.pl51
'^2'
'&27'

'@l'传统数据流模型的分析及改进'@A'

'@n'王永革'@A'
'@n'（南开数学研究所）'@A'

'@M'摘要

数据流计算机作为新一代并行机迅速发展起来，但由于数据流模型本身的局限性，使得商用数据流机器的制造还难于普及。本文通过分析运算的操作字符与非操作字符，改进传统模型的点火规则、引入变量并拓广变量的概念，定义了粗粒度数据流模型ＣＤＦＭ（Coarse granularity Data Flow Model）。在文章的最后一节将这种模型运用于［Hu90］中的ＵＳ　ＭＭＣＭ机器，讨论了如何用模型ＣＤＦＭ以最大的并行性去解决实际问题。'@A'

'@p'§０．引　言'@A'

自从数据流模型问世以来，无论在其理论还是应用方面都得到了迅速发展。特别地对于人们认识“并发”概念起了很大推动作用，直观上讲：事件（计算单元）的发生在时间上连续的是串行计算；事件的发生在时间上有时重叠，对于时间上重叠的这部分在空间上分离的，是并行计算；而并发则是指程序的可能并行执行。在并发计算中，程序的执行有时是串行的，有时是并行的。时间是相对于观察者而言的，也就是说，时间是相对于具体的数据流动而言的。数据流模型帮助人们认识到并发的这一最本质特征，并以最大的可能性利用了算法的静态并行性，所以可以说它在理论上是完美的。但由于数据流模型的充分并行性是以结点的细粒度实现的，因而势必导致了通讯费用的巨额增长，使得难以制造出高效率的数据流机器。近年来，国外很多人尝试通过减少结点的细粒度来降低通讯费用，但这会使模型充分利用并行性这一优点迅速降低。分析其主要原因，是因为人们对“数据准备好”、“数据依赖”这一概念的理解有一定局限性，传统模型中以运算对象都以赋值来理解“数据准备好”。胡国定在［Hu88］、［Hu90］中通过对运算的分析指出：运算对象可分为操作字符于非操作字符，因此我们觉得以如下方式理解“数据准备好”则更好些：运算对象中操作字符都已赋值；而非操作字符不一定赋值，只知道其存放位置就可认为“数据准备好”。基于这一理解，我们引进了粗粒度数据流模型ＣＤＦＭ（Coarse grunularity Data Flow Model），对传统数据流模型作了如下改进：

１．结点的粗粒度性；

２．结点的非纯函数性；

３．结点有内部状态；

４．类似于控制流模型，引入变量的概念并拓广变量的传统理解：即变量在未赋值之前是可以访问的；

５．点火规则的改进：可部分点火。

模型ＣＤＦＭ不仅具有充分并行的性质，而且还具有通讯费用低的优点。　可以用于目前迅速发展起来的“工作站”，并有助于新一代数据流机器的设计。

本文的结构是这样的：§１．分析传统模型的不足之处；§２．给出模型ＣＤＦＭ的定义；§３．ＣＤＦＭ的应用方法及实例。

'@p'　§１．传统数据流模型'@A'

传统的数据流模型为了最大地利用算法本身的静态并行性，依靠数据依赖性决定运算执行的先后次序。虽然各种数据流模型有自己的特征，但它们都具有下边两条共性：（参阅［IE82］）

性质１．细粒度性：每个结点只包含一个简单的算术、逻辑或控制运算。

性质２．结点的函数性：结点没有内部控制状态。结点的运算仅代表一个纯粹的函数，即输出托肯（tokens）仅仅依赖于当前的输入托肯。

这两条性质使得数据流模型具有足够的简洁性与充分的并行性，目前已造出的数据流机器都具有上述二性质（参阅［AN86］、［SK86］）。但是，也正由于这两条性质，使得数据流机器的局限性很大，主要是通讯费用的增长问题。下边作一具体分析。

一、细粒度性导致了巨额通讯费用

在传统数据流模型中，为了最大地利用算法的静态并行性，一项任务（结点）仅仅是一条机器指令。但是，物极必反，在这种过分追求细节的模型中，为了保持运算的正确性，必须花大量的时间去做许多额外的工作：如数据的频繁传递和反复测试各可动结点、决定执行何种任务、以保持其同步性等等。因而反倒不如以前串行算法的速度快，丧失了并行的意义。而且还得指出的是：由于模型的细粒度性，使得在目前的硬件技术下，很难制造出高效率的数据流机器来。

为了克服这种局限性，我们建议采用粗粒度数据流模型，即通过集聚更多的指令扩大结点的规模，形成宏结点。与细粒度相比，粗粒度更适合于当前的技术标准——因为已有的很多处理器能够以较高的效率支持粗粒度，［JD89］讨论了一类粗粒度模型的具体应用。本文将从另一角度探讨如何在粗粒度下寻求较高的效率。

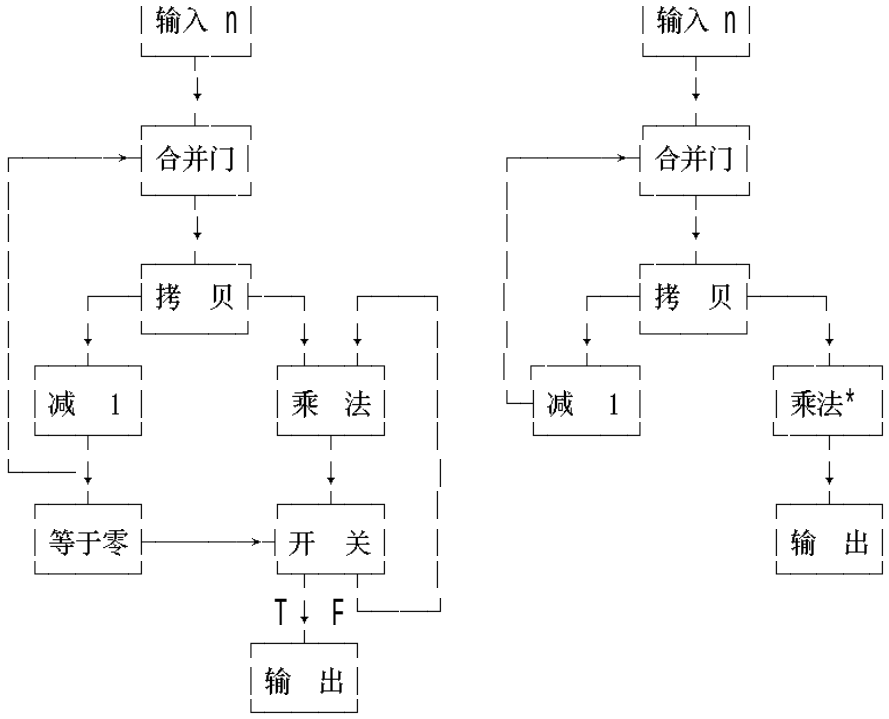
二、等待非操作字符使能执行的运算无法执行

胡国定［Hu88］、［Hu90］的工作表明：运算对象（操作数）可分为操作字符与非操作字符。如对于Post产生式 a'-'1'!'X'-'1'!'a'-'2'!'X'-'2'!'→b'-'1'!'X'-'2'!' 的算域（运算对象）α＝｛a'-'1'!',X'-'1'!',a'-'2'!',X'-'2'!'｝可分为α'-'1'!'＝｛a'-'1'!',a'-'2'!'｝和α'-'2'!'＝｛X'-'1'!',X'-'2'!'｝，α'-'1'!'在运算过程中起决定性作用，它决定了 b'-'1'!' 的输出值及输出中 b'-'1'!', X'-'2'!' 的排列顺序。而α'-'2'!'只是一个数据载体，在运算过程中不起决定作用。因此这个运算的执行只需知道 a'-'1'!',a'-'2'!' 的值，X'-'1'!' 与 X'-'2'!' 的值可暂不考虑，做为变量传递过去。还如对于求 A=B+I, 这里Ｂ为 n 阶方阵，Ｉ为 n 阶单位阵，我们可将算域Ｂ中的元素分为两组：α'-'1'!'＝｛b'-'ii'!' | i=1,…,n｝，α'-'2'!'＝｛b'-'ij'!' | i≠j｝ 其中α'-'1'!'为操作字符，α'-'2'!'为非操作字符。要进行 A=B+I 这个运算，只需知道α'-'1'!'中的值，α'-'2'!'中的值可暂时做为变量传递过去。关于运算中操作字符与非操作字符的划分、应用及进一步研究可参阅［1］、［Hu88］、［Hu90］。

以上分析表明，在粗粒度中如果仍旧运用传统的“数据流”概念，那将使得本来可执行的运算变得无法执行。因此我们必须改进点火规则，以达到对“数据流”的重新认识。

三、结点的函数性降低了运算速度，增加了不必要的通讯设施

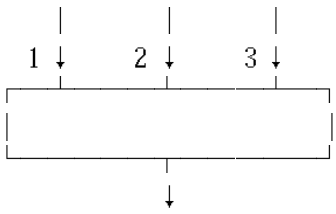
传统数据流模型图中，结点是一个纯粹的函数，即输出托肯仅与当前的输入托肯值有关，而与该结点以前的计算无关，因此我们可以称传统数据流模型的纯函数性为无记忆性。做为一个简单的例子，函数 f(n)=n! 的计算在传统数据流模型中可用图1(a) 表示，图中各个结点都无记忆性，特别地对于“乘法”结点也是如此。这就使得为了保留以前的运算结果必须使用一个开关结点作为信息反馈，因而大大增加了图的复杂性，同时也增加了数据流模型中最为昂贵的同步测试通讯费用。如果允许结点有记忆性，则我们可以构造图 1 (b) 的模型图。在“乘法*”结点中设置内部状态，记忆运算结果，并判断输入是否为零，从而决定是否输出。实质上，此时“乘法*”结点是一个带有存储器的有穷自动机。与图1(a) 相比较，图1(b) 并未减少算法的静态并行性，但它有明显的优越性：减少了通讯设备，废除了多余的数据传送与反馈，从而可大大加快计算速度。



'@M' 图 1． 阶乘的计算图'@A'

四．点火规则的局限性

传统数据流模型分为数据驱动与需求驱动两种。在数据驱动模型中，结点能动当且仅当相对应的托肯到达该结点的各条输入弧上。对于语句“If f'-'1'!'(弧1) then f'-'2'!'(弧2) else f'-'3'!'(弧3)”，我们用图 2 .的宏结点表示，该结点可动当且仅当 f'-'1'!', f'-'2'!' 与 f'-'3'!' 的变量数值分别到达弧1、弧2、弧3。实际上，如果判断 f'-'1'!' 为真，则弧 3 上 f'-'3'!' 的变量输入是无用的；当 f'-'1'!' 为假时，f'-'2'!' 的变量是无用的。因此我们认为有必要采取部分点火的规则。



'@M' 图 2
'@A'

上述四点分析论述了传统数据流模型的一些不尚令人满意的方面。要改进这些不足有一点是肯定的，那就是采用粗粒度数据流模型。至于说如何提高粗粒度数据流模型的效率，是一项有意义的值得探讨的问题，比如说可以采用层次并发的概念，也可以采用多值递归函数的结果。还有一点值得提出的是，如果新的模型以采用粗粒度性、引入内部状态等等概念来克服上述四点不足，则它的语义定义将变得艰难与繁琐。下节我们将尝试给出一个这样的模型。

'@p' § 2．粗粒度数据流模型ＣＤＦＭ'@A'

传统数据流模型中，一个数据流图由结点部分与通道部分组成，通道连接结点。每个通道都标以不同的名字。结点之间以及结点与外界通迅都是通过在通道上传递数据实现的，并且通道上的数据流动是以队列的方式进行的（FIFO）。通道可分为以下三类：

1．外部输入通道，外界向图传送数据；

2．内部通道，结点之间传送数据；

3．外部输出通道，结点向外界发送数据。

数据流图的动作是通过以下方式进行的：结点消耗输入通道上的数据同时在输出通道上产生新数据。

一般的大型数据流图是由若干小图复合而成的，复合方式比较简单：只要将外部输入通道、外部输出通道中同名的进行合并即可。如图3(a)、图3(b)的数据流图合并后可得图3(c) 的数据流图。

为了避免传统模型中的细粒度性和纯函数性（无记忆性），可将每个结点看成带有存储器的有穷自动机，即该结点不仅可以做简单的算术、逻辑或控制运算，而且还可以做较为复杂的运算；不仅具有纯函数性，而且具有历史记忆性。同时为了提高运算速度，避免等待非操作字符，引入变量的概念，让每个

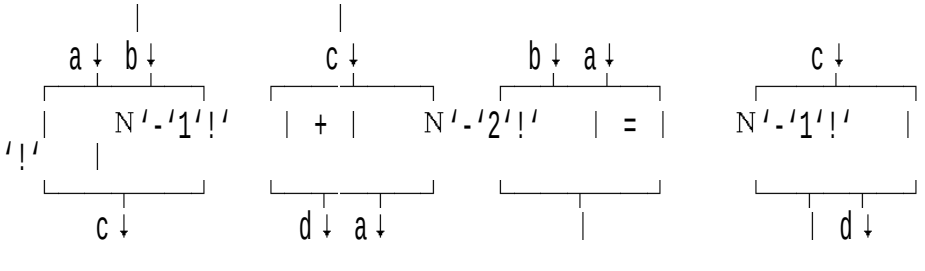


图３．数据流图的合并

结点有两条额外通道与之相连，一条用于读变量的值，另一条用于写变量，并且采用部分点火的规则，形式地有定义２．１：

注：在以下各节，都假定用V'-'1'!'表示数据集，用V'-'2'!'表示变量的地址集合，V表示V'-'1'!'∪V'-'2'!'。

定义２．１　结点 p 的形式规定是十元组　（I'-'p'!', R'-'p'!', W'-'p'!', O'-'p'!', S'-'p'!', s'+'0'!''-'p'!', q'-'waitp'!', Register'-'p'!', G'-'p'!', FS），其中：

I'-'p'!' 是通道集，称为ｐ的输入通道；

R'-'p'!' 是读通道，与地址总线相连（地址总线的概念在后介绍）；

W'-'p'!' 是写通道，与地址总线相连；

O'-'p'!' 是通道集，称为ｐ的输出通道；

S'-'p'!' 是状态集；

s'+'0'!''-'p'!' 是初始状态，s'+'0'!''-'p'!'∈S'-'p'!'；

q'-'waitp'!' 是等待状态，q'-'waitp'!'∈S'-'p'!'；

Register'-'p'!' 是ｐ的内部寄存器集；

G'-'p'!' 是点火集，即下述三种点火规则集合的并：

(1). 点火规则是<s,χ'-'in'!', s', χ'-'out'!',χ'-'w'!'>；

(2). 点火规则是<s, χ'-'in'!', q'-'waitp'!', χ'-'Register'!'>；

(3). 点火规则是<q'-'waitp'!', χ'-'register'!', χ'-'R'!', s', χ'-'out'!',χ'-'w'!'>；

这里 χ'-'in'!' 是从 I'-'p'!' 到 {ε} ∪ V 的映射；χ'-'register'!' 是从 Register 到 V 的映射；s, s'∈S'-'p'!'{q'-'waitp'!'} ；χ'-'R'!' 是该结点进入等待状态时所等待的变量的地址到 V'-'1'!' 的映射；χ'-'out'!' 是从O'-'p'!' 到V的映射；χ'-'W'!' 是该结点新生变量的地址到V'-'1'!'的映射。

下边对各类点火规则给予直观解释：

1．<s, χ'-'in'!', s',χ'-'out'!',χ'-'w'!'>：当结点 p 处于状态 s，每个输入通道i

n'-'i'!'
上的第一个值（记住各通道是ＦＩＦＯ队列）是 χ'-'in'!'(in'-'i'!') 时，结点可被点火；点火过程是这样的：消耗各输入通道 in'-'i'!' 上的输入值 χ'-'in'!'(in'-'i'!')，在每个输
出通道 out'-'j'!' 上产生χ'-'out'!'(out'-'j'!')，并放到 out'-'j'!' 的队列中去；将产生的全局变
量的值以 χ'-'w'!' 的形式通过写通道写入存储器中；最后结点 p 进入状态 s'。

２．<s, χ'-'in'!', q'-'waitp'!', χ'-'Register'!'>：当结点ｐ处于状态ｓ，每个输入通道in'-'i'!'上的第一个值是χ'-'in'!'(in'-'i'!') 时，结点可被点火。但很可能是由于本节点的操作字符有一些是以变量的形式（地址）传送过来的，尚未赋值，所以这
次点火只是进入等待状态 q'-'waitp'!'，并将各输入通道 in'-'i'!' 上的值χ'-'in'!'(in'-'i'!') 及
原状态 s 以χ'-'register'!' 的形式存入内部寄存器 Register 中。进入等待状态
的结点不能再被输入通道上的值点火。

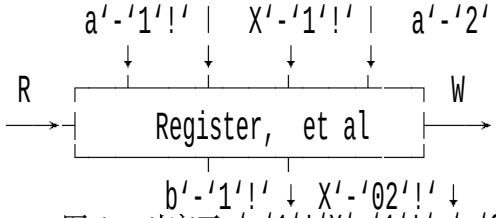
３．<q'-'waitp'!', χ'-'register'!', χ'-'R'!', s', χ'-'out'!', χ'-'w'!'>：进入等待状态的结点
p, 通过读通道去测试它所等待的各变量是否已赋值。如已赋值，则该结点可被激活，激活后，通过通道Ｒ读入变量的值，利用内部寄存器 Register 中保存的以前的输入值及原状态ｓ，在输出通道 out'-'j'!' 的队列尾部产生值χ'-'out'!'(out'-'j'!'),
并将产生的全局变量的值以 χ'-'w'!' 的形式通过写通道写入存储器中，最后结点进入状态ｓ'。

注：我们这里对变量的概念进行了推广，即变量在未赋值之前是可以访问的。

ＦＳ是对点火规则的合理性要求集，这主要依赖于具体的模型。比如，在上述模型中可要求第１、３两种经过中间等待状态而使结点发生的点火与第１种不经过等待状态的直接点火是相容的，也就是说，如果某些操作字符在输入通道上是以变量的形式传送过来使结点进入等待状态，等待这些变量被赋值，赋值之后激活结点而产生的输出值应和第一次操作字符在输入通道上以数值的形式传送过来而使结点以第一种方式点火后所产生的输出值相同（在操作字符相同的情况下）。

为了表达方便，以后我们用映射的图象表示映射本身。如用［(a'-'1'!', b'-'2'!'), (a'-'1'!', b'-'2'!')］表示映射：a'-'1'!'→b'-'1'!', a'-'2'!'→b'-'2'!'；而以 [f; (a'-'1'!', b'-'1'!'), … , (a'-'n'!', b'-'n'!')]
表示映射 f: a'-'i'!'→b'-'i'!'。

例２．１　图４是一个数据流结点的规定：



'@M'　　图４．对应于a'-'1'!'X'-'1'!'a'-'2'!'X'-'2'!'→b'-'1'!'X'-'2'!'的结点 p
'@A'

其中它的形式规定如下：

I'-'p'!'={a'-'1'!', X'-'1'!', a'-'2'!', X'-'2'!'}；
R'-'p'!'={R}；
W'-'p'!'={w}；
O'-'p'!'={b'-'1'!', X'-'02'!'}；

S‘-‘p‘!‘={s‘+‘0‘!‘‘-‘p‘!‘, q‘-‘waitp‘!‘};
s‘+‘0‘!‘‘-‘p‘!‘ 为初始状态;
q‘-‘waitp‘!‘ 为等待状态; G‘-‘p‘!‘=G‘-‘1‘!‘∪G‘-‘2‘!‘∪G‘-‘3‘!‘，其中
G‘-‘1‘!‘={<s‘+‘0‘!‘‘-‘p‘!‘, χ‘-‘in‘!‘, s‘+‘0‘!‘‘-‘p‘!‘, χ‘-‘out‘!‘,χ‘-‘w‘!‘>|
χ‘-‘in‘!‘=[(a‘-‘1‘!‘, x‘-‘1‘!‘), (X‘-‘1‘!‘, y‘-‘1‘!‘), (a‘-‘2‘!‘, x‘-‘2‘!‘), (X‘-‘2‘!‘, y‘-‘2‘!‘)],
χ‘-‘out‘!‘=[(b‘-‘1‘!‘, u=x‘-‘1‘!‘+x‘-‘2‘!‘), (X‘-‘02‘!‘, y‘-‘2‘!‘)],
χ‘-‘w‘!‘=[(address(u), u=x‘-‘1‘!‘+x‘-‘2‘!‘)], x‘-‘i‘!‘ ∈V‘-‘1‘!‘ and y‘-‘i‘!‘∈V};
G‘-‘2‘!‘={<s‘+‘0‘!‘‘-‘p‘!‘, χ‘-‘in‘!‘, q‘-‘waitp‘!‘, χ‘-‘Register‘!‘>|
χ‘-‘in‘!‘=[(a‘-‘1‘!‘, x‘-‘1‘!‘), (X‘-‘1‘!‘, y‘-‘1‘!‘), (a‘-‘2‘!‘, x‘-‘2‘!‘), (X‘-‘2‘!‘,y‘-‘2‘!‘)],
χ‘-‘Register‘!‘=[(r‘-‘1‘!‘, x‘-‘1‘!‘), (r‘-‘2‘!‘, y‘-‘1‘!‘), (r‘-‘3‘!‘, x‘-‘2‘!‘), (r‘-‘4‘!‘, y‘-‘2‘!‘), (r‘-‘5‘!‘, s‘+‘0‘!‘‘-‘p‘!‘)],
(y‘-‘i‘!‘∈V)∧(t((x‘-‘t‘!‘∈V‘-‘2‘!‘)∧((i≠t)→(x‘-‘i‘!‘∈V))), i=1, 2};
G‘-‘3‘!‘={<q‘-‘waitp‘!‘,χ‘-‘Register‘!‘, χ‘-‘R‘!‘, s‘+‘0‘!‘‘-‘p‘!‘,χ‘-‘out‘!‘, χ‘-‘w‘!‘>|
χ‘-‘Register‘!‘=[(r‘-‘1‘!‘, x‘-‘1‘!‘), (r‘-‘2‘!‘, y‘-‘1‘!‘), (r‘-‘3‘!‘, x‘-‘2‘!‘), (r‘-‘4‘!‘, y‘-‘2‘!‘), (r‘-‘5‘!‘, s‘+‘0‘!‘‘-‘p‘!‘)],
(x‘-‘1‘!‘∈V‘-‘2‘!‘∧x‘-‘2‘!‘∈V‘-‘1‘!‘)→(χ‘-‘R‘!‘=[(x‘-‘1‘!‘, z‘-‘1‘!‘)]∧χ‘-‘out‘!‘=[(b‘-‘1‘!‘, u=z‘-‘1‘!‘+x‘-‘2‘!‘), (X‘-‘02‘!‘, y‘-‘2‘!‘)]
∧χ‘-‘w‘!‘=[(adress(u), u=z‘-‘1‘!‘+x‘-‘2‘!‘)]);
(x‘-‘1‘!‘∈V‘-‘1‘!‘∧x‘-‘2‘!‘∈V‘-‘2‘!‘)→(χ‘-‘R‘!‘=[(x‘-‘2‘!‘, z‘-‘2‘!‘)]∧χ‘-‘out‘!‘=[(b‘-‘1‘!‘, u=x‘-‘1‘!‘+z‘-‘2‘!‘), (X‘-‘02‘!‘, y‘-‘2‘!‘)]
∧χ‘-‘w‘!‘=[(adress(u), u=x‘-‘1‘!‘+z‘-‘2‘!‘)]);
(x‘-‘1‘!‘∈V‘-‘2‘!‘∧x‘-‘2‘!‘∈V‘-‘2‘!‘)→(χ‘-‘R‘!‘=[(x‘-‘1‘!‘, z‘-‘1‘!‘), (x‘-‘2‘!‘, z‘-‘2‘!‘)]∧χ‘-‘out‘!‘=[(b‘-‘1‘!‘, u=z‘-‘1‘!‘+z‘-‘2‘!‘),
(X‘-‘02‘!‘, y‘-‘2‘!‘)]∧χ‘-‘w‘!‘=[(adress(u), u=z‘-‘1‘!‘+z‘-‘2‘!‘)]),
x‘-‘i‘!‘∈V, z‘-‘i‘!‘∈V‘-‘1‘!‘}。

FS含有下述三条规则：

规则1：

(χ‘-‘Register‘!‘=[(r‘-‘1‘!‘,x‘-‘1‘!‘), (r‘-‘2‘!‘, y‘-‘1‘!‘), (r‘-‘3‘!‘, x‘-‘2‘!‘), (r‘-‘4‘!‘, y‘-‘2‘!‘), (r‘-‘5‘!‘, s‘+‘0‘!‘‘-‘p‘!‘)]
∧(x‘-‘1‘!‘∈V‘-‘2‘!‘∧x‘-‘2‘!‘∈V‘-‘2‘!‘))
→(((q‘-‘waitp‘!‘,χ‘-‘Register‘!‘,χ‘-‘R‘!‘, s‘+‘0‘!‘‘-‘p‘!‘, χ‘-‘out‘!‘,χ‘-‘w‘!‘)∈G‘-‘3‘!‘)
→((s‘+‘0‘!‘‘-‘p‘!‘,χ‘-‘in‘!‘, s‘+‘0‘!‘‘-‘p‘!‘, χ‘-‘out‘!‘,χ‘-‘w‘!‘)∈G‘-‘1‘!‘))

这里

χ‘-‘R‘!‘=[(x‘-‘1‘!‘, z‘-‘1‘!‘), (x‘-‘2‘!‘, z‘-‘2‘!‘)], χ‘-‘in‘!‘=[(a‘-‘1‘!‘, z‘-‘1‘!‘), (X‘-‘1‘!‘, y‘-‘1‘!‘), (a‘-‘2‘!‘, z‘-‘2‘!‘), (X‘-‘2‘!‘,y‘-‘2‘!‘)]；

规则2：

(χ‘-‘Register‘!‘=[(r‘-‘1‘!‘,x‘-‘1‘!‘), (r‘-‘2‘!‘, y‘-‘1‘!‘), (r‘-‘3‘!‘, x‘-‘2‘!‘), (r‘-‘4‘!‘, y‘-‘2‘!‘), (r‘-‘5‘!‘, s‘+‘0‘!‘‘-‘p‘!‘)]
∧(x‘-‘1‘!‘∈V‘-‘2‘!‘∧x‘-‘2‘!‘∈V‘-‘1‘!‘))
→(((q‘-‘waitp‘!‘,χ‘-‘Register‘!‘,χ‘-‘R‘!‘, s‘+‘0‘!‘‘-‘p‘!‘,χ‘-‘out‘!‘,χ‘-‘w‘!‘)∈G‘-‘3‘!‘)
→((s‘+‘0‘!‘‘-‘p‘!‘,χ‘-‘in‘!‘, s‘+‘0‘!‘‘-‘p‘!‘,χ‘-‘out‘!‘,χ‘-‘w‘!‘)∈G‘-‘1‘!‘))

这里

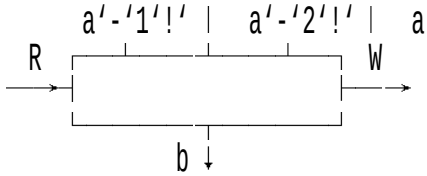
χ‘-‘R‘!‘=[(x‘-‘1‘!‘, z‘-‘1‘!‘)],χ‘-‘in‘!‘=[(a‘-‘1‘!‘, z‘-‘1‘!‘), (X‘-‘1‘!‘, y‘-

'1'!'), (a'-'2'!', x'-'2'!'), (X'-'2'!',y'-'2'!')]；
规则3：
(χ'-'Register'!'=[(r'-'1'!',x'-'1'!'), (r'-'2'!', y'-'1'!'), (r'-'3'!', x'-'2'!'), (r'-'4'!', y'-'2'!'), (r'-'5'!', s'+'0'!''-'p'!')]
∧(x'-'1'!'∈V'-'1'!'∧x'-'2'!'∈V'-'2'!'))
→(((q'-'waitp'!',χ'-'Register'!',χ'-'R'!', s'+'0'!''-'p'!',χ'-'out'!',χ'-'w'!')∈G'-'3'!')
→((s'+'0'!''-'p'!',χ'-'in'!', s'+'0'!''-'p'!',χ'-'out'!',χ'-'w'!')∈G'-'1'!'))
这里
χ'-'R'!'=[(x'-'2'!', z'-'2'!')],χ'-'in'!'=[(a'-'1'!', x'-'1'!'), (X'-'1'!', y'-'1'!'), (a'-'2'!', z'-'2'!'), (X'-'2'!',y'-'2'!')]。　　直观上讲，图4中对应于 a'-'1'!'X'-'1'!'a'-'2'!'X'-'2'!'→b'-'1'!'X'-'2'!' 的结点p是以这样的方式点火的：本结点的 a'-'1'!', a'-'2'!' 是操作字符通道；X'-'1'!', X'-'2'!' 是非操作字符通道。结点处于非等待状态 s'+'0'!''-'p'!' 下，当通道 a'-'1'!', a'-'2'!' 上传送来的是数据（即属于V'-'1'!'）；X'-'1'!', X'-'2'!'上传送来的是数据或变量地址（即属于V）时，结点可被点火，并且在输出通道 b'-'1'!' 上产生新字符（a'-'1'!' 与 a'-'2'!' 中值的和），而在输出通道 X'-'02'!' 上将输入通道X'-'2'!' 中的量拷贝过来（注：X'-'2'!' 中的量可能是已赋值的变量，也可能是尚未赋值的变量即变量的地址∈V'-'2'!'），在此同时，用写通道将变量 u=value(a'-'1'!')+value(a'-'2'!') 的值写到地址 address(u) 中去。同样，结点处于非等待状态下，当 a'-'1'!' 与 a'-'2'!' 上传递过来的量至少有一个是尚未赋值的变量的地址，X'-'1'!' ，X'-'2'!' 上传递过来的是已赋值或未赋值的变量时，结点也可点火，点火后，将 a'-'1'!', a'-'2'!', X'-'1'!', X'-'2'!' 中的量及原状态 s'+'0'!''-'p'!' 拷贝到结点的内部寄存器 Register中，并进入等待状态 q'-'waitp'!' ，此时拒绝接受输入通道中的数据。在等待状态下，结点通过读通道反复测试原 a'-'1'!', a'-'2'!' 中尚未赋值的变量；变量一经赋值，结点马上被激活，利用读通道 R 从存储器中读取变量的值，在输出通道 b'-'1'!' 上产生结点进入等待状态之前通道 a'-'1'!' 与 a'-'2'!' 上变量的和，同时将此值存入 address(u) 中；在输出通道 X'-'02'!' 上将原 X'-'2'!' 的值拷贝过去。

FS中的三条规则是为了保证，不论结点p点火经过中间等待状态与否，最后在 b'-'1'!' 上输出的都是 a'-'1'!' 与 a'-'2'!' 上变量的和，而 X'-'02'!' 上输出的是 X'-'2'!' 中的量。

例2.1说明了我们将要建立的模型可以避免§1中谈到的传统数据流模型的第二点不足：等待非操作字符。下边通过另一个例子说明新模型同样可以克服传统模型的第四点不足：点火规则的局限性。至于传统模型的第一、三点不足新模型显然是克服了的。从而我们可以认为我们将要建立的模型是一种比较理想的模型。

例2.2　图5是一个数据流结点的规定：



'@M'　　图5. 对应于 If f(a'-'1'!') then f(a'-'2'!') else f(a'-'3'!') 的结点 p

'@A'

它的形式规定如下：

I'-'p'!'={ a'-'1'!', a'-'2'!', a'-'3'!' }；

R'-'p'!'={ R }；

W'-'p'!'={ W }；

O'-'p'!'={ b }

S'-'p'!'={ s'+'0'!''-'p'!', q'-'waitp'!', s'-'1'!', s'-'2'!' }；

s'+'0'!''-'p'!' 为初始状态，q'-'waitp'!' 为等待状态；　　Register={ }；

G'-'p'!'=G'-'1'!'∪G'-'2'!'∪G'-'3'!'，其中：

G'-'1'!'={<s'+'0'!''-'p'!', χ'-'in'!', s'-'i'!',χ'-'out'!',χ'-'w'!'>|

χ'-'in'!'=[(a'-'1'!', x), (a'-'2'!', ε), (a'-'3'!', ε)],χ'-'out'!'=[(b, ε)]

,

χ'-'w'!'=[ε], x∈V'-'1'!', f'-'1'!'(x)→(i=1), (f'-'1'!'(x))→(i=2)};

G'-'2'!'={<s'-'1'!', χ'-'in'!', s'+'0'!''-'p'!', χ'-'out'!', χ'-'w'!'>|

χ'-'in'!'=[(a'-'1'!', ε), (a'-'2'!', x), (a'-'3'!', ε)],

χ'-'out'!'=[(b, y)], χ'-'w'!'=[(address(y), y)], x,y∈V'-'1'!' };

G'-'3'!'={<s'-'2'!', χ'-'in'!', s'+'0'!''-'p'!', χ'-'out'!', χ'-'w'!'>|

χ'-'in'!'=[(a'-'1'!', ε), (a'-'2'!', ε), (a'-'3'!', x)],

χ'-'out'!'=[(b, y)],χ'-'w'!'=[(address(y), y)], x,y∈V'-'1'!' };

ＦＳ={ }。

直观上讲，该结点以这样的方式执行语句If f'-'1'!'(a'-'1'!') then f'-'2'!'(a'-'2'!') else
f'-'3'!'(a'-'3'!')：第一步，测试 f'-'1'!'(a'-'1'!') 的值。若 f'-'1'!'(a'-'1'!') 为真，则进入状态 s'-'1'!' ，否
则进入状态 s'-'2'!'。第二步，根据状态为 s'-'1'!' 或 s'-'2'!' 相应地执行 f'-'2'!'(a'-'2'!') 或 f'-'3'!'(a'-'3'!')。

在本节的最后，我们给出数据流图的定义：

定义２．２　数据流网络Ｎ是结点连同通道的集合Ｐ。粗粒度数据流模型ＣＤＦＭ'-'N'!' 是数据流网络Ｎ的一个规定，即ＣＤＦＭ'-'N'!' 是一个十一元组：（I'-'N'!',
O'-'N'!', R'-'N'!', W'-'N'!', Register'-'N'!', Σ'-'N'!',σ'+'0'!''-'N'!', G'-'N'!', ＦＳ'-'N'!', Bus'-'N'!', Memo'-'N'!'），其中

I'-'N'!'=(∪'-'p P'!'I'-'p'!')\050'-'p P'!'O'-'p'!')是 N 的输入通道;

O'-'N'!'=(∪'-'p P'!'O'-'p'!')\050'-'p P'!'I'-'p'!')是 N 的输出通道;

R'-'N'!'=∪'-'p P'!'R'-'p'!';

W'-'N'!'=∪'-'p P'!'W'-'p'!';

Register'-'N'!'=∪'-'p P'!'Register'-'p'!';

ＦＳ'-'N'!'=∪'-'p P'!'ＦＳ'-'p'!'

Memo'-'N'!' 是存储器;

Bus'-'N'!' 是地址总线, 连接 R'-'N'!', W'-'N'!' 中的读写通道与存储器 Memo'-'N'!';

Σ'-'N'!' 是 N 的构形集, 即 P∪C∪Register'-'N'!'∪Memo'-'N'!' 上的映射σ的集合。直
观上讲，对于 Σ'-'N'!' 中的任一构形σ，任给结点 p∈P，σ(p)为 N 的当前状态;
任给通道 c∈C，σ(c) 为通道 c 上的当前数据队列；任给寄存器 Register'-'p'!'
∈Register'-'N'!'，σ(Register'-'p'!')为该寄存器中的当前值;任给 v∈Memo'-'N'!',σ(v)为存
储器单元 v 中的当前值;

σ'+'0'!''-'N'!' ∈Σ'-'N'!',将每个结点 p 映射为 s'+'0'!''-'p'!',将 C,Register,Memo 映射为空值ε;

G'-'N'!' 是变迁集,其可能性有以下三种:

1. 内部变迁τ：σ→σ',如存在结点 p 的点火规则:

<1>. <s,χ'-'in'!', s',χ'-'out'!',χ'-'w'!'>;或

<2>. <s,χ'-'in'!', q'-'waitp'!',χ'-'Register'!'>;或

<3>. <q'-'waitp'!', χ'-'Register'!', χ'-'R'!', s', χ'-'out'!', χ'-'w'!'>
使得τ： σ→ σ'是 p 的一次点火,形式地说(注：这里只对情形<1>.<s,χ'-'in'!',
s', χ'-'out'!',χ'-'w'!'> 加以说明,其余二情况类似),下列四条件成立： A：σ(p)=s,σ
'(p)=s';
σ(p')=σ'(p'), If p≠p';
B：σ(in'-'i'!')=σ'(in'-'i'!')·x'-'i'!', If χ'-'in'!'=[(in'-'1'!', x'-'1'!'),
… ,(in'-'n'!', x'-'n'!')]
and in'-'i'!'∈I'-'p'!';
σ'(out'-'i'!')=y'-'i'!'·σ(out'-'i'!'), If χ'-'out'!'=[(out'-'1'!', y'-'1'!
'), … , (out'-'m'!', y'-'m'!')],
and out'-'i'!'∈O'-'p'!';
σ(c)=σ'(c), If not c∈I'-'p'!'∪O'-'p'!';
C：σ(Register'-'p'!')=σ'(Register'-'p'!') If p∈P;
D：σ'(v'-'i'!')=z'-'i'!', If (v'-'i'!', z'-'i'!')∈χ'-'w'!'; 这里v'-'i'!', v'-
'j'!'∈Memo'-'N'!'。
σ'(v'-'j'!')=σ(v'-'j'!'), If z(v'-'j'!', z)∈χ'-'w'!'.
2．输入变迁τ：σ→σ'，若存在 in'-'i'!'∈I'-'N'!', x∈V'-'1'!' 使得
σ'(in'-'i'!')=x·σ(in'-'i'!'), and σ'(in'-'j'!')=σ(in'-'j'!') for i≠j.
3．输出变迁τ：σ→σ'，若存在 out'-'j'!'∈O'-'N'!', y∈V'-'1'!', 使得
σ(out'-'i'!')=σ'(out'-'i'!')·y and σ(out'-'j'!')=σ'(out'-'j'!') for i≠j
.

有了上述形式化的定义，我们很容易定义数据流模型ＣＤＦＭ'-'N'!' 的一个计算：

σ'-'N'!''+'0'!'→σ'-'1'!'→σ'-'2'!'→ …

通过进一步的深入分析，我们会发现，虽然在我们的模型中多了存储器 Memo，地址总线 Bus，等待状态，结点内部寄存器，全局变量及对未赋值的变量可访问等功能，但这些额外功能只是起到充分利用算法的静态并行性，加速计算速度的作用。也就是说，这些推广是语法上的，在语义上，应与传统模型相类似。即计算的语义仍然依赖于每个输入通道上输入值的顺序、每个输出通道上输出值的顺序及输入与输出之间的序对关系。因而传统模型的各种语义稍作必要的修改就应能适用于我们的模型ＣＤＦＭ，相通在传统模型中各语义所具有的良好性质在模型ＣＤＦＭ中仍然具有，验证这些性质是一件冗长而又乏味的事，在此略去。

'@p' §３ ＣＤＦＭ的应用'@A'

本节将讨论如何用模型ＣＤＦＭ去解决实际问题，也就是说本节的目的在于讨论：如何对于一般的算法利用模型ＣＤＦＭ去寻找它的最大静态并行性。本节的结构是这样的：３.１概述数据流语言ＤＧＬ；３.２介绍ＵＳ ＭＭＣＭ机器；３.３介绍胡国定给出的将ＵＳ ＭＭＣＭ并行化的途径；３.４介绍用ＣＤＦＭ将ＵＳ ＭＭＣＭ并行化的方法，并介绍对于一般问题如何利用ＣＤＦＭ将该问题并行化。

３.１ 有向图语言ＤＧＬ

ＤＧＬ是 R.Jagannathan 等人的小组在ＳＲＩ计算机科学实验室开发出的一种适用于工作站的图形语言[JD89]，它的文本形式的表示可以看成是Ｃ语言的扩充或图形语言的类Ｃ文本化，该实验室也开发出了需求－驱动方式的执行系统，可对ＤＧＬ进行解释、执行，运行效果良好，我们在本节运用类ＤＧＬ 语言来描述模型ＣＤＦＭ。如果对本文的类ＤＧＬ语言进行适当修改，作者相信一定能在ＳＲＩ实验室的执行系统上运行，关于ＤＧＬ的详细描述可参考有关文献。

３.２ 计算模型ＵＳ ＭＭＣＭ

（１）.记号

若B为一集合，N为自然数。则<B>'-'N'!'＝｛α｜α∈B'+'*'!'｜α|≤N｝。
若v表示a'-'1'!'a'-'2'!'a'-'3'!'…a'-'n'!'，a'-'i'!'∈A，则：

v 表示a'-'1'!'a'-'2'!'a'-'3'!'…a'-'n'!'，n＝0时，规定v＝ ε（空字）

v 表示a'-'1'!'a'-'2'!'a'-'3'!'…a'-'n'!'，n＝0时，规定v＝ε（空字）

v 表示a'-'1'!'a'-'2'!'a'-'3'!'…a'-'n'!'，n＝0时，规定v＝ε（空字）

n＝1时，规定v＝a'-'1'!'

（2）,计算模型MMCM

关于MMCM的详细定义参见〔1〕，这里只给出简要描述。

定义：一个MMCM M是一个六元组：

（A，W，Q，Q'-'h'!'，F，P）

其中 <1>. A是一个有穷字母表；

<2>. W A'+'*'!' 为初始字符集；

<3>. Q为有穷状态集； Q'-'h'!' Q为停机状态集；

<4>. F：E'-'1'!'×（Q－Q'-'h'!'）→E'-'2'!'×Q；

这里E'-'1'!' <A'∪V>'-'N'!'，E'-'2'!' <A∪A'∪V∪V'> '-'N'!'，而V＝｛v'-'1'!'，v'-'2'!'，…，

v'-'n'!'｝，V'＝｛v，v，v｜v∈V｝，A'＝｛a｜a∈A｝， E'-'1'!'中的元素

形如α＝v'-'1'!'a'-'1'!'v'-'2'!'a'-'2'!'…v'-'m'!'a'-'m'!'v'-'m+1'!'；2m+1≤N；
若设F（α，q）＝（α'-'1'!'，q'-'1'!'），
则（α'-'1'!'，q'-'1'!'）仅由a'-'1'!',a'-'2'!',…,a'-'m'!',q决定且要求：若v或v，v，v在α中出现，
则v必在α'-'1'!'中出现。

<5>. P为如下形式的算程集合

（α，q'-'0'!'）→（α，q'-'1'!'）→ … →（α，q'-'t'!'）；

我们来看如何实现其中的一步：

（α，q'-'i'!'）→（α，q'-'i+1'!'）；α ∈（A∪A'）'+'*'!'

首先设α =a'-'1'!'a'-'2'!'…a'-'i '!'…a'-'i '!'…a'-'i '!'…a'-'n'!'，令

v'-'1'!'=a'-'1'!'a'-'2'!'…a'-'i-1'!'; v'-'2'!'=a'-'i+1'!'…a'-'i-1'!'; …; v'-'m'!'=a'-'i +1'!'…a'-'i -1'!';v'-'m+1'!'=a …a'-'n'!'.

这样我们得到α =v'-'1'!'a'-'i'!'v'-'2'!'…v'-'m'!'a'-'i'!' v'-'m+1'!', 再利用F得F(α,q'-'i'!')=(α,q'); 将α 中的v'-'i'!'用上述定义的字符代回便得α ;而q'=q'-'i+1'!'。

另外还要求q'-'0'!'为初始状态,q'-'t'!'∈Q'-'h'!'；α 中去掉箭头后便是W的元素。

（3） 一致可分US MMCM机器

定义：一个MMCM 机器M称为一致可分的（Uniformly Separable）如果
它的指令集

F：E'-'1'!'→E'-'2'!'

或写作：(v'-'1'!'a'-'1'!'v'-'2'!'a'-'2'!'…v'-'m'!'a'-'m'!'v'-'m+1'!',q)→(α,q');
满足下列条件

<1>. α的长度只依赖于m、q；

<2>. q'只依赖于m、q；

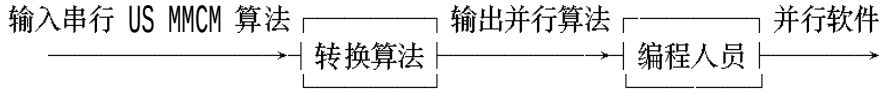
<3>. (V∪V')中的字母哪个在α中出现以及在α中的具体位置只依赖于m、q；

<4>. A′ 中字母在α中出现的位置只依赖于m、q；

<5>. (A∪A′)里的字母哪个在α里出现由a′-′1′!′,a′-′2′!′,…,a′-′m′!′,q决定。

3.3 一致可分US MMCM的并行化

胡国定在[Hu90]中提出了对于一类US MMCM可并行化的充分条件，并给出了从串行到并行的具体转化算法（图6）。



′@M′ 图6. 胡国定之方法

′@A′

本文作者在[1]中描述了这种算法的数据结构及其形式表示，并用 LISP 语言编写了自动转化软件，设

P: α′-′0′!′→α′-′1′!′→ … →α′-′n′!′

是 US MMCM 机器 M 的一个算程，利用[1]中的自动转换软件将该算程P中各步分类得到 I′-′1′!′,I′-′2′!′, … , I′-′l′!′, 满足：

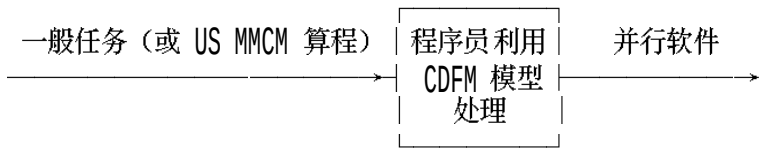
<1>. ∪I′-′i′!′={1*, 2*, … ,n*};

<2>. I′-′i′!′∩I′-′j′!′={ } for i≠j;

<3>. If I′-′i′!′={n′-′1′!′*, n′-′2′!′*, … , n′-′k′!′*}, Then 在算程 P 中, 第 n′-′1′!′*, … ,

n′-′k′!′* 步是可并行执行的。

利用模型CDFM可以省去[Hu90]中的转换算法这一步，而直接在已知



′@M′ 图7. CDFM 模型运用过程′@A′ US MMCM算程的情况下编制并行软件，达到先利用转换算法再编制软件的效果（图7）。当然CDFM不局限于US MMCM机器，对于一般的工作任务，它都是可实用的。

3.4 在 Sun 操作系统下的工作站上执行 US MMCM 机器的算法

设对于 N 长输入，M的算程是：

P: α′-′0′!′→α′-′1′!′→ … →α′-′n′!′

步骤1：对算程P中每一步 i*: α′-′i-1′!′→α′-′i′!′，定义一个CDFM结点 i′-′M′!′。例如，

当 i*: α′-′i-1′!′→ α′-′i′!′具有形式

a′-′1′!′X′-′1′!′a′-′2′!′X′-′2′!′→b′-′1′!′X′-′2′!′

时，其中 a′-′1′!′，a′-′2′!′ 是操作字符，X′-′1′!′，X′-′2′!′ 是非操作字符，b′-′1′!′ 是新生字符。可用

图4及例2.1中的形式规定来充当该产生式 i* 的CDFM结点 i′-′M′!′。

步骤2：将步骤1中所得到的结点及它们的各通道用类DGL语言描述出来。例如，对于对应于产生式 a′-′1′!′X′-′1′!′a′-′2′!′X′-′2′!′→B′-′1′!′X′-′2′!′ 的CDFM结点及例2.1中的

形式规定可用下述图8的类DGL语言来描述。其中图8的程序可分为3段

<1>. 类型定义；

<2>. 各边描述；

<3>. 结点的描述。

步骤3：将步骤2中所得到的各程序块以图9的方式连接起来：

通过上述步骤得到的图9中的类DGL语言可以在SRI 实验室中 Sun OS

操作环境中“需求－驱动”系统下执行；因为是“需求－驱动”方式，所以执行的方式一定和先通过文 [1] 中对算程分类，然后编写并行软件所执行的方式一样，因此我们可以说利用模型ＣＤＦＭ可以省去文 [Hu90] 中提出的转换步骤，大大减轻了编程人员的工作。在本文的结尾，我们给出一个将ＣＤＦＭ用于普通任务的框架步骤：

１．将任务Ｔ分解成若干子任务 T'-'1'!', … ,T'-'k'!';

２．对每个子任务 T'-'i'!'，求出其操作字符输入，非操作字符输入，新产生输出及保留字输出，然后对子任务 T'-'i'!' 给出相应的ＣＤＦＭ结点；

３．将各结点规定及输入、输出通道用类ＤＧＬ语言描述出来；

４．综合第三步中的各子描述形成任务Ｔ的类ＤＧＬ语言；

５．上机调试，形成软件。'@M'

本文是在胡国定先生的鼓励、徐书润副教授和胡久稔副研究员的亲自指导下完成的，作者在此表示衷心的感谢。'@A'

```
             node_i'-'M'!'_Definition
                (in'-'1'!', in'-'2'!', in'-'3'!', in'-'4'!', &out'-'1'!', &out'-'2'
!', R, W);
            typedef char operating;
            typedef char string[100];
            typedef string reserved;
            typedef char newborn;
            typedef rw_channel real;
            genedge operating in'-'1'!', in'-'2'!';
            genedge newborn out'-'1'!';
            genedge reserved in'-'2'!', in'-'4'!', out'-'2'!';
            genedge rw_channel R, W;
            {
            /* in'-'1'!', in'-'2'!', in'-'3'!', in'-'4'!' are the incomming edge of
 node i'-'M'!' */
            /* and out'-'1'!', out'-'2'!' are the outgoing edge of node i'-'M'!' */

            /* R, W are the Read and Write edge of this node resp. */
            /* in the following for each edge specify it's type */
            /* specification */
            edge TYPE edge_name;
                 …  …
            edge TYPE edge_name;
            /* the following is the function description of this node*/
            /* i.e. the S'-'p'!', s'+'0'!''-'p'!', q'-'waitp'!', G'-'p'!', Register
'-'p'!' and FS */
            /* can be defined here */
           function_description_of_this_node
                (p1, p2, p3, p4, &r'-'1'!', &r'-'2'!', Read, Write);
              {
              … …
               }
           }
```

'@M'　　图8.结点 i'-'M'!' 的类 DGL 语言'@A'

```
 /*  typedefpart  */
 ┌──────────────────────────┐
 │  the definition of all types  │
 │    indicated in each node     │
 │   and edge can be put here    │
 └──────────────────────────┘
/* edges part */
 ┌──────────────────────────┐
 │       the definition of       │
 │  all edges will be put here   │
 └──────────────────────────┘
/* nodes part */
 ┌──────────────────────────┐
 │       the definition of       │
 │  all nodes will be put here   │
 └──────────────────────────┘
/* the main part */
main( )
{
 node_i'-'1'!'_definition;
 node_i'-'2'!'_definition;
      … …
 node_i'-'n'!'_definition;
}
```

'@M'　　图9. 对应与 US MMCM 的类 DGL 语言'@A'

'@p' 参考文献:'@A'

[1] 王永革　一个从串行 ＵＳ ＭＭＣＭ 到并行机器的数据结构及其算法,1990.
(待发表)。

[2] 王永革 序佩特里网(Petri)计算能力分析 《软件学报》(已接受发表)
[3] 徐书润 多值递归函数, 《华中理工学院学报》, vol.2,1979, pp8-20.
[Ag86] G.Agha. Actors: A Model of Concurrent Computation in Distributed Systems. MIT Press,Cambridge,Mass., 1986.
[Ag89] G.Agha. Supporting Multiparadigm Programming on Actor Architectures,in E.Odijk M.Rem and J.-C.Syre (Eds.),PARLE'89, LNCS 366. pp.1-20, 1989.
[AN90] Avind and Nikhil,R.S.,"Executing a Program on the MIT Tagged-Token Dataflow Architecture",IEEE Trans. on Comput.,Vol.39,No.3, pp300-318,March 1990.
[BA81] J.D.Brock and W.B.Ackerman. Scenarios: a model of non-determinate computation. In Diaz and Ramos(Eds.), Formalization of Programing Concepts, LNCS 107,pp.252-259, Springer Verlag, 1981.
[BP90] M.Beck and K.Pingali. From control flow to data flow,In D.A.Padua (Ed.), Proc. 1990 ICPP,Vol II Software,pp53-60, The Pennsylvania State University Press,1990.
[Br88] M.Broy. Nondeterministic data flow programs: how to avoid the merge anomaly. Science of Computer Programing, 10:65-85,1988.
[GJ89] Gluck-hiltrop,E.,Johnk,M.,Schurfeld,U.. The stollmann Data Flow Machine, in E.Odijk M.Rem and J.-C.Syre(Eds.) PARLE'89, LNCS 365 pp.216-234.
[Hu88] Hu Guoding(胡国定). On the Mathematical Model of Computing Machine, J. of Computer Sci. and Technology,Vol.3,No.4,1988.
[Hu90] Hu Guoding(胡国定). Parallel Computation Simulating Sequential Computation,1990.(to appear).
[IE82] IEEE Computer Vol.15, No.2, 1982.
[JD89] R.Jagannathan, A.R.Downing, W.T.Zaumen,and R.K.S.Lee. Dataflow-Based Methodology for Coarse-Grain Multiprocessing on a Network of Workstations, in E.C.Plachy, P.M.Kogge(Eds.), Proc.1989 ICPP, pp.II209-216.
[JK89] B.Jonsson, J.N.Kok. Comparing Two Fully Abstract Dataflow Models, in E.Odijk M.Rem and J.-C.Syre(Eds.), PARLE'89, LNCS 366 pp.216-234.
[KP85] R.M.Keller and P.Panangaden. Semantics of networks containing indeterminate operators. In Brookes,Roscoe,and Winskel(Eds.), Seminar on Concurrency 1984, LNCS 197,pp.449-496,1985.
[Pa83] D.Park. The "fairness" problem and nondeterministic computing networks. In de Bakker and van Leeuwen(Eds.), Foundations of Computer Science IV, Part 2, pp.133-161, Amsterdam, 1983. Mathematics Center Tracts 159.
[Sh85] Sharp,S.A. Data Flow Computing, ELLIS HORWOOD LIMITED,1985.
[SK86] Sargeant,J. and Kirkham,C.C., "Stored Data Structures on the Manchester dataflow machine" Proc. 13th Annu. Sympo. Comput. Architecture. pp.235-242, 1986.
[WT90] H.Wu and L.-E.Thorelli. Extending data flow Principles for multiprocessing, In D.A.Padua(Ed.), Proc.1990 ICPP,Vol II Software, pp53-60, The Pennsylvania State University Press, 1990
[Xu89] Xu Shuruen(徐书润). Petri Net-like Languages(to appear)