一种分布对象的并行程序设计框架*

王 晨 周 颖 张德富

(南京大学计算机软件新技术国家重点实验室 南京 210093)
(南京大学计算机科学与技术系 南京 210093)
E-mail: zhangdf@netra.nju.edu.cn

摘要 计算性能和合成性能对于基于工作站网的软件十分重要,但由于缺乏相应的开发环境,现在这类软件在这两方面还做得很不够,尤其是合成性能十分薄弱.该文提出并实现了一种基于分布对象的并行程序设计框架,力图使分布对象能提供高性能的并行计算服务,同时也使并行算法获得一种良好的封装和复用机制.经过一些并行算法的测试,表明该框架具有实用价值.

关键词 分布对象,并行程序设计,工作站网,框架.

中图法分类号 TP311

目前,越来越多的分布并行软件系统将它们的平台确定在异构型工作站网(workstation clusters)之上,为这种环境提供的计算模式也正在从 RPC(remote procedure call)向 CORBA(common object request broker architecture)[1]演化.其总的趋势在于[2],第一,增强网络互操作性(interoperability),第二,增强软件的可合成性(compositional).但对于试图充分利用工作站网上的处理资源进行并行计算而言,这种演进却没有带来实质性的帮助,原因有两个方面:一是缺乏对并行计算模型的支撑,使并行程序的编写依然是件烦琐而低效的事情;二是实现效率不高,缺乏对高速网络程序接口的支持.分布对象由于地理位置的分散而拥有自己的处理资源,也因此具有潜在的并行能力,在分布对象系统中提供并行计算服务也是一种自然的要求.

近年出现的 MPI,PVM,Express 等基于 message-passing 的工作站网络并行计算环境更增加了人们对于在这种平台上进行并行计算的需求.但这几种并行计算环境提供给程序员的基本上都是一个消息传递库,需要程序员自己去匹配进程(或线程)之间的通信.这样做的缺点在于:(1) 如同用汇编语言写串行程序一样,并行程序的书写较为困难,代码难于控制;(2) 软件难以进行复用.

基于这种原因,我们提出一种分布对象的并行程序设计框架 DOPPF(distributed object's parallel programming framework),尝试解决以上问题,并保持分布对象的可复用特性.

1 分布对象的并行性分析

传统的对象计算模式由一串调用链组成,除了初始化对象外,其他对象的激活都通过另一个对象调用它的公共方法来实现,可以说它们的方法调用是遵循一种 invoke by other object 的方式.被调用者称为 supplier 对象,调用者称做被调用者的客户(client).在分布式环境中这种对象调用方式由 RPC 演变成远程方法调用 RMI(remote method invocation)[3].但分布对象关注的只是激活远程对象并获取服务,因此,RMI 也如 RPC 一样没有利用本身的潜在并行性,client 总是不管调用点以下的代码是否与调用的结果相关而阻塞,直到 supplier 的调用方法执行完.我们称这种方式为同步远程方法调用.它的作用有两个方面:一是实现分布对象调用的位置透明

* 本文研究得到国家 863 高科技项目基金资助.作者王晨,1971 年生,博士生,主要研究领域为分布并行系统和面向对象技术.周颖,女,1973 年生,硕士,主要研究领域为分布对象技术.张德富,1937 年生,教授,博士生导师,主要研究领域为并行处理技术和分布式系统.

本文通讯联系人:张德富,南京 210093,南京大学计算机科学与技术系

本文 1997-11-13 收到原稿,1998-03-03 收到修改稿

性;二是使 client 与 supplier 保持一定的同步关系.

为了获得并行效率,我们显然还需要有使 client 和 supplier 能够并行执行的异步调用方式.这需要处理以下几方面的问题:

(1) 使 supplier 能够同时处理多个请求,并在某些时候能将这些请求的处理分布到另外的空闲工作站上,称为对象内的并行.

(2) 对于某些需要在几个请求之间保持 supplier 的状态的 client 而言,supplier 还需要具有向 client 提供原子操作的事务处理(transaction)的机制.这是 CORBA 这类框架中具有的服务.

(3) 使 client 在远程方法调用后可以继续执行,直到需要这次调用的结果,称为对象间并行.

(4) supplier 由于容纳了多个线程的执行,自身状态是不稳定的,也就是说其属性可以被多个线程所改变.其中有些改变不是某些 client 所导致的,但却是它们所需要了解的.这需要 supplier 和 client 间的一种异步通信方式.在分布对象中被称为事件服务(event_service),但其实现效率并不能满足并行计算的需要.

作为一种标准,CORBA 制定了分布环境中软件构件的通用接口形式,但要使这种标准广为接受,就应当使之适应尽量多的应用领域,我们提出的 DOPPF 表明,并行计算是属于这些领域的.参照 CORBA,这个框架更强调:(1) 对并行计算方式的支持和框架本身的实现效率;(2) 接口定义语言中对主动性继承的多种手段,并以此作为实现对象内并行的一种方式.

2　分布对象的并行程序设计框架

2.1　结　构

OO framwork 是一个类的集合,由这些类合作可解决某一问题或提供某种能力.复用一个 framework 意味着继承它的类,并通过重载其中的方法来精化这些类.有两种方式的 OO framework[4]:called framework 和 calling framework.我们的·DOPPF 包含了这两种方式.一个简化的用户视图如图 1 所示.

DOPPF 由客户方和服务方(分布对象的实现)两部分组成,其下是运行时刻支撑环境,主要提供消息传输,用于分布对象定位的名服务等接口.客户方包括客户应用和若干个主动对象的 Stub,采用 called framework 的形式.客户应用实例化由 DOPPF 生成的 Stub 类,就可以创建其实例并通过通常的方法调用享受位于不同地址空间的 supplier 对象提供的服务.而服务方实际上就是一个完整的 supplier 对象,采用 calling framework 的形式.主要体现在两个方面:一方面,DOPPF 生成的控制线程,在运行时根据客户和请求调用程序员实现的 supplier 对象的功能;另一方面,对于 supplier 对象自身状态不稳定的情况,其中的事件服务机制也采用 calling framework 的形式,这是因为事件服务的大部分功能和总体框架均由 Event_service 类定义实现,程序员需要做的只是实现具体的事件调度方法.由该框架管理事件服务的方方面面的工作.

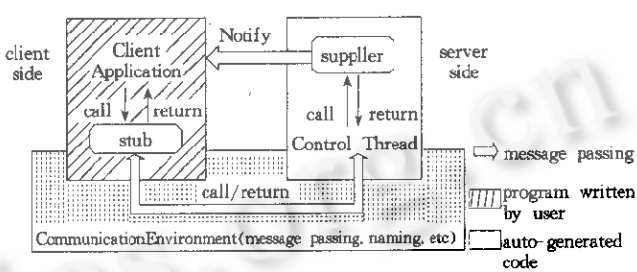

图1　DOPPF视图

图 1 中的 Stub 是 supplier 对象在客户方的映象,Stub 类实际上是一个经过改造的 supplier 类,它和 supplier 的类定义大致相当,有相同的名字、相同的 public 方法和 public 变量,只是内部实现完全不一样.调用其中的方法就相当于向远程 supplier 对象请求执行相应的方法.stub 保证了分布对象的位置透明性.

图 1 中的 Notify 消息是在使用事件服务时,supplier 对象向 client 发送的异步信息.它也需要通过底层通信环境传递给相应的 Stub.将它单独表示是为了和一般方法调用中的返回消息区分出来,因为这一消息并不通过 Control Thread/Stub 通道来传递.

DOPPF 对程序员达到了很高程度的透明性,不论 supplier 是远程对象还是本地对象,客户程序对它的使用

方式都是一样的.消息传递中的数据打包(marshal)、解包(unmarshal)以及目标对象定位等问题都由DOPPF生成的代码自动完成.而描述分布对象的IDL(interface definition language)语法也十分简单.

2.2 接口定义语言(IDL)

DOPPF提供的IDL主要是对C++类定义的扩充,使其能描述分布对象的动态特征.另外,在对象声明中也扩充了其主动和被动属性的转换修饰.在类定义时指定其实例是否为主动对象.而类实现以及客户应用部分不需任何改动.扩充语法如下,其中DOPPF的新增关键字加黑表示,与C++语法相同的部分略去:

⟨class definition⟩ ::=[**Active**]class⟨class name⟩[⟨inheritance list⟩]⟨class body⟩

⟨inheritance list⟩ ::=':'⟨inheritance segment⟩|⟨inheritance list⟩⟨inheritance segment⟩

⟨inheritance segment⟩ ::=[⟨access specifier⟩]⟨class name⟩[⟨active specifier⟩]

⟨access specifier⟩ ::=public|private|protected|**callback**

⟨active specifier⟩ ::=**actively**|**passively**|⟨part specifier⟩

⟨part specifier⟩ ::=[⟨active method⟩][⟨active variable⟩]

⟨active method⟩ ::='['**make active_method**';'⟨method declare list⟩']'

⟨active variable⟩ ::='['**make active variable**';'⟨variable declare list⟩']'

由Active class模板实例化的对象默认为主动对象,但也可以使用这种模板生成被动对象.对象声明的语法做如下扩充:

⟨object declaration⟩::=[**actively**|**passively**]⟨Active class name⟩⟨object name⟩[parameter]';'

2.3 分布对象方法调用的语义

2.3.1 同步调用

主要针对函数形式的方法调用,如ret=active_obj.(arg1,arg2,...),有返回值而且返回值对下面的程序执行有影响,这种调用是同步的,客户必须等待,直到服务方完成调用、返回结果才可继续往下执行.这种调用方式的另一个作用就是使并行执行的程序保持一定的同步关系.

另外,为了减小这种调用的参数传递开销,DOPPF实现中将参数及调用方法无阻塞地传递到分布对象的消息队列中,这样可以减少同步点,同时提高同步调用的效率.

2.3.2 异步调用

主要针对过程形式的方法调用,如active_obj.(arg1,arg2,...),这种调用没有返回值但会引起分布对象状态的改变.过程型方法调用是异步进行的,客户在调用后立即往下执行,不必等待服务方是否完成操作.过程型的调用有利于并行执行,因此在并行要求较高的分布式程序中,应尽量多地使用过程型调用.获取对象状态的方式有两种:① PULL方式,由客户在需要时用DOPPF的预编译器自动生成的本地方法——*get_variable*()获取状态;② PUSH方式,client登记相应的处理函数,由supplier通过事件服务的方式通知状态的改变.同样,参数的传递也是无阻塞的.

2.4 异步消息处理框架

如图2所示,DOPPF生成的stub中包含了一个异步消息的监听线程,当supplier的异步消息到达时,它自动调用对象登记的callback方法.近来出现一些通信结构允许用户进程存取网络界面,并提出了一些异步消息的处理模式,如active-message,single-threaded upcalls以及popup threads.文献[5]在分析了这几种通信模式的表达能力和效率后认为,即使在高速网络条件下,牺牲表达能力以获取效率也是不可取的.DOPPF中的异步消息处理采用了面向popup threads的方式,但为保持对象的封装特性和减小client方并发控制的复杂性,在每个需要事件服务的分布对象的stub中只提供一个监听线程,由这个线程根据不同的事件号调用相应的客户callback句柄.这样,stub中只需处理两个线程的并发控制,降低了死锁发生的可能性.而且由于同一结点上同时运行的线程数减少,处理机的overhead也相应降低.

2.5 继承问题

很多主动对象的实现都避而不谈继承问题,因为它们将控制线程等表示主动行为的机制固定在类定义中,

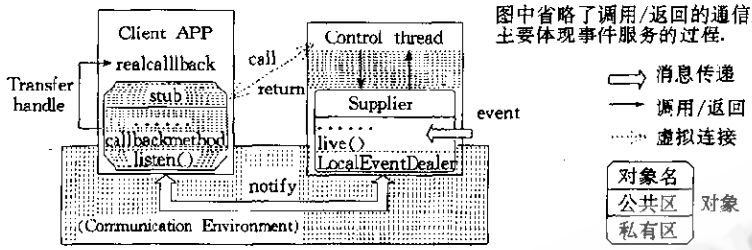

图2　适合并行计算的事件服务的框架

这种实现方式不利于继承.而我们的主动对象模式则不同,定义 **Active** 类和定义传统的 passive 类没有区别,控制线程由 DOPPF 自动生成.分布属性只是作为运行时的一种属性而对类的继承和对象的类型没有影响.至于并发机制则完全由用户控制,当前这方面的研究进展甚微,各种并发机制继承方式带来的复杂性往往要大于提供给用户的好处.如第 2.4 节所述,DOPPF 在客户端的尝试是减少不必要的并发线程数量,以此降低并发控制的复杂程度.因此,**Active** 类完全像 passive 类一样,可支持继承.但由于加入了 **Active** 类,类继承的关系比以前复杂.大致说来,可分为下列 4 种情况:(1) 父类为 **Active**,子类为 passive;(2) 父类为 passive,子类为 passive;(3) 父类为 passive,子类为 **Active**;(4) 父类为 **Active**,子类为 **Active**.其中第 2 种为最一般的继承情况,不加以讨论.其他 3 种情况都涉及到主动对象类的继承,其语法、语义和被动对象类的继承大致相同.考虑到虽然主动对象的行为是在另一个地址空间发生,它的对象类的子类实例中这些行为未尝不可在本地发生,我们在 DOPPF 中增加 4 个行为修饰关键字 **passively**,**actively**,**make_active_method** 和 **make_active_variable**.**passively** 表明无论父类是主动还是被动的,子类中父类的行为都在当前进程的上下文中运行.**actively** 则相反,无论父类是主动还是被动的,子类中父类的行为都以另一个进程的形式(远程或本地)在不同的地址空间运行.后两个关键字介于前两者之间,父类只有部分行为需要在远程发生,即那些由这两个关键字指明的方法和变量.

这种继承机制使得 supplier 可以继承别的主动对象类属性,从而使得 supplier 可以将某些客户请求分布到其他处理机上同时运行,获得对象内的并行性.

下面分别讨论这 3 种情况:

(1) 父类为 **Active**,子类为 passive,语法为

class ⟨class_name⟩[public|private protected]⟨base_name⟩[actively|⟨make_active⟩].

缺省的行为修饰是 **passively**,表示子类的实例是纯粹的被动对象,所有行为均在本地发生.行为修饰为 **actively** 时,表示父类中的远程行为依然在远程发生,而子类自身的行为在本地发生.⟨make_active⟩修饰可将父类的部分行为保留在远程发生.这也是继承方式实现的对象内并行.

(2) 父类为 **Active**,子类为 **Active**,语法为

Active class⟨class_name⟩[public|private|protected]⟨base_name⟩[**passively**|⟨make_active⟩].

缺省的行为修饰是 **actively**,表示子类是纯粹的主动对象,所有行为均发生在远程.行为修饰为 **passively** 时,表示父类的远程行为在本地发生,子类自身定义的行为在远程发生.⟨make_active⟩作用同上.

(3) 父类为 passive,子类为 **Active**,语法为

Active class⟨class_name⟩[public|private|protected]⟨base_name⟩[**actively**][⟨make_active⟩].

缺省的行为修饰是 **passively**,表示父类的行为在本地发生,而子类的行为在远程发生.行为修饰为 **actively** 时,表示子类和父类的行为都在远程发生.⟨make_active⟩作用同上.

2.6　类　库

类库是 OO Framework 的重要组成部分,DOPPF 中的类库分为系统级和算法级两部分.系统级主要提供分布对象的异步事件服务(event service)、并发控制、事务服务(transaction service).算法级的类主要提供一些常用封装好的并行算法,比如矩阵乘、二维图像平滑算法、常微分方程求解等.由于工作站网络环境计算模式相对各种并行机而言,各处理机的连接方式比较统一,因此,工作站网上的并行算法具有比较好的可扩展性

(scalability),适于封装和复用.

2.7 DOPPF的实现技术

使用DOPPF编写分布并行应用程序的过程是这样的:

首先,定义并实现服务方supplier类,注意在定义时用规定的语法描述它的主动特征.其次,用DOPPF预处理器处理supplier类的定义文件,生成客户和服务两方面的stub代码,stub中的通信调用使用PVM提供的函数.再次,实现客户方应用程序,使用supplier的地方,都用client stub中的Stub类替代.最后,分别编译客户和服务方程序.

一个分布并行应用程序的生成过程如图3所示.

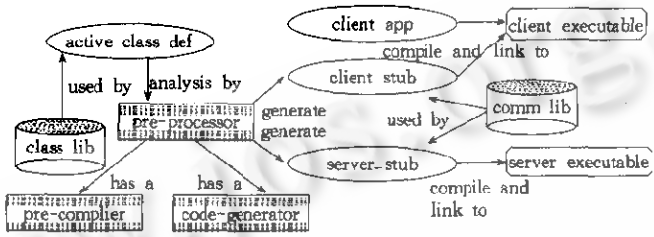

图3 分布并行应用程序生成过程

3 计算实例及分析

Master-Slave是一种使用很广的并行计算模式.在DOPPF中,Master-Slave可以这样理解:客户应用是一个大计算量任务,可分为多个小任务,每个小任务都由一个主动对象(slave)管理.客户应用本身也可能是一个类的实例,Master对象包含几个主动对象分别并行完成一部分任务.Master与Slave可以是同构的(具有相同的类定义,但具有不同的主动或被动修饰),也可以是异构的(具有不同的类定义).下面的例子使用了这种计算模式,通过这种方式,算法获得了较好的封装和复用能力.

3.1 Mandelbrot算法

我们在Ethernet连接的SGI indy,SUN sparc20工作站网上测量了此算法的加速比,最大迭代次数设定为1 024,分别在大小为800*800,400*400,200*200的窗口中显示M集的图像.结果如表1(其中每列表示所用工作站台数)和图4、5所示.2个处理机时并行处理效率比4个时高一些,主要原因是2个处理机时的任务分配是均衡的,而4个处理机时略有不均.数据量大时加速比较高是由于Ethernet上的小报文传递overhead与大报文相差无几.

表1

Speedup	P1	P2	P4	Efficiency	P1	P2	P4
800*800	1.00	1.90	3.63	800*800	1.00	0.95	0.91
400*400	1.00	1.84	3.63	400*400	1.00	0.92	0.91
200*200	1.00	1.74	3.41	200*200	1.00	0.87	0.85

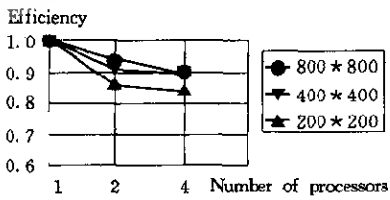

图4 Mandelbrot算法效率曲线

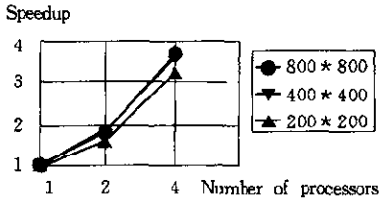

图5 Mandelbrot算法加速曲线

3.2 矩阵乘法

矩阵乘法是大运算量、大通信量的例子. 我们试图从中观测工作站网上计算这类问题的效率. 采用分块算法，$C=[A_1^T,A_2^T][B_1,\ldots,B_{block\_num}]$. 通信量为$\left(\frac{P}{4}+3\right)*n^2*commcost$(通信开销随处理机数的增加而增加). 如表 2 和图 6,7 所示.

表 2

Speedup	P1	P2	P4	P8	Efficiency	P1	P2	P4	P8
128 * 128	1.00	1.34	1.72	2.67	128 * 128	1.00	0.67	0.43	0.33
256 * 256	1.00	1.58	2.22	3.53	256 * 256	1.00	0.79	0.56	0.44
512 * 512	1.00	1.81	3.10	4.94	512 * 512	1.00	0.91	0.78	0.62
640 * 640	1.00	1.75	3.26	5.15	640 * 640	1.00	0.87	0.82	0.69

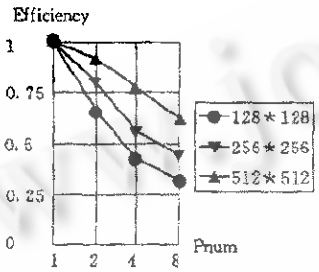

图6　矩阵乘法的效率曲线

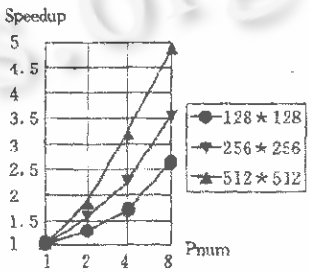

图7　矩阵乘法的加速曲线

表 3、图 8 和图 9 是 128 * 128 的矩阵乘在专用高速网络连接的 TRANSPUTER 上和工作站网上的加速和效率比较.

表 3

Speedup	P1	P2	P4	P8	Efficiency	P1	P2	P4	P8
T128 * 128	1.00	1.48	2.47	2.17	T128 * 128	1.00	0.74	0.62	0.27
128 * 128	1.00	1.34	1.72	2.67	128 * 128	1.00	0.67	0.43	0.33

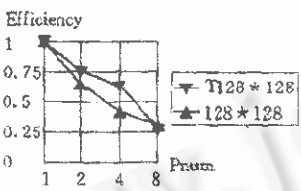

图8　矩阵乘法的效率曲线
(TRANSPUTER和NOW的比较)

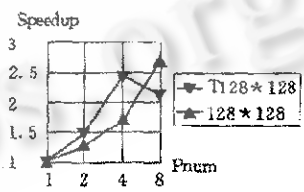

图9　矩阵乘法的加速曲线
(TRANSPUTER和NOW的比较)

通信量的上升是处理机增加而并行处理效率下降的主要原因. 最后给出的是 TRANSPUTER 上进行矩阵运算(128 * 128)的加速比，由于 8 个处理机的 TRANSPUTER 采用环形连接，因此，数据分布的传递路径过长，每个处理机都要参与这些数据传递，导致加速比下降. 另外，TRANSPUTER 上的并行程序设计语言采用 CSP 模型，使传递子矩阵的处理机不能重叠通信时间进行计算，这也是 TRANSPUTER 这类专用网络连接的并行机在某些问题上低效率的原因.

因为框架生成的通信代码是基于 PVM 的，因此，与直接使用 PVM 的非 OO 程序相比，上面两例的效率与其相差不大.

4 结　论

由上可以看出：(1) 在工作站网环境下进行并行计算是能获得比较好的性能的；(2) 利用分布对象构造并行

程序设计框架不会引起并行计算 overhead 的很大增加,或者说它不是降低并行效率的主要因素;(3) 分布对象的并行程序设计框架可以使程序员注重描述问题领域状况,替他们自动完成了通信的匹配过程;(4) 提供多样的分布对象方法调用语义,可以使程序员能根据不同问题选取最好的方式;(5) 面向对象方法提供了良好的软件复用能力,工作站网又具备很好的可扩展性,因此,构造通用的并行算法类库,使用 OO 方法封装复用并行算法能使分布的处理资源得到更加充分的利用,并提高并行软件的开发质量和效率.

参考文献

1 Vinoski S. CORBA: integrating diverse applications within distributed heterogeneous environments. IEEE Communications Magazine, 1997,35(2):46~55

2 Nierstrasz O. Composing active objects. In: Agha G, Wegner P, Yonezawa A eds. Research Directions in Object-based Concurrency. Cambridge, MA: MIT Press, 1993. 151~171

3 Java Remote Method Invocation Specification. available in: ftp: //ftp.javasoft.com/docs/jdk1.2/rmi-spec-JDK1.2ps, 1998

4 Sparks S, Benner K, Faris C. Managing object-oriented framework reuse. IEEE Computer, 1996,29(9):52~61

5 Langendoen K, Bhoedjang R, Bal H. Models for asynchronous message handling. IEEE Concurrency, 1997,5(4):28~38

A Distributed Object Based Framework for Parallel Programming

WANG Chen ZHOU Ying ZHANG De-fu

(*State Key Laboratory for Novel Software Technology Nanjing University Nanjing* 210093)

(*Department of Computer Science and Technology Nanjing University Nanjing* 210093)

Abstract The computational and compositional features are very important in the construction of software for the workstation clusters. However, due to the lack of suitable supporting environment of software development, most existing distributed parallel software systems are weak in these two aspects, especially in the compositional feature. In this paper, a distributed object based framework for parallel programming is proposed. The goals of the framework are: first, getting high parallel computing efficiency; second, constructing a mechanism to encapsulate and reuse parallel program. The framework is tested by some parallel algorithms, the results indicate that the framework is helpful.

Key words Distributed object, parallel programming, workstation clusters, framework.