第46卷第3期 中国科学技术大学学报 Vol. 46, No. 3
2016年3月 JOURNAL OF UNIVERSITY OF SCIENCE AND TECHNOLOGY OF CHINA Mar. 2016

文章编号:0253-2778(2016)3-0200-08

面向 X86 多核处理器的数据流程序任务调度与缓存优化

唐九飞[1],李 鹤[2],于俊清[1,2]

(1. 华中科技大学网络与计算中心, 武汉 430074; 2. 华中科技大学计算机科学与技术学院, 武汉 430074)

摘要:数据流编程作为一种编程模式被广泛地应用于多核处理器系统,其多核处理器的并行调度和对主存的访问延迟对程序的性能有很大的影响. 为此,结合 X86 多核处理器的特点,提出一种数据流程序的任务调度与缓存优化方法. 任务调度优化首先在预处理阶段提高目标程序的局部性和并行粒度;然后利用数据流程序的数据并行、任务并行和流水并行优化核间负载均衡,并构造软件流水调度. 缓存优化针对目标系统的层次性缓存结构特征,通过消除缓存伪共享减少多核并行运行时相互间的干扰,根据逻辑线程间的通信分布实现逻辑线程到处理器核的映射. 以 COStream 作为数据流编程语言,输出经过编译优化后的目标代码. 实验选取数字媒体领域典型的算法进行测试,测试结果表明,编译优化后的测试程序基本达到线性加速比,验证了编译系统的有效性.

关键词:X86 多核处理器; 数据流; 任务调度; 缓存优化

中图分类号:TP311 **文献标识码**:A doi:10.3969/j.issn.0253-2778.2016.03.004

引用格式: TANG Jiufei, LI He, YU Junqing. Research on stream program task scheduling and cache optimization for X86 multi-core processor[J]. Journal of University of Science and Technology of China, 2016,46(3):200-207.

唐九飞,李 鹤,于俊清. 面向 X86 多核处理器的数据流程序任务调度与缓存优化[J]. 中国科学技术大学学报,2016,46(3):200-207.

Research on stream program task scheduling and cache optimization for X86 multi-core processor

TANG Jiufei[1], LI He[2], YU Junqing[1,2]

(1. *Center of Network and Computation, Huazhong University of Science and Technology, Wuhan* 430074, *China*
2. *School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan* 430074, *China*)

Abstract: Stream programming as a kind of programming paradigm widely used in multi-core systems, in which the parallel scheduling and access latency of main memory has a great influence on the performance of the program. To solve this problem, a task scheduling and cache optimization method was proposed for the stream program by combining the characteristics of X86 multi-core processors. To improve the parallel scheduling granularity and locality of the target program, expanded scheduling in the pre-processing phase was used firstly. Then the compiler exploitsed the data parallelism and task parallelism to keep load balancing and construct the corresponding software pipeline scheduling. According to cache hierarchies of the target system, the interference were reduced among parallel scheduling cores were reduced by

收稿日期:2015-08-27;**修回日期**:2015-12-01

基金项目:国家高科技发展(863)计划(2012AA010902), 中国高等学校博士学科点专项基金(20120142110089)资助.

作者简介:唐九飞,男,1970 年生,硕士/工程师. 研究方向:并行处理和编译优化. E-mail: tangjiufei@hust.edu.cn

通讯作者:于俊清, 博士/教授. E-mail: yjqing@hust.edu.cn

eliminating cache false sharing, and the optimized mapping between logic threads and physical cores in line with the inter-thread communication distribution were implemented. The compiler took a COStream dataflow program as input and outputs the optimized program. The common algorithms in media processing technology were chosen as the test programs for the experiment. The experimental result shows that the optimized test programs achieve linear speedup, which indicates the effectiveness of the compiling optimization system.

Key words: X86 multi-core; data flow; task scheduling; cache optimization

0　引言

数据流编程模型使得面向特定领域的开发人员摆脱了繁琐的系统优化工作，在多/众核平台上的应用日益广泛，目前比较成熟的数据流编程语言包括StreamIt[1]、Brook[2]、Cg[3]. 在数据流编程模型中，目标应用表示为一个数据流图，图中的顶点表示计算任务，顶点间的有向边表示计算任务间的通信. 采用数据流编程模型，开发人员只要实现领域相关的算法，而系统优化工作主要由第三方开发的编译优化系统完成[4].

随着多核处理器平台日趋复杂，数据流程序的并行编译优化工作变得更加困难，其问题主要包括多处理器核的并行调度和对主存的访问延迟. 一方面，为了避免浪费处理器核的计算能力，并行调度算法必须考虑各并行控制流之间的同步开销、通信开销以及并行粒度是否适用于目标系统，并确保各处理器核间的任务负载均衡[5]；另一方面，对于多核共同竞争主存访问带宽的多核平台，缓存优化的有效性在很大程度上决定了程序的性能，但是层次性缓存结构和多核并行相互间的影响增加了缓存优化工作的难度和复杂性[6].

本文基于 X86 多核处理器体系结构以及数据流程序的特点，提出了一种数据流程序的任务调度与缓存优化方法，并采用 COStream[7] 数据流编程语言进行验证优化方法的有效性. COStream 采用同步数据流计算模型对 C 语言进行了扩展. 相比 StreamIt 等数据流编程语言，COStream 从 IBM SPL 借鉴了一些语法结构以提高可编程性和代码复用. 另外，为了支持更通用的计算模式，COStream 支持多输入多输出计算单元.

本文的主要贡献有以下 3 点：①采用扩大调度策略提高目标程序的局部性和并行粒度；②设计并实现了数据流程序的任务划分与调度优化；③针对 X86 多核处理器的层次性缓存结构特点，实现了多核核间缓存优化.

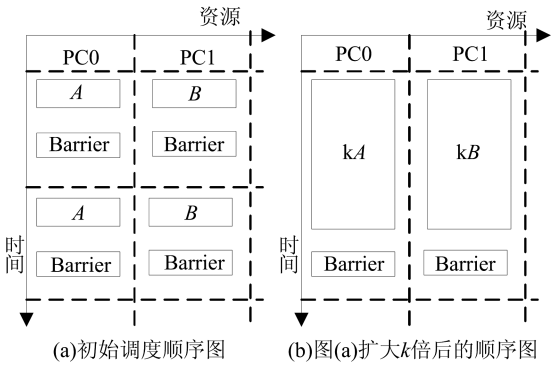
1　数据流程序的任务划分与调度

与传统程序并行化不同，数据流程序并行调度的目标不是最小化目标程序的完成时间，而是为了最大化程序的吞吐率. 除了任务并行性，数据流程序中还存在数据并行性和流水并行性. 国内外相关研究工作表明软件流水调度能够充分利用这三种并行性实现高吞吐率的有效方法[8]. 软件流水调度策略的性能主要取决于两个方面：①并行线程的任务负载均衡；②多核间的同步和通信开销. 本文首先通过数据流图的预处理减少同步和通信开销对程序的影响；然后通过迭代水平分裂算法提高多核核间任务负载，并构造软件流水线调度.

1.1　数据流图的预处理

1.1.1　扩大调度

扩大调度指的是以相同的扩大因子，成倍扩大每个计算单元稳态调度阶段的执行次数. 图 1 给出了使用扩大调度前后的对比图，其中 k 为扩大因子. 在图 1(a)中，每个稳态调度周期处理器核 1 和 2 分别对计算单元 A 和 B 调度一次；在图 1(b)中两个计算单元分别被调度 k 次. 以下介绍扩大因子对目标程序性能的影响以及其确定方法.



(a)初始调度顺序图　(b)图(a)扩大k倍后的顺序图

图 1　扩大调度前后对比

FIg. 1　The comparison before and after the expansion of scheduling

首先,扩大调度可以减少同步开销对程序性能的影响.假定在理想情况下各处理器的计算开销为W_{Per},核间同步开销为W_{sync},则扩大调度k倍使得程序的计算同步比由$W_{\mathrm{Per}}/W_{\mathrm{sync}}$提高为$k\times W_{\mathrm{Per}}/W_{\mathrm{sync}}$.

本文采用下式确定扩大因子k的下限.

$$k\geqslant\frac{W_{\mathrm{Total}}\times\mathrm{ratio}}{N\times W_{\mathrm{sync}}}\tag{1}$$

式中,W_{Total}为目标程序稳态调度总的计算开销,ratio为计算通信比的阈值,N为处理器核的数量.

其次,扩大调度提高了目标程序的局部性,从而减少了处理器核访存延迟.如图1所示,通信缓存区的大小与扩大因子k的大小成正比.为了在提高程序局部性的同时避免由于缓存溢出降低性能,用下式限定扩大因子的上限.

$$k\leqslant\frac{N\times\mathrm{Size}\times Usage}{\mathrm{Com}_{\mathrm{Total}}}\tag{2}$$

式中,Size为目标系统$L1$数据缓存容量的大小,Usage为目标系统$L1$数据缓存中缓冲区所占比例的阈值,N为处理器核的数量,$\mathrm{Com}_{\mathrm{Total}}$为计算单元间总的通信数据量.公式(1)和(2)确定了扩大调度因子的选择区间.为了实现简单,选择在此区间内的最大的且为2的幂次的整数作为最终的扩大调度因子.

1.1.2 相邻计算单元的融合

相邻计算单元的融合是指将两个独立且在数据流图中相邻的计算单元融合为一个计算单元.相邻计算单元融合操作保证了两个计算单元不会被分配到不同的处理器核上调度,这能够减少通信开销对程序的影响;但是过度的融合操作会减少数据流图的计算单元,可能导致任务划分阶段负载的不均衡.

相邻计算单元融合算法的关键在于如何判断是否某对相邻计算单元进行融合.首先,被融合的计算单元必须是单输入单输出的,这是由于过多的通信边将导致复制分裂后计算单元间过多的通信开销;其次,因为存在状态计算单元不能进行数据并行调度,所以不对相应状态计算单元进行融合操作.对于满足上述两个条件的相邻节点,根据两者通信量之和以及融合后的计算量,当通信计算比大于某个阈值时,将其合并为一个计算单元.

1.2 软件流水线调度

1.2.1 基于迭代水平分裂的任务划分算法

在软件流水调度模型中,各线程首先独立完成分配到其上的计算任务;然后进行一次核间同步,程序的吞吐率由计算任务量最大的线程决定.根据预估计的各计算单元的工作量,可以将数据流程序的任务划分问题转换为图划分问题.K路图分配算法(K-way Mesh Partition, KMP[9])是当前比较常用的图划分算法,以负载均衡为目标同时最小化通信开销.

KMP算法并没有利用计算单元的数据并行性进行优化.本文在KMP算法的基础上,采用迭代水平分裂算法,改进划分负载均衡.

迭代水平分裂算法流程如图2所示,主要分为三个步骤.

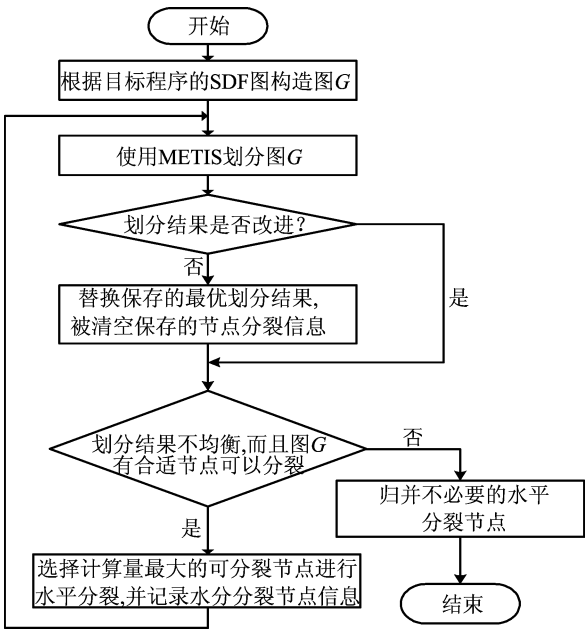


图2 基于迭代水平分裂的任务划分算法

Fig. 2 Task partitioning algorithm based on iterative level splitting

(Ⅰ)构造初始划分.根据计算单元的计算量和稳态调度次数确定对应图顶点的权值,计算单元间的通信量确定图中对应边的权值,然后采用KMP算法获得初始划分结果.

(Ⅱ)迭代分裂改进划分结果.选择计算量最大的可分裂计算单元进行水平分裂以改进划分结果,重复执行该过程,直到负载均衡或者不能进一步分裂为止.

(Ⅲ)撤销不必要的分裂操作.根据第(Ⅱ)步求得的划分结果,将由同一个计算单元分裂但却被划分到同一子集上的不同计算单元进行合并操作.

1.2.2 阶段赋值

经过任务划分后,目标程序的计算单元被分配

到处理器核上执行，即在空间维度上指定了计算任务的分配. 为了消除各划分子集中相邻计算单元的耦合性，需要确定各计算单元被流水调度的阶段号，即在时间维度上指定计算单元的调度.

算法 1.1 用伪代码描述了计算单元的阶段复制算法，其以目标程序对应的数据流图和任务划分结果作为输入. 对于数据流程序，其数据流图的有向边表示计算单元间的数据传输. 算法首先构造数据流图中计算单元节点的拓扑排序以满足计算单元数据读写规则；然后以拓扑序列依次访问计算单元，根据数据读写节点是否被划分到同一个处理器核上确定计算单元的流水调度阶段值.

算法 1.1　阶段赋值算法

输入：SDF 图 G(V, E)，图 G 计算单元到核的映射 mapActor2Partition

输出：图 G 计算单元到阶段号的映射 mapActor2Stage

```
1 topologyOrderOfActors =topologyTravSDF();
2 FOR all actor u IN topologyOrderOfActors DO
3   int maxStageValue =0; int stageValue;
4   FOR all actor v which is a parent of u DO
5     IF (mapActor2Partition[v] ! = mapActor2
       Partition[u]) THEN
6       stageValue = mapActor2Stage[v] + 1;
7     ELSE
8       stageValue = mapActor2Stage[v];
9     ENDIF
10    IF (stageValue > maxStageValue) THEN
11      maxStageValue = stageValue;
12    ENDIF
13  END FOR
14 mapActor2Stage[u]=maxStageValue;
15 END FOR
```

2　数据流程序多核核间的缓存优化

随着处理器芯片处理器核数的增加，为了减少处理器核的访存延迟，缓存系统由传统的单层结构转换为层次性结构[9].

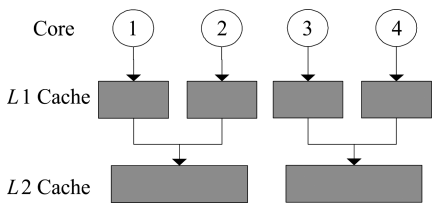


图 3　多核处理器及其层次性缓存结构实例

Fig. 3　Multi core processor and its hierarchical cache architecture

图 3 给出了一个多核处理器的实例，该处理器上有 4 个处理器核以及两级层次性缓存结构，其中第一层缓存处理器核私有，每个 $L1$ Cache 的存储总量为 16kB，处理器访问延迟约为 3 个时钟周期；第二层缓存 $L2$ Cache 由两个处理器核共享，每个 $L2$ Cache 的存储总量为 4MB，处理器访问延迟约为 12 个时钟周期.

资源共享和竞争一直是并行计算领域不可回避的主题. 对于层次性缓存系统，一方面，并行线程竞争有限的缓存空间导致程序执行时间的不可预见性；另一方面，为了保证并行线程对主存数据访问的一致性，底层系统必须增加额外的开销[10]. 针对这些问题，根据数据流程序的特点，本文设计并实现了缓存优化方法.

2.1　多核缓存伪共享的消除

当映射到同一个缓存行的不同变量被多个处理器核并发访问时，多个处理器核更新同一缓存行，这就会导致缓存伪共享的发生[11]. 图 4 给出了数据流程序在并行调度时发生缓存伪共享的实例，其中通信的计算单元 A 和 B 被分配到不同处理器核上调度. 在一稳态调度周期中，数据生成者 A 需要修改缓冲区，数据消费者读取缓冲区，而缓存区被映射到同一缓存行上.

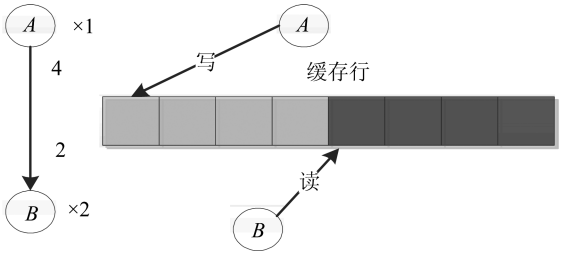


图 4　稳态调度阶段核间缓存伪共享

Fig. 4　Pseudo sharing of kernel cache in steady state scheduling

为了消除稳态调度阶段发生的核间缓存伪共享，对于消费者计算单元的两种数据访问模式，分别采取相应的优化策略：

(Ⅰ)滑动窗口机制是指消费者计算单元每次计算访问的数据量(Peek)大于其消化的数据量(Pop). 为了保证滑动窗口机制数据访问的正确性，本文采用循环缓冲区策略实现计算单元对共享缓冲区的访问. 针对核间缓存伪共享问题，用临时缓冲区和共享缓冲区相结合的方法，即对于生产者计算单元，首先将数据写入临时缓冲区中，在稳态调度完成

后将临时缓冲区中的数据拷贝到对应的共享缓冲区内;对于消费者节点,在稳态调度开始前先将共享缓冲区中的数据拷贝到临时缓冲区中,然后在计算调度时只访问临时缓冲区.由于稳态调度中采用同步数据流的方式,即每次流水线迭代中所有的计算节点运行结束后才会进行下一次迭代,故无更需复杂的锁机制即可实现共享缓冲区.

(Ⅱ)翻转窗口机制是指消费者计算单元每次计算访问的数据量(Peek)等于其消化的数据量(Pop).对于这种数据访问模式,采用二维数组实现计算单元间的通信缓冲区,以减少计算单元对缓冲区的访问开销.二维数组的最高维的大小由通信缓冲区对应的读写计算单元的阶段值的差决定,而最低维的大小在每次稳态调度的通信量的基础上扩充到缓冲的整数倍.另外,通信缓冲区在主存中以目标系统缓存行大小对齐的要求分配.对于图4中计算单元A和B间的通信缓冲区,假定两者阶段值的差为1,通信数据的单位大小为4B,缓冲行的大小为64B,则数组最高维的大小为2,数组最低维的大小为16.

2.2 逻辑线程到多处理器核的映射

当不同处理器核调度的线程共享相同数据集时,共享的缓存存储能够有效提高程序性能.由于并行线程共同竞争导致某个线程访问的数据因其他线程的调度而被替换时,共享的缓存存储会降低程序性能,因此逻辑线程到处理器核的映射策略对程序的性能有着很大的影响.

鉴于任务划分后各线程间通信分布可能存在不对称性,本节设计了逻辑线程到处理器核的映射算法.该算法结合系统层次性缓存拓扑结构,通过将线程间通信密集的线程映射到共享缓存层次较高的处理器核上,从而提高缓存使用效率.图5给出了映射算法的流程.映射过程由层次性结构的顶端开始向下,统计上层缓存存储上的逻辑线程间的通信分布,将数据共享密集的线程映射到同一组中,组的大小为该层单核缓存的处理器核个数.

3 数据流程序目标代码生成

3.1 多核平台的目标代码生成框架

为了完成将数据流程序在多核平台下的并行调度,编译器需要生成对应的目标代码.本文采用C++作为目标语言,这主要是从两个方面考虑:

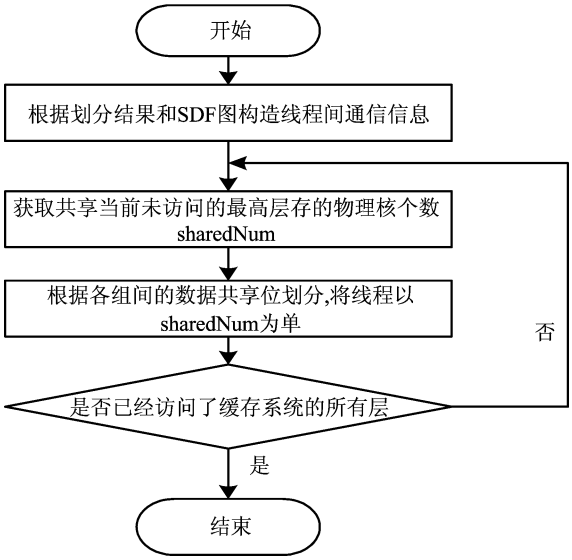


图5 逻辑线程到物理核的映射算法

Fig. 5 Mapping algorithm for logical thread to physical kernel

①C++语言支持多种编程模式,比如函数式编程模式、基于对象编程模式以及模板编程模式,这些模式使得目标代码生成工作变得更为简单;②C++语言在不同平台具有可移植性性以及高性能,本文工作主要就是提高数据流程序的吞吐率,相比Java、C#等更高层的语言,C++无疑更适合.在选定C++作为生成代码目标语言的基础上,需要确定线程库以实现数据流程序的并行调度,编译系统采用Linux下的Pthread线程库实现并行多线程.

使用C++和Pthread库的目标代码框架主要由三个相互关联的部分构成,首先需要使用C++的类封装数据流程序中的计算单元,每个计算单元对应一个类;然后需要完成计算单元间通信的共享缓冲区的定义;最后需要定义并行线程函数,在每个并行线程函数内部完成软件流水调度.

3.2 多核平台的目标代码生成实现

3.2.1 计算单元的封装

在代码生成框架中,编译器采用C++中的类封装计算单元,每个计算单元对应一个类,类名为修饰过的计算单元的名字.每个类只提供初态调度runInitSchedule、稳态调度runSteadySchedule以及构造函数三个接口:初态调度完成对应计算单元内部变量的初始化以及调度该计算单元的初态调度函数;稳态调度完成对应计算单元的稳态调度函数,调度次数为计算单元的稳态调度次数;构造函数用于实例化对应计算单元.

3.2.2　软件流水调度代码生成实现

首先,实例化分配到该线程上的计算单元类.然后进入第一个 for 循环实现计算单元的初始化工作,使用实例化对象完成其初态调度.再进入第二个 for 循环完成稳态调度,定义 stage 数组实现了流水调度的流水填充、流水满以及流水退出三个调度阶段,该数组大小为软件流水调度的最大阶段值.最后,为了满足不同线程上计算单元的数据依赖关系,本文采用 sense-reversing barrier[12] 完成核间同步操作.生成并行线程的 C++伪代码如下:

```
    voidthread_x_fun(){
//实例化 actor 的对象;
  charplStage[stageNum] = {0};
plStage [0] = 1;
  //初态调度
  for(int i=0; i < stageNum; i ++){
    if(stageNum -1 == i){
  //调用阶段号为 stageNum -1 的计算单元的初态调度函数
    }
    ……
    if(0 ==i){
    //调用阶段号为 0 的计算单元的初态调度函数
    }
  }
  //稳态调度
  for(int i =0; i < runTime + stageNum -1;i++){
    if(plStage [stageNum -1]){
      //调用阶段号为 stageNum -1 的计算单元的稳态调度函数
    }
    ……
    if(plStage [0]){
    //调用阶段号为 0 的计算单元的稳态调度函数
    }
    //移动 stage,实现流水线填充与排空
    for(int j = stageNum -1;j>=1; j --)
plStage [j] = pipelineStage [j-1];
    //判断是否进入流水线排空阶段
    if(i == stageNum -1)
plStage[0] = 0;
barrierSync();//同步操作
  }
}
```

3.2.3　共享缓冲区定义代码生成实现

针对 X86 多核架构,本文采用共享缓冲区实现计算任务间的通信,减少不必要的数据传输开销.正如 2.1 节介绍的,对于消费者计算单元两种不同的数据访问模式,采用不同的实现:

(Ⅰ)对于使用滑动窗口机制的数据访问模式,采用循环缓冲区的策略管理计算节点间的通信数据.目标代码中使用了 Buffer、Producer 和 Consumer 三个模板类实现循环缓冲区的管理,其中 Buffer 模板类负责实现缓冲区的分配,并提供 Producer 和 Consumer 模板类对缓冲区的访问接口,而 Producer 模板类和 Consumer 模板类分别负责封装的计算单元对缓冲区的访问.

(Ⅱ)对于使用翻转窗口机制的数据访问模式,采用二维数组管理计算单元间的通信数据.相比循环缓冲区的实现方式,使用二维数组对通信数据的访问延迟基本可以忽略.二维数组的高维大小由缓冲区对应的生成者和消费者的流水调度阶段差决定,当通信计算单元被划分到同一个处理器核上时,低维大小由每次稳态调度计算单元访问的数据量决定;否则,采用数据填充策略消除缓存伪共享.

4　实验结果与分析

4.1　实验平台和测试程序

实验采用 X86-64 架构的通用多核服务器作为测试平台.该服务器配有 2 颗 4 核的 2.40 GHz Intel Xeon E5620 CPU,最大支持内存 48GB.

在 Intel Xeon E5620 CPU 的缓存系统中,该处理器上每个处理器有一个私有的 *L*1 Cache,其大小为 32kB,另外处理器核两个为一组共享一个大小为 64MB 的 *L*2 Cache.实验平台上使用的 Linux 系统的内核版本为 2.6.18,GCC 版本为 4.1.2,G++版本为 4.1.2(编译器采用 C++作为目标代码,需要调用 G++编译器生成可执行程序),Metis 版本为 5.0.

为了对本文流编译优化策略的有效性进行测试和分析,选取 12 个数字多媒体领域的典型算法作为测试程序.各测试程序的功能、规模和结构特征如表 1 所示,其中 SDF 图的节点数目和通信边反映了并行调度的复杂性.综合该表数据可以看出,测试程序的规模和复杂度多样化,能够检验 COStream 流编译系统是否有效.

表1 测试用例信息

Tab. 1 Test case information

测试程序名	简要描述	SDF图结点数目
BitonicSort	二分排序程序	48
DES	DES加密算法程序	55
DCT	离散余弦变换程序	40
Beamformer	滤波器程序	57
VocoderTopLevel	比特率降低声码器	116
Filterbank	多速率信号处理滤波程序	85
ChannelVocoder	频道话路编码器	57
FFT	傅里叶变换程序	17
Serpent_full	蛇加密程序	60
FMRadio	调频收音机程序	31
Tde_pp	透明数据加密程序	54
MPEGdecoder	视频解码程序	39

4.2 实验结果与分析比较

(Ⅰ)扩大调度的有效性

本文采用扩大调度策略提高程序的并行粒度和局部性,其中扩大因子是根据程序的计算负载和通信量确定的.图6给出了设定扩大因子分别为1、2、4和8时测试程序的吞吐率,实验数据以扩大因子为1(即没有扩大调度)的测试程序吞吐率为基准.

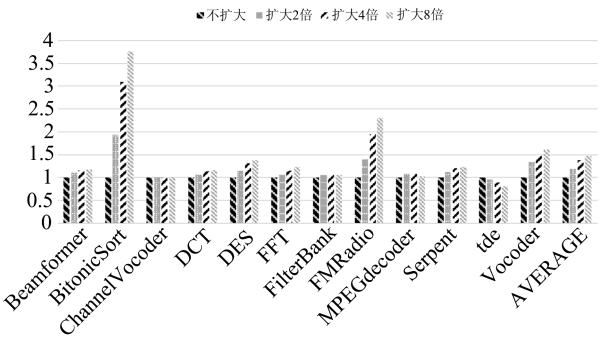


图6 不同扩大因子测试程序吞吐率比较

Fig. 6 Comparison of the throughput ratio in different expansion factor testing

从图6可以看出,随着扩大因子的增长,不同测试程序的性能有不同的变化,如BitonicSort的性能呈明显的上升趋势,这是因为扩大调度增加了每个流水调度周期的计算量,从而提高了程序的计算同步比;tde的性能呈现明显的下降趋势,其原因是该测试程序的通信量很大,使得扩大调度后缓冲区过大产生溢出,导致缓存系统使用效率低下.

为了验证扩大因子确定方法是否合理,实验还比较了在单核下使用扩大调度优化前后的测试程序的吞吐率.图7给出了实验结果对比,其中数据以在单核下不使用扩大调度优化的吞吐率为基准.从图7可以看出,12个使用程序自动生成扩大因子的测试程序中,有8个使用扩大调度后性能提高超过了10%.另外4个程序没有被扩大调度,即扩大因子为1,所以性能没有明显变化.

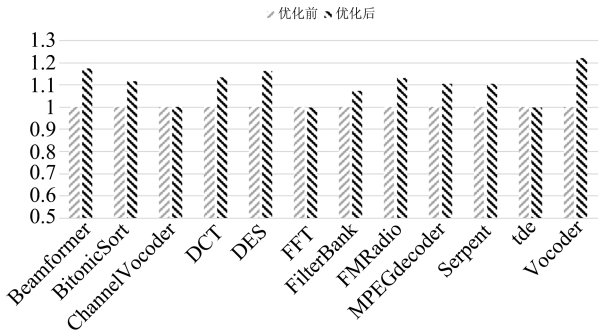


图7 测试程序在单核下扩大调度前后性能的比较

Fig. 7 Performance comparison of the test program before and after the expansion of scheduling in single core

对比图6和图7,发现只有极个别使用固定扩大因子的测试程序效果好于使用自动生成扩大因子的测试程序.这主要是程序的特性决定的.BitonicSort程序中通信开销相对于计算开销,所占比重很小,故而扩大的越大,执行的效果越好,并且自动生成扩大因子的方式适用于大部分应用.

(Ⅱ)加速比

图8给出了测试程序经过编译优化后的加速比示意图.从图8可以看出,测试程序的平均加速比基本呈现线性增长趋势,在2、4及8核下的平均加速比分别为2.1x、4x和7x.

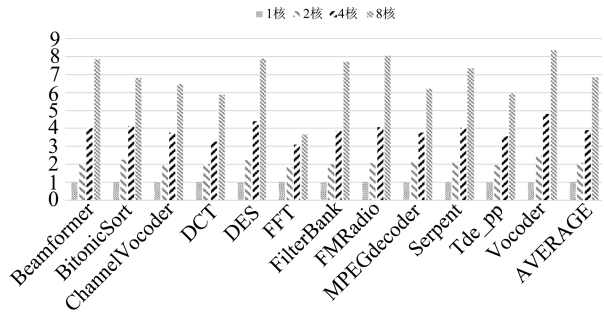


图8 测试程序的加速比图

Fig. 8 Acceleration ratio of test program

测试程序的性能主要由划分结果的负载均衡和连通性决定.在8核下,BeamFormer、DES、FMRadio以及Vocoder四个测试程序的加速比约

为 8x，FFT 的加速比不到 4x. 通过分析这些测试程序的划分结果以及实际运行时各线程的计算开销，BeamFormer 等程序的划分图连通性比较好，而 FFT 的划分图的连通性很差.

图 9 给出了 FFT 测试程序在 8 核下并行调度时各线程的计算同步比. 由图 9 可知，线程间的计算负载很不均衡. 理论上，任务划分阶段采用的迭代水平分裂算法能够保证各处理器核间的计算负载相对平衡. 然而，在实际并行调度时，多核间的相互干扰会使得计算单元的计算负载与静态估计有一定的误差，这样便造成了实际运行时核间的负载不均衡.

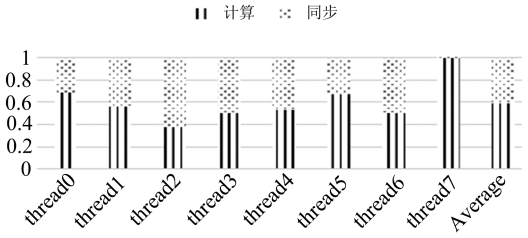


图 9　FFT 程序 8 核下各个线程的计算同步比

Fig. 9　Synchronization ratio in FFT program 8 core

5　结论

本文提出并实现了一种面向 X86 多核处理器的数据流程序任务调度与缓存优化方法，该方法通过对数据流程序预处理提高了并行粒度和局部性；然后利用程序特有的数据并行性改进负载均衡，并针对 X86 多核体系结构的缓存结构特征进行核间缓存优化. 实验结果验证了该方法的有效性. 由于并行调度时处理器核间的相互干扰会导致计算单元的工作量与静态估计时存在误差，从而降低运行时核间负载均衡. 如何设计在多核平台下的划分方法减少这种误差是将来需要进一步研究的工作.

参考文献(References)

[1] THIES W, KARCZMAREK M, AMARASINGHE S. StreamIt: A language for streaming applications[C]// Proceedings of the 11th International Conference on Compiler Construction. London, UK: ACM Press, 2002: 179-196.

[2] BUCK I, FOLEY T, HOM D, et al. Brook for GPUs: Stream computing on graphics hardware[J]. ACM Transactions on Graphics, 2004, 23(3): 777-786.

[3] MARK W R, GLANVILLE R S, AKELEY K, et al. Cg: A system for programming graphics hardware in a C-like language[J]. ACM Transactions on Graphics, 2003, 22(3): 896-907.

[4] KUDLUR M, MAHLKE S. Orchestrating the execution of stream programs on multicore platforms [J]. ACM SIGPLAN Notices, 2008, 43(6): 114-124.

[5] GORDON M, THIES W, AMARASINGHE S. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs[C]// Proceedings of 14th International Conference on Architectural Support for Programming Languages and Operating Systems. San Jose, USA: ACM Press, 2006: 151-162.

[6] ZHURAVLEV S, BLAGODUROV S, FEDOROVA A. Addressing shared resource contention in multicore processors via scheduling[C]// Proceedings of the 15th Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems. New York, USA: ACM Press, 2010: 129-142.

[7] 张维维，魏海涛，于俊清，等. COstream：一种面向数据流的编程语言和编译器实现[J]. 计算机学报，2013，36(10)：1993-2006.
ZHANG Weiwei, WEI Haitao, YU Junqing, et al. COstream: A language for datafloe application and complier[J]. Chinese Journal of Computers, 2013, 36(10): 1993-2006.

[8] THIELE L, BACIVAROV I, HAID W, et al. Mapping applications to tiled multiprocessor embedded systems[C]// Proceedings of the 7th Application of Concurrency to System Design. Bratislava, Slovak: ACM Press, 2007: 29-40.

[9] ABOU-RJEILI A, KARYPIS G. Multilevel algorithms for partitioning power-law graphs[C]// Proceedings of 20th International Parallel and Distributed Processing Symposium. Washington USA: IEEE Press, 2006: 1-10.

[10] MOLKA D, HACKENBERG D, SCHONE R, et al. Memory performance and cache coherency effects on an Intel Nehalem multiprocessor system [C]// Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques. Raleigh, USA: IEEE Press, 2009: 261-270.

[11] TORRELLAS J, LAM H S, HENNESSY J L. False sharing and spatial locality in multiprocessor caches [J]. IEEE Transactions on Computers, 1994, 43(6): 651-663.

[12] MELLOR-CRUMMEY J, SCOTT M. L Algorithms for scalable synchronization on shared-memory multiprocessors[J]. ACM Transactions on Computer Systems, 1991, 9(1): 21-65.