身特点。有些程序是计算密集型的，该类程序本身具有很好的并行性特点，计算任务之间没有直接的相互依赖关系，很适合利用大规模并行方式进行处理，当增加线程数时，能更好的发挥众核处理器的并行处理优势。可用的线程数越多，程序的计算性能越好，可表现出更好的加速比效果；有些程序是访存密集型的，该类应用在程序执行时要不断的访问存储系统，进行数据的读写操作，频繁的访存操作会影响程序计算性能的提升，当启用更多的线程并行执行时，会进一步增加存储访问量，从而会因共享存储容量及其存储带宽限制而不能相应的提升计算性能；有些程序是通信密集型的应用，该类应用在程序执行过程中，计算任务之间要频繁的进行信息交互，如果执行过程中启用过多的线程，会因线程之间高信息交互引入过多的额外通信开销而影响程序的整体计算性能；还有的应用程序在执行过程中会发生阶段性变化，不同的执行阶段会表现出不同的应用特点。所以只有根据程序本身特点，动态的设置线程数才能获得理想的计算性能。

2）系统架构特点及操作系统级影响因素。这方面影响性能可扩展性的主要因素有：线程数、cache miss率、带宽利用率、线程上下文切换率、线程迁移率。以上因素中线程数的变化对程序性能的影响起着主要的作用，因为当线程数变化时，会引起其它因素相应的变化。具体表现为：当程序线程数增加时，由于更多的线程争用共享cache，会导致cache缺失率的增加[54]；由于更多的线程争用带宽会而引起额外的传输延迟；由于众核处理器引入了同时多线程技术（SMP），同一处理核上要并发的执行多个线程，当某一个线程由于资源得不到满足，就要切换其它就绪状态的线程来执行，如果线程数增加，则这种线程上下文切换率也会随之增加，从而引入过高的额外开销，影响程序的性能；同时在众多的处理核之间，要根据资源利用情况，通过不断的在处理核之间进行线程的迁移来充分利用计算资源，当增加线程后，线程迁移也会随之增加，从而引入额外开销影响程序性能。根据主成分分析理论，以上因素对程序性能的影响都能间接的通过线程数表现出来，所以线程数是影响程序性能的主因素，可以通过线程数的变化来研究程序性能的变化。

如何根据运行平台架构特征结合具体应用程序运行时的特点，在程序运行过程中实时动态的设置最优的线程数，使影响程序效能的各个因素都能达到一个合理状态，保证提升异构众核系统计算性能的同时降低能耗。例如对图3-1中的Ｉ类应用，只要将线程数设置成达到最大性能时的最小线程数，而将其它闲置的处理核关闭，就可以大幅降低系统的能耗；对第Ⅱ类应用要尽量满足其对处理核数的要求，以最大程度的提升程序性能；对第Ⅲ类应用，要能检测到使程序性能发生转变的线程数，从而防止程序性能由于不合理的线程数设置而受到影响；第Ⅳ类应用，要能根据实时检测到的程序阶段性变化的特点，在程序不同的执行阶段设置不同的线程数，从而达到提升程序整体性能降低系统能耗的目的。

## 最优线程数预测模型TCPM

由以上分析可知，在异构众核系统下线程数是影响程序性能的主要因素。为了将异构众核系统架构特征和应用程序运行特点相结合，实时的预测出最优的线程数，对应用程序的线程数进行动态调整，以达到最优的计算性能，设计了最优线程数预测模型TCPM。具体通过动态分析线程数如何影响程序性能，通过扩展Amdahl定律，结合主成分分析理论抽象影响因素，得出包含未知参数的线程数分析模型，然后再利用回归分析理论结合最小二乘法对分析模型中的未知参数进行估计，最后推导出最优线程数预测模型。

### 性能指标定义

根据文献[55]中反映多线程应用程序运行时的性能指标，定义如下反映程序执行性能并用于后面预测模型的性能指标。

Ts：表示程序串行执行时间；

Tp：表示程序并行执行时间；

T0：表示总的并行额外开销；

TT(Turnaround time)：程序周转时间，指程序在整个执行过程中的所消耗的总时间；

TTn：当启用n个应用线程执行程序时，程序的周转时间；

TT1：单个应用线程运行时，程序的周转时间；

SIP：程序包含的总指令数；

IPS1：当单个应用线程运行时，程序每秒钟所处理的指令数；

IPSn：当n个应用线程运行时，程序每秒钟所处理的指令数。

### 预测模型理论基础

程序在执行过程中，无论是单处理器方式还是多处理器方式，程序总的指令数（SIP）是固定不变的，以并行加速比定律为基础，同时考虑由于共享资源竞争、线程同步、线程切换、线程迁移等诸多因素造成的额外通信开销对系统性能的影响，参考文献[56][57][58]，对已有加速比模型进行扩展后，构造反映程序线程数和性能之间关系的模型。

#### 加速比模型及其扩展

（1）固定问题规模加速比模型

Amdahl定律[59]描述了在固定问题规模的情况下，应用程序的加速比不随处理器核数目的增加而无限上升，而是受限于应用程序中串行部分的比例，加速比的极限是程序中串行部分比例的倒数。Amdahl定律的具体定义如下：

当. （3-1）

式中的*Sn*表示固定问题规模时的加速比；Ts代表单处理器执行时间；Tp代表多处理器并行执行时间；*Sn*代表加速比；ω表示工作负载；α表示程序中的串行部分所占比例；*n*表示处理核数（或硬件线程数）。该加速比主要描述了在问题规模一定，不考虑所有的数据传输、通信等额外开销的前提下并行加速比上限，同时明确的表明并行应用程序中的顺序部分是计算性能的瓶颈。当并行度达到一定的规模时，必须要降低程序中顺序部分所占的比例，才能进一步提高计算性能。主要从程序本身的串行和并行所占比例两方面考虑了程序对加速比的影响，没有考虑其他影响因素。

在基本的Amdahl定律的基础上，进一步考虑不同线程之间的数据传输、通信等开销时，固定问题规模的加速比可扩展为：

 当. （3-2）

式中的T0表示数据传输通信总开销；表示平均开销。从式（3-2）可以看出，为了提高固定问题规模下的并行程序的加速比，不仅要减小程序顺序比例，同时也要减小平均开销对计算性能造成的负面影响，所以并行程序的性能除了受限于程序顺序比例的瓶颈，也受限于额外通信开销。

（2）固定时间的加速比模型

固定时间加速比模型，是以提高处理问题的求解精度为目的，将问题规模扩大，使所有扩增的计算资源都处理于忙碌状态，提高系统计算资源利用率，在同样的时间内完成更大规模的计算量，使所处理问题得到更高精度的求解。

Amdahl定律的前提是所处理的问题规模（工作负载）固定，当机器规模扩大时，问题规模不能随之扩大以匹配已扩大的可用的计算能力。为了消除固定问题规模的约束，通过在扩大机器规模的同时扩大相应问题规模的方法来获取加速比的改善。Gustafson[60]提出固定时间的加速比模型，即Gustafson定律，具体定义如下：

. （3-3）

式中的St表示固定时间的加速比；ω表示原求解问题的工作负载；α为程序中串行部分所占比例；1-α表示程序中可并行部分所占比例。设在单处理器单线程情况下，执行原求解问题工作负载时所用时间为ω。在多核多线程情况下将问题规模扩展为w=αω+(1-α)nω，且在多核多线程情况下执行时间仍固定为Tp=ω，但扩展规模后的计算负载在单处理核单线程方式下执行时间为Ts=αω+(1-α)nω，可以得（3-3）所示的固定时间加速比公式。以上定律成立的关键是可并行化部分的工作负载从(1-α)ω扩展到(1-α)nω，而串行部分αω保持不变。

Gustanfson定律说明如果工作负载可扩展到可以维持固定执行时间的前提下，那么固定时间加速比便是并行度n的线性函数。也说明当所处理的问题规模扩展到与可用的已扩展的计算能力相匹配时，程序中的顺序部分不再是影响性能提升的瓶颈。

Gustanfson定律没有考虑由于扩展问题规模或增加计算并行度时引入额外通信开销对加速比的影响。在考虑额外开销To的情况下，Gustanfson定律可扩展为以下形式：

. （3-4）

式中，如果要进一步提高加速比，不但要考虑加大并行度，同时也要考虑减小平均通信开销。而平均通信开销又和不同线程之间的信息交互、数据传输、存储访问等有直接关系。所以要做到并行度和额外开销之间较好的平衡，才能获得理想的加速比。

（3）存储受限的加速比模型

存储受限的加速比模型主要思想是用尽所有的存储器容量来最大化的增加处理问题规模，来求解尽可能大的问题。该模型称为Sun-Ni定律[61]，具体定义如下：

. （3-5）

上式中*Sc*表示存储受限的加速比，*G(n)*表示存储容量增加n倍时工作负载的增加量。当考虑所有的额外开销时该加速比可扩展为：

. （3-6）

存储受限的加速比模型，是在存储容量一定的条件下，尽可能的求解大的问题，通过扩展负载，来提高加速比，获得更高的精度和更好的计算资源利用率，但在扩展计算规模时相应的额外开销也会随之增加，反过来会影响加速比的提高。

Sun-Ni定律在取不同的*G(n)*值时可分为以下三种情况：

a）*G(n)*=1，相当于问题规模固定时的加速比模型。此时受限于存储器的加速比模型与Amdahl定律等效；

b）*G(n)=n*时，此时相当于存储容量增加n倍，工作负载也增加n倍的情况，等价于Gustafson定律；

c）G(n)>n，此时相当于计算负载增加量大于存储容量所能支持的程度，此时受限于存储器的加速比（式3-6）将比固定负载加速比（式3-2）及固定时间加速比（式3-4）都高。

#### 预测模型理论依据

（1）考虑额外开销的多核多线程加速比

在分析以上不同加速比模型的基础上，针对多核多线程并行情况下如何减少固定负载的执行时间问题，同时充分考虑多核多线程并行处理时引入的额外开销，得出以下计算加速比的扩展公式：

 （3-7）

式中的表示所要处理的总任务数；表示所使用的硬件线程数；分别表示所处理任务中的串行任务比例系数、并行任务比例系数及额外开销对性能的影响因子，同时满足约束条件：。随着并行度的增加，引入的额外通信开销也随之增加，反过来会影响程序的计算性能。

（2）预测模型理论依据推导

令*f*代表多处理核多线程相对于单处理核单线程运行时的相对周转时间，则有：

 （3-8）

结合式（3-7）得：

 （3-9）

又因当多个线程并行执行时程序的周转时间，单个线程执行时程序的周转时间，其中IPSn和IPS1分别为多个线程和单线程运行时程序每秒钟所能处理的指令数，SIP为程序总的指令数。则多线程相对单线程运行时的相对周转时间可表示为：

 （3-10）

结合式（3-9）和（3-10）可得：

 （3-11）

本文最优线程数预测模型TCPM，以式（3-11）为基础，结合回归分析理论，利用最小二乘法构造程序线程数n的预测模型。根据公式（3-11），要预测未知参数n的值，先要估计出未知参数α、β、γ的值，然后用估计出的值计算线程数n。因多个线程并行执行时程序周转时间TTn和单线程执行时程序的周转时间TT1只有在程序整个执行结束后才可以获得，所以用这两个参数值进行预测是不可行的，也没有实际意义。而程序每秒钟所处理的指令数IPS1和IPSn的值可以在程序运行过程中实时动态的获取到，可以作为最小二乘法求解预测模型未知参数α、β、γ的经验值。所以本文通过对程序每秒钟所处理的指令数IPS1和IPSn的值进行采样，以回归分析理论来为基础，用最小二乘法的方法来构建具体的最优线程数的预测模型。下面以公式（3-11）为基础构造线程数预测模型。

### 预测模型构造

#### 回归分析理论及最小二乘法

（1）回归分析，是研究变量之间相关关系的一种数理统计方法，基于观测数据，建立变量间适当的依赖关系，分析数据内在规律。可用于预报、控制等问题，是广泛应用的数据分析方法。具体来说，回归分析是对具有因果关系的影响因素（自变量）和预测对象（因变量）所进行的数理统计分析处理。从一组数据出发，确定某些变量之间的定量关系式，即建立数学模型并估计其中的未知参数，估计参数时常用最小二乘法。

（2）最小二乘法，是一种数学优化技术，可以从一组测定的数据中寻求变量之间的依赖关系，通过最小化误差的平方和寻找数据的最佳函数匹配，这种函数关系称为经验公式。利用最小二乘法可以简便地求得未知的数据，并使得这些求得的数据与实际数据之间误差平方和最小，是从一组测量值中求解未知量的可依赖的方法。

#### 预测模型构造过程

由公式（3-11）及实际多核多线程运行时的情况可知，多核多线程相对于单核单线程运行时的相对周转时间*f*和线程数*n*之间存在着非确定性关系，即不同的应用程序在启用相同数据量的线程数运行时，因程序的不同应用特点，不同应用程序会表现出不同的加速效果；即使同一程序，由于程序在运行时所表现出的阶段性变化，在不同的运行阶段也会表现出不同的加速效果。所以*f*和*n*之间存在着非确定性关系，同时作为自变量的*n*是可控变量，而与之相关的因变量*f*是随机变量，故*f*和*n*之间的这种非确定性关系满足回归分析的条件，*f*和*n*之间的相关关系可利用回归分析理论进行分析。所以本文以回归分析理论为基础，来构造最优线程数预测模型。最优线程数预测模型具体构造过程如下。

（1）确定回归函数

由公式（3-11）可得：

 (3-12)

式中n为分配的线程数，是可控的自变量；*f(n)*是与自变量*n*相关的随机变量，当n取确定的值时，*f(n)*有一确定的分布与之对应；α、β是不依赖于*n*的未知参数。根据回归分析理论，（3-12）式可作为*f*关于*n*的回归函数。

（2）未知参数*α、β、γ*估计

通过抽样测试程序在不同线程数执行情况下*IPS1*和*IPSn*的值，由公式(3-10)计算出相对的周转时间。通过测试得到多组(*fi ,n*)值，然后利用最小二乘法，估计出相应预测模型中的系数α、β、γ的值。具体方法如下：

1. 利用最小二乘法得：

 （3-13）

1. 为求使偏差S最小的系数α、β，分别对α、β相对于（3-13）求偏导后，根据极值定理，建立以下偏微分方程组：

 （3-14）

令

 （3-15）

变换后得：

（3-16）

1. 将所测得的不同（*fi , n*）值，代入(3-16)，建立只包含未知参数*α、β*的方程组，然后解方程组后即可得到*α、β*的估计值。

（3）最优线程数预测模型

利用极值定理，对公式(3-11)，求使相对周转时间最小的值。求解过程如下：

 （3-17）

 （3-18）

 （3-19）

公式（3-19）即为最终利用最小二乘法后获得的线程数预测模型。

#### 预测模型物理意义

最优线程数预测模型TCPM直接反映了线程数和应用程序串行比例系数、并行任务并行度比例系数、额外开销对性能影响因子*α、*β、γ之间的相关关系。从式（3-19）可知，当β值变大时，即应用程序中可并行执行部分对程序性能影响变大，通过模型计算后，获得的线程数n也相应的增加，从而可启用更多的多核多线程计算资源来实现更高的并行性以获得较好的计算性能；当β值变小时，说明应用程序的并行执行部分对计算性能的影响降低，此时通过模型计算后得到的线程数n也相应减小，可通过减少启用的线程数来合理的利用多核多线程计算资源在达到最佳的计性能的同时降低整体能耗。

TCPM线程数预测模型也很好的反映了由于多线程并行执行时所引发的额外开销和线程数之间的相关关系。当并行额外开销因子γ变小时，说明不同线程之间由于共享资源竞争、信息交互等引起的额外开销对计算性能的影响降低，此时通过模型计算后，获得的线程数n随之增大，则可以通过设置更多数量的线程数以更好的利用并行计算资源来提高计算性能；反之，如果γ值变大，则说明不同线程之间相互作用引起的额外开销对计算性能的影响变大，通过模型计算后，得到的线程数n相应的减小，此时可以通过减少启用的线程数来降低额外开销对计算性能的影响。可以通过实时对应用程序运行状态检测，估计出最优线程数预测模型的影响因子，计算出最优的线程数，从而通过动态的对启用的线程数进行调整，以实现充分利用计算资源，在提高计算性能的同时降低系统的整体能耗。

## TCPM预测模型有效性验证

为了验证本文所构建的最优线程数预测模型TCPM，通过实际测试，应用假设检验的方法来验证所得出的预测模型的有效性。同时从预测精度和预测线程数时引入的额外开销方面定量的评测TCPM的有效性。

### 样本值测试

为了全面分析线程数对程序性能的影响，选取了PARSEC基准测试程序集中具有代表性的*balckscholes*，*ferret*，*streamcluster*，*canneal*四个benchmark程序，四个测试程序具有不同的应用特征，分别代表计算受限的应用、通信受限的应用、访存受限的应用、非规则应用。通过测试以上四个基准测试程序在不同线程数（Ti）情况下运行时间，计算多线程相对于单线程运行时的相对周转时间*f*，具体测试数据如表3-1所示。

表3-1 不同线程数情况下多线程相对于单线程运行时的相对周转时间（时间/秒）

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Benchmark | T1 | T8 | T24 | T48 | T72 | T96 | T120 |
| 执行时间  *blackscholes*  相对周转时间 | 872.76 | 274.68 | 214.78 | 199.43 | 197.39 | 195.63 | 193.95 |
| 1.00 | 0.29 | 0.25 | 0.23 | 0.23 | 0.22 | 0.22 |
|  |  |  |  |  |  |  |  |
| 执行时间  *ferret*  相对周转时间 | 327.61 | 41.64 | 27.29 | 25.45 | 22.91 | 21.45 | 21.01 |
| 1.00 | 0.13 | 0.08 | 0.08 | 0.07 | 0.07 | 0.06 |
|  |  |  |  |  |  |  |  |
| 执行时间  *streamcluster*  相对周转时间 | 763.07 | 211.93 | 163.54 | 114.71 | 103.88 | 135.96 | 170.91 |
| 1.00 | 0.28 | 0.21 | 0.15 | 0.14 | 0.14 | 0.22 |
|  |  |  |  |  |  |  |  |
| 执行时间  *canneal*  相对周转时间 | 775.58 | 207.75 | 140.86 | 147.46 | 145.50 | 179.76 | 185.67 |
| 1.00 | 0.27 | 0.18 | 0.19 | 0.19 | 0.23 | 0.24 |

### 假设检验

为了检验TCPM线程数预测模型的有效性，对上面四个典型的基准测试程序分别在取不同线程数的情况下相对于单线程运行时的相对周转时间的值进行抽样，根据抽样获得的样本值，利用假设检验理论对线程数预测模型进行有效性检验。

下面以*blackscholes*基准测试程序为例来对线程数预测模型进行有效性检验，具体检验过程如下。

#### 求正规方程组

将所测得的*blacksholes*基准测试程在线程数为1、8、24、48、72、96、120时的相对周转时间*f*代入到方程组（3-16）中，得到如下用于假设检验的正规方程组：

 （3-20）

#### 计算未知参数估计值

通过求解方程组（3-20），计算出未知参数α、β、γ的估计值：







#### 求残差平方和，计算方差的估计值

由回归分析理论得到残差平方和的计算公式如下：

 （3-21）

将已求得的*α、*β、γ估计值及测得的不同的*fi*值代入上式（3-21），得出具体的残差平方和为：

 （3-22）

根据回归分析理理论，用以下公式来估计方差：

 （3-23）

将以上求出的相应数值代入上式得方差的估计值为：

 (3-24）

#### 取检验统计量

根据多线程并行程序执行时的实际意义，在多线程并行应用时，线程之间的相互作用对程序的性能影响是客观存在的，即回归函数中γ值不应为零，否则就与实际意义相悖。因此作如下的检验假设：

Ｈ0：γ＝0，H1：

又因该检验属于方差未知的假设检验，根据假设检验理论，采取*T*检验法进行检验，取检验统计量：

 （3-25）

其中Ｎ为样本数量，。

#### T检验

根据假设检验理论，对于，对给定的显著水平λ，通过抽样测试后从样本值计算出统计量的观测值*t*，若，则拒绝H0：，认为回归分析显著，从而说明求得的回归方程有实用价值；若，则接受H0：，即认为回归分析不显著，从而说明求得的回归方程无实际意义。

因为







所以



在显著水平时，查数学表得。

因为所以拒绝原假设H0：，认为回归分析显著，即所得出的线程数预测模型是有效的。

同时按以上的方法，利用其他三个基准测试程序的样本值，对线程数预测模型进行假设检验，分别得到t值为：24.467、36.557、43.235，在显著性水平时，三个观测值都大于，同样得到了拒绝原假设H0：的检验结果。所以整个假设检验过程，验证了本文线程数预测模型TCPM是有效可用的。

利用本文所提出的最优线程数预测模型TCPM进行线程数预测时，通过实时动态的获取在不同线程数情况下程序系统级状态信息*IPS1*和*IPSn*的值，利用公式(3-10)计算出相对周转时间*f*的值，结合公式(3-16)计算出*α*、*β*、*γ*的估计值，再根据公式(3-19)计算出最优线程数*n*的预测值，然后根据预测出的最优线程数来调整应用程序实际运行的线程数。获取反映程序运行状态的有效状态信息，对获得准确的线程数预测值至关重要，TCPM最优线程数预测模型要和具体的动态运行时状态信息检测机制配合才能获得较精确的最优线程预测值。在下一章中，将会涉及相应的应用程序运行时状态信息检测机制。

### 预测精度及额外开销评测

为了定量的验证本文最优线程数预测模型TCPM的预测精度和引入的额外开销，通过对PARSEC基准测试程序测试，比较在理想的最优线程数情况下相对串行执行时的性能加速比和利用本文TCPM最优线程数预测模型后所达到的加速比，评测TCPM预测模型的预测精度；比较在使用本文TCPM最优线程数预测模型相对于系统默认的线程数设置策略下的性能提升率和引入的额外开销，来评测TCPM预测模型额外开销对性能的影响。具体的测试环境及测试方法见4.6节。

#### 预测精度评测结果

通过计算不同基准测试程序分别利用本文TCPM预测模型和操作系统默认的线程数设置策略后程序相对于理想线程数情况下程序计算性能的比例，来作为评价两种策略计算精度的指标[62][63]。操作系统通常情况下默认设置的线程数为硬件平台所支持的最大硬件线程数。不同基准测试程序理想的线程数通过实验方法反复测试对比后获得。

表3-2给出了在理想线程数（Optimal）、操作系统默认线程数(OS\_Default)、基于本文预测线程数模型（TCPM）三种策略下，不同基准测试程序相应的线程数及相对于串行程序的加速比。图3-2给出了OS默认策略和本文TCPM所达到的计算精度对比。

表3-2 三种策略下程序执行时的线程数及相对串程序的加速比

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Benchmark | Optimal | | OS\_Default | | TCPM | |
| #threads | Speedup | #threads | Speedup | #threads | Speedup |
| blackscholes | 240 | 4.57 | 240 | 4.57 | 144 | 4.53 |
| ferret | 192 | 17.33 | 240 | 17.04 | 168 | 17.21 |
| Streamcluster | 72 | 7.35 | 240 | 3.01 | 72 | 7.25 |
| raytrace | 192 | 3.16 | 240 | 3.1 | 144 | 3.1 |
| bodytrack | 120 | 17.94 | 240 | 8.21 | 96 | 16.27 |
| vips | 72 | 34.41 | 240 | 18.58 | 72 | 34.35 |
| cannel | 144 | 5.66 | 240 | 2.73 | 48 | 5.22 |
| freqmine | 240 | 29.87 | 240 | 29.87 | 216 | 28.31 |
| X264 | 168 | 23.48 | 240 | 16.93 | 72 | 23.41 |
| swaptions | 168 | 54.35 | 240 | 38.02 | 144 | 52.21 |
| blackscholes | 240 | 4.57 | 240 | 4.57 | 144 | 4.53 |
| Average | 161 | 54.35 | 240 | 14.206 | 118 | 19.19 |

图3-2 OS默认方法和利用TCPM方法所达到的计算精度对比

从表3-2中可以看出，三种策略下平均启用的线程数分别为161、240、118，TCPM在整个程序运行过程中所启用的硬件线程数最小。从图3-2操作系统默认方法和利用本文TCPM方法所达到的计算精度对比中可以看出，整体上TCPM计算精度平均达到理想最优情况下的97%，而OS默认策略平均计算精度达到理想最优线程数情况下的73%。文献[43]线程映射框架中性能预测精度为87.4%。从而说明本文TCPM预测模型在预测线程数精度方面是有效的。利用TCPM可以在启用最小的线程数的情况下，达到理想计算性能的97%。

#### 额外开销评测结果

图3-3给出了利用本文TCPM线程数预测模型后，不同基准测试程序计算性能相对于OS默认线程数设置策略计算性能的提升率及由于预测线程数所引入的相应额外开销。

图3-3 TCPM额外开销和性能提升率对比

从图3-3中可以看出，基准测试程序平均计算性能相对提升了48.6%，而引入的平均额外开销为2.03%。引入的额外开销相对于所获得的性能收益比例来说，完全可以忽略。同时已有文献中基于搜索比较确定最优线程数的方法[44]，基于机器学习预测线

程数方法[63]、基于多元回归预测方法[69]，引入的额外开销达到了4%以上。TCPM预测模型可以在保证预测精度的情况下，实现简单、高效的最优线程数预测。

## 本章小结

本章针对在众核系统下如何设置合理线程数的问题，研究多核多线程应用程序运行时不同线程之间相互作用对程序性能的影响，分析出现多样性对应关系的原因，抽象影响因素，通过扩展Amdahl定律，结合回归分析理论，提出了一种面向众核系统的最优线程数预测模型TCPM。主要利用程序运行时的状态信息来估计最优线程数模型的未知参数，根据估计得到的参数值，直接计算最优线程数。最后通过假设检验验证了所提最优线程数预测模型的有效性，同时对所提出的预测模型的预测精度和引入的额外开销进行了相应的评测。本文构造的最优线程数模型可以在保证预测精度的情况下，实现高效、简单的最优线程数预测，可作为下一章实现众核系统下多线程并行应用线程数动态调整映射机制的基础，来指导线程数的实时动态调整，以高效的利用众核系统的计算资源，提高整个系统的计算效能。

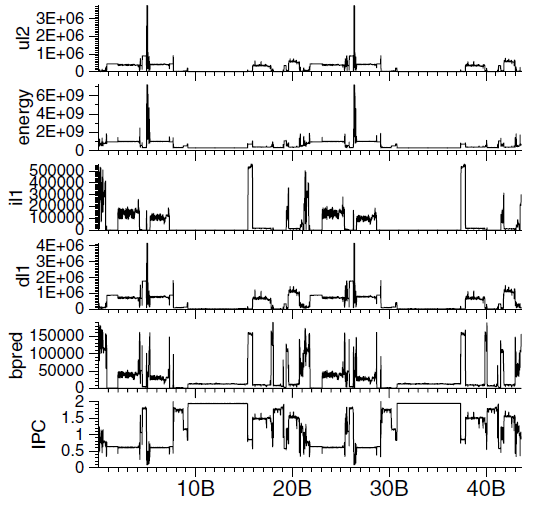
# 基于程序阶段性变化的线程数动态调整映射机制

本章针对如何根据应用程序运行时不同计算阶段对计算资源不同需求、合理设置线程数的问题，通过实时检测程序运行状态及阶段性变化，根据收集到的反映程序运行状态硬件性能计数器信息，在上一章所提出的最优线程数预测模型的基础上，提出了一种在应用程序运行过程中对线程数进行动态调整的映射机制DPTM（Dynamic Prediction Thread Mapping），并在Intel MIC异构众核系统上实现了DPTM映射机制。DPTM通过在程序运行过程中，动态对程序线程数进行调整，以高效的利用异构众核系统计算资源，达到在提高应用程序计算性能的同时，降低系统整体能耗的目的。

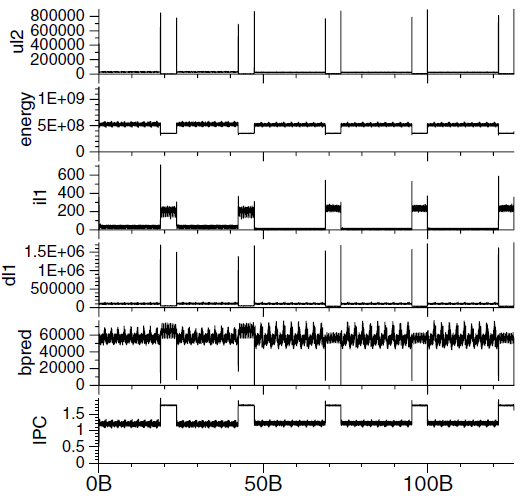
## 引言

异构众核系统下，主处理器一般采用通用的片上多核处理器（CMP），因处理器核数有限，通常在CMP上将线程数设置为具体的处理器核数，就能达到较好的加速效果。协处理器通常采用新型众核处理器（如GPU，MIC），由于处理器核数众多（通常几十、上百个核），同时处理核内部又引入了同时多线程，这使得影响因素更加复杂，如果设置的线程数过多，会因线程间争用共享资源导致计算性能下降，同时启用太多的处理核会造成过高的能源消耗；设置的线程数过少，则因异构系统处理核资源得不到充分利用，影响计算性能的提升。又因主处理器和协处理器具有不同的硬件架构，如何根据分配到的任务特点，结合硬件架构特征合理的设置线程数，将计算任务映射到众核处理器具体的处理核上，对充分利用不同处理器的硬件资源、高效发挥异构众核系统的计算能力及提高异构众核系统整体效能具有重要意义。根据程序的运行行为，结合具体运行平台架构特征为多线程应用程序设置合理的线程数，目前仍然是一个重要的、没有很好解决的、开放的研究问题[64][65][66][67]。

应用程序有各自的计算特征，根据不同的计算特征可将应用程序划分为计算密集型应用、访存密集型应用、通信密集型应用、I/O密集型应用。同时，在应用程序具体执行过程中，同一个应用程序在不同执行阶段也会表现出不同的运行特点。Sherwood等在文献[68]中给出了如图4-1所示的不同应用程序在不同执行时间，所表现出的不同计算特征。图4-1中应用程序在执行过程中所表现出来的阶段性变化，主要通过在一定时间间隔内采样程序执行时的硬件性能指标信息来量化。图中在每十亿指令间隔内采样一次硬件性能指标值。分别用每时钟周期指令数IPC（instruction per cycle），分支预测失效次数bpred（number of branch mispredicts），数据cache缺失率dl1，指令cache缺失率il1，能耗energy，L2 cache缺失率来反映程序运行时的计算特征变化情况。从图中可以看出两个应用程序在执行过程中表现出了明显的阶段性变化特征，有些区间程序表现出了较强的计算密集型特性，有些区间表现出了访存密集型特性，有些计算特征还会规律性重复出现。所以为了进一步提高程序的执行效能，必须要根据应用程序不同执行阶段对计算资源的不同需求，动态的分配计算资源才能使计算负载并行特征较好的和计算资源相匹配，从而高效的利用硬件计算资源，提高程序的计算性能、降低系统能耗。



(a) 程序gcc执行过程中计算特征阶段性变化情况



（b）程序gzip执行过程中计算特征阶段性变化情况

图4-1 不同应用程序运行时计算牲征阶段性变化情况

根据应用程序在多核/众核系统下的运行特点，为多线程应用程序设置合理的线程数，实现高效的线程映射方面，目前已有大量的研究工作。已有的研究工作所采用的方法总体上可以归为两大类。一类根据收集程序运行过程中硬件性能计数器的状态信息，检测程序运行过程中的阶段性变化，通过比较在不同线程配置情况下的性能来选择最优线程数。该类方法本质上是一种静态比较的方法来调节并行性，会引入过高的额外开销，不能很好的适应具有不同硬件架构特征、众多处理核数量、复杂存储层次异构众核系统下的程序动态并行性调节；另一类是基于编译系统的线程数动态调节机制，该类方法通过在编译阶段分析程序特点，得到反映程序运行特点的执行模式，然后在程序运行过程中基于执行模式动态的组合线程，以减少额外开销和共享资源冲突，提高系统性能、降低能耗。但基于动态编译调节线程数，实现复杂，主要根据在编译阶段分析得到的静态执行模式来实现动态的线程数调节，没有考虑程序在具体运行平台上执行时的动态阶段性变化的特性。

由以上分析可知，已有的线程数映射机制，主要存在以下问题：通过静态比较获得最优线程数的方法会引入过高的额外开销，不能很好的适应多线程应用程序在异构众核系统下动态实时的并行性调节；基于编译系统的线程数动态调节机制，实现复杂，要依赖于特定硬件系统，通用性受到限制，同时动态编译时主要根据程序本身的静态特点抽取执行模式，不能较好的反映程序在运行过程中动态及阶段性变化的特性，不能实时根据程序运行状态变化来合理的设置线程数，对程序阶段性变化明显的应用程序，不能获得理想的映射效果。

本章针对已有的线程数调节机制存在的额外开销高、通用性差、不能实时调节异构众核系统下运行程序并行性的问题，提出了一种面向异构众核系统，基于程序阶段性变化的线程数动态调节映射机制DPTM。在结合具体异构众核系统架构特征的基础上，同时考虑应用程序本身的特点和程序在具体运行时动态变化特性，通过实时的检测程序运行状态及阶段性变化，仅根据收集到的反映程序运行状态的少量硬件性能计数器信息，利用所提出的数学模型直接计算出最优线程数的预测值，然后对程序的并行性进行调整。同时借助相应的DVFS[74]技术对处于空闲状态的硬件线程或处理核进行控制，使其处于关闭或低功耗状态，以降低系统的整体能耗。

## 线程数动态调整机制处理流程

本文在上一章线程数预测模型的基础上，根据程序运行时状态变化，设计了线程数动态调整映射机制DPTM。具体的线程数动态调整映射机制DPTM处理流程如图4-2所示。

首先在异构众核平台上预运行程序，通过设置不同的线程数来检测程序运行状态的变化情况，检测过程的主要任务是采集在不同线程数情况下，程序运行时的系统级状态信息IPS1和IPSn的值。当采集到规定要求的状态信息后，通过上一章中的最优线程数预测模型，计算出预测模型中的未知参数、，然后利用公式(3-19)计算出最优线程数的预测值n。计算出线程数n后，以n为最优的估计值设置程序线程数后重新运行程序。此时影响程序性能的各个因素如上下文切换、线程迁移、cache利用率、共享带宽利用率等都基本处在一个合理的状态[73][102]，可避免由于无任何反映程序运行状态信息指导，根据经验值设置的不合理多线程应用程序线程数而引起的共享资源争用，同步、传输延迟等对程序性能的影响。考虑到程序动态阶段性变化对程序计算性能的影响，在程序运行过程中继续采集程序状态信息，根据程序阶段性变化检测算法，检测程序运行状态。当检测到程序运行发生阶段性变化时，重新根据当前的状态信息估计新的能够适应当前程序运行状态的合理线程数，实时动态的对程序运行状态进行调整，同时结合系统相应的动态电压及频率调整机制DVFS[75][76]对空闲的硬件线程及处理核进行控制，使其处于低功耗或关闭状态，保证在获得最佳性能的同时最大程度的降低系统的能耗，以充分利用异构众核系统的计算资源来提升程序的效能[95][96][98]。这样不断的迭代，直到完成整个程序的执行。



图4-2 线程数动态调整映射机制DPTM处理流程

在整个程序执行过程中，线程数动态调整映射机制DPTM，能根据程序在运行过程中的阶段性变化，利用实时采集到的反映程序运行状态的性能计数器信息，根据最优线程数预测模型估计最优线程数，之后动态的对程序运行时的线程数进行调整，从而根据多线程应用程序的运行行为及不同执行阶段对计算资源不同需求实时分配合理的硬件线程和处理核，在充分利用众核系统的计算资源提升计算性能的同时降低系统能耗。

## 基于程序阶段性变化的线程数动态调整映射框架

使用Linux OS内核提供的性能检测工具，在线的采集能够反映程序行为、与处理器微架构相关的状态信息，实时动态检测运行程序阶段性变化特征，利用预测模型计算出不同阶段的最佳线程数，根据最佳线程数动态的调整分配到应用程序的处理核资源，实现计算资源的合理分配。以Intel 开源的OpenMP运行时库为基础，通过扩展Intel OpenMP\_Runtime Library[77]，以动态运行时库的形式，实现针对异构众核系统，提高异构系统资源利用率和应用程序性能、降低系统整体能耗的动态线程映射机制。

### DPTM映射框架

本文基于程序阶段性变化的线程数动态调整DPTM映射框架，主要针对主处理器+协处理器异构众核系统，以Intel MIC异构众核系统下offload[78][79][80]运行模式为基础，以动态运行时库的形式进行原型系统设计。程序在运行时，首先在异构系统主处理器端进行任务划分，由主处理器控制整个程序的运行和任务分配。当程序执行过程中遇到循环处理部分时，主处理器将循环部分分派给协处理器，利用协处理器高并行计算能力来加速循环部分执行。MIC异构众核系统下DPTM线程数动态调整映射机制总体框架如图4-3所示。MIC协处理器端接收到主机端CPU分派的循环部分并行计算任务时，首先在MIC协处理器上，在不同线程数情况下预运行程序，同时在程序执行过程中收集反映程序运行状态的性能计数器信息，当所收集的信息达到预先设定的采样量后，将采集到的状态信息反馈到主机端，由主机端主进程根据状态信息，利用线程预测模型计算出最优线程数的预测值，然后用最优线程数来控制MIC端程序的并行执行。在MIC端并行执行程序的过程中，运行时系统在一定的时间间隔内持续收集程序状态信息，同时检测程序的运行状态，一旦检测到程序发生了明显的阶段性变化，运行时系统将程序当前状态信息反馈给主机端，由主机端主进程根据接收到的当前程序运行状态信息，利用线程数预测模型重新计算最优线程数，之后利用新的线程数预测值对MIC端运行程序的并行性进行动态调整，从而实现根据程序运行时的阶段性变化及不同阶段对计算资源的不同需求，来调整并行程序的线程数，实现对众核处理器处理核及硬件线程实时控制，以高效的利用众核处理器计算资源执行程序，最大化的提高程序的计算效能。

当MIC端完成所分派的并行任务后，将计算结果返回到CPU主机端。此时整个应用程序在CPU主机端主进程的控制下继续顺序执行。在主程序执行过程中，如果遇到循环计算部分，则CPU端主进程继续将循环部分计算任务分派给MIC协处理器，并以当前所记录的最优线程数来控制MIC端并行度，来并行执行计算任务。同时在程序执行过程中，仍然收集程序运行状态信息，检测程序的运行状态，如果检测到程序发生了阶段性变化，则将当前收集到的状态信息反馈到主机端，主机端根据新的状态信息实时计算出最优线程数来对MIC协处理器端的并行性进行动态调整[97]。这样CPU端和MIC端相互协同，不断的迭代直到计算任务全部完成。

图4-3 MIC异构系统下DPTM映射框架

### 状态信息采集

程序运行的状态信息是预测模型预测最优线程数的基础，同时也是判断程序运行是否发生阶段性变化的依据。采样到的状态信息的正确与否直接决定着最优线程数预测值的正确性及对线程数动态调整的准确性，所以既要保证状态信息采样的正确性，又要保证采样的高效性。

在整个动态预测线程映射过程中所要采样的系统状态信息有：单线程运行时程序每秒钟执行的指令数IPS1、N个线程运行时程序每秒执行的指令数IPSN、线程上下文切换率（context-switches）、线程迁移率（thread-migration）、cache缺失率（cache-misses）、CPU利用率（cpu-cycles）、带宽利用率（bus-cycles）。具体对硬件性能计算器信息进行采样时，使用Perf Event性能测试工具，该工具是随Linux内核代码一同发布和维护的性能诊断工具，由内核社区维护和发展。图4-4给出了具体在Linux kernel module中通过改变内核控制寄存器CR4的PCE位，实现在用户空间直接使用rdpmc指令读取系统性能检测单元数据信息的代码。



图4-4 设置控制寄存器CR4的PCE位为有效状态的代码

Perf工具不仅可以用于应用程序的性能统计分析，也可以应用于内核代码的性能统计和分析。通过该工具可以实时访问系统性能检测单元（PMU），来提取以上所需的状态信息[81][82][83]。但直接使用Perf工具采集程序状态信息，由于要通过系统调用才能实现，会引入额外的系统调用开销。为了降低额外开销，在具体采集程序性能指标时通过设计Linux kernel module改变内核控制寄存器CR4的PCE（Performance-Monitoring Counter enable）位，使其从无效状态变为有效状态，这样可以直接在用户空间使用rdpmc指令来读取相关性能计数器信息，减少由于系统调用产生的过高的额外开销[84]。图4-4给出的具体在Linux kernel module中通过改变内核控制寄存器CR4的PCE位，实现在用户空间直接使用rdpmc指令读取系统性能检测单元数据信息的代码，因PCE位在控制寄存器CR4的第8位，默认为无效状态，图中代码主要实现设置该位为有效状态的功能。

在具体进行状态信息采集时，通过具体参数设置来控制采样间隔，根据经验值[52]设定采样间隔为100毫秒。在整个程序执行过程中持续对以上状态信息进行采样，采样信息以全局变量的形式保存。为了提高采样信息的有效性，仅保留最近两次的状态信息，对过早的状态信息不进行存储。其中状态信息IPS1和IPSN是预测最优线程数的主要性能指标，为有效的获得采样值，通过分析第3.3节中不同测试程序不同的测试结果，得出当线程数为8、24、48、120、168、240时程序性能有较明显的变化，所以在预运行程序时只对这六个线程数下的IPS值进行测试，可以在提高状态采集效率的同时保证在CPU端有足够的信息进行精确的线程预测。

### 程序阶段性变化动态检测

由于计算负载、输入数据、程序运行特点都会在程序执行过程中发生变化，所以可能会导致应用程序的运行状态发生阶段性变化[85][86][87][88][89]。如果能在程序执行过程中，根据程序不同执行阶段对计算资源的不同需求，实现合理的计算资源分配，则可以进一步提高计算资源利用率、提升计算效能[90][91][92][93]。为了实现这一目标，DPTM通过实时的检测程序阶段性变化，对应用程序线程数进行动态调整来实现合理的计算资源分配。

程序阶段性变化大多发生在程序的不同循环部分。根据这一特点，DPTM在MIC异构众核系统offload运行模式下，利用在MIC端收集到的应用程序循环部分并行执行时的运行状态信息，检测程序阶段性变化。主要根据采集到的线程上下文切换率、线程迁移率、cache缺失率、CPU利用率、带宽利用率五个指标进行程序阶段性变化检测[86]。

程序在执行过程中，DPTM每隔100毫秒读取一次系统内核提供的性能计算数器中的程序运行状态信息：context-switches、thread-migration、cache-misses、cpu-cycles、bus-cycles，将当前读取的状态信息和上一次保存的状态信息值进行比较，计算出各个性能指标的相对变化率：Δcpu-cycles、Δcontext-switches、Δthread-migration、Δcache-misses、Δbus-cycles。然后通过将所计算出的不同性能指标相对变化率和预先设定的相应阈值进行比较，根据检测算法做出程序是否发生阶段性变化的判断。

#### 性能指标阈值

用于判断程序是否发生阶段性变化的性能指标阈值分别为：ThresholdΔcpu-cycle、ThresholdΔcontex-switches、ThresholdΔthread-migration、ThresholdΔbus-cycles、ThresholdΔcache-miss。以上阈值主要通过实验测试和观察的方法获得。具体获取性能指标阈值的过程如下：

（1）从PARSEC[48]基准测试集中选择五个具有不同运行特点的基准程序：bodytrack，x264，canneal，blackscholes，streamcluster来进行测试，通过对比分析后获得相应的性能指标阈值。其中bodytrack基准测试程序对线程的迁移比较敏感，用它可以较好的测试出线程迁移率的阈值ThresholdΔthread-migration；x264对线程的上下文切换比较敏感，用它主要测试线程上下文切换率阈值ThresholdΔcontex-switches；canneal对带宽较敏感，用它来测试总线带宽变化率阈值ThresholdΔbus-cycles；blackscholes属计算密集型应用，对CPU时钟周期很敏感，用它来测试CPU周期变化率的阈值；cache miss率对streamcluster基准程序影响比较大，用它来测试cache miss变化率的阈值ThresholdΔcache-miss。

（2）在MIC众核系统上，以native运行模式执行上面五个基准测试程序，分别测试每个基准程序所对应的性能指标值。每隔100毫秒对相应的性能指标采样一次，同时计算最近两次相应指标的变化率，如果观察到程序的计算性能有明显的变化，则记录此时对应的性能指标变化率，这样不断的执行、分析，直到整个测试程序执行完成，最后取引起程序计算性能有明显变化的最大的性能指标变化率为最终的阈值。判断程序计算性能时，用硬件性能指标IPC (Instruction per cycle)来衡量，同时设定如果IPC值的变化率超过30%，就认为程序的计算性能发生了明显的变化。

通过以上测试观察方法得到如表4-1所示的，用于判断程序阶段性变化的性能指标相对变化率阈值。

表4-1性能指标相对变化率阈值

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 性能指标 | cpu-cycles | context-switches | thread-migration | cache-misses | bus-cycles |
| 相对变化率阈值 | 60% | 30% | 50% | 30% | 50% |

#### 程序阶段性变化检测算法

在以上性能指标相对变化率阈值的基础之上，设计程序阶段性变化动态检测算法，来检测应用程序执行过程中的阶段性变化。具体如算法4-1所示。

算法的具体检测过程如下：

首先比较当前采样到的反映CPU利用率的性能指标cpu-cycles值和上一次保存的采样值之间的变化率，如果cpu-cycles的变化率Δcpu-cycles小于相应的阈值ThresholdΔcpu-cycle，则认为程序的运行行为没有发生阶段性变化，程序继续执行。如果比较发现cpu-cycles的变化率Δcpu-cycles大于或等于相应的阈值ThresholdΔcpu-cycle，则表明CPU的利用率发生了明显的变化。单纯的CPU利用率的变化可能是因为计算任务量发生了变化，不能充分说明程序的运行行为发生了改变[104]。此时需要结合其他的性能指标，进一步判断程序的运行状态是否发生了阶段性变化，具体判断方法如下：

1. 如果线程的上下文切换率指标变化率Δcontext-switches大于阈值ThresholdΔcontex-switches，表明原始的线程数已不能较好的适应当前程序运行状况，操作系统为了更好的利用计算资源，进行了明显的线程上下文切换，由此可以判断出程序发生了阶段性变化；
2. 若上下文切换率没有明显的变化，则需要判断线程迁移率及带宽的利用率，如果线程迁移率和带宽利用率变化同时大于其相应的阈值，则表明此时线程数不能和处理核计算资源较好的匹配，导致了不同处理核间线程过高的迁移及过高的带宽利用，说明程序的运行行为可能从计算密集向存储密集型转变[94]，反之亦然，所以由此可以判断出程序发生了阶段性变化；
3. 若以上两点都不能满足，则继续判断cache miss变化率是否大于其阈值，如果cache miss变化率Δcache-misses大于其阈值，则表明当前线程之间发生了明显的共享cache争用[100]，程序的运行行为发生了阶段性变化，所以应该调整线程数使其更好的适配程序当前运行状况。

如果以上三点都不满足，则说明程序的运行行为没有发生阶段性变化，继续按原线程数执行程序。

|  |
| --- |
| **算法4-1**: 程序阶段性变化检测算法 |
| **Input**: Δcpu-cycles, Δcontext-switches, Δthread-migration, Δcache-misses, Δbus-cycles Threshold  **Output**: IPS采样值  **Begin**  1: **if**  Δcpu-cycles< Thresholdcpu-cycle **then**  2: 无异常，采样、收集程序运行状态信息；  3: 继续执行程序；  4: **else**  5:  **if**  Δcontext-switches> Threshold\_contex-switches **then**  6: 有异常，采样、收集程序运行的性能状态信息；  7: 向CPU反馈异常状态及IPS采样值；  8:  **else**  9: **if**  Δthread-migration> Thresholdthread-migration∩Δbus-cycles> Thresholdbus-cycles **then**  10: 有异常，采样、收集程序运行的性能状态信息；  11: 向CPU反馈异常状态及IPS采样值；  12: **else**  13: **if**  Δcache-misses> Thresholdcache-miss **then**  14: 有异常，采样、收集程序运行的性能状态信息；  15: 向CPU反馈异常状态及IPS采样值；  16: **else**  17: 无异常，继续运行程序；  18: 采样、收集程序运行状态信息；  19: **endif**  20: **endif**  21: **endif**  22: **endif**  **End** |

## DPTM映射框架实现

DPTM在MIC协处理器端检测到程序运行发生了阶段性变化后，将MIC端程序运行时的状态信息及IPS采样值反馈到CPU主机端，CPU端主进程根据接收到采样信息，计算出预测模型中的未知参数，并根据预测模型计算出要分配的最佳线程数。之后动态调整在MIC端启动的活动线程数。在程序执行过程中，同时将闲置的处理核设为非活动状态，以降低能耗[110][111][112]。在程序运行过程中按一定的时间间隔，持续的对程序的运行状态信息进行采样，并检测程序阶段性变化，根据程序阶段性变化动态的对程序的线程数进行调整。以Intel开源的OpenMP运行时库为基础，通过扩展Intel OpenMP\_Runtime Library[77]，以动态运行时库的形式实现DPTM线程映射。

### OpenMP运行时库扩展策略

本文DPTM映射实现时，通过扩展Intel OpenMP运行时库，在运行过程中加入一些控制语句来实现运行时线程数动态调整。在Intel OpenMP库内部，OpenMP控制功能主要通过 \_\_kmpc\_fork\_call 控制原语标识的代码部分来实现。

在源程序中，只要用制导语句 #pragma omp parallel标识的要并行执行的代码部分，都会调用OpenMP内部用控制原语 \_\_kmpc\_fork\_call 标识的代码，实现对OpenMP并行程序具体执行过程的控制。在具体扩展运行库时，主要通过在 \_\_kmpc\_fork\_call 控制原语标识的部分加入相应的控制代码，以实现对程序运行时的性能状态信息采集、程序阶段性变化的检测、动态调整线程数的功能。

主要的控制语句为：

读性能状态指标信息：read\_pmc()；

收集性能状态信息：collect\_status\_information()；

返回状态信息到主机端：return\_status\_information(host)；

动态设置线程数：omp\_set\_mum\_threads(opt\_number\_threads)；

检测程序运行状态阶段性变化：detecting\_running\_exception()；

具体功能调用如图4-5所示。



图4-5 扩展OpenMP运行时库后的内部控制功能调用

### MIC异构系统下的DPTM实现

在Intel MIC异构众核系统下实现DPTM映射框架时，采用了主机端CPU和MIC协处理器端相互协同的运行模式offload[78]。应用程序的整体运行由CPU主进程进行控制，应用程序的循环执行部分则分派给MIC协处理器并行执行。基于OpenMP并行编程模型实现具体的并行化，同时通过扩展OpenMP运行时库，实现对程序运行时状态信息采集、程序阶段性变化检测、动态调整线程数等控制[97]。因应用程序的循环执行部分占据了整个程序计算量的主体部分[112]，OpenMP并行化主要集中在程序的循环部分，所以DPTM动态并行性调整主要针对应用程序的循环部分进行。

整个应用程序在主机端CPU主进程的控制之下执行。在执行过程中，当遇到循环执行部分时，则将循环部分offload到MIC协处理器端执行，然后根据DPTM映射策略实现动态线程数调整。MIC异构众核系统下具体的DPTM动态预测线程映射实现如图4-6所示。

图4-6 MIC异构系统下DPTM实现

DPTM实现主要包含以下五个部分：

（1）**HOST SIDE 1** 代码段：应用程序以offload 模式运行，整个应用程序在主机端CPU主进程控制下执行。首先在主机端执行程序，当遇到程序中的循环部分时，CPU主进程将其offload到MIC 协处理器上执行，如算法中的 #pragma offload target (MIC SIDE 1) 实现将其后面的for循环代码offload到MIC端执行，然后执行指导语句 #pragma offload\_wait target(mic)等待MIC端的运行结果[79][80]。此时循环部分程序在MIC协处理器的控制下在MIC端执行。

（2）**MIC SIDE 1** 代码段：MIC协处理器接收到计算任务后，首先检测OS内核提供的控制寄存器CR4的PCE位是否已经置1，如果没有置位则调用初始化内核模块init\_module() 对其进行置位，目的是为了能够直接在用户空间使用rdpmc指令来读取相关性能计数器信息。然后调用预运行实现函数pre\_running\_program() 对程序进行预运行，调用函数read\_pmc()读取程序性能计数器信息，并调用函数collect\_status\_information() 收集状态信息后，调用return\_status\_information(HOST SIDE 2)函数将收集到的状态信息反馈给CPU主进程。

（3）**HOST SIDE 2** 代码段：CPU主进程接收到MIC端反馈的程序运行状态信息后，根据状态信息，利用线程数预测模型，估计出最优线程数，具体线程数预测由函数：predicting\_optimal\_number\_threads (status\_info, opt\_number\_threads) 实现。接着将最优线程数通过制导语句：#pragma offload\_transfer target(MIC SIDE 2) in(opt\_number\_threads) 发送到MIC端，从而控制MIC端线程并行度。

（4） **MIC SIDE 2**代码段：MIC端根据CPU端发送的最优线程数重新执行程序，在程序执行过程中，在一定的时间间隔内，对程序运行状态信息进行一次采样。根据采样信息检测程序是否发生了动态的阶段性变化，具体检测通过函数：detecting\_running\_exception() 实现。如果检测到异常则将当前的状态信息反馈到CPU端，由CPU主进程重新对最佳线程数进行计算，计算出最佳线程数后返回给MIC端，MIC端根据接收到的最佳线程数对程序线程重新调整后继续执行程序；若无异常发生，则继续执行程序，直到MIC端所有的程序执行完成后，将最后的执行结果返回给CPU主进程。

（5）**HOST SIDE 3** 代码段：CPU主进程接收到MIC协处理器返回的当前循环部分的最终计算结果后，继续根据主程序流程运行整个应用程序的后续部分，同时如果后续又有循环执行部分则继续按前面的操作方法将新的循环代码offload到MIC协处理器上执行。这样不断迭代直到整个应用程序执行完成。

## 实验评测及分析

### 基准测试程序及测试环境

#### 基准测试程序

通过对PARSEC[50]基准测试集中的应用程序进行测试来验证、评测本文动态线程数调整机制DPTM。具体用于测试的benchmark程序及其特征如表4-2所示。

表4-2基准测试程序及其特征

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 基准测试程序 | 应用领域 | 主要特征 | 处理方式 | 输入数据集 |
| blackscholes | 金融分析 | 计算受限，规则应用 | 数据并行 | native |
| ferret | 相似性搜索 | 计算受限，规则应用 | 流水线 | native |
| Streamcluster | 数据挖掘 | 通信受限，非规则应用 | 数据并行 | native |
| raytrace | 图形渲染 | 计算受限，非规则应用 | 数据并行 | native |
| bodytrack | 计算机视觉 | 计算受限，非规则应用 | 数据并行 | native |
| vips | 多媒体处理 | 带宽受限，非规则应用 | 数据并行 | native |
| cannel | 工程应用 | 计算受限，规则应用 | 数据并行 | native |
| freqmine | 数据挖掘 | 计算受限，规则应用 | 数据并行 | native |
| X264 | 多媒体处理 | 访存受限，非规则应用 | 流水线 | native |
| swaptions | 金融分析 | 计算受限，规则应用 | 数据并行 | native |

#### 测试环境

MIC异构众核运行平台由2路8核的E5-2670 CPU和Xeon Phi 7110P MIC卡组成，CPU主存和加速卡之间采用通道带宽为x16的PCI-E总线进行数据传输，最高传输速度可达到16GB/s。操作系统：Red Hat Enterprise Linux Server release 6.3.

开发编译环境：Intel parallel\_studio\_xe\_2013\_update3\_intel64软件包。

CPU和MIC协处理器具体性能参数如表4-3所示。

表4-3硬件系统性能参数

|  |  |  |
| --- | --- | --- |
| 性能参数 | Xeon Phi 7110P MIC | E5-2670 CPU |
| 处理器核数 | 61 | 2\*8 |
| 线程数/核 | 4 | 2 |
| 处理器时钟频率 | 1.238GHz | 2.60GHz |
| Cache 大小 | 30.5MB | 20MB |
| 内存大小 | 8GB | 64 GB |
| 单精度浮点性能 | 2130GFlops | 665.6GFlops |
| 双精度浮点性能 | 1208GFlops | 332.8GFlops |
| 功率 | 300W | 115W |

#### 测试方法

为了从计算性能、能耗、能效、开销四个方面综合评测本文DPTM映射方法的有效性[89]，主要从以下方面进行测试：

（1）测试在并程序执行时操作系统默认的线程数设置策略下，不同benchmark程序相对于串行程序的性能加速比。通常操作系统默认的线程数为处理器所能支持的最大硬件线程数[49]，该线程数默认设置策略在后续描述中表示为：OS\_Default。本文实验中MIC众核处理器所支持的最大硬件线程数为240。

（2）测试在不同的线程数情况下，所有的benchmark程序达到最优性能时（后续描述中表示为：Optimal\_Performance）的线程数及对应的加速比，作为理想的线程数设置标准；

（3）测试利用本文动态线程数调整映射机制DPTM后，所有benchmark程序所达到的性能加速比及对应的线程数，并和前面两种方法进行对比；

（4）测试本文线程数动态调整映射机制DPTM的额外开销及DPTM，OS\_Default，Optimal\_Performance三种映射方法下的能耗，对比三种映射方法下的能效。

### 测试结果及分析

测试结果主要从以下三方面进行对比分析：（1）本文动态线程数调整映射机制（DPTM）、最优性能（Optimal\_Performance）、线程数等于最大支持的硬件线程（OS\_Default）三种情况之间的性能差异；（2）三种情况下的能耗、能效比较；（3）比较DPTM下的额外开销对性能的影响，验证DPTM的有效性。

#### 计算性能评测

图4-7给出了Optimal\_Performance、OS\_Default及DPTM三种映射情况下，不同benchmark程序相对于单线程时所达到的加速比及对应的线程数（条形框上方的数字表示对应的线程数）。总体上DPTM的加速性能相对于OS\_Default提升了34.35%，同时达到最优性能（Optimal\_Performance）的96.8%。三种映射情况下平均启用的线程数分别为：161、240、118。由于DPTM能根据程序阶段性变化动态调整线程数，所以整个程序运行过程中所启用的硬件线程数最少，从而可以大大降低由于启用不必要的硬件线程而造成的额外能耗开销。

图4-7 计算性能及对应线程数对比

从图4-7中可以看出，对大部分测试程序，DPTM可以获得比OS\_Default较好的计算性能，主要原因是这些程序大部分都是存储受限或通信受限的应用，当启用太多线程时，会增加同步开销、引起共享cache冲突、加大对共享存储带宽的争用[105]，从而影响程序计算性能。DPTM由于能根据程序阶段性变化对线程数进行动态调整，所以可以取得较好的计算性能。而对某些基准测试程序，如*blackscholes*、*ferret*、*raytrace*、*freqmine*，DPTM和OS\_Default相比，计算性能并没有提升，甚至有略微下降，主要原因是这些程序属于计算密集型的应用，其计算性能会随着线程数的增加而保持上升的趋势，所以当启用最大线程数240时可以获得较好的计算性能。

#### 能耗评测

通过测试对比基准程序在不同线程映射方法下，在MIC协处理器上运行时的能耗，来评测本文DPTM 映射方法在能耗方面的有效性。通过每隔100毫秒读取Linux系统文件/sys/class/micras/power中的信息来获取MIC处理器的功耗（power）信息，最后计算出程序运行过程中的平均功耗，和运行时间(T)相乘后得到程序运行时的总体能耗(Energy comsumption)。具体通过最优线程数映射(Optimal\_Performance)和本文DPTM映射方法分别和最大硬件线程数映射（OS\_Default）方法之间的相对能耗指标（*Normalized energy consumption*）来进行分析对比。因为在最大硬件线程数情况下系统的能耗最高，所以其它两种映射情况下的能耗以它为基准进行比较。具体的相对能耗计算公式如下：

（4-1）

图4-8比较了最优线程数映射方法（Optimal\_Performance）和本文DPTM映射方法相对于系统默认的最大硬件线程映射方法（OS\_Default）的能耗比。

图4-8 相对能耗比较

从图中可以看出，最优性能情况下的能耗平均达到最大线程时能耗的53.7%,而本文DPTM映射机制由于能根据程序阶段性变化动态调节线程数，可以在程序不同阶段设置最合理的线程数，从而避免启用不必要的线程，使整个程序执行过程中启用的线程总数最小，从图中可以看到，本文DPTM映射机制能耗总体上低于最优性能映射的情况，平均只达到最大硬件线程映射方法总体能耗的41%。

#### 能效评测

为了进一步分析三种线程映射情况下，程序的计算性能和能耗的关系[90]，本节采用归一化的能效指标来衡量计算性能和能耗之间的关系[108][109][110]。具体的归一化的能效指标定义如下：

（4-2）

式中，；。其中*Energy-performance efficiency*表示能效；*Speedup*表示加速比，主要用来度量程序的计算性能*Performance*；Energy consumption表示能耗；Power表示整个应用程序执行过程中的总功耗；*P0*表示单个线程单位时间内的平均功耗；T表示程序执行时间；表示程序执行过程中每个线程平均执行时间；*Nth*表示线程数。由于*P0*在程序执行过程中是一个常数，所以也是一常数，故在公式（4-2）的基础上，得到公式（4-3）所示的用来度量能效的归一化指标Normalized enefgy efficiency：

 (4-3）

根据公式（4-3），计算了三种线程映射情况下不同基准程序的归一化的能效指标。该归一化的能效指标值越大，说明在相同的能耗情况下获得了更好的计算性能，也说明计算资源利用的越好[92]。

在具体测试时，DPTM线程映射方法由于在整个程序运行过程中，线程数要根据程序运行的阶段性变化动态调整，所以采用了整个过程中所启用的平均线程数作为计算归一化能效指标的值。加速比为各个基准程序在三种映射情况下，最终得到的相对于单线程运行时的加速比值，执行时间为执行完整个程序所用的平均时间。为了保证测试的准确性，所有的测试数据都是经过10轮测试后所取得的平均值。

图4-9给出了不同基准测试程序在三种线程映射情况下归一化后的能效比较。从图中可以看出，不同应用情况下，除streamcluster基准程序和最优线程数映射方法相比略低一点，vips基准程序和最优线程数映射方法相等外，其它基准程序DPTM的能效值都高于其它两种映射方法。从不同应用程序在三种线程情况下的几何平均值的比较可以看出，总体上DPTM线程映射机制取得了较好的效能。

(a)

(b)

(c)

图4-9 能效比较

#### 开销评测

本文DPTM映射机制，在具体实现时，由于要不断的根据程序运行状态变化来动态调整线程数，会引入一定的额外开销。额外开销主要来自状态信息的传输及最优线程数的计算。通过统计这两部分在整个映射过程中的所花费的时间，来评测DPTM所引入的额外开销对整个程序计算性能的影响程度。在测试中我们用DPTM预测时间来表示由于传输程序性能状态信息及预测最优线程数所花费额外时间。

表4-4给出了不同benchmark测试程序在本文DPTM映射机制下整个执行所用的时间、引入的预测时间、相对于最大硬件线程数映射方法所获得的性能提升及能耗降低比例、引入的额外开销所占总执行时间的比例。

表4-4动态预线程映射机制额外开销及性能提升比较（时间/秒）

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Benchmark | Total  time | DPTM prediction  time | DPTM #threads | Performance  improvement | Energy  reduction | Overhead |
| blackscholes | 192.63 | 0.83 | 144 | **-0.9%** | 39.5% | 0.43% |
| ferret | 20.80 | 0.08 | 168 | **1.0%** | 30.7% | 0.38% |
| Streamcluster | 70.95 | 2.82 | 72 | 140.9% | 87.5% | 3.97% |
| raytrace | 87.04 | 0.37 | 144 | **0.0%** | 40.0% | 0.42% |
| bodytrack | 37.05 | 1.06 | 96 | 98.2% | 79.8% | 2.86% |
| vips | 6.13 | 0.12 | 72 | 84.9% | 83.8% | 1.97% |
| cannel | 148.18 | 5.51 | 48 | 91.2% | 89.5% | 3.72% |
| freqmine | 37.23 | 0.15 | 216 | **-5.2%** | 25.0% | 0.39% |
| X264 | 6.90 | 0.13 | 72 | 38.3% | 78.3% | 1.89% |
| swaptions | 7.92 | 0.34 | 144 | 37.3% | 56.3% | 4.26% |
| Average | - | - | - | 48.6% | 59.0% | 2.03% |

从表中可以看到*blacksholes*、*ferret*、*raytrace*、*freqmine*四个测试程序在本文DPTM映射机制下和最大线程数情况相比性能没有提升，且性能还有所下降，这主要是因为这三个基准程序都是计算密集型的应用，计算性能随着线程数增加持续增加，在最大线程数情况下可以获得最优性能。而本文DPTM机制在预测过程中为了达到最优的效能，只预测到了达到近似最优性能下的最小线程数，所以加速比略低于最大线程时的加速比，但可以获得较高的效能。从图中可以看出，这四个基准测试程序的能耗都有明显下降且能耗下降的幅度远远大于性能下降的幅度和引入的额外开销比例。这充分说明本文DPTM尽管在单纯的计算性能指标上在某些应用程序上略差于最大硬件线程映射策略，但从整体能效上相比却有较大优势。

对其余的基准测试程序，本文DPTM机制在提升性能的同时能耗也明显降低，且性能提升率和能耗下降程度远大于引入的额外开销比例。

总体上来说，对所采用的十个基准测试程序，本文DPTM映射机制相对于最大硬件线程数映射策略，平均性能提升了48.6%，平均能耗降低了59%，而平均引入的额外开销为2.03%，引入的额外开销相对于获得的能效收益的比例来说完全是可以忽略的。文献[43]中所提方法计算性能提升率为17%，文献[69]中所提方法计算性能提升率为17.9%，能耗降低率为26.7%。这也说明了本文DPTM映射机制的有效性。

## 本章小结

本章提出线程数动态调整映射机制DPTM，通过实时的检测程序运行状态及阶段性变化，根据收集到的反映程序运行状态硬件性能计数器信息，利用所提出的数学模型直接计算出最优线程数的预测值，然后根据计算出的最优线程数对程序的并行性进行调节，以降低系统的整体能耗。DPTM映射机制从运行时开销、预测精度、计算性能提升、能耗降低等方面相比已有的方法都有一定的优势。同时在MIC异构众核系统上以运行时的方式实现了DPTM，可以根据程序运行状态及阶段性的变化，实时的对运行程序的线程数进行调节，使程序本身的运行特点和具体的运行平台硬件架构较好的匹配，在充分利用计算资源提高程序计算性能的同时降低系统的整体能耗。DPTM既能依据程序的动态特性实时的控制线程数，提升程序的性能，又可通过合理控制启动的线程数和处理核数，减少由于启用过多的线程和处理核而造成的额外能耗，达到降低系统整体能耗的目的；同时基于线程预测模型进行预测线程数，不会引入过高的额外开销，从而可高效的利用异构众核系统的计算能力来提升应用程序的性能，提高整个异构众核并行系统的能效。基准程序测试表明，DPTM线程映射机制，在提升计算性能和降低能耗方面是有效的，可以根据程序运行过程中阶段性变化及程序不同执行阶段对计算资源不同需求的特点，动态的对程序的线程数进行调整，以较低的开销达到高性能和低功耗的目的。

# 基于数据重用距离的线程数据相关性度量方法

随着多核/众核技术的发展，众核处理器片内集成的处理器核数越来越多，进一步加剧了多个处理核对片上共享计算资源（如共享cache，共享带宽）的争用。同时众核处理器采用了多级层次化的存储结构，导致其存储访问更加复杂。如果将无数据相关性的多个线程分配到同一处理核上，会因不同线程访问不同数据，导致共享cache数据的频繁换入换出，造成过高的共享存储访问冲突，增加额外传输开销，影响程序整体性能。在将应用程序映射到众核处理器时，只有将应用程序数据局部性和处理器存储架构有效的结合起来，实现应用程序到处理核的合理映射，才能提高计算资源利用率，提升应用程序的计算性能。

研究不同线程之间的数据局部性及存储访问特点，定量分析不同线程之间的数据相关性，对实现应用线程数据局部性和应用平台架构空间局部性较好的匹配、减少存储访问延迟及由于共享资源竞争而造成的过高额外开销、提高异构众核系统的整体能效具有重要意义。为了能准确的量化反映应用程序不同线程之间数据局部性关系的数据相关性，本章研究了数据重用距离计算方法，基于数据重用距离提出了一种线程间数据相关性度量方法RDAQ (Data Reuse Distance based Data Affinity Quantifying)。

## 引言

在异构众核系统上运行具体程序时，如何合理的将不同线程映射到具体处理核上，降低由于不同线程之间存储访问、信息交互造成的额外开销，对程序性能、异构系统整体能耗都具有重要意义[113][114][115]。在主处理器+协处理器的新型异构众核系统下，由于主、协处理器各自拥有独立的存储系统，同时两类处理器内部又采用层次化的存储结构，导致其存储访问更加复杂。如MIC众核处理器，其处理核内部硬件线程共享核内的L1 cache，不同处理核之间通过环形总线全局共享L2 cache。处理核内部和处理核间的数据传输延迟相差很大，L1 cache 访问延迟为1个时钟周期，而L2 cache访问延迟为11个时钟周期。如果在程序运行过程中任务分配不合理，将具有频繁信息交互的线程分配到不同处理核的硬件线程之上，则线程之间交互信息时由于要通过环形总线经L2 cache后再到L1 cache，会引入较高的存储访问延迟，造成高的数据传输开销；如果将无数据相关性的多个线程分配到同一处理核上，则会因不同线程访问不同数据，导致共享cache数据的频繁换入换出，造成过高的共享存储访问冲突，增加额外的传输开销，影响程序的整体性能。在将应用程序映射到众核处理器时，不但要考虑处理器的存储结构特点，同时也要考虑应用程序本身的存储访问特点，只有将应用程序数据局部性和处理器存储架构有效的结合起来，实现应用程序到处理核的合理映射，才能降低不同线程之间共享存储访问冲突、减少额外传输开销、提高计算资源利用率、提升应用程序的计算性能、降低异构系统整体能耗[116]。

如何将应用程序线程间数据相关性和异构众核系统物理架构特点结合，实现应用线程数据局部性和应用平台架构空间局部性较好的匹配，减少存储访问延迟及由于共享资源竞争而造成的过高额外开销，在保证充分利用处理核计算资源的同时尽量降低系统能耗，实现高效的线程映射，对提高异构众核系统的整体能效具有重要意义。

程序局部性一般从时间和空间两方面考虑。时间局部性指同一数据在一定的时间间隔内被连续访问的性质；空间局部性指相邻存储位置的数据被连续访问的性质。

下面给出具体量化程序局部性时用到的主要评测指标[124]：

（1）固定窗口内存占用量（Footprint），指在某一规定的时间窗口内，所访问的不同数据的数量，该指标数值越小说明数据局部性越好；

（2）固定数据量访问时间（Volume fill time），指程序访问给定数量的数据时，所用的平均时间，该值越小表明数据的局部性越好；

（3）Cache缺失间隔（Inter-miss time），指在给定的cache大小的情况下，两个连续cache缺失之间的时间间隔，该值越大表示程序的数据局部性越好；

（4）Cache缺失率（Miss ratio），指引起cache 缺失的访问数据占总访问数据的比例，该值越小表示程序的数据局部性越好；

（5）数据重用距离（Data reuse distance），表示在两个连续的相同访问数据间隔内，所访问的不同数据的个数，该值越小表示程序的数据局部性越好。

以上五个评价程序数据局部性的指标中，数据重用距离由于能直观准确的反映程序的数据局部性，被广泛的应用在性能分析、程序优化、虚拟存储及高速缓存管理、网络数据传输分析等方面。在具体应用时可以通过基于训练的预测、基于剖分及编译器的静态分析、基于采样的动态运行时分析等方法来获取具体反映数据局部性的评测指标值。

通过2.4.3节分析，程序数据局部性的方法可以归纳为三类：基于预测的分析方法、基于编译器的静态剖分方法、基于运行时的动态采样分析方法。通过预测的方法来分析数据局部性，因预测精度受限于训练样本的大小，不能精确反映程序的数据局部性关系，但可以快速预测数据局部性，可以用在对预测精度要求不高的情况下；基于编译器的静态剖分数据重用距离的方法，一般可以获得比较准确的数据局部性，但要依赖于具体编译器或特定硬件的支持，通用性受到限制；基于运行时的采样分析方法，因要在程序运行过程中通过剖分、收集特定的硬件性能计数器信息来分析程序的局部性，可获得比较准确的数据局部性，但该方法不但需要特定的硬件支持，同时会引入额外的运行时开销，因抽取数据局部性而引入的额外开销会影响因数据局部性而获得的程序计算性能提升的收益，所以通常在具体应用时，要在其开销和性能收益之间做较好的平衡。

本章针对以上问题，通过剖分不同线程存储访问数据信息，设计线程数据重用距离计算方法。根据数据重用距离信息，定量分析线程内部数据局部性特点及不同线程之间的数据相关性，该数据局部性及数据相关性与具体的运行平台无关，反映了程序固有的线程内及线程间的数据相关性。具体采用模式分类的方法将不同线程归并为不同的局部性模式，根据线程不同的局部性模式，用相关性矩阵（Affinity矩阵）来度量不同线程间的数据相关性。本文数据相关性度量方法，可以应用在存储及高速缓存管理、数据传输、线程映射等方面，通过定量分析数据局部性，为其优化提供理论基础和参考依据。

## 数据重用距离理论

数据重用距离统计了对相同访问数据最近两次访问间隔内不同访问数据的个数，是一个能较好反映应用程序数据局部性的指标[125][127]。数据重用距离小，表明所访问数据有较好的数据局部性，被重用的频度高；数据重用距离大，表明所访问数据的数据局部性差，被重用的频度低。数据重用距离的概念自从1970年被Mattson[136]等提出来以后，被广泛的应用在cache miss率预测[137]、程序阶段性检测[138]、数据层次优化[139]、虚拟存储管理[[140]、I/O性能优化[141]等领域。如在cache性能优化中，用来判定数据cache 是否命中。对一个大小为N的全相联cache，如果数据的重用距离D小于N，则可以判断该数据为cache命中，反之则为cache不命中。通过计算数据重用距离，可以为性能分析、程序优化、虚拟存储及高速缓存管理、网络数据传输分析等提供依据。

### 数据重用距离的形式化表示

**定义5-1**： 为从时间点s开始到时间e点结束时间段内按顺序访问的数据序列，具体定义如下：

 (5-1)

**定义5-2**：为一数据序列内所访问的不同数据元素的个数，具体定义如下：

 (5-2)

**定义5-3**：为数据序列中数据w所访问的时间点集合，具体定义如下：

 (5-3)

**定义5-4**： 为数据序列中数据w最近一次访问的时间点，具体定义如下：

 (5-4)

**定义5-5**：结合以上定义，所访问数据的重用距离即当前所访问数据和最近一次访问该数据时间间隔内，所访问的不同数据个数，计算公式如下：

 (5-5)

### 数据重用距离实例

设在一定的时间段内，程序所访问的数据序列如表5-1所示。根据以上公式(5-1)至(5-5)，通过对该时段内，所访问的数据进行统计后，分别统计每一个访问的数据元素相对其最近一次访问时间间隔内不同数据元素的个数，即为所访问数据的数据重用距离。

表5-1 访问数据序列

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 时间点 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 访问数据 | a | b | c | a | b | d | b | c | d | f |

表5-2给出了表5-1中数据序列对应元素的数据重用距离统计结果。为数据序列中数据w在时间点i之前，最近一次访问的时间点。为数据w在时间点i处的数据重用距离。

表5-2 数据重用距离结果

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 时间点i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 访问数据 | a | b | c | a | b | d | b | c | d | f |
|  | 0 | 0 | 0 | 0 | 1 | 0 | 4 | 2 | 5 | 0 |
|  | ∞ | ∞ | ∞ | 2 | 2 | ∞ | 1 | 3 | 2 | ∞ |

## 数据重用距离计算

为计算程序所访问数据的重用距离，先要收集程序所有的访问数据。一般通过扫描程序，利用剖分的方式将程序所访问的数据存储在特定的数据结构中，然后根据所用的数据结构特点采用不同的方法，计算在一定时间间隔内不同数据的重用距离，来评测数据的局部性。

传统数据重用距离计算主要使用基于栈和树两种方法实现。基于栈的算法由于计算时要从栈顶开始依次顺序遍历所有的数据，效率较低；基于树的算法可以利用树的一些特殊性质减少冗余的遍历，具有较高的计算效率。

### 数据重用距离计算方法

为了在异构众核系统下实现高效、准确的重用距离计算，本文采用基于树的算法来计算数据重用距离。通过构造平衡二叉树的方法来实现整个应用线程数据重用距离的计算。通过并行的对每个应用线程进行剖分扫描后获得所访问的存储访问数据，将收集到的访问数据同时插入到对应平衡二叉树的相应数据结点上，在插入数据结点的过程中同时计算所插入的访问数据重用距离，同时将计算得到的数据重用距离信息记录在相应的数据结点内，当扫描完整个应用线程后，随即生成一棵记录了应用线程全部不同访问数据及其对应重用距离信息的平衡二叉树。然后通过遍历该平衡二叉树，计算包含树中所有数据结点的平均重用距离，该平均数据重用距离反映了整个线程的数据局部性关系。

具体实现时，在异构众核系统主机端采用多线程并行方式剖分不同线程的数据存储访问信息，同时并行计算不同线程的数据重用距离。具体参考文献[128][133]中的方法，使用Intel Pin API编写Pin工具，并行统计每个线程的存储访问数据、计算线程内每个数据的重用距离，根据线程内不同数据的重用距离信息，计算反映整个线程数据局部性的线程平均数据重用距离。

### 线程访问数据收集方法

采用Intel Pin API 编写Pin工具[142]实现对线程不同访问数据的收集。Pin是Intel公司开发的应用广泛的动态插桩工具，为用户提供了一套丰富的应用程序接口（API），隐藏底层指令集实现细节，使用户能够方便地获取程序执行过程中的上下文信息（如各种寄存器的状态值），且大多数API都是跨平台的。Pin用户只需根据API编写实现特定功能的插桩代码，Pin将自动在执行到插桩代码时保存线程寄存器状态，并在执行完毕后恢复寄存器状态，保证线程能够正确往下执行。由于Pin的易用性、灵活性及丰富功能，使得Pin成为学术界影响力最大的插桩工具之一。

#### Pin软件架构

Pin工作方式是利用即时编译器（just in time compile）动态产生可被运行的代码，在产生代码的同时注入插桩代码，注入的插桩代码不仅可以是指令级的，也可以是基本块级的。Pin可以在目标程序中的任意位置动态注入用C/C++编写的函数调用，而不需要进行任何重编译和重链接。Pin还具有寄存器重置、指令优化、程序动态分析等能力，它不但能够在进程入口点获取对进程的控制，还允许对一个正在运行中的进程进行插桩。要实现对一应用程序的插桩，需要图5-1中所示的应用（application）、Pin工具（Pintool）、Pin引擎三个部分相互配合才能实现。其中应用是要进行插桩的目标程序，Pin是对应用程序进行即时编译并实现插桩的引擎，Pintool包含了用于具体插桩和分析的程序例程，该程序例程同时被封装成了连接库，可以被Pin直接调用。Pin引擎主要由虚拟机（VM），代码cache，插桩应用接口（Instrumentation APIs）三个主要部分组成。其中VM又由即时编译器（JIT Compiler）、模拟器（Emulation Unit）、分配器（Dispatcher）组成。当Pin引擎获得应用程序的控制权后，便由VM协调它内部的部件来执行具体的操作。JIT Compiler主要完成对应用程序编译和插桩，之后被分配器调用，将编译后的代码及所收集到的状态信息保存在代码cache中，该信息可以直接通过应用接口返回到应用程序，从而实现对程序应用状态信息的收集。Pin具体的软件架构[143]如图5-1所示。



图5-1 Pin 软件架构

#### Pintool工具实例

下面通过一个对应用程序内存进行读写的Pin工具实例，说明Pin工具的编写方法及对Pin API的调用关系。具体实现代码如图5-2所示。

在min() 程序段中，首先调用PIN\_Init(argc, argv) 函数对Pintool进行初始化，然后通过trace=fopen(“pinatrace.out”, “w”) 语句打开保存所要收集的程序状态信息的目标文件。之后用INS\_AddInstrumentFunction(Instruction, 0)语句声明所要调用的插桩函数Instruction，同时调用PIN\_StartProgram()函数，启动pin工具对应用程序进行动态插桩。整个插桩结束后，调用函数PIN\_AddFiniFunction(Fini, 0)，完成对插庄后的相应原信息的保存、状态恢复等处理。

在具体的插桩函数Instruction(INS ins, VOID \*v)中，通过语句INS\_IsMemoryRead(ins)和INS\_IsMemoryWrite(ins) 对内存的读和写进行判断，之后分别调用pintool提供的INS\_InsertCall(ins, IPOINT\_BEFORE，AFUNPTR(Address)，IARG\_MEMORYREAD\_EA, IARG\_END) 应用程序接口函数实现具体的插桩操作。

本文利用Pin工具，通过在相应的程序代码中插入插桩代码，实现对线程访问数据的剖分，并通过设计平衡二叉树插入算法，将剖分得到的访问数据插入到所定义的平衡二叉树相应结点的数据结构中，同时在插入数据结点的过程中计算对应数据的重用距离。整个线程的访问数据统计是一个在二叉树中插入数据节点、生成二叉树的过程。

|  |
| --- |
| #include <stdio.h>  #include “pin.H”  FILE \* trace;  VOID Address(VOID \* addr)  {  fprintf(trace,”%p\n”, addr);  }  VOID Instruction(INS ins, VOID \*v)  {  if (INS\_IsMemoryRead(ins))  {  INS\_InsertCall(ins, IPOINT\_BEFORE, AFUNPTR(Address),  IARG\_MEMORYREAD\_EA,  IARG\_END);  }  if (INS\_IsMemoryWrite(ins))  {  INS\_InsertCall(ins, IPOINT\_BEFORE, AFUNPTR(Address),  IARG\_MEMORYWRITE\_EA,  IARG\_END);  }  }  VOID Fini(INT32 code, VOID \*v){ fclose(trace); }  int main(int argc, char \*argv[])  {  PIN\_Init(argc, argv);  trace = fopen(“pinatrace.out”, “w”);  INS\_AddInstrumentFunction(Instruction, 0);  PIN\_StartProgram();  PIN\_AddFiniFunction(Fini, 0);  return 0;  } |

图5-2 对内存读写进行剖分的Pin工具实例

### 线程访问数据统计

已有工作，如文献[128]中，为了提高基于树的重用距离算法遍历数据结点的速度，利用hash表来辅助计算重用距离。该方法在搜索目标结点时，先查询hash表，根据在hash表中查找到的目标结点关键字，通过直接搜索二叉树来统计数据重用距离，以提高目标结点的搜索速度。但该方法在计算重用距离之前必须要构建相应的hash表，将二叉树的搜索开销转嫁到查找hash表上，没能从根本上提高目标结点的搜索效率。而构造hash表，从时间和空间两方面引入了额外开销。本文统计线程访问数据时，采用将访问数据插入二叉树的方式，在插入数据结点的同时计算数据重用距离，并将相应数据重用信息记录在对应的数据结点内。当扫描完线程的所有访问数据后，随即生成包含线程访问数据信息及数据重用距离的平衡二叉树，从空间和时间两方面降低数据重用距离计算的复杂度。

#### 数据结构定义

本文通过构造平衡二叉树，实现应用线程访问数据的统计及重用距离的计算。采用二叉链表形式存储具体的平衡二叉树，所用的二叉树结点数据结构定义如下：

typedef struct node

{

int TS;

float Element;

int Frequency;

int Weight;

int RD;

struct node \*left\_child, \*right\_child;

}BT;

以上定义的平衡二叉树结点用于存储访问到的不同数据信息。二叉树每个结点所包含的数据项具体存储以下信息**：**

TS (Time stamp)：时间戳，记录数据被访问的先后次序；

Element：记录访问的数据元素；

Frequency：频度，记录数据被访问的次数，用于统计整个线程的数据访问量；

Weight：权重，记录当前结点所包含的子结点个数，用于计算重用距离；

RD (Reuse Distance)：数据重用距离；

Left\_child：结点左子树指针；

right\_child：结点右子树指针。

#### 统计数据访问信息

整个线程数据重用距离计算是一个在平衡二叉树中插入结点、删除结点和遍历结点的过程。当依次扫描完线程所访问的数据后，随即生成一棵记录了线程数据重用距离信息的平衡二叉树。通过遍历该二叉树，计算出整个线程的平均数据重用距离，以此量化线程内数据局部性。

线程数据访问信息的统计，通过在二叉树中插入结点的方式实现。具体插入数据结点时，采用以数据时间戳（时间戳代表数据被访问的次序）为主键的平衡二叉树中序插入算法。每插入一个结点的同时，将相关数据项信息记录在该结点的数据结构中，节点数据项包括以上定义的完整结构体信息node (TS；Element；Frequency；Weight；RD；left-child；right-child)。

在插入一个新数据之前，先遍历当前生成的二叉树，判断该数据是否已经记录在二叉树中，若已在当前二叉树中，则计算该数据的重用距离，计算完重用距离后，将以前存储该数据的结点从已有二叉树中删除，并重新调整二叉树，之后将当前数据结点插入到树中相应位置；若待插入的数据不在当前二叉树中，则直接按标准的插入算法将数据结点插入到二叉树中。这样不断迭代直到将线程所有访问的数据及相关信息都记录在二叉树中，即完成对整个线程访问数据的统计。最后生成的平衡二叉树所包含的结点就是线程所访问的所有不同数据，同时每个结点信息中记录了该结点数据被访问的次数及数据的重用距离。

#### 数据重用信息统计算法

本文数据重用信息统计算法包含：扫描访问数据、数据结点的插入、原数据结点删除、数据重用度的统计、数据重用距离计算五个过程。

具体的数据重用信息统计算法如下：

（1）初始化：定义数据结点数据结构node (TS；Element；Frequency；Weight；RD；left-child；right-child)及空二叉树.

（2）调用Pin工具扫描线程所访问的数据变量，记录扫描到的数据时间戳ti及数据变量di.

（3）给待插入结点数据项赋初值：TS=ti；Element=di；Frequency=1；Weight=1；RD=∞.

（4）判断待插入结点数据是否已包含在当前二叉树中，中序遍历当前二叉树，比较待插入数据是否包含在树中已有结点中：

i. 如果待插入结点的数据已包含在当前二叉树中，则：

(a) 调用数据重用距离计算函数（见算法1），计算当前数据重用距离RD.

(b) 统计待插入结点数据重用频度：当前查找到的包含待插入结点数据变量的对应数据结点数据访问频度值加1后，赋值给待插入数据结点对应的数据项变量Frequency.

(c) 删除当前平衡二叉树中查找到的包含待插入结点数据的已有数据结点.

(d) 调整平衡二叉树，将待插入结点插入到当前平衡二叉树中.

ii. 如果待插入结点的数据没有包含在当前二叉树中，则直接将待插入结点插入到当前平衡二叉树中.

（5）重复（2）、（3）、（4）步，直到扫描完线程的所有访问数据。最后生成在结点中包含线程所有不同的访问数据、对应数据的访问频度、对应数据重用距离信息的平衡二叉树.

（6）对树中重用距离为∞的结点进行调整，将其重用距离统一调整为树中总的结点个数M（即树中根结点weight数据项的值），表示当前最大的重用距离.

（7）完成数据重用信息的统计.

### 基于平衡二叉树的数据重用距离计算

计算访问数据重用距离，就是统计最近两次相同访问数据之间不同访问数据的个数。本文整个访问数据重用距离的计算过程，是一个高效遍历已生成的平衡二叉树的过程。在计算具体访问数据重用距离时，结合平衡二叉树的存储结构，先在已有的存储了当前访问数据的平衡二叉树中，查找包含和当前访问数据变量相同的数据结点，找到后通过比较数据结点时间戳，结合平衡二叉树的特性，通过高效的遍历平衡二叉树来统计当前访问数据和其有相同访问数据结点之间的数据结点个数，从而完成对当前数据重用距离的计算。因当前已有的平衡二叉树，是在统计访问数据时，以访问数据的时间戳为主键构造的，所以可以只通过比较访问数据时间戳，结合平衡二叉树数据结构特点，便可高效的完成数据重用距离的计算。

在上一节统计访问数据的过程中，每插入一个数据结点之前，先计算出该数据当前重用距离及数据的重用次数，并将计算结果赋值给待插入数据结点相应的数据项变量，用于度量数据的局部性。本文将数据重用距离计算过程封装成一个独立的函数，在子树生成过程中直接调用，具体实现如算法5-1所示。算法中输入变量Nroot表示二叉树根节点指针，Ntarget表示当前数据结点（目标结点）指针，RD为最终计算出的数据重用距离。计算数据重用距离RDC算法，具体执行过程如下：

1）在当前树中查找包含待插入数据的结点，如果当前树中没有包含待插入数据的结点，说明该数据为首次访问，重用距离置为∞；

2）如果在当前树中找到包含要插入数据的结点，分两种情况进行计算：

（1）若查找到的目标结点时间戳小于根结点时间戳，由平衡二叉树属性可以判断出该结点属于根结点左子树结点。这时根结点右子树结点数加上根结点左子树中时间戳大于目标结点时间戳的所有结点个数即为待插入结点的数据重用距离.

重用距离计算方法：根结点右孩子结点weight值（根结点右子树结点个数）加根结点左子树中时间戳大于目标结点时间戳的结点个数rd.

（2）若查找到的目标结点时间戳大于根结点时间戳，表明目标结点属于根结点右子树结点，这时目标结点根结点右子树中，时间戳大于目标结点时间戳的结点个数rd，即为待插入结点数据重用距离.

3） 比较计算出的待插入结点数据重用距离和当前查找到的目标结点数据重用距离，将两者中较小值作为待插入结点最终的数据重用距离RD，以保证重用距离计算的有效性；

4）根结点左子树（或右子树）中，时间戳大于包含待插入数据结点（该结点称为目标结点）时间戳的数据结点个数计算方法如下（最终计算出的结点个数为rd）：

（1）rd赋初值为0；

（2）将当前包含待插入数据结点（目标结点）的右孩子结点weight 值赋给rd，并将目标结点作为当前结点；

（3）回溯到当前结点父结点；

（4）如果当前父结点不为根结点且其时间戳大于目标结点时间戳，则将当前父结点weight 值和其左孩子weight值相减后加到变量rd中（相当统计了当前结点右兄弟结点个数）。将当前结点父结点作为当前结点，转到步骤（3）继续执行；

（5）如果当前父结点时间戳小于目标结点时间戳，则将当前父结点作为当前结点，直接转到步骤（3）继续执行；

（6）如果当前父结点为根结点，则计算完成，当前rd值，即为根结点左子树（或右子树）中，时间戳大于当前结点时间戳的结点个数。

计算数据重用距离的RDC具体算法如下：

|  |
| --- |
| **算法 5-1：** 数据重用距离计算RDC算法 |
| 1 **Input**：Nroot, Ntarget  2  **Output**：RD  3  **Begin**  4 RD=0;  5 **if** ((Ntarget->TS)<(Nroot->TS)) **then**  //目标结点时间戳小于根结点时间戳，目标结点在根结点左子树中  6 RD=(Nroot->right\_child.weight)+1+(Ntarget->right\_child.weight);  7 p=Ntarget->parent;  8 **while** (p!=Nroot)  9 **if** ((p->TS)>(Ntarget->TS)) **then**  10 RD+=(p->weight)-(p->left\_child.weight);  11 p=p->parent;  12  **else**  13 p=p->parent;  14 **end if**  15 **end while**  //将目标结点和计算出的待插入结点中较小的重用距离，作为  //待插入结点最终的重用距离值  16 RD=（RD<Ntarget->RD)? RD: Ntarget->RD;  17 return RD;  18 **else**  //目标结点在根结点右子树中  19 RD=Ntarget->right\_child.weight;  20 p=p->parent;  21 **while** (p!=T)  22 **if** ((p->TS)>(Ntarget->TS)) **then**  23 RD+=(p->weight)-(p->left\_child.weight);  24 p=p->parent;  25 **else**  26 p=p->parent;  27 **end if**  28 **end while**  29 RD=（RD<Ntarget->RD) ? RD: Ntarget->RD;  30 return RD;  31 **end if**  32  **End** |

### 算法复杂度分析

设线程访问数据总数为N，不同的访问数据个数为M。因整个算法实现过程主要包括平衡二叉树节点的插入、遍历、删除操作，在整个操作过程中没有引入额外存储开销。整个算法时间开销主要在搜索待插入数据元素及计算重用距离上，一旦查找到满足条件的数据结点，则通过从当前结点回溯的方法计算出待插入数据的重用距离。因树中结点只存储了最近一次不同的访问数据，要从当前树中查找包含待插入数据的结点，必须从根结点开始依次搜索左子树、右子树，只到查找到满足条件的数据结点为止，最坏情况下要遍历当前树中所有结点M，搜索目标结点时间复杂度为O(M)。搜索到目标结点后计算待插入数据的重用距离时只对整个二叉树的左或右子树进行回溯，计算重用距离的时间复杂度为O(log (M/2))。整个算法的时间复杂度为：О(N·(M+log(M/2)))。由于算法在计算数据重用距离时只是比较当前遍历到的结点数据和待插入数据，在比较和计算过程中没有引入额外的存储开销，空间复杂度为：О(N)。

### 线程数据重用距离计算实例

下面用一个具体实例，说明本文统计线程访问数据信息和计算数据重用距离的过程。设收集到的线程待访问的数据序列如表5-3所示，其中Frequency和Reuse distance相应的值，通过执行上文中的重用距离算法后获得。

表5-3 线程数据访问序列

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Time stamp | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| Element | d | b | c | e | a | b | e | c | f | a |
| Frequency | 1 | 1 | 1 | 1 | 1 | 2 | 2 | 2 | 1 | 2 |
| Reuse distance | ∞ | ∞ | ∞ | ∞ | ∞ | 3 | 2 | 3 | ∞ | 4 |

访问数据的统计及重用距离计算过程如图5-3所示。图中分步骤显示了将表5-3中线程所访问的数据依次插入到平衡二叉树中，并同时计算各个数据重用距离及重用频度的过程，主要包括二叉树的搜索、删除、插入操作，最后生成了如图（i）所示的，只包含了不同访问数据d, b, c, e, f, a及其相应重用距离信息结点的平衡二叉树。

图5-3 线程数据信息统计过程示意图

### 线程平均数据重用距离计算

生成反映线程数据重用距离的平衡二叉树后，通过遍历每个线程对应的二叉树，统计树中所有结点的数据重用距离总和，来计算线程的平均数据重用距离。设线程总数为K，每个线程的平均数据重用距离为RDj (j=1, 2, …, K)，线程内部每个数据的重用距离为rdi，线程访问的不同数据总数为M（平衡二叉树结点个数），则线程平均数据重用距离为：

. （5-6）

线程平均数据重用距离反映了每个线程内的数据局部性特点。平均重用距离越大，表明线程内部数据重用率越低，反之表明线程内部数据重用率高、数据局部性好。本文用线程平均数据重用距离指标来度量线程内部数据局部性强弱。

## 数据相关性判定及度量

在完成应用线程的重用距离计算后，根据线程平均数据重用距离，分析反映不同线程内部数据局部性特点的不同局部性模式。以线程数据局部性模式为依据，将线程归并为不同的模式类。分析每个模式类内不同线程间的数据相关性，通过统计不同线程间访问的相同数据的数据量，用线程相关性矩阵（Affinity Matrix）来度量不同线程之间的数据相关性。

### 线程内数据局部性模式分析

为了能定量的分析不同应用线程之间的数据相关性，首先根据反映不同应用线程内部数据局部性的平均数据重用距离，将不同线程归并为不同的模式类，然后在数据局部性模式类的基础上，分析量化不同线程之间的数据相关性。

不同的模式代表了不同的数据局部性特征，用数据重用距离信息进行量化，根据数据存储访问特点，设置反映不同数据访存特点的数据重用距离阈值Dmin及Dmax。以重用距离阈值为基准，将数据重用距离划分为三个不同的区间，每个重用距离区间对应一种局部性模式。通过将不同线程平均数据重用距离和重用距离阈值比较后，标识出线程所属的局部性模式。

#### 数据局部性模式定义

通过分析，定义以下三种反映不同数据局部特征的数据局部性模式。

**定义5-6： DSP** (Data Sharing Pattern)，为共享数据模式，且满足条件：**。

以上定义中RDj为应用线程平均数据重用距离，Dmin为数据重用距离阈值。只要线程的数据重用距离小于该阈值，则将线程归并为该模式。该模式下，线程所访问的数据有较强的时间局部性，数据间有较强的数据相关性，属于这种模式的线程要尽量分配到同一处理核不同硬件线程上，同时线程所访问数据要尽量分配到和线程执行指令相同的存储位置。

**定义5-7： DIP** (Data Isolation Pattern)，为数据无关模式，且满足条件：。

以上定义中Dmax为数据重用距离阈值，只要线程数据重用距离超过该阈值，则将该线程归并为该模式。该模式下，所访问数据的时间局部性差，数据之间相互独立，无数据相关性，属于该模式的线程要尽量分配到不同处理器不同处理核的不同硬件线程之上。

**定义5-8：DDP** (Data Dependency Pattern)，为数据依赖模式，且满足条件：。

在以上定义中，只要线程的数据重用距离介于Dmin和Dmax之间，则可将该线程归并为该模式。在该模式下，线程所访问数据有一定的时间局部性，数据之间存在数据依赖关系，有一定的数据相关性。属于该模式的线程可分配到同一处理器不同处理核的不同硬件线程上。

#### 数据重用距离阈值确定

数据重用距离阈值的选取对算法性能有重要的影响。如果阈值Dmin选取的太小，会将一些有一定数据相关性的线程排除在DSP(Data Sharing Pattern)模式类之外，若阈值Dmin选取的太大，则会将一些没有较好数据相关性的线程包含在DSP模式类中。同理对阈值Dmax的选取，同样存在使线程局部性模式分类不准确的问题，这会直接影响后续线程之间数据相关性度量的准确性。数据重用距离阈值Dmin和Dmax设置通过实验测试的方法获得。通过对不同基准测试程序进行多次测试后，计算出各个程序的数据重用距离信息，分析不同程序的数据局部性特性和数据重用距离的关系，比较所测得的具有较强数据共享性程序的数据重用距离，将其中最大的数据重用距离值作为阈值Dmin；比较具有相互独立访存关系程序的数据重用距离，将其中最小的数据重用距离作为阈值Dmax。本文实验中Dmin和Dmax分别为每个测试时间间隔内数据访问量的50%和85%。

### 线程间数据相关性度量

分析出各个线程数据局部性模式后，将不同线程归并为不同的模式类。在模式类的基础上，分析不同模式类内不同线程间的数据相关性。因为属于同一模式类的线程之间，具有更强的数据相关性，这些线程之间的数据相关性会直接影响具体线程映射时的性能。而对属于不同模式类的线程之间的相关性，其对线程映射时的性能影响一般很小，可以忽略不计[136]。本文主要通过比较同一个模式类内不同线程之间所访问的相同数据个数，并用线程相关性矩阵（Affinity矩阵）来记录通过比较统计得出的共享数据量。最后得到能完整反映程序不同应用线程之间数据相关性的Affinity矩阵，以此来度量不同线程间的数据相关性。

#### Affinity矩阵

Affinity矩阵(如图5-4所示)，反映了不同线程间的数据共享特性，矩阵行标和列标分别代表不同的线程ID，矩阵中的每个元素值代表对应行列所指线程间的数据共享量，矩阵元素值越大表明对应线程之间数据共享性越好，线程间的相关性越强。

#### Affinity矩阵的计算过程

Affinity矩阵的具体计算过程如下。首先比较不同线程间的共享数据，统计各个线程之间的共享数据个数，作为线程相关性矩阵中对应元素的值。以不同的模式类为基础，结合反映线程不同数据访问量的平衡二叉树，通过比较同一模式类内不同线程之间相同的访问数据，统计不同线程之间共享数据量。为了提高比较效率，利用并行化的方法进行比较，同时只统计属于相同模式类内不同线程之间的共享数据量，不同模式类之间线程局部性对整个程序的局部性影响很小，可以忽略不计。通过比较不同线程对应的平衡二叉树，计算出两二叉树相同数据结点的个数，即为对应两线程之间的共享数据量，同时将计算出的共享数据量记入线程相关性矩阵对应元素中。最终构造出反映程序线程间数据相关性的Affinity矩阵。

#### Affinity图

计算出线程Affiity矩阵后再将该矩阵转换成能直观反映线程间数据相关性的相关性图（Affinity图，如图5-5所示）。相关性图是一个顶点代表不同的线程ID，边代表对应两线程间数据共享量的无向图。

具体Affinity图的数据结构定义如下：

#define MaxVertexNum N

typedef char VertexType

typedef int EdgeType

typedef struct

{

VertexType Vexs[MaxVertexNum];

Edgetype edges[MaxVertexNum][ MaxVertexNum];

int V, E;

}AffinityGraph;

以上数据结构定义中，Ｎ在应用中对应具体处理器所能支持的最大硬件线程数，本文中所用Intel MIC 处理器所支持的硬件线程数为240。

图5-4和5-5给出了一个8线程应用程序，通过访问数据信息统计后所得到的反映线程相关性的Affinity矩阵及对应的Affinity图。

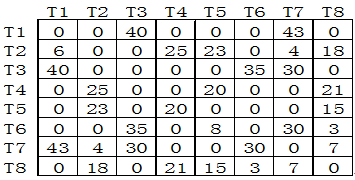
 

图5-4 Affinity 矩阵 图5-5 Affinity 图

在得到线程相关性图后，下一步是如何将线程间的数据相关性和具体运行平台物理存储层次架构相结合，将应用线程数据局部性和应用平台架构空间局部性较好的匹配，实现线程到具体运行平台的合理映射，减少存储访问延迟及由于共享资源竞争而造成的过高额外开销，在充分利用处理核计算资源的同时尽量降低系统能耗。

本章所提出的反映不同线程之间数据相关性的Affinity矩阵及对应的Affinity图可以准确的量化应用程序不同线程之间的数据相关性，可以为实现线程到具体运行平台合理映射，提供简单、准确、高效的应用线程数据局部性及应用线程之间数据相关性的度量方法。可为实现存储及高速缓存管理、数据传输优化、线程到处理核的合理映射等提供理论基础和参考依据。

## 数据相关性度量方法评测

### 数据局部性分析方法比较

#### 不同数据局部性分析方法比较

通过定性的对本文所提数据相关性度量方法和已有文献[126][128][132][133][134]中所提的数据局部性分析方法进行相比，说明本文方法的优势。

文献[126]在考虑多核共享cache环境下不同线程相互作用对数据重用距离影响的基础上，提出了数据并发性重用距离的概念。设计了一种概率模型，通过剖分或模拟，将并发重用距离和应用程序的数据局部性关联起来，简化了并发性重用距离的计算，预测共享cache在同时多线程时引起的cache缺失率（cache misses），实现了在CMP多线程环境下利用数据重用距离分析程序的数据局部性。但该方法所设计的概率模型只针对并发性重用距离和共享计算资源依赖性较小的一类多线程应用，通用性受到限制。而本文方法所提出的线程数据相关性度量方法通过在程序执行前计算不同线程静态重用距离，用线程相关性矩阵来度量不同线程之间的数据相关性，度量线程之间的数据相关性不受具体应用程序特点及不同线程之间相互作用的影响，能较准确的反映程序本身固有的数据局部性特点，通用性更好。文献[128]提出的基于预测数据重用距离来获取基于循环并行程序数据局部性的方法，用来评测和分析超大规模片上多核处理器存储性能，该方法克服了传统通过采样和模拟，抽取数据局部性方法不能适应超大规模片上多核处理器存储性能分析和评测的问题，为选择最优cache配置提供依据。所提出的预测数据重用距离的方法，主要基于循环并行程序线程对称性，高效预测在不同cache及处理核配置下存储访问数据的并发重用距离及私有cache栈的数据重用距离来获取反映cache性能评测指标，从而为设计合理的cache层次提供指导。通过预测的方法来分析数据局部性，因预测精度受限于训练样本的大小，不能精确的反映程序的数据局部性关系，可以用在对预测精度要求不高的情况下。同时该方法主要基于循环并行程序线程对称性来预测数据局部性，数据局部性预测受到一定的限制。和本文数据相关性方法相比，该方法所提出的预测数据重用距离的方法存在通用性受限和准确性不高的问题。

文献[132]在同时考虑应用程序和目标机器特点的基础上，提出了利用机器学习的方法自动产生反映程序数据局部性的AutoFinity策略，实现将应用程序线程合理分配到不同的处理核上，提高共享计算资源利用率，减少额外通信开销，提升应用程序的计算性能。该方法预先通过剖分、采样获取样本数据，对样本数据进行训练后获取应用程序运行特征，根据运行特征设定具体的线程到处理核的分配策略库，然后在程序运行的过程中通过和策略库匹配来确定程序具体的线程到处理核的分配方法。如果运行程序应用特征在分配策略库中无法匹配，则要对程序进行重新剖分、抽样、训练后扩充分配策略库。该方法和本文方法相比，由于预先要通过收集特定的硬件性能计数器信息，对应用程序进行剖分、采样、训练后分析程序的局部性，不但需要特定的硬件支持，而且会引入较高的额外的开销；另外在程序执行过程中通过匹配方法在已有的分配策略库中寻找相应的分配策略，不能反映程序本身固有的数据局部性特点，同时还会引入较高的运行时开销。

文献[134]提出了一种程序固有的数据局部性及改变程序的执行顺序后程序潜在数据局部性的动态分析方法，用于判定改变程序执行顺序后程序数据局部性提高的潜力。通过剖分程序、计算重用距离的方法构建反映程序不同执行部分依赖关系的有向无环图，通过对图的分割实现对程序迭代空间的划分，以此改变程序执行顺序，从而提高程序执行时的数据局部性。该方法主要思想是通过改变程序的执行顺序来提升程序的局部性，最终达到提升程序计算性能的目的。文中分析数据重用距离时仍然使用基于栈的数据重用距离计算方法，计算时要从栈顶开始依次顺序遍历所有的数据，和本文提出的基于平衡二叉树的计算方法相比，重用距离的计算效率较低。文献[135]提出了一种基于kernel的重用距离计算方法，该方法通过检测kernel输入和输出特征，可以快速的评测数据局部性，从而克服传统数据重用距离计算方法面临的插桩及计算重用距离高开销挑战。但该方法由于建立在基于kernel的较粗粒度的重用距离分析基础之上，主要通过检测kernel输入和输出特征来评测数据局部性，和本文基于数据重用距离的数据相关性度量方法相比，数据局部性的评测准确性上存在一定的不足。

通过以上和已有文献数据局部性分析方法相比，本文RDAQ方法不需要特定的硬件支持，直接通过计算数据重用距离来分析和度量数据局部性，实现简单；不受具体应用程序特点及不同线程之间相互作用的影响，通过基于平衡二叉的方法来并行的计算数据重用距离，能较准确、高效的量化程序本身固有的数据局部性特点，具有更好的通用性。

#### 算法复杂性比较

本文线程数据相关性度量方法中统计数据重用距离信息算法的时间复杂度为O(N·(M+log(M/2)))。基于栈的传统数据重用距离信息统计算法的时间复杂度为：(N2·M)。其中N为线程访问数据的总量，M为线程所访问的不同的变量数，通常M远小于N。所以在统计数据重用距离信息算法时间复杂度方面和已有方法相比也有一定的提高。

### 数据相关性度量方法有效性评测

为了定量的分析本文数据相关性度量方法的有效性，主要从RDAQ数据相关性度量时间、利用RDAQ后Cache 缺失率的变化两方面对其有效性进行定量的评测。具体的测试环境如6.6.1节所示，基准测试集使用PARSEC测试集中的OpenMP并行程序，输入集使用simsmall[50]。

#### 度量时间评测

通过比较在不同基准测试程序下，利用本文数据相关性度量方法RDAQ统计整个数据重用距离信息所用时间和在操作系统默认线程映射机制下各个基准测试程序以OpenMP并行方式执行时所用时间，以及两者之间的比值来作为评测本文数据相关性度量方法的有效性的指标[134]。

图5-6给出了不同的基准测试程序在OS默认映射机制（ﬁrst-touch policy）下，以OpenMP方式并行执行时所用时间和本文数据相关性度量方法RDAQ统计不同线程之间数据相关性所用时间比较。从图中可以看出，本文数据相关性度量方法统计数据相关性信息所用时间远远小于程序执行时间。图5-7给出了不同基准测试程序用本文RDAQ方法度量数据相关性时所用时间占整个程序执行时间的比例，以该比例值作为衡量数据相关性度量方法的相对开销的指标。从图中可以看出最大开销为8.7%，最小开销为7.1%，平均开销为8%。整个开销在10%以下，RDAQ方法引入的开销完全在可接收的范围内。文献[135]中所提及的通过动态插桩的方法分析数据局部性所增加的程序额外开销在20%以上。这也充分说明了本文线程数据相关性度量方法的有效性。

图5-6 OS默认映射机制下程序执行时间与RDAQ统计数据相关性所用时间比较

图5-7 不同基准程序下RDAQ相对于OS默认映射机制下程序执行时间比例

#### Cache 缺失率评测

为了进一步验证RDAQ数据相关性度量方法的有效性，通过测试在不同的基准测试程序下，使用本文RDAQ方法的线程映射、使用OpenMP所支持的Compact和Scatter映射、理想的映射（Oracle）四种情况下相对于OS默认映射机制（fisrt-touch policy）下的最后一级Cache缺失率相对变化率来评测RDAQ的有效性。

图5-8给出了Compact、Scatter、Oracle、RDAQ四种线程映射方法下相对于OS默认线程映射方法下，L2 Cache 缺失率变化情况。在不同基准测试情况下，OpenMP支持的线程映射方法Compact和Scatter与OS默认的线程映射机制（fisrt-touch policy）相比，L2 Cache缺失率分别增加了3%和2%。主要原因是OpenMP支持的线程映射方法，在进行线程映射时没有考虑具体应用线程之间的数据相关性，只从处理核的硬件线程结构方面考虑应用线程的映射，不能较好的利用应用线程之间的数据局部性和相关性来降低cache缺失率。和OS默认的线程映射机制相比，Oracle和RDAQ映射方法的平均L2 Cahce缺失率分别降低了12%和10.4%。RDAQ方法在降低Cache缺失率方面达到了理想映射方法的87%，说明RDAQ在度量线程数据相关性方面是有效的。

图5-8 不同映射方法下最后一级Cache缺失率相对OS默认映射下的相对变化率

通过以上分析比较，本文提出的基于数据重用距离的线程数据相关性度量方法，可以在不需要特定硬件支持的情况下，实现简单、准确、有效的应用程序线程间数据相关性的量化，通过定量分析应用程序不同应用线程之间数据相关性，可为实现存储及高速缓存管理、数据传输优化、线程到处理核的合理映射等提供理论基础和参考依据。

## 本章小结

简单、高效、准确的定量分析应用线程间的数据相关性，对实现应用线程数据局部性和应用平台架构空间局部性较好的匹配、减少存储访问延迟及由于共享资源竞争而造成的过高额外开销、提高异构众核系统的整体能效具有重要意义。本章通过剖分不同线程存储访问数据信息，设计并行线程数据重用距离计算方法，计算不同线程的数据重用距离；在数据重用距离的基础上，通过定义反映不同数据相关性的局部性模式，将不同的线程归并为不同的模式类；在不同模式类的基础上通过比较不同线程间数据共享量，用相关性矩阵来记录不同线程之间的数据共享量，以此来度量不同线间的数据相关性。同时将线程相关性矩阵转换成相应的相关性图，来直观的反映不同应用线程之间的数据相关性。该数据相关性和具体的运行平台无关，反映了程序不同线程之间本身具有的数据依赖关系。本文提出的基于数据重用距离的线程数据相关性度量方法，可以在不需要特定硬件支持的情况下，实现简单、准确、高效的程序数据局部性的量化。通过对应用程序程序不同线程数据相关性的定量分析，可为实现存储及高速缓存管理、数据传输优化、线程到处理核的合理映射等提供理论基础和参考依据，以实现应用线程数据局部性和应用平台架构空间局部性较好的匹配、减少存储访问延迟及由于共享资源竞争而造成的过高额外开销、提高异构众核系统的整体能效。

# 基于数据相关性的线程分组映射机制

为了能根据应用程序不同线程之间的数据相关性，结合异构众核架构特征，将线程合理的分配到相应的处理核上，减少由于存储访问延迟及共享资源竞争造成的过高额外开销，提高程序的整体计算效能，本章提出了一种基于数据相关性的线程分组映射机制DagTM（Data Affinity Grouping based Thread Mapping）。将程序本身固有的数据相关性和异构众核系统硬件架构特点相结合，将具有较好数据相关性的任务分配到同一个处理核的相邻硬件线程之上，使线程之间的数据相关性和运行平台的架构特征较好匹配，减少由于不合理的任务到硬件线程映射造成的过高额外通信和访存开销，实现应用线程到处理核的合理映射。

## 引言

根据应用程序不同线程之间的相关性特点，结合具体运行平台硬件架构特征，将线程合理的分配到相应的处理核上，使线程间的数据相关性和运行平台的物理架构特点较好的匹配，降低由于存储访问延迟、共享资源竞争、数据传输而造成的过高额外开销，提高程序的整体计算效能，是线程到处理核映射时要解决的一个关键问题[127-132]。已有的根据程序局部性特点进行计算任务到处理核映射的研究工作，主要可分为两大类：一类属于静态映射方法，一般通过对反映程序数据及存储访问相关性的信息进行剖分，对剖分信息进行离线分析，根据分析结果分配任务，不考虑具体的运行平台物理架构特点，直接根据经验值将线程映射到处理核上运行（如OpenMP应用程序中的Affinity属性参数都是根据经验值进行静态设置的），这种直接静态映射方法，不能客观反映不同线程在具体运行平台上运行时的数据相关性特点。在处理核数目有限的传统同构片上多核处理器系统上，运行单一程序时，直接静态映射方法能获得较好的计算性能，但在处理核数目众多且结构复杂的异构众核系统上，由于同时要考虑计算性能和系统的整体能耗，这种简单的静态线程映射方法不能满足异构众核系统高效能的计算需求；另一类属于动态映射方法，在程序执行过程中，动态检测程序运行状态，根据剖分到的程序运行时的状态信息、结合线程之间的数据局部性，通过将应用线程迁移到能较好适配当前线程数据局部性和处理器存储层次特点的处理核上运行，达到动态调整线程、实现线程到处理核合理映射、降低存储访问及通信开销的目的。但动态剖分程序局部性及动态迁移线程实现复杂，同时又会引入额外的运行时开销。需要在由动态迁移线程及动态调整线程所引入的额外运行时开销和所获得的性能收益之间做较好的平衡[150][151][152][153]。

针对应用线程到处理核合理映射的问题，如何在进行任务映射时，在尽量低的开销和成本的前提下，将程序本身固有的数据局部性和异构众核系统硬件架构特点相结合，将具有较好数据局部性的任务分配到同一个处理核的相邻硬件线程之上，使程序的数据局部性和运行平台的架构特点较好的匹配，减少由于不合理的任务到硬件线程映射造成的过高额外通信和存储访问开销，对充分利用异构众核系统的计算资源，提升应用程序的整体计算效能具有重要的现实意义。

本章针对以上问题，在同时考虑应用程序本身应用特点及运行平台硬件架构特点的基础上，提出了一种基于数据相关性的线程分组映射机制DagTM。通过计算不同线程数据重用距离来分析应用程序本身的数据相关性，根据不同应用线程内部和不同线程之间的数据相关性进行线程分组，通过线程组到处理核的绑定完成线程到处理核的映射，以实现应用线程数据局部性和处理核空间局部性的较好匹配，达到减少不同线程间的共享资源冲突，提高共享资源的利用率，降低数据传输开销，提高系统的整体计算效能的目的。

## DagTM线程分组映射机制

为了将应用线程数据相关性和具体的硬件架构相结合，实现应用线程到处理器不同处理核的合理映射，本文设计基于数据相关性的线程分组映射机制DagTM。

首先对应用程序进行逻辑任务（进程/线程）划分。根据运行平台所支持的最大硬件线程数，将应用程序划分为相应数量的应用线程。逻辑划分计算任务时，采用简单的均匀分配计算任务的方法，不考虑计算任务内部数据局部性特点，数据局部性在后续应用线程逻辑分组时考虑，以简单快速的完成计算任务到应用线程的逻辑划分。

完成计算任务到应用线程的逻辑划分后，采用上一章设计的线程间数据相关性度量方法，对不同线程的数据相关性进行度量。通过剖分不同线程的存储访问数据，计算线程的数据重用距离。根据数据重用距离信息，分析线程内部数据局部性特点及不同线程之间的数据相关性。该数据局部性和数据相关性和具体的运行平台无关，反映了程序固有的数据依赖关系。具体采用模式分类的方法将不同的线程归并为不同的局部性模式，根据线程不同的局部性模式，用相关性矩阵（Affinity矩阵）来度量不同线程间的数据相关性。

最后依据相关性矩阵对应用线程进行逻辑分组，并结合具体众核系统硬件架构特征，将不同线程分配到能使程序局部性和硬件架构特点较好匹配的处理核上，实现计算任务到处理核的合理映射。DagTM映射机制处理过程如图6-1所示。



图6-1 DagTM线程分组映射机制处理过程

## 线程逻辑分组

为了能将应用程序的不同应用线程合理的分配到众核处理器不同的处理核上，实现分配到同一处理核的应用线程之间有较好的数据相关性，而分配到不同的处理核的应用线程之间数据相关性尽可能的小。这样可以充分利用众核处理器同一处理核内不同硬件线程之间的共享cache，从而提高cache利用率；同时众核处理器不同处理核硬件线程之间由于分配了数据相关性小的应用线程，则可以在很大程度上降低处理核间的信息交互和数据传输，从而降低总通信量。最终可以通过提高众核处理器处理核内部共享cache的利率，降低处理核间信息交互及数据传输量，同时从众核处理器处理核内和处理核间两个层面综合考虑来提升应用程序的整体计算效能。

针对同时包含数量众多的处理核，不同处理核内部又同时支持共享内部cache的多硬件线程的众核处理器架构特征，为了实现应用线程数据相关性能和众核处理器架构特点较好的匹配，更好的发挥众核处理器的计算能力，本文根据不同线程间数据相关性及众核处理器单处理核所能支持的最大硬件线程数，通过将应用程序不同线程划分为不同的线程组，再将不同的线程组映射到不同的处理核的策略，实现线程到处理核的映射。

如何将不同应用线程划分为不同的线程组，同时保证处于同一组内的线程间有较好的数据共享性，本质上是一个组合优化问题。在进行线程分组时，即要考虑每个线程对和其具有竞争关系线程的存储访问影响，同时又要考虑其它线程对当前线程存储访问的影响。根据上一章线程间数据相关性度量方法，在得到反映应用程序不同线程之间数据相关性的Affinity图后，结合众核处理器处理核所能同时支持的最大硬件线程数，将应用线程分为K个线程组，保证每个组内线程之间有较好的数据共享性。本文在线程相关性图的基础上，将线程的逻辑分组问题抽象成一个图的分解问题，即将线程相关性图分解为满足上述要求的K棵子树，实现将具有高数据共享的线程划分到同一个线程组内，将具有较强数据访问争用关系的线程划分到不同的线程组内，从而降低因不同线程之间相互竞争共享计算资源引起的高存储访问冲突。具体通过设计Affinity子树生成算法实现满足上述目标的线程逻辑分组。

### Affinity子树生成算法

#### 算法设计思想

根据不同应用线程之间的数据相关性，将不同线程分成不同的线程组，保证分组后，每一个组内线程之间有较好的数据共享性，同时属于不同组的线程之间数据相关性尽可能的小。利用上一章所设计的线程间数据相关性度量方法，在得到反映不同线程数据相关性的Affinity图后，在Affinity图的其基础上，设计Affinity子树生成算法，通过将Affinity图分解成不同的子树来实现线程的分组。具体设计思想如下。

（1）设G=(V，E)是一个无向连通的加权图，即线程相关性图（Affinity图），其中顶点V表示不同线程的集合，边E代表线程间数据相关性的集合。图中每条边(Ti, Tj)∈E，且都有一个权值ω(Ti,Tj)，表示两线程间的共享数据量，如图6-2中（a）所示。图G的总顶点数为Nt，代表总的线程数；Np表示生成的每棵子树所要包含的节点数，即分组后每个线程组所包含的线程个数，该线程数对应具体运行平台处理器所支持的最大硬件线程数；K表示最终生成的子树个数。

（2）从图G上生成K棵子树，每棵子树用STk表示。生成的每棵子树最多包含Np个节点，同时保证当前生成的每棵子树边的权值之和最大，即满足如下约束条件：

i. . （6-1）

ii. . （6-2）

（3）通过借鉴Prim最小生成树算法，从无向连通图中生成满足上述约束条件的K棵子树。

#### 算法实现

根据以上算法设计思想，设计了具体的Affinity子树生成算法，从而实现将Affinity图分解成K棵子树，每棵子树对应一个线程组。从而实现将应用线程分成K个线程组。具体算法如算法6-1所示。

|  |
| --- |
| **算法6-1：**Affinity子树生成算法 |
| 1 **Input**: G=(V, E), Np.  2 **Output**: STk,(k=1, 2, …, n).  3 **Begin**  4 K=1;  5 Search\_max\_weight\_edge(G, E, STk);  //搜索最大权值的边，并将其加入到当前子树 STk中.  6 **while** (V !=NULL)  7 {  8 **do**  9 {  10 Search\_adjacent\_max\_weigh\_edge(G, Ei , STk );  //搜索连接当前边Ei的最大权值的边，并将其加入到当前子树STk中.  11 } **while** ( Vi < Np && V != NULL )  12 **end do**  13 Delete\_adjacent\_edge(G, STk );  // 删除没有包含在当前子树 STk中的邻接边.  14 Generate\_subtree( STk );  // 产生新的子树STk.  15 K++;  16 Search\_max\_weight\_remain\_edge(G, E-Ek-1 , STk );  //在图G剩余的边中搜索当前权值最大的边.  17 }  18 **end while**  19 **End**. |

算法具体实现过程如下：

输入：一个加权无向连通图G=(V，E)，其中顶点集合为V，边集合为E。子树中所包含的最大顶点数为Np.

1. 查找图G中最大权值的边，将该边及相应的顶点加入到当前子树的集合中；
2. 以最大权值的边为基础，搜索与该边顶点相连的其它边中权值最大的一条边，作为当前要生成子树的边，并将该边及其对应顶点加入到当前子树集合中；
3. 判断当前生成的子树已包含的顶点个数：

a）如果当前生成树中顶点个数小于Np，且图G中剩余顶点集合不为空，则继续查找和当前子树所包含顶点关联的其余顶点中最大权值的边。查到当前最大权值的边后，返回第ii步继续执行；

b）如果当前生成树中顶点个数等于Np，则表明已生成一棵完整的子树。删除没有包含在当前子树边集中，且和当前子树所含顶点有关联的边，将当前子树从图Ｇ中分离出来，生成一棵完整的子树STk，如图6-2中（d）所示；

1. 在图G剩余顶点构成的边集中查找当前权值最大的边，之后转到第ii步继续执行。如图6-2中（e）所示；
2. 如果图G中剩余顶点构成的边集为空，则整个子树生成过程结束，如图6-2中（g）所示.

输出：生成的不同子树STk，如图6-2中（h）所示。

### Affinity子树生成实例

下面用一个包含了八个线程的应用程序的Affinity图为例，来示意整个子树生成过程。本实例中处理器所支持的最大硬件线程数为4，最终完成子树生成后，将图中所包含的线程分成两个逻辑组。具体的子树生成过程实例如图6-2中（a）到（h）所示。

（a） （b） （c）

（d） （e） （f）

（g） （h）

图6-2 Affinity 子树生成过程

以上Affinity子树生成过程中，首先在图中查找到与线程T1共享数据量最大的线程T7，接着在图中剩余的所有边中，继续查找到当前与线程T1和T7数据共享量最大的线程T3，随后查找到线程T6。此时组内的线程数已经达到最大线程数的要求，随即将线程T1、T7、T3、T6从图中分离出去，生成单独的一棵子树，也即完成一个逻辑分组。接着在剩余的图节点中，继续按算法搜索另一组满足条件的线程，从而生成包含数据结点T2、T4、T5、T8的一棵新的子树。最终完成整个图的分解，生成两棵独立的子树，也即将整个线程分成独立的两个线程组。

### Affinity子树生成算法复杂性分析

Affinity子树生成算法的复杂性和整个Affinity图G=(V，E)的顶点数V及边数E有关。时间复杂度主要和图中所包含的边数有关，初始比较时要比较所有边的权值，后续比较中每次比较的次数会逐减。如果边数为n，则每次最多的比较次数依次为：n, n-1, n-2, …, 2, 1.故算法总的时间复杂度为О((n2/2+n/2)。为了能快速确定图中边是否存在，采用邻接矩阵存储加权无向连通的相关性图G，同时为了降低空间复杂度，根据无向连通图的邻接矩阵对角线上下部分对称的特性，只存储邻接矩阵对角线及其以上部分的信息，从而使占用的存储空间减少一半，算法的空间复杂度为О(V2/2)。

## 线程分组映射及实现

在完成了应用线程的逻辑分组以后，不同的应用线程被划分成了若干不同的逻辑组，属于同一逻辑组内的线程之间有较好的数据相关性、具有较好的数据共享特性，而属于不同逻辑组的线程之间的数据相关性差，线程之间数据共享量小。之后结合具体的硬件架构特点，按照一定的映射规则，将不同的线程组映射到具体的处理核上，以降低额外数据传输开销，减少数据访问冲突，提高程序的计算效能。

### 线程组到处理核的映射规则

结合具体硬件平台存储层次架构，将线程相关性子图中的线程映射到具体众核处理器不同处理核的不同硬件线程之上。借鉴文献[155][158]中的映射算法，以线程相关性子图和硬件架构存储层次图为输入，通过线程到处理核的静态绑定,实现线程到处理核硬件线程的映射。具体映射规则如下：

**规则6-1**：将同一线程组中的应用线程尽量分配到同一个处理核的不同硬件线程之上。如果该处理核线程已经全部分配，则将应用线程分配到相邻处理核的硬件线程之上。使具有数据共享的线程充分利用处理核内的共享cache，减少不必要的数据复制、数据传输，降低存储访问延迟，提高共享cache的利用率。

**规则6-2**：将不同线程组中的应用线程分配到不同处理核的硬件线程之上，使无共享数据的线程分散到具有独立cache空间的不同处理核上，避免在相同处理核上运行无数据相关性的应用线程时，在同一共享cache上复制大量不同处理数据引起的高传输延迟及共享cache冲突，以提高不同处理核共享cache空间利用率，减少额外的传输延迟。

### 线程分组映射实现

本文应用线程分组映射机制DagTM实现，分为线程内数据局部性检测、线程间数据相关性度量、线程分组、线程组到处理核映射及执行四个部分，具体实现框架如图6-3所示。



图6-3 DagTM线程分组映射实现框架

第一步检测不同应用线程的内部数据局部性，利用上一章的方法来计算数据重用距离，根据不同应用线程重用距离将不同线程归并为不同的局部性模式类，不同的局部性模式类反映了不同的数据局部性特点，属于同一模式类的线程具有相拟的数据局部性；第二步，在完成线程的数据局部性检测后，分析不同线程间的数据相关性，并对线程间的数据相关性进行量化。分别对属于不同模式类内线程之间的共享数据量进行统计后，得出能量化不同模式类内数据相关性的不同Affinity矩阵，之后将不同的Affinity 矩阵合并后，形成可量化整个应用程序不同线程数据相关性的总的Affinity矩阵及相就应的Affinity图；第三步，在上面得到的反映整个应用程序不同线程之间数据相关性的Affinity图的基础上，利用Affinity子树生成算法将Affinity图分解成不同的Affinity子树，每棵子树代表了一个独立的线程组，从而实现基于线程数据相关性的线程分组；第四步，结合具体的硬件架特征，根据设定的线程映射规则，通过静态绑定的方式，将不同线程组内的线程映射到众核处理器不同的处理核上执行。

### 线程分组映射实例

为了说明本文应用线程到具体处理核硬件线程完整的映射过程，用一个8线程并行应用程序到具有两级cache的Intel MIC架构众核处理器上的映射为例，说明线程具体的映射过程。

#### 构造Affinity矩阵及Affinity图

（1）首先计算出线程平均数据重用距离，标识出不同线程的局部性模式；以线程数据局部性模式为依据，将线程归并为不同的模式类；通过统计不同模式类内不同线程间相同的访问数据量，构造线程相关性矩阵（Affinity Matrix），如图6-4所示。

（2）将相关性矩阵（Affinity Matrix）转换成相应的相关性图（Affinity Graph），如图6-5所示。该图能直观的反映出各个线程之间的数据相关性。图的顶点代表不同的线程，边代表两线程之间的数据相关性，边的权值表示两线程间的数据共享量。

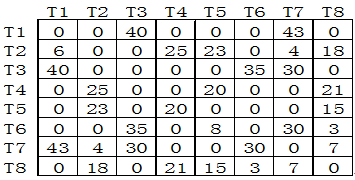
 

图6-4 线程Affinity 矩阵 图6-5 线程Affinity 图

#### 线程的逻辑分组

根据运行平台硬件架构存储层次图，结合不同线程之间的数据相关性，利用Affinity图子树生成算法，将线程分成不同的逻辑组，每个逻辑组对应不同的处理核。本实例以Intel MIC架构众核处理器为基础（具体存储架构如图6-6所示），实现线程逻辑分组，即在Affinity图的基础上生成不同的子树。最终生成的子树个数K由总的线程数Nt和所运行的处理器每个处理核所能支持的最大硬件线程数NP共同决定：K=Nt/Np。



图6-6 MIC 处理器存储层次图

本实例中，根据MIC众核处理器每个核所支持的最大4个硬件线程将8个应用线程分成两组。分组后生成的子树，要保证每个组内的线程之间具有较高数据共享，同时组间数据相关性尽量小。具体分组实现过程如图6-2所示，最终的线程逻辑分组结果如图6-2（h）所示。

#### 线程到处理核的映射

完成线程的逻辑分组后，根据设定的线程映射规则，将每个逻辑线程组内的线程分派到每个处理核不同的硬件线程之上，实现线程到处理核的静态映射，保证分配到同一处理核上的线程之间有较好的数据局部性，不同核线程间的数据相关性最小，从而减少共享存储冲突，提高共享cache利用率，降低数据存储访问延迟，提高程序运行效率。本实例中完成应用线程到处理核硬件线程映射后的结果如图6-7（c）。



（a） OpenMP内置的Compact映射模式



（b） OpenMP内置的Scatter映射模式



（c） DagTM映射方法

图6-7 不同线程到处理核硬件线程映射方式比较

图6-7直观的比较了传统的OpenMP并行运行时库支持的应用线程到处理核映射方法和本文基于数据相关性的线程分组映射DagTM不同的映射结果。从图中可以看出，OpenMP所支持的Compact映射模式只考虑充分利用每一个处理核资源，不考虑线程之间的数据局部性关系，会将具有高数据共享性的线程映射到不同的处理核上，从而引起过高的额外存储访问。如图6-7（a）所示，具有高共享量的线程T1、T3、T6、T7被分派到两个不同的处理核上，因同时要在处理核core1和core2的片内cache上存储相同共享数据拷贝，增加了额外的数据存储开销。图6-7（b）中OpenMP所支持的Scatter映射方式主要考虑负载平衡，将所有线程均匀的分布在不同的处理核上，而不考虑线程之间的数据局部性。同样会引入过高的额外数据存储访问，同时还存在处理核不能充分利用，系统能耗过高的问题。图6-7（c）中，本文DagTM映射方法，在映射时首先考虑不同线程之间的数据局部性，根据处理核硬件线程将具有不同数据局部性的线程分为不同的逻辑组，然后将不同的逻辑组映射到不同的处理核，从而可利用线程之间的数据局部性来提高线程之间的数据共享，减少额外的数据访问及数据传输，同时在映射时也会充分利用处理核资源，以提高每个处理核的资源利用率，降低系统的整体能耗。

## 实验评测及分析

### 实验环境及方法

#### 实验环境

采用PARSEC[32]基准测试集，具体用于测试的benchmark程序及其特征如第4章表4-2所示。

实验平台采用MIC异构众核系统：由2路8核E5-2670 CPU和一块Xeon Phi 7110P MIC卡组成，CPU主存和加速卡之间采用通道带宽为x16的PCI-E总线进行数据传输，最高传输速度可达到16GB/s。CPU和MIC协处理器具体性能参数如表4-3所示。操作系统：Red Hat Enterprise Linux Server release 6.3. MIC开发编译环境：Intel parallel\_studio\_xe\_2013\_update3\_intel64软件包，gcc 4.4.6。程序运行时的性能指标通过PAPI\_5.4.1[144][145]性能测试工具获取。

#### 评测方法

对不同基准测试程序，分别用本文线程映射方法DagTM，OpenMP内置的多线程映射方法Compact、Scatter，理想情况下最优的映射方法（Oracle）及文献[156]提出的kMAF方法从以下三方面进行测试对比。

（1）计算性能评测：分别测试以上五种映射方法，在不同基准测试程序下的执行时间及最后一级cache misses率，并以OS默认映射方法（ﬁrst-touch policy）的执行时间及最后一级cache misses率为基准，归一化后比较不同映射情况下的性能差异。

（2）能效评测：分别测试以上五种映射方法在不同基准测试程序下，在运行平台最大支持线程数情况下的能耗，以OS默认映射方法下能耗为其准，归一化后比较不同映射方法下的能耗差异。

（3）额外开销评测：测试本文DagTM映射方法在不同的基准测试程序下，线程分组所需时间占整个程序执行时间的比值，评测本文DagTM映射方法在程序执行前，线程分组所引入的额外时间开销。

### 测试结果及分析

#### 计算性能测试结果

图6-8显示了不同基准测试程序，在本文分组映射DagTM及其他四种映射方法下，和OS默认映射方法相比较的计算性能相对提升率，用不同映射方法下基准测试程序相对于OS默认映射方法（ﬁrst-touch policy）下的计算时间的相对减少率来度量。从图6-8可以看出，本文DagTM平均计算性能提升了近14%，最优映射方法Oracle计算性能提升了17%，Compact、Scatter、kMAF三种映射方法计算性能分别提升了2%、 3%、 12%。 DagTM计算性能达到了Oracle映射的82.4%，优于其它三种映射方法。同时从图中可以看出，Compact和Scatter两种方法在所有的测试程序下性能提升都低于DagTM和kMAF，有些情况下甚至低于OS默认的映射方法（如Blackscholes、x264、Dedup、Facesim），主要原因是Compact和Scatter完全没有考虑不同线程之间的数据相关性，会引起不同线程之间共享资源冲突及额外的数据传输开销。同时本文DagTM在有些情况下计算性能不如kMAF（如Streamcluster、Raytrace、Freqmine、Dedup），因kMAF主要根据程序运行时不同线程之间的数据局部性来动态调整线程到处理核的映射，对运行行为在执行过程中变化较大的程序，若动态调整线程映射获得的计算性能收益大于引入的额外开销时，会有明显的计算性能提升。而对于在运行过程中计算行为变化不大的程序，由于引入的额外动态映射开销，会抵销部分性能收益，所以性能提升不如本文DagTM映射方法。

图6-8 不同映射方法相对于OS默认映射方法的性能提升

#### L2 cache缺失率测试结果

图6-9显示了不同映射方法相对于OS默认方法下，最后一级cache misses率的变化对比。从图6-9中可以看出，Compact、Scatter、Oracle、DagTM、kMAF五种映射方法下L2 Cache缺失率相对于OS默认方法，Compact和Scatter映射方法不但没有降低L2 Cache缺失率，反而比OS默认方法分别增加了3%和2%。主要原因是OpenMP支持的这两种方法，没有具体考虑应用程序不同线程之间本身的数据相关性，只关注如何充分利用处理核硬件线程和处理核间负载平衡问题，所以出现最后一级cache缺失率略微增加的情况。Oracle、DagTM、kMAF三种映射方法下L2 Cache缺失率平均分别降低了12%、8%、11%。总体上，kMAF方法的Cache缺失率的降低程度最接近最优的映射方法Oracle，优于其他三种方法。这主要因为kMAF是根据程序运行时数据局部性动态调整线程到处理核的映射，可以实时反映cache的利用情况，从而降低cache misses率。但由于要动态检测后获取线程的实时数据局部性，会引入一定的额外运行时开销，这些额外的开销会抵消部分由于高的cache利用率而获得的性能收益，从而影响整体的计算性能。所以如图6-8所示，kMAF整个计算性能并不优于本文DagTM映射方法。

图6-9 不同映射方法相对于OS默认映射方法的L2 cache misses 率降低比例

#### 能耗测试结果

能耗也是众核系统考虑的主要问题，一方面通过充分利用计算资源、减少额外的数据传输来降低能耗；另一方面可以通过DVFS技术动态调整处理核的工作状态，采用物理的方法来降低能耗。本文主要通过充分利用计算资源，减少计算时间来相对的降低系统的能耗。

如图6-10所示，本文DagTM方法平均能耗相对于OS默认方法降低了10.3%，Compact、Scatter、kMAF三种方法相对能耗分别降低了2.3%，3.2%，8.5%，最优映射方法Oracle的相对能耗降低了12.4%。由于DagTM方法在线程映射时考虑了数据相关性，可以减少数据访问冲突，提高共享资源的利用率，降低数据传输开销，减少程序的运行时间，相应的减少了系统的整体能耗。

图6-10 不同映射方法相对于OS默认映射方法的能耗降低比例

#### 预处理开销测试结果

图6-11显示了不同基准测试程序下，利用本文DagTM映射方法时，执行线程分组所用时间占整个程序运行时间的比例。

图6-11 DagTM预处理开销

本文DagTM线程分组引入的额外时间平均占总执行时间的11%。由于Compact和Scatter映射方法不考虑程序本身的局部性特点，采用直接线程映射的方法，在程序运行前及运行过程中都不会引入额外的开销；Oracle方法主要通过穷举对比方法获得最佳的映射方式，只作为一种理想的映射标准，引入的开销最大，不能作为实际可用的映射方法；kMAF在程序运行过程中，根据程序运行状态动态调整线程到处理核的映射来提高程序性能，会引入额外的运行时开销，这种开销会直接影响程序的执行性能，要在性能和开销之间做很好的平衡；本文DagTM映射方法在程序运行前结合程序本身的局部性和运行平台架构特点将线程分组后直接实现映射，在程序运行过程中不会引入额外的开销，同时又在可接受的预处理开销的情况下实现线程分组，可以克服Scatter及Compact不考虑程序本身局部性特点的不足，在不引入额外运行时开销的情况下可以获得和kMAF相当的性能提升。

## 本章小结

本章针对众核系统下线程到处理核的映射问题，以应用程序不同线程数据相关性特点为基础，结合具体众核系统硬件架构特点，将应用线程合理的映射到众核处理器不同的处理核上，达到减少不同线程间的共享资源冲突，提高共享资源的利用率，降低数据传输开销，提高系统的整体计算效能的目的。通过计算数据重用距离来判定线程内部的数据局部性；用数据相关性矩阵度量线程间的数据相关性；根据线程相关性图，利用最小生成树实现不同线程的逻辑分组；最后通过线程组到处理核的绑定完成线程到处理核的映射。实验评测表明，本文线程分组映射，在提升计算性能和降低能耗方面是有效的，可以根据应用程序不同线程之间的数据相关性特点，将不同线程合理的映射到具体众核处理器的不同处理核上，在不引入额外运行时开销的基础上，提高系统的计算性能，降低系统的整体能耗。

# 结论与展望

随着多核、众核技术的迅速发展，出现了具有不同硬件架构、不同计算模式、高并行计算能力的异构众核处理器，为构建高性能计算机系统提供了更高性能的硬件支持。基于异构众核处理器搭建异构超级计算机已经成为高性能计算领域一个重要的发展趋势，这也将成为构建E级计算机系统的主要架构趋势。由于异构系统不同计算部件的异构性，进一步加剧了并行应用程序的开发难度，使异构众核系统的高计算潜能的发挥面临严峻挑战。

提升系统性能，改善系统可编程性是新型异构系统亟待解决的关键问题。如何将应用程序的计算任务合理的映射到异构众核系统处理器的具体处理核上，使应用程序运行特点和硬件架构特征较好的匹配，对异构众核系统来说至关重要，将直接影响能否充分利用异构众核系统的计算资源，能否有效提升并行应用程序的计算效能。

## 本文工作总结

在新型异构众核系统下，利用协处理器对应用程序进行加速过程中面临的具体问题是：如何充分发挥协处理器的高效处理能力；如何使主处理器和协处理器较好的协作；如何提高应用程序的计算性能、降低系统能耗。

本文主要针对新型异构众核系统高计算能力不能充分发挥的问题，结合新型异构众核计算机系统层次化存储架构特点，通过研究线程到处理核资源的合理映射机制来提升异构众核系统的计算效能，从计算任务到处理单元的映射层面来优化异构众核系统的计算效能。

在对相关研究工作充分调研分析的基础上，主要从合理的线程数设置，数据局部性及线程之间数据相关性方面，研究如何将应用程序的特点和具体运行平台硬件架构特征相结合，实现应用程序并行特征和运行平台计算资源合理匹配，以高效执行更高级别的细粒度并行任务，从而充分利用异构众核系统的计算资源来提升计算效能。具体对异构众核系统线程映射时的线程数预测、线程数动态调整、数据局部性及线程数据相关性度量、基于线程数据相关性的线程到处理核的映射进行了深入的研究，主要研究内容和所取得创新成果如下：

（1）针对异构众核系统下如何为多线程应用程序设置合理线程数的问题，研究多核多线程运行时不同线程之间相互作用对程序性能的影响，分析出现多样性对应关系的原因，抽象影响因素。在充分考虑多线程执行时不同线程之间的相互作用对程序性能造成额外开销的基础上，通过扩展Amdahl定律，结合回归分析理论，利用最小二乘法构造最优线程数预测模型TCPM。通过假设检验、预测精度及额外开销测试，验证TCPM预测模型的有效性。验证结果表明利用TCPM预测模型可以在保证预测精度的情况下，实现简单、准确、有效的最优线程数预测。

（2）针对应用程序在执行过程中，由于运行行为发生阶段性变化而对计算资源有不同需求的问题，在分析已有的动态调整线程数映射机制存在额外开销高、通用性差、依赖于特定硬件、不能实时调节异构众核系统下运行程序并行性问题的基础上，提出了面向异构众核系统，基于程序阶段性变化的线程数动态调整机制DPTM。DPTM以线程数预测模型为基础，通过动态检测程序阶段性变化，实时检测程序运行状态信息，根据收集到的反映程序运行状态硬件性能计数器信息，利用所提出的数学模型直接计算出最优线程数；根据计算出的最优线程数对程序执行过程中的线程数进行动态调整，使程序在执行过程中并行性达到最佳状态；结合操作系统的计算资源管理机制，使闲置的处理核及硬件线程处于低功耗或关闭状态，从而在充分利用众核系统计算资源提升程序计算性能的同时，降低系统的整体能耗，提升异构众核系统的整体效能。通过在Intel MIC异构众核系统上实现DPTM映射框架，并对PARSEC基准测试程序进行测试。实验测试结果表明，DPTM线程映射机制在充分利用众核系统计算资源提升程序计算性能的同时，降低系统的整体能耗，提升了异构众核系统的整体效能。

（3）针对如何准确的定量分析反映应用程序不同线程之间数据局部性关系的数据相关性，将应用程序线程数据相关性和异构众核系统物理架构特点相结合，实现应用线程数据局部性和应用平台架构空间局部性较好的匹配，减少存储访问延迟及共享资源竞争造成的过高额外开销，在研究了不同线程之间的数据局部性及存储访问特点的基础上，提出了基于数据重用距离的线程数据相关性度量方法RDAQ。具体通过剖分不同线程存储访问数据信息，设计线程数据重用距离计算方法，计算不同线程的数据重用距离；在数据重用距离的基础上，通过定义反映不同数据相关性的局部性模式，将不同的线程归并为不同的模式类；在不同模式类的基础上通过比较不同线程间数据共享量，用相关性矩阵来度量不同线程间的数据相关性。实验评测结果表明，RDAQ可以实现对应用程序不同线程之间数据相关性的简单、准确、高效的量化。

（4）针对如何根据不同线程之间数据相关性特点，结合异构众核架构特征将线程合理分配到相应处理核上，减少存储访问延迟及由于共享资源竞争而造成的过高额外通信开销，实现高能效的线程映射，提高程序的整体计算效能，提出了基于数据相关性的线程分组映射机制DagTM。根据应用程序线程之间的数据相关性，对应用线程进行逻辑分组，结合异构众核系统硬件架构特点，通过线程组到处理核静态绑定实现线程到处理核的映射。具体实现线程逻辑分组时，在线程相关性图的基础上，将线程的逻辑分组问题抽象成一个图的分解问题，通过设计数据相关性子树生成算法，将线程相关性图分解为满足一定要求的K棵子树，实现将具有高数据共享的线程划分到同一个线程组内，将具有较强数据访问争用关系的线程划分到不同的线程组内，从而降低因不同线程之间相互竞争共享计算资源引起的高存储访问开销。最后通过线程组到处理核的绑定，将应用线程合理的映射到众核处理器不同的处理核上高效执行。基准测试程序实验结果表明，DagTM在不产生额外运行时开销的基础上，与传统映射机制和已有的方法相比，在提升计算性能的同时降低了系统的整体能耗。

## 下一步工作展望

为了适应异构众核系统表现出的处理器核数越来越多、存储层次越来越复杂、应用程序多样性变化、及对整体计算效能要求越来越高的发展趋势[163][164][165][166]，在后续工作中，针对异构众核系统能效优化问题，进一步从以下方面深入研究，以更好的适应异构众核系统及应用发展变化的需要。

（1）针对应用程序多样性变化的不规则的多线程多应用程序情况，在本文提出的基于程序阶段性变化的线程数动态调整机制DPTM的基础上，研究更高效的，面向新型异构众核系统线程映射机制，不但要能对不同应用程序的线程数进行动态调整，而且要考虑如何将多个应用程序合理的调度到不同处理核上，将具有不同执行特点的计算任分配到不同处理能力的处理核上，使异构众核系统不同处理器高效协同计算，充分利用闲置的处理核资源，进一步提升异构众核系统的整体计算效能。

（2）在本文提出的基于线程数据相关性的线程映射机制DagTM的基础上，进一步研究动态和静态相结合的应用线程到处理核的映射机制。同时考虑程序在执行过程中线程相关性动态变化的因素，根据线程数据相关性动态变化设计相应的线程迁移策略，通过静态和动态相结合的方式进一步提升应用程序的计算性能。

（3）为了高效的利用异构众核处理器计算资源，要进一步深入研究如何根据运行任务及当前负载情况，结合DVFS技术，实时的对异构众核系统不同处理核的运行状态进行控制的动态计算资源管理机制，进一步降低异构众核系统整体能耗。

（4）为了进一步提高异构众核系统整体效能，同时要考虑影响系统计算效能的其它因素，如众核处理器互联网络拓扑、处理核间的通信机制等，将这此因素和线程映射机制相结合，同时整合所提的线程映射技术，将其逐渐集成在异构众核系统编程框架原型系统中，从而全面提升异构众核系统的整体效能。

# 致 谢

光阴飞逝，在交大已整整度过了五个春秋，自己的博士研究生涯也已走近尾声。回顾在交大度过的每个日日夜夜，历历在目，感慨万千！五年的博士生涯，给了我放松思考，提升自我的机会，也给了我让自己思想和能力积累、沉淀的机会。五年的博士研究生活的历练，自己不仅在知识积累和理论水平上有了一定的提高，同时开阔了自己的学术视野，磨砺了自己的意志，培养了自己独立的科研能力。这些将会是我人生最宝贵的财富，也将使自己在今后的生活和科研中终身受益！

至此论文完成之际，谨向所有给予我关心、鼓励、帮助、指导的各位师长、同学、朋友表示最诚挚的感谢！

首先向我的导师董小社教授表示最诚挚的感谢和最崇高的敬意！在我整个博士研究生期间，董老师从生活、科研各方面都给了我悉心的指导和无私的帮助，同时也为自己提供了很多学习和锻炼的机会，从各个方面培养了自己独立从事科研的能力。在董老师无私的关怀和指导下，使我得以顺利完成论文的选题、研究和撰写等工作。董老师严谨的治学态度，敏锐的学术洞察力，高瞻远瞩的学术视野，勤勉的科研作风，一丝不苟的工作态度，和蔼可亲、平易近人的高尚人格，积极向上的生活态度，诲人不倦的崇高师德，潜移默化的影响着我，使我深受教益。在此，对尊敬的恩师给予我的辛勤培养表示衷心的感谢!

感谢伍卫国教授、张兴军教授、朱正东高工在科研、学习、生活方面给予的关心和帮助！

感谢陈衡老师在修改论文过程中给予的帮助和指导！

感谢师兄王寅峰副教授在科研和生活方面给予的关心和帮助！

感谢教研室郑豪、孙健、李亮、白秀秀、王龙翔、刘强、吴树森、周墨颂、党博超等博士在生活、学习和科研过程中给予的关心与帮助！

感谢教研室其他师弟师妹们，他们是张保、刘超、刘袁、辛龙、颜康、陈宝可、王强、张译之、魏晓林、魏洪昌、黄锴、崔红琳等，因为你们使我的博士研究生活更丰富多彩。

感谢林强博士，张学军博士，对我的关心、帮助、鼓励和支持！

特别感谢我的父母，是你们一直默默地支持我走过了这漫长、富有挑战的求学之路，我所取得的任何成绩都是你们无私奉献的结果！同时感谢我的姐姐和妹妹，正因为有你们的支持、关心和鼓励，我才能安心的进行科研，顺利完成学业。你们默默的支持、鼓励、关心也是我不懈奋斗、不断进步的直接动力！

最后，谨向百忙之中为本论文审查、评审、预答辩、答辩及讨论评定提出宝贵意见及付出辛勤劳动的老师致以最真挚的谢意！

# 参考文献

1. Chen W. The demands and challenges of exascale computing: an interview with Zuoning Chen[J]. National Science Review, 2016, 3(1): 64-67.
2. Brodtkorb AR, Dyken C, Hagen TR, et al. State-of-the-art in heterogeneous computing[J]. SCIENTIFIC PROGRAMMING,2010,18(1):1-33.
3. Owens JD, Luebke D, Govindaraju N, et al. A survey of general purpose computation on graphics hardware[J] Computer Graphics Forum, 2007, 26(1): 80-113.
4. Brodtkorb AR, Hagen TR, Sætra ML. Graphics processing unit (GPU) programming strategies and trends in GPU computing[J]. Journal of Parallel and Distributed Computing, 2013, 73(1): 4-13.
5. 王海峰，陈庆奎. 图形处理器通用计算关键技术研究综述[J].计算机学报，2013，36(4):757-772.
6. 刘颖, 吕方, 王蕾,等. 异构并行编程模型研究与进展[J]. 软件学报, 2014(7):1459-1475.
7. Lee S, Min SJ, Eigenmann R. OpenMP to GPGPU: a compiler framework for automatic translation and optimization[J]. ACM Sigplan Notices, 2009, 44(4): 101-110.
8. Han TD, Abdelrahman TS. hiCUDA: High-level GPGPU programming[J]. IEEE Transactions on Parallel and Distributed Systems, 2011, 22(1): 78-90.
9. Baskaran MM, Ramanujam J, Sadayappan P. Automatic C-to-CUDA code generation for affine programs[J]. Compiler Construction, 2010, 6011: 244-263.
10. Linderman MD, Collins JD, Wang H, et al. Merge: a programming model for heterogeneous multi-core systems[C]//Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems. ACM, 2008: 287-296.
11. Dubach C, Cheng P, Rabbah R, et al. Compiling a high-level language for GPUs:(via language support for architectures and compilers)[C]//Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation. ACM, 2012:1-12.
12. Liu W, Lewis B, Zhou X, et al. A balanced programming model for emerging heterogeneous multicore systems[C]//Proceedings of the 2nd USENIX Conference on Hot Topics in Parallelism. USENIX Association, 2010:3-3.
13. Gelado I, Stone J E, Cabezas J, et al. An asymmetric distributed shared memory model for heterogeneous parallel systems[C]//Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems(ASPLOS). ACM, 2010: 347-358.
14. Jang B, Schaa D, Mistry P, et al. Exploiting memory access patterns to improve memory performance in data-parallel architectures[J]. IEEE Transactions on Parallel and Distributed Systems, 2011, 22(1): 105-118.
15. Ryoo S, Rodrigues CI, Baghsorkhi SS, et al. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA[C]//Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. ACM, 2008:73-82.
16. Baskaran MM, Bondhugula U, Krishnamoorthy S, et al. A compiler framework for optimization of affine loop nests for GPGPUs[C]//Proceedings of the 22nd Annual International Conference on Supercomputing. ACM, 2008:225-234.
17. Jang B, Schaa D, Mistry P, et al. Exploiting memory access patterns to improve memory performance in data-parallel architectures[J]. IEEE Transactions on Parallel and Distributed Systems, 2011, 22(1):105-11.
18. Sundaram N, Raghunathan A, Chakradhar S T. A framework for efficient and scalable execution of domain-specific templates on GPUs[C]//Proceedings of the IEEE International Symposium on Parallel & Distributed Processing. IEEE, 2009:1-12.
19. Liu Y, Zhang EZ, Shen X. A cross-input adaptive framework for GPU program optimizations[C]// Proceedings of the International Symposium on Parallel & Distributed Processing. IEEE, 2009:1-10.
20. Yang Y, Xiang P, Mantor M, et al. CPU-assisted GPGPU on fused CPU-GPU architectures[C]// Proceedings of the 18th International Symposium on High-Performance Computer Architecture. IEEE Computer Society, 2012:1-12.
21. Jeffers J, Reinders J. Intel Xeon Phi Coprocessor High Performance Programming[M]. Newnes, 2013.
22. Lee J, Lakshminarayana N B, Kim H, et al. Many-thread aware prefetching mechanisms for gpgpu applications[C]//Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture. IEEE, 2010:213-224.
23. Yang Y, Xiang P, Kong J, et al. A GPGPU compiler for memory optimization and parallelism management[C]//Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation. ACM, 2010:86-97.
24. Yang Y, Xiang P, Mantor M, et al. Shared memory multiplexing: a novel way to improve GPGPU throughput[C]//Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques. ACM, 2012:283-292.
25. 张保, 董小社, 白秀秀, 等. CPU-GPU 系统中基于剖分的全局性能优化方法[J]. 西安交通大学学报, 2012, 46(2): 17-23.
26. Wang PH, Collins JD, Chinya GN, et al. EXOCHI: architecture and programming environment for a heterogeneous multi-core multithreaded system[C]//Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation. ACM, 2007:156-166.
27. Luk CK, Hong S, Kim H. Qilin: exploiting parallelism on heterogeneous multi-processors with adaptive mapping[C]//Proceedings of the the 42nd Annual IEEE/ACM International Symposium on Microarchitecture. IEEE, 2009:45-55.
28. Jablin TB, Prabhu P, Jablin JA, et al. Automatic CPU-GPU communication management and optimization[C]//Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation. ACM, 2011:142-151.
29. Lee J, Kim J, Seo S, et al. An OpenCL framework for heterogeneous multicores with local memory[C]//Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques(PACT). ACM, 2010:193-204.
30. Reyes R, López-Rodríguez I, Fumero JJ, et al. accULL: An OpenACC implementation with CUDA and OpenCL support[J]. Euro-Par Parallel Processing Lecture Notes in Computer Science, 2012, 7484:871-882.
31. Wienke S, Springer P, Terboven C, et al. OpenACC-first experiences with real-world applications[J]. Euro-Par Parallel Processing Lecture Notes in Computer Science, 2012, 7484: 859-870.
32. Chen C, Yang C, Tang T, et al. OpenACC to Intel offload: automatic translation and optimization[J]. Computer Engineering and Technology Communications in Computer and Information Science, 2013, 396:111-120.
33. Top 500 supercomputer sites, November 2014, available at:http://www.top500.org/.
34. Kothapalli K, Banerjee DS, Narayanan P J, et al. CPU and/or GPU: Revisiting the GPU Vs. CPU Myth[J]. Eprint Arxiv, 2013.

|  |
| --- |
| 1. Daga M, Aji AM, Feng W. On the efficacy of a fused cpu+gpu processor (or apu) for parallel computing[C]//Proceedings of the Symposium on Application Accelerators in High-Performance Computing. IEEE, 2011:141-149. 2. Liu X, Smelyanskiy M, Chow E, et al. Efficient sparse matrix-vector multiplication on x86-based many-core processors[C]//Proceedings of the International Conference on Supercomputing. ACM, 2013:273-282. 3. Saini S. An early performance evaluation of many integrated core architecture based SGI rackable computing system[C]//Proceedings of the High Performance Computing, Networking, Storage and Analysis (SC). IEEE, 2013:1-12. 4. Bakhoda A, Yuan GL, Fung WWL, et al. Analyzing CUDA workloads using a detailed GPU simulator[C]//Proceedings of the International Symposium on Performance Analysis of Systems and Software. IEEE, 2009:163-174. 5. Stratton JA, Rodrigues C, Sung IJ, et al. Parboil: a revised benchmark suite for scientific and commercial throughput computing[J]. Center for Reliable and High-Performance Computing, 2012. 6. Buchty R, Heuveline V, Karl W, et al. A survey on hardware-aware and heterogeneous computing on multicore processors and accelerators[J]. Concurrency & Computation Practice & Experience, 2012, 24(7):663–675. 7. Che S, Boyer M, Meng J, et al. Rodinia: A benchmark suite for heterogeneous computing. In Proceedings of IISWC, pages 44-54, 2009 8. Etinski M, Corbalan J, Labarta J, et al. Understanding the future of energy-performance trade-off via DVFS in HPC environments[J]. Journal of Parallel and Distributed Computing, 2012, 72(4):579-590. 9. 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. |

1. Pusukuri KK, Gupta R, Bhuyan LN. Thread reinforcer: Dynamically determining number of threads via os level monitoring[C]//Proceedings of the IEEE International Symposium on Workload Characterization(IISWC). 2011:116-125.
2. Sasaki H, Tanimoto T, Inoue K, et al. Scalability-based manycore partitioning[C]//Proceedings of the ACM 21st International Conference on Parallel Architectures and Compilation Techniques(PACT). 2012:107-116.
3. Liu C, Thanarungroj P, Gaudiot JL. How many cores do we need to run a parallel workload: A test drive of the Intel SCC platform?[J]. J. Parallel Distrib. Comput. 2014,74(7):2582-2595.
4. Bienia C. Benchmarking modern multiprocessors. PhD thesis, Princeton University, 2011.
5. 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.
6. Bienia C, Kumar S, Li K. PARSEC vs. SPLASH-2: A quantitative comparison of two multithreaded benchmark suites on chip-multiprocessors[C]//Proceedings of the IEEE International Symposium on Workload Characterization(IISWC). 2008: 47-56.
7. Bienia C, Li K. Parsec 2.0: A new benchmark suite for chip-multiprocessors[C]//Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation. 2011.
8. Black F, Scholes M. The Pricing of Options and Corporate Liabilities.[J]. Journal of Political Economy, 1973, 81(3):637-659.
9. Banerjee P. Parallel algorithms for VLSI computer-aided design[J]. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.
10. Grahne G, Zhu J. Efficiently Using Prefix-trees in Mining Frequent Itemsets[C]//Proceedings of the third IEEE International Conference on Frequent Itemset Mining Implementations(FIMI). 2003:90.
11. Qureshi MK, Patt YN. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches[C]//Proceedings of the 39th Annual International Symposium on Microarchitecture. IEEE/ACM, 2006: 423-432.
12. Eyerman S, Eeckhout L. System-level performance metrics for multiprogram workloads[J]. IEEE MICRO, 2008,28(3):42-53.
13. Hill MD, Marty MR. Amdahl's Law in the Multicore Era[J]. Computer, 2008,41(7):33-38.
14. Sun XH, Chen Y. Reevaluating Amdahl's law in the multicore era[J]. Journal of Parallel and Distributed Computing. 2010,70(2):183-188.
15. Yao E, Bao Y, Tan G, et al. Extending Amdahl's law in the multicore era[J]. ACM SIGMETRICS Performance Evaluation Review,2009,37(2):24-26.
16. Amdahl GM. Validity of the single processor approach to achieving large scale computing capabilities[C]//Proceedings of the spring joint computer conference. ACM, 1967: 483-485.
17. Gustafson JL. Reevaluating Amdahl's law[J]. Communications of the ACM, 1988, 31(5): 532-533.
18. Sun XH, Ni LM. Scalable problems and memory-bounded speedup[J]. Journal of Parallel and Distributed Computing, 1993, 19(1): 27-37.
19. Kai H, Xu Z. Scalable Parallel Computing: Technology, Architecture, Programming[J]. Scalable Computing Practice & Experience, 1998, 2(1).
20. Karan S, Matthew CM, Sally AM, et al. Comparing Scalability Prediction Strategies on an SMP of CMPs[C]// Proceeding of the International Euro-Par Conference on Parallel Processing. Springer-Verlag, 2010:45-49.
21. 王之元. 并行计算可扩展性分析与优化[D]. 国防科学技术大学, 2011.
22. Schmidl D, Cramer T, Wienke S, et al. Assessing the performance of OpenMP programs on the intel xeon phi[C]//Proceedings of the 19th International Conference on Euro-Par 2013 Parallel Processing. Springer Berlin Heidelberg, 2013:547-558.
23. Moore RW, Childers BR. Building and using application utility models to dynamically choose thread counts[J]. Journal of Supercomputing, 2014, 68(3):1184-1213.
24. Emani MK, Wang Z, O'Boyle MFP. Smart, adaptive mapping of parallelism in the presence of external workload[C]//Proceedings of the IEEE/Acm International Symposium on Code Generation and Optimization. IEEE, 2013:1-10.
25. Sherwood T, Sair S, Calder B. Phase tracking and prediction[C]//Proceedings of the 28th International Symposium on Computer Architecture(ISCA). IEEE, 2003.
26. Curtis-Maury M, Blagojevic F, Antonopoulos CD, et al. Prediction-based power-performance adaptation of multithreaded scientific codes[J]. IEEE Transaction on Parallel and Distributed Systems. 2008 (10):1396-1410.
27. Lee J, Wu H, Ravichandran M, et al. Thread tailor: dynamically weaving threads together for efficient, adaptive parallel applications[C]// Proceedings of the 37th Annual International Symposium on Computer architecture(ISCA). ACM, 2010:270-279.
28. Heirman W, Carlson TE, Craeynest KV, et al. Undersubscribed threading on clustered cache architectures[C]//Proceedings of the 20th International Symposium on High Performance Computer Architecture(HPCA). Orlando, FL, USA: IEEE, 2014:678-689.
29. Heirman W, Carlson TE, Craeynest KV, et al. Automatic SMT threading for OpenMP applications on the Intel Xeon Phi co-processor[C]//Proceedings of the 4th International Workshop on Runtime and Operating Systems for Supercomputers. Munich, Germany:ACM, 2014:7.
30. Kandemir M, Yemliha T, Muralidhara SP, et al. Cache topology aware computation mapping for multicores[C]// Proceedings of the ACM SIGPLAN conference on Programming language design and implementation(PLDI). Toronto, Ontario, Canada:ACM, 2010:74-85.
31. Cochran R, Hankendi C, Coskun A, et al. Pack & Cap: adaptive DVFS and thread packing under power caps[C]//Porceedings ot the IEEE/ACM 44th Annual International Symposium on Microarchitecture (MICRO). New York, NY, USA: IEEE/ACM, 2011:175-185.
32. Ge R, Feng X, Song S, et al. Powerpack: energy profiling and analysis of high-performance systems and applications[J]. IEEE Transactions on Parallel and Distributed Systems, 2010, 21: 658–671.
33. Li L, Martinez J. Dynamic power-performance adaptation of parallel computation on chip multiprocessors[C]//Proceedings of the 12th International Symposium on High Performance Computer Architecture(HPCA).IEEE, 2006: 77-87.
34. Intel OpenMP runtime library. Available at http://www.openmprtl.org/.
35. James J, Reinders J. Intel’s Xeon Phi Coprocessor High-Performance Programming[M]. USA:Elsevier Inc.2013.
36. Newburn CJ, Dmitriev S, Narayanaswamy R, et al. Offload Compiler Runtime for the Intel Xeon Phi Coprocessor[C]//Proceedings of the IEEE 27th Parallel and Distributed Processing Symposium Workshops & PhD Forum(IPDPSW). Cambridge, MA, USA: IEEE, 2013:1231-1225.
37. Huang M, Lai C, Shi X, et al. Study of parallel programming models on computer clusters with Intel MIC coprocessors[J]. International Journal of High Performance Computing Applications, 2015.
38. 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.
39. 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.
40. 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.
41. Weaver VM. Linux perf\_event features and overhead[C]//Proceedings of the 2nd International Workshop on Performance Analysis of Workload Optimized Systems, FastPath. 2013: 80.
42. Zhang Z, Chang J M. A Cool Scheduler for Multi-Core Systems Exploiting Program Phases[J]. IEEE Transactions on Computers, 2014, 63(63):1061-1073.
43. Fang Z, Li J, Zhang W, et al. Improving dynamic prediction accuracy through multi-level phase analysis[J]. Acm Sigplan Notices, 2012, 47(5):89-98.
44. Zhang W, Li J, Li Y, et al. Multilevel Phase Analysis[J]. Acm Transactions on Embedded Computing Systems(TECS), 2015, 14(2):1-29.
45. Sobue K, Tsumura T, Matsuo H. An Efficient Thread Recombining at Program Phase Changes[C]// Proceedings ot the 3rd International Conference on NETWORKING and Computing. IEEE, 2012:316-320.
46. Sondag T, Rajan H. Phase-based tuning for better utilization of performance-asymmetric multicore processors[C]// Proceedings ot the IEEE/ACM International Symposium on Code Generation and Optimization. IEEE, 2011:11-20.
47. Zhang W, Liu F, Fan R. Cache matching: thread scheduling to maximize data reuse[C]// Proceedings of the International Symposium on High Performance Computer Architecture(HPCA). Tampa, Florida: 2014:1-8.
48. Nicolau A, Kejariwal A. How Many Threads to Spawn during Program Multithreading?[J] Languages and Compilers for Parallel Computing. Springer Berlin Heidelberg, 2011:166-183.
49. Castro M, Góes LF W, Méhaut JF. Adaptive thread mapping strategies for transactional memory applications[J]. Journal of Parallel and Distributed Computing, 2014, 74(9):2845-2859.
50. Grewe D, Wang Z, O'Boyle M F P. A workload-aware mapping approach for data-parallel programs[C]//Proceedings ot the International Conference on High PERFORMANCE and Embedded Architectures and Compilers. ACM, 2011:117-126.
51. Korch M, Rauber T, Scholtes C. Memory-Intensive Applications on a Many-Core Processor[C]// Proceedings of the 2011 IEEE International Conference on High Performance Computing and Communications. IEEE Computer Society, 2011:126-134.
52. Tallent NR, Mellor-Crummey JM. Effective Performance Measurement and Analysis of Multithreaded Applications[J]. Acm Sigplan Notices, 2009, 44(4):229-240.
53. Curtis-Maury M, Shah A, Blagojevic F, et al. Prediction models for multi-dimensional power-performance optimization on many cores[C]//Proceedings of the International Conference on Parallel Architectures and Compilation Techniques(PACT). ACM, 2008:250-259.
54. Becchi M, Crowley P. Dynamic thread assignment on heterogeneous multiprocessor architectures[C]//Proceedings of the 3rd conference on Computing frontiers. New York, NY, USA:ACM, 2006:29-40.
55. Saripalli V, Sun G, Mishra A, et al. Exploiting Heterogeneity for Energy Efficiency in Chip Multiprocessors[J]. IEEE Transaction on Emerging & Selected Topics in Circuits & Systems, 2011, 1(2):109-119.
56. Sarikaya R, Isci C, Buyuktosunoglu A. Runtime Application Behavior Prediction Using a Statistical Metric Model[J]. IEEE Transactions on Computers, 2013, 62(3):575-588.
57. Das R, Ausavarungnirun R, Mutlu O, et al. Application-to-core mapping policies to reduce memory interference in multi-core systems[C]//Proceedings of the 19th International Symposium on High Performance Computer Architecture(HPCA). ACM, 2012:107-118.
58. Nishtala R, Mossé D, Petrucci V. Energy-aware thread co-location in heterogeneous multicore processors[C]// Proceedings of the International Conference on Embedded Software. Montreal, Quebec, Canada: ACM, 2013:1-9.
59. Drebes A, Heydemann K, Drach N, et al. Topology-Aware and Dependence-Aware Scheduling and Memory Allocation for Task-Parallel Languages[J]. Acm Transactions on Architecture & Code Optimization, 2014, 11(3):1-25.
60. Andrade, Diego, Fraguela, et al. Accurate prediction of the behavior of multithreaded applications in shared caches[J]. Parallel Computing, 2013, 39(1):36–57.
61. Shen X, Zhong Y, Ding C. Predicting locality phases for dynamic memory optimization[J]. Journal of Parallel & Distributed Computing, 2007, 67(7):783-796.
62. Feliu J, Sahuquillo J, Petit S, et al. Understanding Cache Hierarchy Contention in CMPs to Improve Job Scheduling[C]//Proceedings of the International Symposium on Parallel and Distributed Processing. IEEE, 2012:508-519.
63. Wang W, Dey T, Mars J, et al. Performance analysis of thread mappings with a holistic view of the hardware resources[C]//Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). IEEE, 2012:156-167.
64. Li D, De Supinski BR, Schulz M, et al. Strategies for Energy Efficient Resource Management of Hybrid Programming Models[J]. IEEE Transactions on Parallel & Distributed Systems, 2013, 24(24):144-157.
65. Singh K, Bhadauria M, McKee SA. Real time power estimation and thread scheduling via performance counters[J]. ACM SIGARCH Computer Architecture News, 2009, 37(2): 46-55.
66. Cho S, Melhem RG. On the Interplay of Parallelization, Program Performance, and Energy Consumption[J]. IEEE Transactions on Parallel & Distributed Systems, 2009, 21(3):342-353.
67. Ge R, Feng X, Burtscher M, et al. Performance and Energy Modeling for Cooperative Hybrid Computing[C]//Proceedings of the International Conference on Networking, Architecture, and Storage. IEEE, 2014:232-241.
68. Winter JA, Albonesi DH, Shoemaker CA. Scalable thread scheduling and global power management for heterogeneous many-core architectures[C]//Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT). ACM, 2010:29-40.
69. Sasaki H, Imamura S, Inoue K. Coordinated power-performance optimization in manycores[C]// Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT), IEEE, 2013: 51-61.
70. Pusukuri KK, Gupta R, Bhuyan LN. ADAPT: A framework for coscheduling multithreaded programs[J]. ACM Transactions on Architecture and Code Optimization, 2013, 9(4):167-169.
71. 王磊, 刘道福, 陈云霁等. 片上多核处理器共享资源分配与调度策略研究综述[J]. 计算机研究与发展, 2013, 50(10):2212-2227.
72. 林闯, 田源, 姚敏. 绿色网络和绿色评价:节能机制、模型和评价[J]. 计算机学报, 2011, 34(4):593-612.
73. 吴俊杰. 层次存储的访问分析与优化方法研究——重用性、相似性与亲和性[D]. 国防科学技术大学, 2009.
74. Weinberg J, Mccracken MO, Strohmaier E, et al. Quantifying locality in the memory access patterns of hpc applications[C]//Proceedings of the ACM/IEEE International Conference on Supercomputing. IEEE: 2005:50-50.
75. Zhang C, Ding C, Ogihara M, et al. A hierarchical model of data locality[M]// ACM SIGPLAN Notices. ACM, 2006:16-29.
76. Gu X, Christopher I, Bai T, et al. A component model of spatial locality[C]//Proceedings of the International Symposium on Memory Management. ACM, 2009:99-108.
77. Jiang Y, Tian K, Shen X. Combining locality analysis with online proactive job co-scheduling in chip multiprocessors[C]//Proceedings of the 5th International Conference on High Performance Embedded Architectures and Compilers. Pisa,Italy: 2010:201-215.
78. 唐 滔 杨学军 林一松. 基于迭代序的流程序局部性分析和优化[J]. 计算机研究与发展, 2012, 49(6): 1363-1375.
79. Cruz EHM, Diener M, Alves MAZ, et al. Optimizing Memory Locality Using a Locality-Aware Page Table[C]// Proceedings of the International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD). 2014:198-205.
80. Gupta S, Zhou HY. Spatial Locality-Aware Cache Partitioning for Effective Cache Sharing[C]//Proceedings of the 44th International Conference on Parallel Processing (ICPP). Beijing, China:2015: 150-159.
81. Xiang XY, Ding C, Luo H, et al. HOTL: a higher order theory of locality[C]//Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). ACM, 2013:343-356.
82. Ding C, Zhong Y. Predicting whole-program locality with reuse distance analysis[C]//Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation(PLDI). San Diego, CA: ACM, 2003: 245-257.
83. Zhong Y, Shen XP, Ding C. Program Locality Analysis Using Reuse Distance[J]. ACM Transactions on Programming Languages and Systems,2009,31(20):341-351.
84. Jiang YL, Zhang EZ, Tian K, et al. Is reuse distance applicable to data locality analysis on chip multiprocessors?[C]//Proceedings of the 19th Joint European Conference on Theory and Practice of Software. Paphos, Cyprus, 2010:264-282.
85. Niu QP, Dinan J, Lu QD, et al. PARDA: A Fast Parallel Reuse Distance Analysis Algorithm[C]//Proceedings of the 26th International Parallel and Distributed Processing Symposium (IPDPS). IEEE, 2012:1284-1294.
86. Wu MJ, Yeung D. Identifying optimal multicore cache hierarchies for loop-based parallel programs via reuse distance analysis[C]//Proceedings of the SIGPLAN Workshop on Memory Systems Performance and Correctness. ACM, 2012:2-11.
87. Wu MJ, Yeung D. Efficient Reuse Distance Analysis of Multicore Scaling for Loop-Based Parallel Programs. ACM Transactions on Computer Systems (TOCS), 2013, 31(1):1-37.
88. Nugteren G, Van den Braak GJ, Corporaal H, et al. A detailed GPU cache model based on reuse distance theory[C]//Proceedings of the 20th International Symposium on High Performance Computer Architecture (HPCA). IEEE, 2014: 37-38.
89. Moore RW, Childers BR. Automatic generation of program affinity policies using machine learning[C]//Proceedings of the 22nd International Conference on Compiler Construction. Rome, Italy, 2013:184-203.
90. Schuff DL, Kulkarni M, Pai VS. Accelerating multicore reuse distance analysis with sampling and parallelization[C]// Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques(PACT). Vienna, Austria: ACM, 2010:53-64.
91. Fauzia N, Elango V, Ravishankar M, et al. Beyond reuse distance analysis: Dynamic analysis for characterization of data locality potential[J]. Acm Transactions on Architecture & Code Optimization(TACO), 2013, 10(4):95-95.
92. Pericas M, Taura K, Matsuoka S. Scalable analysis of multicore data reuse and sharing[C]// Proceedings of the 28th International Conference on Supercomputing. Munich, Germany: ACM, 2014:353-362.
93. Mattson RL, Gecsei J, Slutz DR, et al. Evaluation techniques for storage hierarchies[J]. IBM Systems Journal, 1970, 9(2):78-117.
94. Zhong Y, Dropsho SG, Ding C. Miss Rate Prediction across All Program Inputs[C]//Proceedings of the International Conference on Parallel Architectures and Compilation Techniques(PACT). IEEE Computer Society, 2003:79-79.
95. Shen X, Zhong Y, Ding C. Locality phase prediction[J]. Acm Sigops Operating Systems Review, 2004, 38(5):165-176.
96. Zhong Y, Orlovich M, Shen X, et al. Array Regrouping and Structure Splitting Using Whole-Program Reference Affinity[J]. Acm Sigplan Notices, 2004, 39(6):255-266.
97. Cascaval C, Duesterwald E, Sweeney PF, et al. Multiple Page Size Modeling and Optimization[C]//Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques(PACT). IEEE Computer Society, 2005:339 - 349.
98. Jiang S, Zhang X. Making LRU friendly to weak locality workloads: a novel replacement algorithm to improve buffer cache performance[J]. IEEE Transactions on Computers(TOC), 2005, 54(8):939-952.
99. Bach M, Charney M, Cohn R, et al. Analyzing Parallel Programs with Pin[J]. IEEE Computer, 2010, 43(3):34-41.
100. Luk CK, Cohn R, Muth R, et al. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation[C]//Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation(PLDI). ACM, 2005:190-200.
101. Dey T, Wang W, Davidson JW, et al. Characterizing multi-threaded applications based on shared-resource contention[C]//Proceedings of the International Symposium on PERFORMANCE Analysis of Systems and Software. IEEE, 2011:76-86.
102. Saez JC, Fedorova A, Koufaty D, et al. Leveraging core specialization via os scheduling to improve performance on asymmetric multicore systems[J]. Acm Transactions on Computer Systems, 2012, 30(2):121-135.
103. Moore RW, Childers BR. Program affinity performance models for performance and utilization[C]// Proceedings of the conference on Design, Automation & Test in Europe. Dresden, Germany:2014:1-4.
104. Zhong Q, Guan X, Huang T, et al. Affinity-aware DMA buffer management for reducing off-chip memory access[C]//Proceedings of the ACM Symposium on Applied Computing. ACM, 2012:1588-1593.
105. Ren B, Ravi N, Yang Y, et al. Automating and optimizing data transfers for many-core coprocessors[C]// Proceedings of the Acm International Conference on Supercomputing. Munich, Germany: ACM, 2014:12.
106. Vydyanathan N, Krishnamoorthy S, Sabin GM, et al. An integrated approach to locality-conscious processor allocation and scheduling of mixed-parallel applications[J]. IEEE Transactions on Parallel and Distributed Systems(TPDS), 2009, 20(8): 1158-1172.
107. Rodrigues R, Koren I, Kundu S. Does the sharing of execution units improve performance/power of multicores?[J]. Acm Transactions on Embedded Computing Systems, 2015, 14(1):1-24.
108. Pusukuri KK, Vengerov D, Fedorova A, et al. FACT: a framework for adaptive contention-aware thread migrations[C]//Proceedings of the International Conference on Computing Frontiers. ACM, 2011:127.
109. Wang Z, Tournavitis G, Franke B, et al. Integrating profile-driven parallelism detection and machine-learning-based mapping[J]. Acm Transactions on Architecture & Code Optimization(TACO), 2009, 11(1):730-753.
110. Schuff DL, Parsons BS, Pai VS. Multicore-aware reuse distance analysis[C]//Proceedings of the International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW). IEEE, 2010:1-8.
111. Zhang Y, Kandemir M, Yemliha T. Studying inter-core data reuse in multicores[C]//Proceedings of the ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems. ACM, 2011:25-36.
112. Muralidhara SP, Kandemir M, Kislal O. Reuse distance based performance modeling and workload mapping[C]// Proceedings of the 9th conference on Computing Frontiers. Cagliari, Italy:2012.
113. Tam D, Azimi R, Stumm M. Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors[J]. Acm Sigops Operating Systems Review, 2007, 41(3):47-58.
114. Diener M, Cruz EMH, Navaux POA, et al. kMAF: automatic kernel-level management of thread and data affinity[C]//Proceedings of the 23rd International Conference on Parallel Architectures and Compilation Techniques(PACT). Edmonton, AB, Canada: ACM, 2014.
115. Ding W, Zhang Y, Kandemir M, et al. Locality-aware mapping and scheduling for multicores[C]// Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). IEEE, 2013:1-12.
116. Fu W, Chen T, Wang C, et al. Optimizing memory access traffic via runtime thread migration for on-chip distributed memory systems[J]. Journal of Supercomputing, 2014, 69(3):1491-1516.
117. Eduardo HM. Cruz MD, Marco AZ, et al. Dynamic thread mapping of shared memory applications by exploiting cache coherence protocols[J]. Journal of Parallel and Distributed Computing, 2014, 74(3):2215-2228.
118. Terpstra D, Jagode H, You H, et al. Collecting Performance Data with PAPI-C[J]. Tools for High Performance Computing. Springer Berlin Heidelberg, 2009:157-173.
119. Weaver VM, Johnson M, Kasichayanula K, et al. Measuring Energy and Power with PAPI[C]//Proceedings of the International Conference on Parallel Processing Workshops. IEEE, 2012:262-268.
120. Majo Z, Gross TR. A library for portable and composable data locality optimizations for NUMA systems[C]//Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), New York, NY, USA: ACM, 2015:227-238.
121. Tillenius M, Larsson E, Badia R M, et al. Resource-Aware Task Scheduling[J]. Acm Transactions on Embedded Computing Systems, 2015, 14(1):1-25.
122. Wu J. Energy-efficient scheduling of real-time tasks with shared resources[J]. Future Generation Computer Systems, 2016, 56(C):179-191.
123. Cheng SW, Chang CW, Chen J, et al. Many-Core Real-Time Task Scheduling with Scratchpad Memory[J]. IEEE Transactions on Parallel and Distributed Systems, 2016:1-1.

# 攻读学位期间取得的研究成果

**学术论文：**

[1] Tao Ju, Weiguo Wu, Hen Chen, Zhengdong Zhu Xiaoshe Dong. Thread Count Prediction Model:Dynamically Adjusting Threads for Heterogeneous Many-Core Systems[C]// Proceedings of the 21st IEEE International Conference on Parallel and Distributed Systems (ICPADS 2015), Melbourne, Australia, December 14-17, 2015:459-464.(EI：20161802336953). (CCF认定的C类国际会议)

[2] 巨涛，朱正东，董小社．异构众核系统及其编程模型与性能优化技术研究综述[J].电子学报，2015,43(1):111-119. (EI: 20151500737465).

[3] Tao Ju, Zhengdong Zhu, Yinfeng Wang, Liang Li, Xiaoshe Dong. Thread Mapping and Parallel Optimization for MIC Heterogeneous Parallel Systems[C]// Proceedings of the 14th International Conference on Algorithms and Architectures for Parallel Processing(ICA3PP2014), Dalian, China, August 24-27, 2014:300-311.(EI: 20143518106623, ISTP:000345508000023).（CCF 认定的C类国际会议）

[4] Tao Ju, Xiaoshe Dong, Endong Wang, Liang Li, Zhengdong Zhu. Parallel Optimization Strategies for MIC Heterogeneous Parallel Systems[C]// Proceedings of the 6th International Symposium on Parallel Architecture, Algorithms and Programming(PAAP2014), Beijing, China, July 13-15, 2014: pp204-209.(EI: 201443118264).

[5] 巨涛，张兴军，陈衡，董小社. 面向众核系统的线程分组映射方法[J]. 西安交通大学学报.(已录用，2016年10月出刊).

[6] Tao Ju, Xiaoshe Dong, Heng Chen,Xingjun Zhang. Data Affinity Based Threads Grouping Mapping for Many-Core Systems[J]. Energies（SCI源刊，西安交大认定的国际知名期刊）(Under review).

[7] Tao Ju, Xiaoshe Dong, Heng Chen, Xingjun Zhang. Energy-Efficient Thread Mapping for Heterogeneous Many-Core Systems via Dynamically Adjusting the Thread Count[J]. Concurrency and Computation:Practice and Experience.（SCI源刊，西安交大认定的国际知名期刊） (Under review).

**专利：**

[8] 董小社、巨涛、张兴军、陈衡等. 一种用于众核系统的数据相关性线程分组映射方法, 2016/3/30, 中华人民共和国国家知识产权局. 申请号：201610188032.8.

[9] 董小社、巨涛、朱正东等. 一种用于异构众核系统下的应用程序线程数动态调整方法，2015/2/28,中华人民共和国国家知识产权局. 申请号: 201510090555.4. 公开号：CN104657219A.

**参与的主要科研项目：**

[10] 国家自然科学基金面上项目“面向异构众核系统的统一编程框架研究”，（编号：61572394），2016/01~2019/12.

[11] 国家自然科学基金面上项目 “新型异构多核系统的渐近拟合优化技术研究”，（编号：61173039），2011/01~2015/12.

[12] 国家863计划项目“新型多核／众核处理器编程与运行支撑环境”，（编号：2012AA010904），2012/7-2015/12.

[13] 国家重点研发计划高性能重点专项“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”子课题“面向E级计算机的大型流体机械并行计算软件研制” （课题编号：2016YFB0200902），2016/7~2020/12.项目申报工作.

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

(本声明的版权归西安交通大学所有，未经许可，任何单位及任何个人不得擅自使用)