一种并行分布对象的互操作模型*

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@nju.edu.cn

摘要 并行软件设计本身的复杂性使它的复用成为一个引人注目的问题.分布对象技术不仅可以将并行软件封装成相应的构件,而且提供了利用各种异构系统进行并行计算的可能性,但这样往往会使这些构件的互操作的效率有所降低.文章提出的并行分布对象互操作模型试图解决这一问题.这个模型与分布对象的旧有模型兼容,并且实际测试结果表明,它还能挖掘出并行分布对象间的更多并行性.

关键词 并行计算,分布对象,互操作,数据并行,任务并行.

中图法分类号 TP311

随着计算机网络在人们生活中的地位越来越重要,也随着软件工业的迅速发展,分布对象的思想和技术标准(OMG的CORBA[1])已日渐为人所接受,但要使这种思想和技术标准真正并且充分地发挥作用却需要使它们适应尽可能多的应用领域,满足这些领域的特定需求.这方面的研究是当前分布对象技术的主要研究方向,涉及内容包括CORBA的实时性[2]、容错性[3]和对移动计算[4]的支持等.近年来,基于分布对象的并行计算也引起了较多关注[5~7],这主要有以下几个原因.

(1) 面向对象技术是改变旧有并行程序设计方法粗糙状况的一种有效手段.

(2) 分布对象技术本身的特征是解决并行软件互操作问题的一种有效手段.

(3) 使用分布对象技术能够将分散在异构系统中的处理资源组织起来并发挥它们的并行计算能力,这样的异构系统可能是大规模的并行计算机处理机群,也可能是高速网络连接的工作站网络.

一些基于异构大型计算机的协作计算展示了分布对象技术在这方面的前景[8].

对于通常的分布对象来说,接口定义语言(interface definition language,简称IDL)所描述的接口方法提供了客户足够的信息来满足自己的需要.但如果这些分布对象采用并行方式来实现,而客户出于效率的考虑要直接与并行实现体中的某一个进程或线程协作完成一项工作时,接口定义语言就很难给出令客户满意的信息了.为此,我们给出了一种并行分布对象的互操作模型,使分布对象的并行实现之间能够比较充分地互相连接,保持比较好的并行效率.

与此相关的研究有两类.一类是解决CORBA本身的实现效率,如RT ORB[4],工作做在底层.另一类试图用CORBA来解决数据并行程序的互操作,如PARDIS[5],POOMA[9],这类工作都才开始进行.与之相比,我们的模型概括了任务并行和数据并行两种方式,提出了一种统一的互操作模式,也就是说,这种模型具有更强的互操作能力.

---

* 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,主要研究领域为分布对象技术.周颖,女,1973年生,硕士,主要研究领域为分布对象技术,Internet计算.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.

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1998-05-21收到原稿,1998-09-21收到修改稿

1 分布对象的并行实现

现在较为常用的并行程序设计模式以多程序多数据流(multi-program multi-data,简称MPMD)和单程序多数据流(single program multi-data,简称SPMD)为主,前者发掘任务并行性(task parallelism),后者发掘数据并行性(data parallelism),两者组合应用于并行程序设计也比较常见.任务并行性通过几个任务并发计算不同的数据集合来获取,每一个集合上的运算被称为一个任务,相反地,数据并行性通过并发计算同一个数据集合的不同部分而获得.几个数据并行的任务的组合可以成为一个任务并行模式,几个任务并行的任务的组合也可以构成更大的任务并行模式,而几个数据并行的任务和几个任务并行的任务之间也能组成一种任务并行模式.这种组合是现在并行计算实际应用中很常见的现象,因为计算问题通常不是很单一的,各种组合都能够发掘出更多的并行性.但并行计算机、并行计算机群或者并行支撑环境之间的差异使得在很多情况下的组合并行计算变得十分复杂,很难做到有良好的扩展性.而分布对象技术正好可以弥补并行软件在这方面的不足,它可以通过一种中介语言(IDL)将各种异构分布对象无缝地集成为一个系统,IDL通过编译器映射成各种具体的实现语言.

为了描述的严密和进一步探讨的方便,我们借助CCS(calculus of communicating system)[10]来描述分布对象的并行实现.分布对象可以表示成如下形式[11,12]:

$$O=((V1|V2\ldots|Vn|M1|M2\ldots|Mk)\backslash ACC(V1)\backslash\ldots\backslash ACC(Vn)|OB)\backslash ACC(M1)\ldots\backslash ACC(Mk),$$

其中Vi表示域变量,Mi表示对象中的接口方法.OB表示一种进程间消息传递到方法调用的映射方式,在实现中通常是一个调度器(scheduler).$ACC(V_i)$表示方法对变量的存取,$ACC(M_i)$表示调度器对方法的激活."|"是并发代理的组合(compose)操作符,"\"是限制操作符(restriction),用于对外部观察者隐藏相应的内部事件.

这里,$V=put_v x.Loc_v(x)$,其中$Loc_v(y)=put_v x.Loc_v(x)+\overline{get_v y}.Loc_v(y)$."+"表示一种或关系,$Loc(y)$是存放$y$的地址.这个定义表明,首次使用变量前必须对其进行初始化.

$$M=call_{Mj}.arg_{Mj1}x1.\ldots.arg_{Mjn}xn.Body_{Mj}(x1,\ldots,xn)|M.$$

对于方法间互斥执行的分布对象而言,其调度器定义为

$$OB_O=\sum_{C,M,j}(call_{C.O.M.j}a1\ldots an.\overline{call_{M.1}}.\overline{arg_{M.}a1}\ldots\overline{arg_{M.1}an}.\overline{reply_{M.1}}x.reply_{C.O.M.j}.x.\overline{done_{M.1}}.OB_O),$$

其中下标C标志调用对象O的客户,M标志C所调用的对象O中的方法,j标志同一客户对此对象方法的各次调用.这表明了每次调度器收到客户的调度请求后($call_{C.O.M.j}a1\ldots an$)将调度请求传递给相应的方法($\overline{call_{M.1}}$),它总是等待方法执行完返回之后才接收另外的客户请求.

对于方法间可以并行执行的分布对象而言,其调度器定义为

$$OB'_O=\sum_{C.M}(call_{C.O.M.1}a1\ldots an.\overline{call_{M.1}}.\overline{arg_{M.1}a1}\ldots\overline{arg_{M.1}an}.$$
$$reply_{M.1}.x.reply_{C.O.M.j}x.\overline{done_{M.1}}.OB_O)|OB'_O[f_{o.M}],$$

其中$f_{o.M}$是个替换函数.

$$f_{o.M}=\{call_{C.O.M.i+1}/call_{C.O.M.i},call_{M.i+1}/call_{M.i},arg_{M.i+1}a_k/arg_{M.i}a_k,done_{M.i+1}/$$
$$done_{M.i},reply_{C.M.i+1}x/reply_{C.M.i}x\}.$$

通过替换,调度器可以标志出来自不同客户的多个请求,并同时处理它们,由此可以得到一种对象内方法间的并行性,这是一种任务间的并行.但上面给出的V的定义中并不能保证并行执行的顺序性,如果第i次方法调用中会写变量Vi,这个结果未必能反应到随后执行的第$i+1$次方法调用中,因为第$i+1$次方法与第i次方法并行执行,而且可能在第i次方法写变量之前就使用了变量Vi.下面的定义通过一个方法依赖关系分析器给出两个消息$nordep(j)$和$rdep(j)$,这两个消息分别表示方法j和j之前执行的方法对变量V不存在和存在依赖关系,由此保证方法间并行执行的顺序性.当并行执行的方法j存取变量V时发现与前面启动的方法存在读写依赖关系,并且前面方法尚未完成对变量V的写入,则阻塞等待前面启动方法写入的完成消息.改进后的变量定义如下:

$$Loc_v(y)=(put_{v,j}x.(nordep_vj.\overline{block_v(i>j)}.Loc_v(x).+rdep_vj.block_vj.Loc_v(x))+\overline{get_{v,j}y}.Loc_v(y))|Loc_v(y).$$

从上面给出的分布对象并行执行方式我们可以看出,对于以任务并行方式实现的分布对象,客户请求完全通过调度器散发到各个方法执行,各个方法的执行结果也通过调度器回收并转发给客户,这不仅增加了通信开销,而且使调度器成为通信的瓶颈,会导致分布对象协作起来后效率降低.对于数据并行方式实现的分布对象,客户提供的原始数据也需要通过调度器传递给某个方法,由这个方法将数据传递到各个结点,结果的返回也有一个调度器的回收过程,这也形成了一个瓶颈.

要解决这个问题,需要对并行分布对象作一些特殊处理,使得各个并行执行体能够为客户所感知,同时又要保持它与CORBA等分布对象框架的兼容性.

对分布对象中数据并行的实现,可以将分布数据表示为*DSeq*.

$$DSeq=Vi1|Vi2...|Vin|Mi1|Mi2...|Min\backslash ACC(Vi1)\backslash ACC(Vi2)...\backslash ACC(Vin),$$

其中$Vi1,...,Vin$是变量Vi分散在各个结点上的部分,$Mi1,...,Min$则是Mi(存取数据的方法)运行在各个结点上的部分.事实上,可以把*DSeq*看做一组由Mik组成的并行的任务.

我们将任务并行的分布对象表示为*PObj*.

$$PObj=((V1|V2...|Vn|M1|M2...|Mk)\backslash ACC(V1)\backslash...\backslash ACC(Vn)|OB'|M1'|M2'...|Mk')\backslash ACC(M1)...\backslash ACC(Mk),$$

其中$M'j=call_{C,O,Mj,port,1}.a1...an.\overline{call_{Mj,1}}.\overline{arg_{Mj,1}a1}...\overline{arg_{Mj,1}an}.\overline{reply_{Mj,1}}x.reply_{C,O,Mj,p}.x.\overline{done_{Mj,1}}.M'j$.

外部消息$call_{C,O,Mj,port}.a1...an$与$call_{C,O,M,1}.a1...an$略有不同,需要给出某个方法对应的端口号,也可以将后者(递交给*OB*的消息)看做对某个特殊端口的调用,这个端口可以处理所有的方法调用请求,是一个通用端口.而前者是专用端口,只能处理对绑定它的方法的请求.由于方法的调度是动态的,端口信息也时常变化,我们给出一个port对象来刻画这种变化.

$$Port_{mj}(y)=put_{mj}x.Port_{mj}(x)+\overline{getp_{mj}y}.Port_{mj}(y).$$

如图1(a)所示,当某个方法mj被调度器在某个处理结点启动之后,*PObj*设置相应方法的端口信息,客户需要使用端口直接请求该方法的服务时,从*Port*获取这个信息,然后向这个端口送出*call*请求,不需要直接通过端口获取服务时,客户可以把端口设为一个特殊值送出调用请求,这个请求会先传递给*PObj*的调度器,然后由调度器分发给相应方法.为保持与原有分布对象调用方式的兼容,我们将port服务从(a)转换到(b).容易知道,对于客户来说(外部观察者),即两种表示方法对观察者等价.首先,$getp_{mj}$也可以看成是一种call调用,因为不改变*PObj*状态,不影响其他调用的结果;其次,调度器*OB*的转发消息对外部是透明的.属于内部状态转换;另外,在把*port*视为一个参数的情况下,$M'1|M'2...|M'k$和*OB*的入口消息相同;再次,通过$M'i$通道传递到Mi的消息最终得到的状态与通过*OB*通道传递到Mi的消息是一致的(因为Mi的消息接口对*OB*和$M'i$是相同的),这就意味着通过$M'1|M'2...|M'k$通道的消息使分布对象本身能达到的状态与这些消息通过*OB*通道使分布对象达到的状态对外部观察者是等价的.

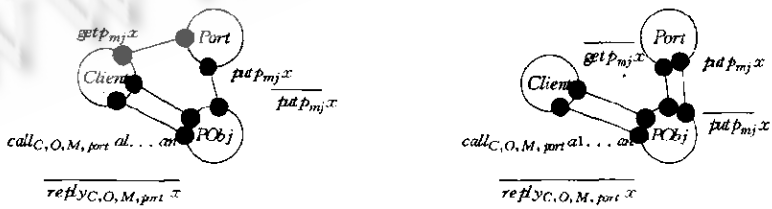

图1　*Port*与*client*和*PObj*的关系

下面,我们将详细说明这个模型的实现.

2 互操作模型

为实现数据并行对象和任务并行对象的互操作，我们给出了两个类：*DSeq* 和 *PObj*. 前者是数据并行程序中分布数据的一种统一接口，可以作为任务并行实现的分布对象的父类；后者对并行实现的对象的互操作接口进行了抽象.

2.1 以任务并行方式实现的分布对象的互操作

图2(b)中给出了 *PObj* 的结构. 黑色箭头代表通常的分布对象将网络消息映射到对象方法的消息传递途径，白色箭头代表并行程序间的互操作通道. *PObj* 中定义了一个类型 *port*，*port* 中包含了客户直接调用 *PObj* 中已被调度器调度运行于某个处理器的方法的相应信息.

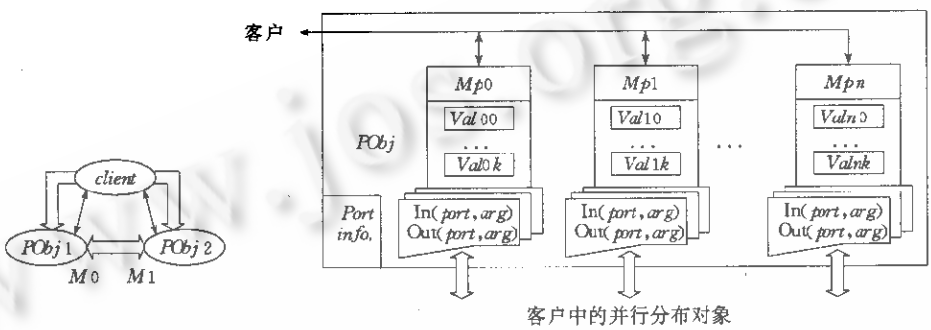

图2 任务并行分布对象的互操作模型

```
class port
{
private:
    addr_t          addr;       //该方法依附的进程地址
    int             occupier;   //当前使用此端口的客户 id
    para_list       pl;         //输入此端口的参数表
    ret_type        rt;         //返回类型
    int             ret_len;    //返回数据长度
public:
    int     in(VAL *arg);       //向端口写入相应的数据
    int     out(VAL *ret);      //从端口取出相应的数据
    int     bind(port p);       //与另外的端口相连
};
```

其中的地址部分由全局进程通信地址(主机名和进程号)、对象名和方法名这3部分构成. 这个定义既包含了输入又包含了返回，实际上是一个双向端口，可以表示上面所说的 *call* 和 *reply* 消息. 相应于 *port* 的操作是 in，out 和 bind. in 传递对方法的调用请求和调用参数，out 则从端口取出返回值，bind 可以将端口串连起来. 由图2可以看出，方法可以被多个 *port* 和调度器传递来的调用请求同时激活，产生并发执行. 我们限制了这种在同一个处理器上的并发，以减少用于同步的额外开销. 方法的端口在一个客户释放之前对其他请求关闭，只有调度器能生成互相并行的对象方法执行体.

PObj 分别为调度器和客户提供了一个调用接口：

```
class PObj
{
protected:
    int     bind_port(int method_id);       //供调度器记录并行执行方法的端口信息
    void    unbind_port(int_method_id);     //供调度器释放端口
```

```
    int       dispatch(int method_id, int node_id);     //将方法分布到某个处理结点
public:
    id_list   get_method_id(String str_method_call);
    port      get_port(int str_method_id);              //供客户根据方法号获取相应端口信息
};
```

这里的 method_id 对应于方法的每一个并行执行体,通过 get_method_id 可以从方法的调用形式(方法名加上参数格式,返回类型的串)获得相应的 method_id 值.一个调用形式可能对应于多个 method_id,因为同一个方法可能被调度器同时指派到多个处理器上运行.客户可以从中任意选取一个发送请求,也可以调用运行系统提供的负载评价机制以获取负载最轻的结点发送请求.

客户请求到达之后,调度器将它所请求的方法运行在空闲处理器上,并调用 bind_port 将方法的端口信息记录下来.端口信息表的变化通过分布对象事件服务[13]的形式通知客户,以后当客户需要直接向这些已经启动的方法请求服务时,再通过端口操作发出请求并传递参数.当调度器传递的请求执行完且端口中没有其他请求的情况下,方法执行体消亡,相应的端口通过 unbind_port 被调度器释放.

对于客户方同时也含有另一个或多个任务并行对象的情况来说,端口的绑定是很有用处的.如图2(a)所示,两个对象可以将一些方法执行体的端口连接起来,*PObj1*中的方法 *m*0可以与 *PObj2*中的方法 *m*1在某一个端口相连,这样,*m*0端口的输出就成为 *m*1端口的输入,而其输入也就是 *m*1端口的输出.同时,客户(client)也可以有两条途径来获取分布对象的服务.通过这种方式,我们可以使任务并行分布对象间获得比较全面的互操作能力.

2.2　以数据并行方式实现的分布对象的互操作

数据并行方式实现的分布对象的互操作主要提供分布数据的互操作,也就是说提供这些数据的多种传输途径,这些数据为并行语句使用.我们把这些分布的数据看做一个对象.图3给出了相应的类结构,这个结构跟图2相似,因为分布数据对象也可以看做在各个结点上向应用程序提供数据服务的任务并行对象.与图2不同的是,这个 *DSeq* 对象融合了许多数组操作和数据分布函数,另一方面,*DSeq* 的方法不由调度器动态分配到各个处理器,而是在对象构造时随数据分布到各个处理结点(通过 dispatch)并向数据并行程序提供服务.从 *DSeq* 中还获得数据分布的信息(*get_part_info*).Partition 允许客户给出数据的划分比例(如 *part.strsplit*="1:2:2:4"表示将数据按1:2:2:4的比例分布到4个处理结点上),也可以给出按行或列划分的说明(如 *part.rowsplit*=*BLOCK* 表示将数据按行依次放置到各个处理结点,行的长度为 *BLOCK*),每一个数据块包含一个端口倾听应用程序的数据请求,并向它们发送相应数据,还包含一个与调度器的接口,允许客户向其分发数据.

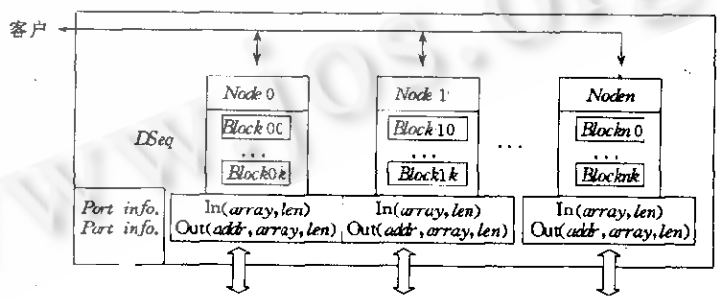

图3　数据并行分布对象的互操作模型

DSeq 的接口部分如下:

```
class DSeq
{
public:
    DSeq(T data, int len, Partitions part);      //根据 part 初始化数据的分布
    ~DSeq();                                     //终止各个结点的服务程序执行
    DSeq& operatore=(const DSeq &d);
```

```
    Port   operator[](int index);                    //重载下标操作符,使之返回相应数据结点的端口
    T *    local_data(Port);                         //从本地端口读取数据
    Int    local_length(Port);                       //获取本地数据长度
public:
    Partitions get_part_info();                      //获取数据分布信息
    Int    updatedata(T data, int len, Partition part);  //更新数据对象的数据
};
```

这个类重载了相应的操作符,使并行语句存取数据资源时能获取数据所在结点的端口,从而通过端口读写对数据进行操作.

3 实例及性能分析

我们使用上面提出的模型构造了2D-FFT算法,并在Ethernet连接的4台SGI indy工作站网络上测出实际运算结果,把这些结果与通常的做法进行了比较.

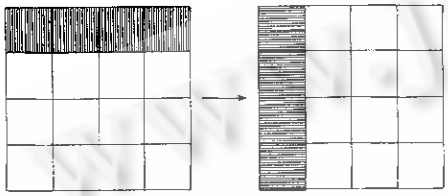

图4 2D-FFT的并行计算示意:黑色部分表示放在处理结点0上的数据

如图4所示,通常的2D-FFT并行算法先由master将数据按行分布到各个处理结点,各个结点上的slave同时计算各行的FFT变换值,然后搜集计算结果到master,再将结果矩阵转置,把转置矩阵再按行分到各个slave计算,最后回收的结果就是整个矩阵的二维快速傅里叶变换.显然,回收发送的过程是并行计算的瓶颈,网络传输时间无法被计算所重叠.

为了使网络传输最大限度地为计算时间所覆盖,我们将4台工作站分为两组,每一组上运行一个分布对象,第1组并行计算一个512×512矩阵每一行的FFT,计算完后回收结果进行转置,然后传递到第2组的分布对象每一个运行方法的端口,第2个对象的再方法并行计算这些值的FFT,并输出结果.两个分布对象的互操作构成了一种流水线结构,部分覆盖了网络传输时间.表1给出了这种方法的计算时间、使用一个分布对象计算的时间,以及使用两个分布对象通过调度器传递数据计算时间的比较,表2给出了相应加速比的比较.

表1 计算时间比较

计算次数	使用一个分布对象的并行计算时间(μs)	使用两个分布对象的并行计算时间(通过调度器通信)(μs)	使用两个分布对象的并行计算时间(通过端口通信)(μs)
4	58 522 953	47 210 621	43 866 150
8	118 216 291	103 456 452	94 836 858

表2 并行加速比比较*

计算次数	使用一个分布对象的并行计算加速比	使用两个分布对象的并行计算加速比(通过调度器通信)	使用两个分布对象的并行计算加速比(通过端口通信)
4	1.43	1.76	1.90
8	1.38	1.58	1.72

从表中可以看出,使用两个分布对象的互操作模型计算2D-FFT比只使用一个分布对象并行计算的加速比高,而通过端口通信减少了通过第2个分布对象调度器传递数据的时间,因而能获得比仅通过调度器进行互操作

* 整个加速比较低的主要原因是,通过我们测试用的10Mbps的Ethernet传递一个数组的通信开销相对于这个数组的计算时间过大,无法使计算时间完全覆盖通信时间,而这种情况在网络带宽有所提高的情况下会明显改进.另外,随着计算次数的增加,加速比下降的原因也是由于通信时间在整个计算过程中占了无法覆盖的比重.计算次数越多,通信所用掉的时间也就越多.

更高的加速比.这也表明,我们的互操作模型是有效的.

4 结 论

并行软件设计本身的复杂性使它的复用成为一个引人注目的问题.分布对象技术不仅可以将并行软件封装成相应的构件,而且提供了利用各种异构系统进行并行计算的可能性,但这往往使得这些构件的互操作的效率有所降低.我们提出的并行分布对象互操作模型试图解决这一问题.这个模型,如第1节所述,是与分布对象的旧有模型观察等价的;如第3部分的实例所示,这个模型能够挖掘出并行分布对象间更多的并行性.

参考文献

1 OMG. The common object request broker: architecture and specification. February 1998, http://www.omg.org

2 Schmidt D C *et al*. A high-performance end system architecture for real-time CORBA. IEEE Communications Magazine, 1997,35(2):72~77

3 Maffeis S *et al*. Constructing reliable distributed communication systems with CORBA. IEEE Communications Magazine, 1997,35(2):56~60

4 Chen Larry T, Suda Tatsuya. Designing mobile computing systems using distributed objects. IEEE Communications Magazine, 1997,35(2):62~70

5 Keahey Katarzyna, Gannon Dennis. PARDIS: a parallel approach to CORBA. Technical Report, ftp://ftp.cs.indiana.edu/pub/techreports/TR475.ps.Z

6 Nierstrasz Oscar. Regular types for active objects. In: Nierstrasz O, Tsichritzis D eds. Object-oriented Software Composition. London: Prentice Hall, Inc., 1995. 99~121

7 Keahey Katarzyna. CORBA compliance and parallel supercomputing. http://www.acl.lanl.gov

8 Norman M L, Beckman P *et al*. Galaxies collide on the I-WAY: an example of heterogeneous wide-area collaborative supercomputing. International Journal of Supercomputer Applications and High Performance Computing, 1997,10(2):132~144

9 Williams T J, Reynders J V W *et al*. POOMA User Guide. http://www.acl.lang.gov/pooma/doc/userguide/pooma-UserGuide.ps

10 Milner R. Communication and Concurrency. London: Prentice Hall, 1989

11 Nierstrasz Oscar. Towards an object calculus. In: Tokoro M, Nierstrasz O, Wegner P *et al* eds. Proceedings of the ECOOP'91 Workshop on Object-Based Concurrent Computing. LNCS612, Geneva (Switzerland): Springer-Verlag, 1992. 1~20

12 Nierstrasz Oscar, Papathomas Michael. Towards a type theory for active objects. ACM OOPS Messenger, 1991,2(2):89~93

13 OMG. CORBA services: common object service specification. November 1997, http://www.omg.org

A Model for Interoperability of Distributed Objects' Parallel Implementations

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 complexity of developing parallel software makes it attractive to research their reusability. The distributed object technology provides some benefits in this field because it can encapsulate parallel programs to components and exploit the parallelism in the various heterogeneous systems. However, the drawback to use this technology to do parallel computing is obvious because of the limit interoperability among those components. A model of interoperability is proposed in this paper. This model keeps compliant with the common distributed object model while gaining more parallelism from the interoperability of distributed objects.

Key words Parallel computing, distributed object, interoperability, data parallelism, task parallelism.