ISSN 1000-9825, CODEN RUXUEW
Journal of Software, Vol.19, No.9, September 2008, pp.2181−2190
DOI: 10.3724/SP.J.1001.2008.02181

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel/Fax: +86-10-62562563

数据流 Java 并行程序设计模型的设计、实现及运行时优化*

刘 驼[1,2+], 范 彬[1,2], 吴承勇[1], 张兆庆[1]

[1](中国科学院 计算技术研究所 计算机系统结构重点实验室,北京 100190)

[2](中国科学院 研究生院,北京 100049)

Dataflow-Style Java Parallel Programming Model and Runtime Optimization

LIU Tao[1,2+], FAN Bin[1,2], WU Cheng-Yong[1], ZHANG Zhao-Qing[1]

[1](Key Laboratory of Computer System and Architecture, Institute of Computing Technology, The Chinese Academy of Sciences, Beijing 100190, China)

[2](Graduate University, The Chinese Academy of Sciences, Beijing 100049, China)

+ Corresponding author: E-mail: liutao@ict.ac.cn

Liu T, Fan B, Wu CY, Zhang ZQ. Dataflow-Style Java parallel programming model and runtime optimization. *Journal of Software*, 2008,19(9):2181−2190. http://www.jos.org.cn/1000-9825/19/2181.htm

Abstract: This paper presents a dataflow-style Java parallel programming model with a runtime profile based thread duplication algorithm to exploit data level parallelism. Furthermore, a new dataflow polymorphism feature is introduced. This model has been implemented in an open source Java virtual machine. Evaluations on real machine show good speedup for benchmark applications.

Key words: dataflow; parallel programming model; managed runtime environment; runtime optimization

摘 要: 提出了一种具有数据流特征的 Java 并行程序设计模型,并针对该模型提出了一种基于运行时信息反馈的自适应优化算法,使得运行时系统可以利用数据流程序所暴露出的数据并行性,加速程序的运行.此外,在该模型中加入了数据流多态的概念,扩展了该模型的面向对象特性.在一个实际的开放源码 Java 虚拟机中实现了上述程序设计模型及优化方法.在实际多核多线程机器上的实验结果表明,所提出的程序设计模型及优化能够充分利用硬件的并行处理能力,显著地提高了程序的性能.

关键词: 数据流;并行程序设计模型;可管理运行时环境;运行时优化

中图法分类号: TP314 **文献标识码**: A

传统的提高单个处理器性能的方法,包括提高主频、采用复杂硬件发掘指令级并行性等已经走到了尽头.当今工业界和学术研究的主流已经转向通过提高单个芯片上处理单元的数目来提高处理器的性能[1−5].根据预测,片上的处理器核的数目每 2~3 年将会翻一番.在不久的将来,单个芯片上集成上百甚至上千个处理单元的多核处理器就会出现在市场上.

在过去的几十年里,人们提出了多种并行程序设计模型.现行主流的基于共享内存的并行程序设计模型是基于锁和信号量的多线程模型,但使用这一模型编程容易出错,并且难以开发和调试.同时,由于线程之间隐式

* Supported by the National Basic Research Program of China under Grant No.2005CB321602 (国家重点基础研究发展计划(973))
Received 2007-02-07; Accepted 2007-04-25

地对共享资源进行竞争和共享,其可扩展性也很差;另一个可能的研究方向是通过编译技术将串行程序自动并行化来提高程序性能,然而到目前为止,这一方法只在规则的科学计算等领域获得了有限的成功.而 Patterson 等研究者[6]指出,当处理器核数目达到 16 或 32 时,现在的程序设计模型和范式就会面临如同现在指令级并行类似的困难.因此,如何找到合适的面向多核多线程处理器的程序设计模型以发掘应用中的并行性,充分利用硬件提供的并行处理能力从而加速软件运行,是一个亟需解决的问题.

数据流模型是一种并行程序设计模型.它在数字信号处理、图像和多媒体处理、网络数据处理等领域都获得了成功.

Java 语言提供了一个基于管程(monitor)的多线程并行模型.这个模型易于理解,但在实际应用中存在很多问题:容易出错、难以验证、扩展性差等等.许多研究者都注意到了这个问题[7−10].他们的解决办法大多是在 Java 的多线程模型之上,以类库的形式实现一个更加安全和抽象的并行模型[11−14].但这些程序设计模型都是某种理论模型的实现,对程序员编程有严格的限制,转换的成本很高.同时,由于是采用 Java 类库的方式实现,并行性能有限,且虚拟机难以获得并行模型所特有的运行信息和特征,无法做出相应的优化.另一种方法是,在 Java 中通过扩展类库接口提供对本地并行库的调用[15−17].这类方法不用改变 Java 语言规范,易于实现和移植,但是往往也难以获得很好的性能.即使采用 JNI(Java 本地方法接口)的实现方法,由于会频繁跨越 Java 方法和本地方法之间的调用接口,也会带来很大的开销.同时,由于 JNI 接口的实现机制,也难以获得很好的并行性.而且,这类方法也难以利用程序模型的运行时特征进行相应的优化.

本文提出了一种具有数据流特征的 Java 并行程序设计模型,并提出了一种称为数据流多态的语言特征.与其他 Java 并行模型实现不同的是,我们的模型采用了类库与虚拟机内部机制协同设计的方案,将模型中与性能密切相关的和便于虚拟机进行动态优化的部分设计在虚拟机内部,通过程序设计模型上的语义限制来减少虚拟机运行时分析和优化的开销.

我们的数据流 Java 程序设计模型实现基于开放源码的 Java 虚拟机 Apache Harmony DRLVM[18].我们提出了一种自适应的启发式线程复制算法.根据该算法,基于运行时收集的反馈信息,虚拟机可以利用多核处理器的并行处理能力,通过动态线程复制来加速应用程序的运行.实验结果表明,这种自适应的线程复制机制能够发掘程序中的数据并行性,最大化地利用处理器的计算能力,加速程序运行,并获得良好的加速比.

本文第 1 节介绍数据流 Java 程序设计模型的特征.第 2 节描述运行时自适应线程复制算法及其实现.第 3 节是我们的实验结果及分析.第 4 节和第 5 节是相关工作和总结.

1 数据流 Java 程序设计模型

1.1 传统数据流模型

数据流模型很早就被提了出来[19,20].一般来说,一个数据流程序由多个 actor 组成.传统的细粒度数据流模型中,actor 的粒度是一个操作,而在粗粒度的数据流模型中,actor 的粒度可以是一个函数.actor 之间只能通过先入先出的缓冲队列进行通信.每个 actor 有一个相应的触发规则(firing rule)集合,当其中某一规则满足时,该 actor 被触发,读取输入队列上的数据,产生输出数据.actor 是没有内部状态的,它的行为只由输入数据和触发规则决定.类似的模型还有进程网络(process network).每个进程是一小段串行程序,进程之间只能通过先入先出的缓冲队列进行同步和通信.当一个进程读一个空队列或者写一个满队列时,它会被阻塞,直到操作完成.

1.2 数据流Java

在数据流和进程网络模型中,各个运行单元之间的同步和通信是通过显式的数据传递来完成的.由于禁止了运行单元之间的隐式数据共享,避免了多线程模型的数据竞争和冲突,有利于程序的形式化分析和验证.数据流模型能够帮助程序员自然地表达应用程序的内部并行性,减少编译器并行化分析和优化的难度.基于这些考虑,我们提出了数据流 Java 模型.

数据流 Java 中最小的独立运行的单元叫做组件(component),它对应于我们通常的进程或线程.组件内只能串行执行.一个数据流 Java 程序可以拥有多个组件,各个组件之间可以独立运行.

组件可以定义自己的输入和输出端口(port),用于和外部通信.其中,输入端口分为两种:普通输入端口和参数端口.一旦组件从普通输入端口接收了一个对象,那么这个对象就被该组件所独占,其他组件无法访问该对象.当一个对象被组件从输出端口发送给其他组件之后,该组件就不能再访问这个对象,而接收到这个对象的组件可以进行访问.参数端口用于向组件传递初始化参数.与普通输入端口不同的是,从参数端口接收到的参数对象不一定是独占的,可能会有多个组件共享.

组件之间的显式数据通信只能通过输入和输出端口之间进行.通信时,数据对象的发送和接收是异步的、先入先出的.当某个组件通过一个输出端口对多个组件的普通输入端口发送数据对象时,可以有两种发送方式:将数据对象复制多个副本后发送到所有组件;或者以轮转方式依次发送.如果是对多个组件的参数端口发送数据对象,那么也可以有两种方式:将数据对象的引用发送给所有组件共享;或者以复制的方式发送.

数据流 Java 采用隐式多线程模型,程序员需要知道组件运行时可能有多个副本同时运行.如果访问参数端口传递的共享的数据对象,则需要保证操作是原子的.

多个组件可以组成一个网络(network).如果网络与外部没有通信端口,则它是闭合的(closed)[21],是一个独立的数据流 Java 程序;反之,它是开放的,可以与其他组件或者开放网络组成更大的网络.

这种混合式的并行模型对于多核多线程体系结构而言更为理想:在细粒度上,保持每个组件的串行特征,使得许多串行算法和遗产代码可以重用;在粗粒度上,通过引入数据流特征,使得组件之间具有良好的并行特性,运行时很容易映射到硬件的线程级并行.组件只能通过通信通道和共享参数对象进行通信,多核多线程体系结构中,处理单元之间的高速通信机制能够极大地减少通信带来的开销和依赖.另一方面,不同的多核多线程处理器,其硬件线程的处理能力和对高速缓存的使用方案差别很大,会直接影响到应用程序的性能.数据流 Java 运行时,系统可以根据程序的运行时特征和体系结构的不同特点做出适应性的优化.

对于共享内存的多核多线程体系结构,引入共享的参数对象并不会带来性能上的明显开销.这种有限制的共享可以增加程序设计模型的表达能力,使其适用于更多的问题领域.

1.3 内存模型

数据流 Java 采用分布式和共享式结合的内存模型.每一个组件有自己独立的局部内存空间,这个内存空间中只存在该组件私有的数据对象.组件分配的数据对象,初始时都位于其局部内存空间中.当一个数据对象通过通信管道被发送到另一个组件时,它会被移动到新组件的局部内存空间中.移动的时机由动态内存管理机制决定,但必须在旧组件发送完对象之后,以及新组件接收到这个对象之前.

在局部内存空间之外,各个组件之间拥有共享的全局内存空间.全局内存空间中的对象都是多个组件共享的.一个组件私有的对象如果通过参数端口传递给多个组件共享,则该对象被移动到全局内存空间.当一个组件将私有的对象的引用赋值给一个共享对象时,该私有对象成为共享对象,将被移动到全局内存空间.任何一个组件对全局内存空间的访问和修改操作必须是原子的,并且对其他组件可见.

对于多核处理器而言,组件的局部内存空间可以容易地映射到单个处理单元的局部存储器上,全局内存空间可以映射为多个处理单元共享的存储器.由于多核处理器内部的通信延迟和通信带宽都远远优于对称多处理器,因此,局部内存空间之间的数据对象的传输可以通过处理单元之间的快速通信机制来完成.

1.4 编程接口

数据流 Java 的编程接口目前是以库的方式实现的,以 JavaFBP[22]为基础进行扩展.数据流 Java 程序中,用户需要显式定义组件.组件中可以声明输入和输出的端口.端口可以进行相应的接收和发送操作.每个组件有一个 execute 函数用于指定该组件在生存期中的执行逻辑.组件可以有内部状态,也可以没有内部状态.无内部状态的组件通常都有一个显式的无限循环.setPorts 函数用于设定每一个端口的名字和传输的数据对象类型.图 1 所示是一个用于从输入端口 in 向输出端口 out 转发数据的名为 Forward 的组件.

组件的连接关系由程序员在网络中定义.有 3 种不同的连接原语:connect 用于把不同的输入和输出端口绑定在一起,形成一条数据通道;initialize 用参数对象连接参数端口;initializeConnect 用于连接普通输出端口和参

数输入端口.如果程序员特别指定,不同组件的参数输入端口可以获得共享对象的引用.图 2 是一个简单的网络.

```
Public class Forward extends Component {
  InputPort in;
  OutputPort out.

  Protected void execute() throws Throwable {
    Type p;
    while ((p=in.receive())!=null) {
      out.send(p);
    }
  }
  protected void setPorts() {
    in=setInput("IN",Type);
    out=setOutput("OUT",Type);
  }
}
```

Fig.1 Definition of component

图 1 组件定义

```
Public class App extends Network {
  Protected void define() throws Throwable {
    Connect(component("Source",Source.class),
              Port("OUT"),
            Component("Compute",Compute.class),
              Port("IN"));

    initialize(new Integer(5),
      component("Source"), port("DATA"));
    …
  }
}
```

Fig.2 Definition of network

图 2 网络定义

如果程序员需要在某一点分裂(split)数据以利用程序的数据和任务并行性[23],可以直接将一个输出端口和多个输入端口相连接,数据默认会以轮转的方式自动分配.也可以设定以复制的方式分裂数据.通过连接也可以合并数据,但是数据的合并顺序是非确定性的(nondeterministic).

通过这些连接原语,程序员可以用 3 种最基本的结构组成网络,它们分别是流水线(组件顺序连接)、并行路径(组件通过分裂合并方式连接)和循环(以回边的方式连接形成环).这 3 种结构可以互相包含,组成更为复杂的网络结构.

1.5 数据流多态

数据流 Java 仍然保持了 Java 的面向对象特征.为了提高数据流 Java 组件的可重用性和可维护性,我们提出了一种称为数据流多态的新特性.如图 3 所示,Rectangle 和 Circle 分别是 Shape 的一个子类.*A* 的输出端口类型设置为 Shape,*B*,*C* 和 *D* 的输入端口类型为 Rectangle,Circle 和 Shape.*A* 发送的数据对象会在运行时根据其实例类型决定目的端口.如果是 Shape 的子类,会被发送到 *B* 或者 *C*,否则才发送到 *D*.这个特性使得程序员可以在配置网络时,通过变换连接方式改变程序的行为.同时,也易于程序的增量式开发和维护.这也使得 *B*,*C* 和 *D* 的代码可以独立开发.如果要为 Shape 增加一个子类,只需要定义处理新子类的组件,然后与 *A* 的输出端口连接,而不需要修改 *A* 的代码.我们称这种特性为数据流多态.

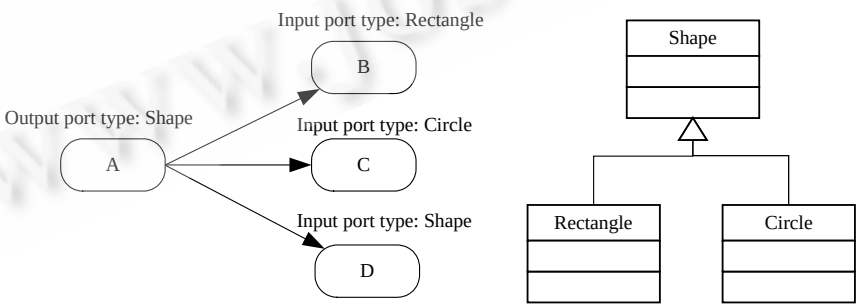

Fig.3 Polymorphism of dataflow

图 3 数据流多态

数据流多态的实现,需要在 *A* 的输出端口维护一个我们称之为分派树(dispatch tree)的数据结构,如图 4 所示.分派树在运行时构建网络的时候创建.输入端口类型必须是输出端口类型本身或者其子类,一个类型只允许

有一个端口.运行时,系统根据类的层次关系建立一个反向的树结构.树的每个节点记录它的类型和对应的出口.输出端口在发送数据对象时,根据分派树从上向下进行匹配,然后向匹配的端口进行发送.

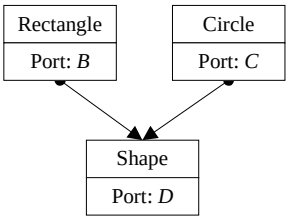

Fig.4 Dispatch tree

图 4 分派树

1.6 环正常结束状态检测

数据流 Java 中允许环的存在,程序员通过调用 connect 原语,可以在网络中构造环.但在数据流 Java 程序中,环可能发生死锁的情况.图 5 所示是一个带有环的数据流 Java 程序的数据流图.当组件 *A* 结束后,*B* 和 *C* 互相在等待着对方的数据,程序陷入了死锁状态.但这其实是程序的一个正常终止状态,我们需要运行时检测这种状态,使程序能够正常结束.

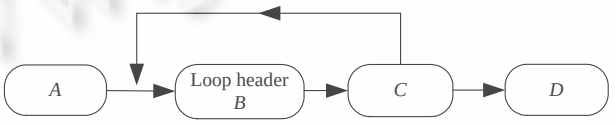

Fig.5 Dataflow graph with loop

图 5 带环的数据流图

无嵌套环死锁检测的算法见算法 1.由于节点处于阻塞和非阻塞两种状态,而数据对象可能存储在输入/输出队列上,因此,检测算法必须保证环没有其他输入边,并且环上不存在数据对象.检测成功之后,从头节点开始依次结束组件,可以使程序正常结束.

算法 1. 检测数据流图上的无嵌套环是否处于死锁终止状态:*DetectLoopDeadlock*( ).

输入:数据流图,环头节点(loop header);

输出:死锁终止返回 true,否则返回 false.

(1) if 头节点除了循环之外还有其他输入边存在 then {return false}

(2) 从环头节点开始,拓扑序遍历循环体上的节点 *n*

(3) for each *n* do

(4) {if (*n* 是非阻塞的 or *n* 的输入边有数据对象 or *n* 的输出边有数据对象) then {return false}}

(5) return true

2 自适应线程复制

2.1 系统设计框架

为了高效地支持数据流程序设计模型以及数据流程序的运行时自适应优化,我们采用了扩展库和 Java 虚拟机相结合的设计方案.如图 6 所示.数据流 Java 的编程接口(dataflow Java API)以扩展库的方式提供给程序员.扩展库将组件映射为虚拟机中的 Java 线程.与数据流 Java 程序性能和优化相关的部分都位于虚拟机中.通信通道(communication channel)提供线程之间的非阻塞高度并发通信机制,支持一对一、一对多和多对一通信.运行时,系统在程序启动时会根据通信通道的连接关系构造程序对应的数据流图.Profiler 通过采样的方式统计通信通道的信息,并标注到数据流图上.系统监视器(system monitor)收集系统负载等信息.即时编译器(JIT compiler)

进行插桩代码,统计线程的对象访问信息,由 Profiler 收集.代价模型(cost model)根据运行时获得的各种信息发现程序的性能瓶颈,决定是否进行线程复制.线程复制后,数据流图会相应更新.

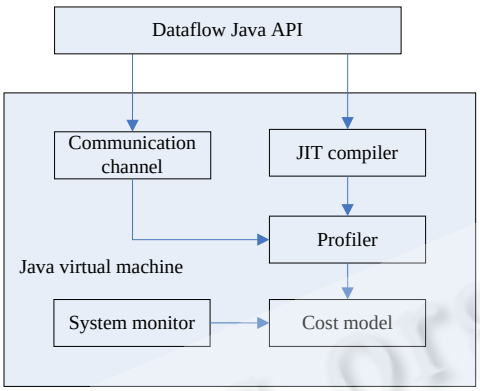

Fig.6 System framework of dataflow Java

图 6 数据流 Java 的系统框架设计

2.2 自适应复制算法

一个数据流程序由多个组件构成,其中大多数的组件都是无状态的(stateless).也就是说,它们由一个无限循环构成,循环迭代之间无依赖关系.这体现了数据流程序的数据并行性.在运行时,我们可以将组件的不同迭代分布到多个线程执行,利用底层硬件的多线程支持来发掘高层的数据并行性.可以通过编译时分析是否有循环迭代间依赖以及对参数对象的操作来识别组件是否是无状态的.目前,我们的库实现方式中需要程序员显式地指定组件的状态属性.

要通过自适应的线程复制来加速程序运行,主要需要解决两个问题:首先,如何发现程序真正的性能瓶颈;其次,如何正确估计在当前系统负载下,瓶颈线程的理想复制数目.

数据流图不仅能够准确反映程序内在的逻辑关系,还是运行时实际的数据通路,因而是性能预测和评估的重要依据.程序运行时,数据流图的每条边会记录流经的数据对象的数目,还会标记每条边的平均等待队列的长度.边上流经的数据对象的数目越多,表明与之相连的组件的相对执行频率越高,边的权值也越大.数据流图中具有较大权值的路径集合,就构成了程序的关键路径.在关键路径上,如果通过采样发现某个节点的输入边等待队列很长而输出边等待队列很短,并且它是无状态节点,那么这个节点是可复制的瓶颈节点.

我们以启发式的算法来估算理想的线程复制数目.公式如下:

$$N_p=F_p\times N+\mathrm{Max}(\Sigma RB_{succ}-\Sigma RB_{prev},0)\times f_1-SW_p\times f_2+\mathrm{Max}(\Sigma C_{prev}-Th_{upper},0)\times f_3+\mathrm{Max}(Th_{lower}-\Sigma C_{succ},0)\times f_4,$$

其中,N_p 是估算的线程应复制数目,F_p 是该线程的负载占关键路径上总负载的比例,N 是系统总的处理单元的数目,RB_{prev} 和 RB_{succ} 分别是关键路径上前后相邻线程的读阻塞次数,SW_p 是该线程在执行中对共享数据对象的平均写操作次数,C_{prev} 和 C_{succ} 分别是关键路径上前后相邻线程通信通道的平均数据队列长度,Th_{upper} 和 Th_{upper} 是设定的阈值,f_1~f_4 是加权修正系数.

算法 2 假设在处理器数目足够的情况下,采用平衡每级流水线工作量的方法估算线程节点的所占总负载的比例 F_p,算法复杂度为 $O(N)$,N 为数据流图上线程节点个数.每个线程的负载可以通过每两次有效读写通信通道之间的时间间隔(也就是平均迭代执行时间)来测量.线程的读阻塞次数由 Profiler 获得,记录在数据流图的节点上.对共享数据的写操作由即时编译器产生的插桩代码收集.

算法 2. 计算节点在数据流图上关键路径中所占的负载比例:*ComputeFraction*( ).

输入:数据流图,节点 n;

输出:节点 n 在关键路径上所有节点中所占负载比例.

(1) $w\leftarrow n$ 的负载,$f\leftarrow 1.0$

(2) 找到节点 n 所在的外层结构 p,如果 p 不存在,跳转(6)

(3) if p 是流水线 then {$w \leftarrow \Sigma$流水线上所有节点负载;$f \leftarrow f \times n$ 的负载/w;$n \leftarrow p$;跳转(2)}

(4) if p 是并行路径 then {$w \leftarrow$Max(任一并行路径上节点负载之和);$f \leftarrow f \times n$ 的负载/w;$n \leftarrow p$;跳转(2)}

(5) if p 是循环 then {$w \leftarrow \Sigma$循环体中节点负载;$f \leftarrow f \times n$ 的负载/w;$n \leftarrow p$;跳转(2)}

(6) return f

在进行复制时,每隔一段时间,系统查找是否出现新的瓶颈线程.如果系统处理器有空闲,则进行复制.对新的瓶颈线程和复制过的线程,估算每个线程需要复制的数目 N_p,每次以 N_p 与当前实际数目差值的某个固定比例来复制或取消线程.当数据流程序处于不同运行阶段时,它的负载和瓶颈节点都可能发生变化,这个复制算法可以自适应地根据程序的运行状态调整多个瓶颈线程之间的复制比例.计算负载变小的线程,它的副本数目会减少,以提高运行时的数据局部性,减少线程调度的开销.

创建运行线程的副本时可以有两种方法:一种是直接复制当前线程的镜像,一种是开启新线程.如果直接复制,首先需要暂停线程,否则,线程的运行时栈是变化的.其次,线程暂停时必须没有处理数据,否则,这些数据会在复制后被重复处理.需要复制的数据结构包括栈、线程控制结构、堆中引用对象等等.这种方法需要暂停线程,而复制的线程往往是程序的瓶颈节点,暂停之后会阻塞整个程序的运行,所以我们采用直接创建新线程的方法.当无状态的组件运行时,系统标记参数端口输入的对象.当需要复制时,直接创建该组件的新线程实例,然后向参数端口发送标记对象.标记对象可以进行复制,也可以共享,取决于程序员设定的语义.最后用原子操作连接新的数据通道,并更新数据流图.这种方法可以保证在复制过程中,原程序无中断持续运行.

3 实验结果和分析

我们在实际的硬件上进行了实验.实验平台为 Dell PowerEdge 2900,含两个 Intel 至强 ™ 双核处理器,支持超线程.操作系统是 Redhat Enterprise Linux 4,内核版本是 2.6.9-5.ELsmp.测试例子采用基于 Raja[24]的光线追踪程序和 SPECjbb 2000,由我们改写成数据流 Java 的形式.光线追踪程序的两个测试图形如图 7 所示.

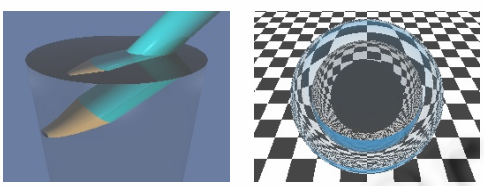

Fig.7 Ray tracing test cases

图 7 光线追踪测试图形

实验结果如图 8 所示.其中,Sequential program 是 Raja 和 SPECjbb 2000 的单线程版本,Dataflow Java 是我们改写之后的版本,Thread duplication 是进行复制之后获得的最高性能.可以看到,相对于串行版本,我们改写后的数据流 Java 程序增加了约 13%~21%的运行时间,这一方面是因为我们的实现是基于扩展库的方式,在运行时会增加程序的开销;另一方面,数据流 Java 程序会生成很多临时对象,也增加了程序运行时间.此外,由于我们只是直接使用操作系统线程来映射数据流组件,因此,操作系统的进程调度无法按照数据流程序最理想的方案调度线程运行,也会对性能有影响.通过自动线程复制,测试程序的运行时间大为缩短.相对于原程序获得了 2.43 的平均加速比,相对于改写后的数据流 Java 版本,平均加速比为 2.97.

测试程序的可扩展性如图 9 所示.当线程数超过 4 以后,程序的可扩展性降低,这说明超线程技术对性能的影响有限,不如独立的处理核.SPECjbb 2000 的可扩展性较差,这是因为我们改写的数据流 Java 版本是按照事务请求的类型来分类处理的,对同一个仓库的不同类型请求可能需要同时被处理.而 SPECjbb 中对仓库的访问是需要同步的,这就会产生竞争.复制的线程数越多,竞争对程序性能的影响就越明显.另一方面我们也发现,数据流程序运行时会产生大量生存周期较短的数据对象,对垃圾收集器的压力很大.而常用的垃圾收集算法运行数据流程序时开销很大,而且需要暂停所有线程,降低了数据流程序的并行性.

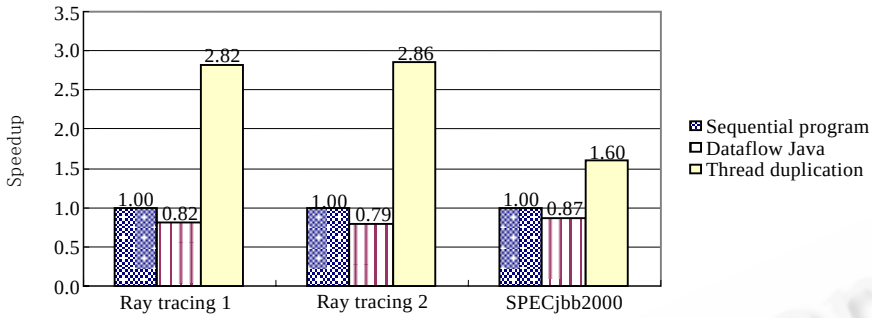

Fig.8 Experimental results

图 8 实验结果

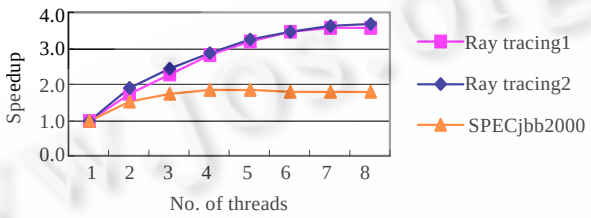

Fig.9 Scalability

图 9 可扩展性

4 相关工作

近年来,研究者提出和实现了一些与数据流相关的程序设计模型和环境.StreamIt[25]主要基于同步数据流(synchronous data flow)的程序设计模型,同时通过引入 Teleport message,加入了一些动态特征.但 StreamIt 模型比较简单,所有分析和优化只需在编译时进行.这样虽然简化了难度,但减小了模型的表达能力,一般只适用于 DSP 等流式计算领域.Brook[26]是面向图像处理器(GPU)的数据流程序设计模型.JCSP[11]是通信串行进程(communication sequential process)的一个 Java 实现.但它们在这个实现中引入了许多非 CSP 的特征:如带缓冲的通信队列、栅栏(barrier)和 CREW(concurrent read exclusive write)锁等高级同步机制.带缓冲的通信队列使得 JCSP 能够支持数据流模型的特征.CTJ[12]同样是 CSP 模型的一个 Java 实现,但它更偏重于实时处理.JavaFBP[21]是基于一种流式程序设计方法学(flow-based programming methodology)的 Java 库实现.它定义了一些用于编写商业应用的程序设计概念和接口,但性能较差.Ptolemy[27]支持许多种与数据流相关的程序设计模型.Click[28]和 Shangri-la[29]是针对网络处理领域的数据流程序设计语言和环境.数据流 Java 与这些模型的不同之处在于,它主要注重运行时系统与程序设计模型的协同设计,如何利用程序设计模型的运行时特征进行优化,同时尽量扩展对通用问题领域的表达能力,不局限于某些特定领域.袁伟等人[30]提出了具有原型和类体的 Dual-Object 模型,用于解决基于远程对象调用的分布式对象效率问题,主要面向分布式系统.数据流 Java 主要面向多核多线程体系结构.

Parks[21]采用主线程以较低优先级运行的方式来检测死锁的发生,但在多处理器上运行时会占用一个空闲的处理器.Stevens[31]的实现中,所有线程一起维护一个读阻塞和一个写阻塞计数器.当计数器之和等于线程总数时,检测到死锁发生,但每个线程维护计数器有运行时开销.我们的方法在死锁没有发生时没有运行时开销,也不会占用空闲处理器,但只对程序结束时的无嵌套环死锁检测有效.

Gummaraju 和 Rosenblum[32]提出了将数据流程序映射到通用处理器体系结构的方法.Gordon 等人[23]提出了数据流程序中的流水线并行、任务并行和数据并行.他们采用启发式算法,在编译时计算 StreamIt 程序中每个 filter 的工作负载来决定如何合并 filter,然后进行复制.这是一种简单的静态估算方法,只处理顺序和分裂合并

(split-join)结构.与静态分析方法不同,数据流 Java 是根据程序运行时的多种反馈信息自适应地预测复制线程的数目,不需要预先知道目标机器的处理器数目,也不必重新编译.Chan[33]在函数调用时开启异步线程来发掘 Java 程序的方法级并行性.它通过结合编译时分析和运行时检测技术来确定线程之间的依赖关系.于勐等人[34]采用编译分析方法,发现 Java 程序中对象内和对象间的并行性来加速程序.然而,编译程序只能发掘有限的程序并行性,数据流 Java 使程序员能够更自然地表达程序内部的并行性,给编译程序和运行时系统提供了更多的优化机会.

5 总 结

本文提出了一种具有数据流特征的 Java 并行程序设计模型和数据流多态的新语言特性,并在实际的 Java 虚拟机中实现了该模型.通过程序设计模型与运行时系统协同设计的方式,使得运行时系统可以获得程序的运行时特征,进行有针对性的优化.并提出了一种自适应的线程复制算法,它基于程序运行时的各种信息反馈,动态预测复制线程的数目,利用程序的数据并行性提高程序性能.实验结果表明,该线程复制算法能够根据系统负载,自适应地加速数据流 Java 程序的运行,获得了良好的加速比.

我们下一步的工作是继续扩展数据流 Java 程序设计模型,增加对通用问题领域的表达能力.开发更多数据流 Java 的应用程序,设计适用于数据流 Java 程序的垃圾收集算法.

References:

[1] Hofstee HP. Power efficient processor architecture and the cell processor. In: Proc. of the HPCA. 2005. 258−262. http://portal.acm.org/citation.cfm?id=1043423

[2] Kongetira P, Aingaran K, Olukotun K. Niagara: A 32-way multithreaded sparc processor. IEEE Micro, 2005,25(2):21−29.

[3] Raza Microelectronics, Inc. http://www.razamicroelectronics.com/products/xlr.htm

[4] Andrews J, Baker N. Xbox 360 system architecture. IEEE Micro, 2006,26(2):25−37.

[5] Talor MB, Kim J, Miller J, Wentzlaff D, Ghodrat F, Greenwald B, Hoffmann H, Johnson P, Lee JW, Lee W, Ma A, Saraf A, Seneski M, Shnidman N, Strumpen V, Frank M, Amarasinghe S, Agarwal A. The raw microprocessor: A computational fabric for software circuits and general purpose programs. IEEE Micro, 2002,22(2):25−35.

[6] Asanovic K, Bodik R, Catanzaro BC, Gebis JJ, Husbands P, Keutzer K, Patterson DA, Plishker WL, Shalf J, Williams SW, Yelick KA. The landscape of parallel computing research: A view from Berkeley, EECS. Technical Report, No.UCB/EECS-2006-183, Berkeley: University of California at Berkeley, 2006. http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html

[7] Hilderink G, Broenink J, Vervoort W, Bakkers A. Communicating Java threads. In: Bakkers A, ed. Proc. of the WoTUG 20 Conf. on Parallel Programming and Java. IOS Press, 1997. 48−76.

[8] JavaPP. 2007. http://www.cs.bris.ac.uk/~alan/javapp.html

[9] Lewis T. If Java is the answer, what was the question? IEEE Computer, 1997,30(3):133−136.

[10] Lee EA. The problem with threads. Technical Report, No.UCB/EECS-2006-1, Berkeley: University of California, 2006. http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-1.html

[11] Communicating sequential processes for Java (JCSP). 2007. http://www.cs.kent.ac.uk/projects/ofa/jcsp/

[12] Communicating threads for Java (CTJ) 2007. http://www.ce.utwente.nl/javapp/information/CTJ/main.html

[13] Jada. 2007. http://www.cs.unibo.it/~rossi/jada

[14] JOMP. 2007. http://www.epcc.ed.ac.uk/research/jomp/index_1.html

[15] JavaMPI. 2007. http://perun.hscs.wmin.ac.uk/JavaMPI/

[16] JPVM. 2007. http://www.cs.virginia.edu/~ajf2j/jpvm.html

[17] mpiJava. 2007. http://www.hpjava.org/mpiJava.html

[18] Apache harmony DRLVM. 2007. http://harmony.apache.org/subcomponents/drlvm/

[19] Mcllroy MC. Coroutines. Int'l Report. Murray Hill: Bell Telephone Laboratories, 1968.

[20] Adams DA. A computation model with data flow sequencing [Ph.D. Thesis]. Stanford University, 1969.

[21] Parks TM. Bounded scheduling of process networks [Ph.D. Thesis]. Berkeley: University of California, 1995.

[22] Flow-Based programming. 2007. http://www.jpaulmorrison.com/fbp/

[23] Gordon MI, Thies W, Amarasinghe S. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. ACM SIGARCH Computer Architecture News, 2006,34(5):151−162.

[24] Raja. 2007. http://raja.sourceforge.net/

[25] StreamIt. 2007. http://cag.csail.mit.edu/streamit/

[26] BrookGpu. 2007. http://graphics.stanford.edu/projects/brookgpu/

[27] Ptolemy II. 2007. http://ptolemy.eecs.berkeley.edu/ptolemyII/

[28] Kohler E, Morris R, Chen B, Jannotti J, Kaashoek MF. The click modular router. ACM TCS, 2000,18(3):263−297.

[29] Chen M, Li XF, Lian R, Lin J, Liu LX, Liu T, Ju R. Shangri-La: Achieving high performance from compiled network applications while enabling ease of programming. ACM SIGPLAN Notices, 2005,40(6):224−236.

[30] Yuan W, Sun Y. Dual-Object: Approach to object-oriented parallel programming. Journal of Software, 1998,9(1):47−52 (in Chinese with English abstract).

[31] Stevens RS, Wan M, Laramie P, Parks TM, Lee EA. Implementation of process networks in Java. Technical Report, No. UCB/ERL M97/84, Berkeley: University of California, 1997.

[32] Gummaraju J, Rosenblum M. Stream programming on general-purpose processors. In: Proc. of the 38th Annual IEEE/ACM Int'l Symp. on Microarchitecture. Washington: IEEE Computer Society, 2005. 343−354. http://portal.acm.org/citation.cfm?id=1099547.1100561

[33] Chan B, Abdelrahman TS. Run-Time support for the automatic parallelization of Java programs. Journal of Supercomputing, 2004, 28(1):91−117.

[34] Yu M, Chen G, Yang X, Xie L, Guo M. JAPS-II: An automatic parallelizing compiler for Java. Journal of Software, 2002,13(4): 739−747 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/13/739.pdf

附中文参考文献:

[30] 袁伟,孙永强.Dual-Object:面向对象的并行程序设计.软件学报,1998,9(1):47−52.

[34] 于勐,陈贵海,阳雪林,谢立,过敏意.JAVA 并行化编译器 JAPS-II.软件学报,2002,13(4):739−747. http://www.jos.org.cn/1000-9825/13/739.pdf

刘弢(1978－),男,重庆人,博士生,主要研究领域为编译优化技术,运行时系统,程序设计模型.

吴承勇(1968－),男,博士,研究员,主要研究领域为指令级并行编译,代码生成与优化,公共编译基础设施.

范彬(1983－),男,硕士生,主要研究领域为编译优化技术,运行时系统.

张兆庆(1938－),女,研究员,博士生导师,主要研究领域为编译优化技术.